IPv6 in Practice

Benedikt Stockebrand

IPv6 in Practice

A Unixer's Guide to the Next Generation Internet

With 53 Figures

Springer

Benedikt Stockebrand

contact@benedikt-stockebrand.net
www.benedikt-stockebrand.net

Library of Congress Control Number: 2006934616

ISBN-10 3-540-24524-3 Springer Berlin Heidelberg New York
ISBN-13 978-3-540-24524-7 Springer Berlin Heidelberg New York

Typesetting: By the Author
Production: LE-TEX Jelonek, Schmidt & Vöckler GbR, Leipzig
Cover design: KünkelLopka Werbeagentur, Heidelberg

Printed on acid-free paper 45/3100/YL - 5 4 3 2 1 0

To my parents

Preface

In the Beginning there was—Frustration

Back in early 2000 I first tried to get seriously started with IPv6. But I couldn't find any documentation that helped me to understand how to make it work in my usual environment. Being swamped with work at my then job I eventually gave up, frustrated for the first time.

In 2002 Silvia Hagen published the first edition of "IPv6 Essentials" [52]. Expecting a hands-on guide to IPv6 I bought it, only to be frustrated again: The book told me a lot more about the IPv6 protocol than I expected but virtually nothing about how to make it work.

This time I didn't give up. I read the book and learned a lot about the underlying concepts. With this knowledge I managed to understand the IPv6-related documentation available for individual Unixen, like Peter Bieringer's Linux IPv6 Howto [10] or the FreeBSD and Solaris online documentation.

It was much like studying mechanical engineering just to learn how to ride a bicycle. So I started teaching others how to get IPv6 up and running at conferences and various training courses. During that time I wrote a first training manuscript and an article series [104, 105, 106] on IPv6 administration.

Since then IPv6 has noticeably matured. Not only have the core protocol specifications become reasonably stable, but the actual implementations have reached a usable state. This made it possible to turn the training course manuscript into something less volatile: The book you are now reading.

What This Book is *Not* About But Why You Might Want to Read It Anyway

This book is *not* about

- basic Unix and TCP/IP network administration,

- what the fifth bit in the fifty-sixth byte of a neighbor discovery request packet means,
- how to make IPv6 work on dedicated router hardware, or Microsoft Windows, or
- any of the fancy new features people talk or write their PhD thesis about but never bother to implement at a production-grade level.

Instead it addresses the Unix-based implementations available today. It tries to tell you how to sit on a bicycle, put your feet on the pedals and get rolling without hurting yourself and innocent bystanders more than necessary—and never mind how that fancy gearbox[1] works.

So if you want to learn about IPv6 by making it work, this book is written for you.

The Unixen Considered

This book itself explains how to configure and run IPv6 on three different Unixen: Debian GNU/Linux, FreeBSD and Solaris. These three differ in many respects:

Debian Sarge *Since the Sarge release most applications support IPv6, but Linux in general is still missing some important IPv6 features, like an IPv6-capable port mapper, so some features available with the other Unixen are still missing. Additionally, configuring IPv6 in the network configuration files is still awkward.*

There is work underway to replace the current IPv6 implementation with a port of the KAME stack from the BSDs; the project is called USAGI. *At this time the USAGI stack is still considered experimental and doesn't generally ship with Linux distributions, so we don't consider it yet.*

FreeBSD 6.1 *FreeBSD was the primary development environment of the* KAME *project, which implemented IPv6 for the BSDs.*

The IPv6 implementation has been integrated into the system quite smoothly. Some deprecated features, like automatic tunnels, have been silently removed, which may cause occasional problems with older installations that still want to use these features.

Solaris 10 *IPv6 support has been available with Solaris for some time and is quite mature. The major drawbacks are that in some cases it doesn't implement all the more recent changes in the specifications and that its handling is sometimes noticeably different than with the other Unixen.* 1

Together these three give a fair overview of IPv6 with Unix. Beyond them, a number of other Unixen, as well as updates to the three shown in this

[1] See `http://www.rohloff.de/en/technical/speedhub/index.html` if you really want to know about the gearbox.

book, will be covered in online supplements available from my home page at `http://www.benedikt-stockebrand.net/` together with an errata list and an online copy of the book's index. So if your Personal Pet Unix is missing, take a look there and you may find what you need.

How to Read This Book

Since you won't learn how to ride a bicycle without having a bicycle at hand, you will need a test environment. It is easiest to use virtual machines, like Xen or (as in my case) VMware.

Throughout this book you will see a variety of Unixen in a number of test setups, plus a few more Unixen at my home page. I recommend you first stick with your Personal Pet Unix. Dealing with IPv6 will be difficult enough in a few cases; using an unfamiliar Unix at the same time will only cause unnecessary pain.

The chapters are arranged in a way to put things to work as soon as possible.

The first part deals with fundamental topics that are virtually impossible to skip. There are however sections called either "Inside IPv6" or "Packet Filter Considerations" which you probably want to ignore on first reading. The "Inside IPv6" sections provide some details of the inner workings of IPv6 that are sometimes useful for debugging or just interesting by themselves. The "Packet Filter Considerations" provide additional information necessary to set up a packet filter, from protocol details like port numbers to architectural suggestions.

The following parts address topics that may be irrelevant to you, so feel free to skip whatever you don't need. If you care about security however, a basic understanding of these topics and their security implications is essential, so you want at least to skim these parts.

Finally, there are two appendices, one giving a crash course on DNS administration with BIND and the other providing a list of various well-known addresses and port numbers, plus a bibliography and an index.

Security Considerations

When you do your very first steps with IPv6 you don't want to bother about packet filter configuration and other security measures just yet. Neither do you want to disrupt network operation within your company network.

So *please* first use IPv6 in a test-only environment disconnected from production environments or the Internet. There are some interactions between IPv4 and IPv6 and we can't deal with them right from the start.

If you really have to start with IPv6 in a production environment, read the first three parts in full, so you know about the most relevant security issues

with IPv6 itself and the interactions between IPv4 and IPv6. Only afterwards start to use IPv6 in your environment.

Trying things in a test environment, making them work, and only afterwards dealing with packet filters and other security issues is obviously preferable; use packet filters from the start only if you absolutely have to.

Typographic Conventions

Throughout the book you will find sections that deal with implementation-specific details. They look like this:

Debian Sarge *is a Linux distribution particularly popular with Linux administrators and developers.*

FreeBSD 6.1 *comes with the KAME stack, probably the most complete IPv6 implementation available.*

Solaris 10 *has implemented IPv6 quite early. IPv6 support is well integrated, but sometimes the handling is slightly unusual.* 2

The number at the bottom refers to the related section in the online supplements covering additional implementations.

Shell transcripts ("screen shots") look like this unfortunate specimen:

```
# nice --20 rm -rf / &
# fg
^C^C^C^C^C^C^[^[^[^[^[^\^\^\^\^\^\^\^\
```

Following Bourne shell standards a hash mark ("`#`") as a prompt indicates that the commands shown must be run as root while a dollar sign ("`$`") implies that the commands don't require root privileges.

File listings look like this:

/etc/resolv.conf

```
domain example.com
nameserver 2001:db8::1
```

Occasionally you will find variables within both shell transcripts and file listings, appearing as "⟨*Interface Name*⟩". More often however you will find examples like "`eth0`" instead. The highlighted background marks those items that you will likely need to adapt to your needs or that will look different on your system.

When we've set up something, there is usually a checklist following. It shows how to ensure in a systematic way that everything works as expected. Let's say that you have just logged in:

- ☐ Read the "Last logged in" message to make sure nobody else used your account since you last logged in.

- ☐ Check your disk quotas to make sure you still have enough space left.
- ☐ Read your e-mail for messages from your administrator (if you are a user) or your users (if you are an administrator).

These lists usually don't tell you in detail how to fix a problem, but following them usually helps either to ensure that something works as expected or to find out more precisely what the actual problem is.

Network plans look like figure 0.1. Routers are drawn as circles while hosts (or "non-routers") are square shaped—we defer the exact definition of hosts and routers to section 4.3.2. Individual subnets are always drawn as oblong boxes, even though the coax cabling this presentation is derived from is rarely used anymore. Contiguous sets of subnets and routers like the Big Bad Internet above are called *clouds* and drawn as such.

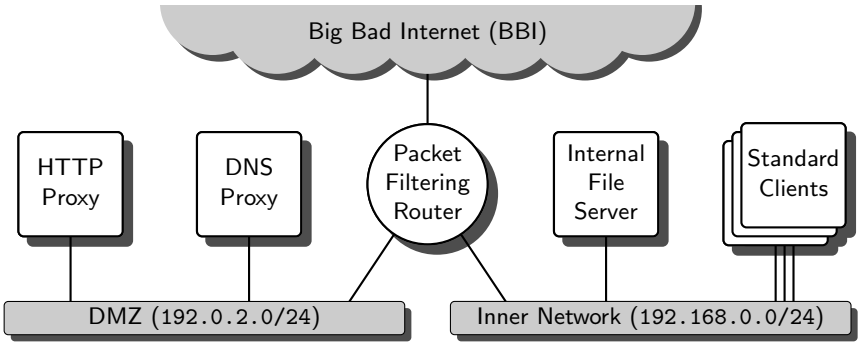

Fig. 0.1. A sample network plan

Whenever we look at how IPv6 works, we'll see protocol flow diagrams that look like figure 0.2. This example shows the TCP "three way handshake", which applies to IPv6 as well as IPv4.

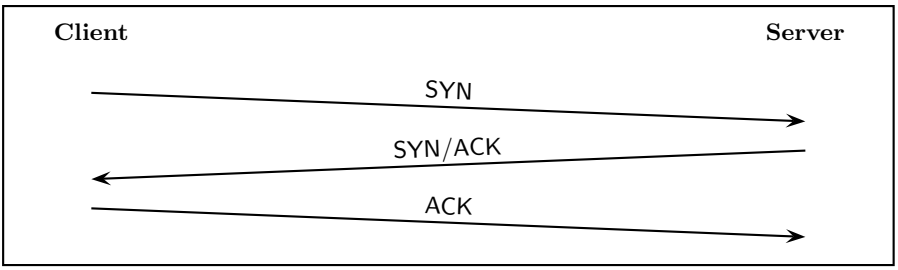

Fig. 0.2. The TCP three way handshake as a protocol flow diagram

Occasionally we do things that are potentially insecure or address security problems in existing implementations. Whenever you see a warning like

For your first attempt to ride a bicycle choose a location easily and quickly accessible to an ambulance but away from major traffic. Make sure to wear a helmet, gloves, properly padded protective clothes and safety goggles.

please make sure you understand what it means before you proceed. Similarly, open problems that are yet unresolved look like this:

So far, no reliable strategy is known how to learn bicycling without getting more or less seriously hurt. Research is still continuing and there is hope that virtual reality will eventually solve this problem.

Acknowledgments

This book wouldn't have happened without a number of people who helped me through a number of difficult stages.

Before I even started to think about writing this book, the unnamed participants of various workshops showed me what aspects of IPv6 they were interested in and let me refine the organization and presentation of IPv6 administration in the way that this book is written.

René Schönfeldt and Bert Ungerer convinced me to write an article series for iX magazine and made me believe that it might just be feasible to turn the training manuscripts into a book. Silvia Hagen, who didn't even know me at that time, told me quite honestly that she didn't think it was; she was right in 2004 and almost right in 2006.

Dr Frank Schmidt convinced me to start writing. When he left Springer, Jutta-Maria Fleschutz took over his job of guiding a certain debutant writer through the book-writing process and helped me to deliver a printable manuscript.

All that time the JOIN IPv6 mailing list was a low-volume high-signal forum that repeatedly helped me out when I was stuck or unsure if I was heading in the right direction. Especially the discussions with Gert Doering, Jeroen Massar, Pim van Pelt and the now disbanded JOIN IPv6 team were immensely helpful to me.

Dr Peter Bieringer, Reiner Krapohl and Wolfgang Zenker spent hours and days of proofreading the raw manuscript, providing a treasure of comments and suggestions. They pointed out various mistakes and a number of ambiguous or just awkward wordings without dispiriting me. Of course, all remaining mistakes are mine alone.

Thank you all for your support.

Darmstadt, July 2006 Benedikt Stockebrand

About the Author

Benedikt Stockebrand received his "Diplom-Informatiker" degree at Dortmund University in Germany. He has been using, operating and designing Unix-centric TCP/IP networks since 1989.

His professional career started as a programmer and system and network administrator. Having gathered some experience, he gradually shifted his focus to system architecture and design, turning whatever software into performant, scalable, reliable, secure and generally datacenter-ready environments. During this time he also worked as an instructor, enjoying it so much that in 2002 he changed his professional focus again and started to work full-time as a trainer, consultant and occasional IT journalist all over the world, specializing in the operational aspects of IPv6.

In his spare time he travels a lot—so far alone 21 000 km (13 000 miles) by bicycle—and occasionally goes scuba diving.

If you want to contact him, for example if you have a question, found an error in this book or look for a freelance IPv6 trainer, he can be reached both by e-mail as `<contact@benedikt-stockebrand.de>` or through his personal web site at `http://www.benedikt-stockebrand.net/`.

Contents

Part I

Getting Started

1

A Quick Overview of IPv6

To understand what IPv6 is and what it is not, what features to look out for, and how it fits into the TCP/IP stack, this chapter provides a rough overview.

1.1 Terminology: IP, IPv4, IPv6 and the Internet

When we talk about "traditional IP" from now on, we use the term *IPv4*, which is short for *Internet protocol, version 4* as of RFC 791 [32] and related documents.

Its successor protocol is called *IPv6*, or *Internet protocol, version 6*. It is defined in RFC 2460 [24] and related standards.

Whenever we talk about *IP*, from now on we talk about the "Internet Protocol" family in general. This includes all network layer protocols from the TCP/IP stack, as explained later on in section 1.3: IPv4, IPv6 and any future successor to both.

On a similar line, when we talk about the *Internet*, we talk about the global network connected using IP. The *Internet4* is the part of the Internet that uses IPv4 and the *Internet6* is the part that uses IPv6. The Internet4 and Internet6 are not strictly disjoint, but this distinction is very helpful when we address the issues concerned with the interoperation of both.

Finally there are *protocol families* or *address families* that denote an entire family of protocols using the same addressing scheme. The *INET* address family includes IPv4 as well as all protocols running on top of IPv4, like TCP or UDP over IPv4. Similarly, the *INET6* protocol family includes IPv6 and all other protocols using IPv6 addresses or running on top of IPv6.

1.2 The "IPv6 Sales Pitch"

What are the differences that make IPv6 superior to IPv4? The most visible differences fall into two categories: Changes that solve fundamental inade-

quacies of traditional IPv4 and new features that were first introduced with IPv6.

The features resolving fundamental problems with IPv4 that made a redesign necessary include these:

Larger address space Probably the most essential advantage of IPv6 over IPv4 is its enlarged address space. While IPv4 addresses are 32 bits long, IPv6 uses 128 bit addresses. These long addresses resolve the address scarcity issues getting more severe every day.

Abolition of NAT With IPv6 there is no need to connect multiple machines to the Internet using a single address and *network address translation* (*NAT*). Without NAT, *end-to-end connectivity* becomes available again, allowing machines to connect to each other without intermediate "broker" services, like mail exchangers/relays, web proxies, DNS forwarders or SIP gatekeepers, that are run by a service provider.

At first glance this doesn't seem like much of an advantage, but at this time its consequences are barely fathomable, making services possible that are difficult even to imagine to our NAT-conditioned minds.

Simplified address structure With the large address space there is no more need for configurable network masks, thus simplifying network configuration and disposing of an ever annoying source of misconfiguration.

Simplified address configuration The large address space allows for a simplified address configuration mechanism, providing a service similar to the dynamic host configuration protocol (DHCP) but avoiding the need to maintain state information about address leases.

Replacing DHCP with a minimum-configuration, stateless mechanism simplifies network configuration even more and eliminates another common cause of network problems.

Simplified address renumbering With the address configuration mechanism it is perfectly feasible to change addresses throughout an entire network during normal operations without touching or even rebooting any machine connected.

IPv4 network renumberings put a network temporarily down and require a serious effort, thus making network reorganizations expensive and risky. This problem ties many customers to their Internet service providers (ISPs). With IPv6 it is feasible to reorganize networks or switch ISPs without disruption of network services.

Improved multicast The multicast address range has been vastly extended, making use of a wide range of "scopes" that define the domain within which an address is used. Multicasts as well as multicast routing are base features of IPv6.

Routed multicasts are a functionality necessary to build "self-configuring" network services and more efficient "intelligent broadcast" services like "Internet Radio", among other things.

Abolition of broadcast With the extended multicast functionality IPv6 doesn't have any further need for IPv4-style broadcasts.
This makes IPv6 invulnerable to attacks that use remote broadcasts such as "ping bounce" or "smurf" denial of service attacks, while it still supports all the "reasonable" features that IPv4 broadcasts are used for. As another advantage over broadcasts, multicasts are only processed on those nodes which have actively signalled that they are interested in the particular multicast group. This reduces the load on all other machines.

Streamlined routing tables With IPv4, address ranges were assigned in an ad-hoc style and for unlimited time. Medium to large organizations obtained *provider-independent addresses* (*PI addresses*) and then connected through one or several ISPs, leading to an excessive growth of routing table entries in the "backbone" routers at the top network service providers. With IPv4 addresses becoming ever more precious and renumberings being virtually infeasible these organizations refuse to release these addresses they hold.
IPv6 doesn't provide PI addresses, it makes renumberings easy and far less risky, it only assigns addresses on a non-permanent basis and provides such an abundance of addresses that hoarding them doesn't make sense. As a consequence, routing tables in the core routers are several orders of growth shorter with IPv6 than with IPv4; and even when the Internet6 grows, the routing tables will mostly stay at at their current size.

All these features are deeply incorporated into the IPv6 design, making them readily available.

In addition, some more advanced features were standardized that don't solve a problem with existing IPv4 but implement new functionalities:

Network traffic security with IPsec The standards expect a full implementation of IPv6 to include network layer encryption and authentication using IPsec as a mandatory feature. Among other advantages of fully integrated network traffic encryption this provides the means to encrypt traffic even within a local network, thus providing protection from insiders trying to sniff network traffic.
IPsec has been backported to IPv4 as an optional feature with little or no loss of functionality. More or less usable implementations are available though the key exchange protocols still show interoperation problems.
While Microsoft Windows XP (SP2) currently limits itself to the "NULL" encryption algorithm, other implementations do provide strong end-to-end encryption.

Mobile IPv6 The IPv6 standards include a feature called "Mobile IPv6". This allows "roaming" while maintaining a "home" network address at all times, keeping all existing network connections open even while the underlying network connectivity changes. While Mobile IPv6 has a number of mind-boggling security implications, "roaming" provides the base technology for a wide range of mobile applications.

The standards for mobile IPv6 have been released fairly late. Implementations are based on preliminary drafts of the standards and should be considered experimental.

IPv4 offers a similar optional feature; it has been added to IPv4 only lately though, severely restricting its functionality compared to IPv6.

Quality of service (QoS) support Several standards addressing quality of service have been released that specify how near-realtime functionality can be incorporated into IPv6. While quality of service is still an emerging technology, near-realtime applications like IP telephony may well make good use of this feature.

Implementations are not yet readily available; with the political issues involved it remains questionable if end-to-end quality of service support will ever become generally available.

The near-realtime features defined for IPv6 haven't been backported to IPv4 and it is unlikely they will ever be.

IPsec may be considered the most mature of these features, but even IPsec isn't fully usable in a production environment. Certificate-based authentication and multicast support are still missing from implementations.

Even though mobile IPv6 and quality of service are very exciting—and scary in the case of mobile IPv6—they are neither essential to the setup and operation of IPv6 nor are they stable enough to be used in a production environment yet.

1.3 IPv6 and the TCP/IP Stack

What exactly is IPv6? You may have a reasonable idea of what the "standard" TCP/IP stack looks like. Maybe you've read the standard "TCP/IP Illustrated" by the late W. Richard Stevens [103], or any other of the wide

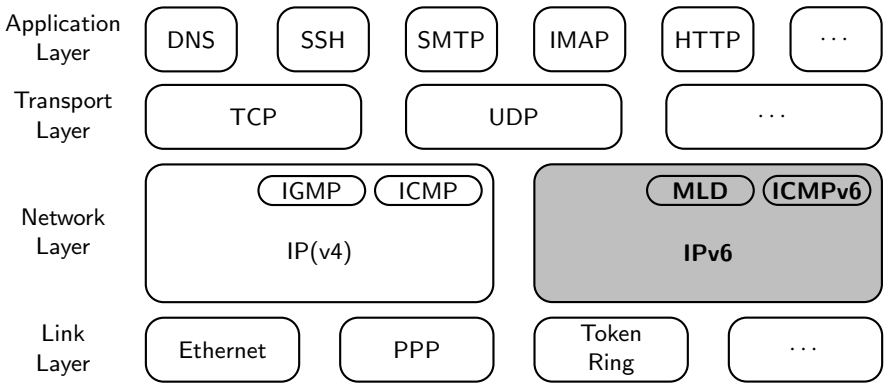

Fig. 1.1. IPv6 and its role in the traditional TCP/IP stack

range of introductory books on TCP/IP. So except for the highlighted IPv6 part, figure 1.1 may look reasonably familiar to you. If you have never seen it, this is how it works: The network stack is organized in four different layers, communicating only with the layers immediately above and below them (except in one case we'll see below). Every layer provides a specific functionality to the layers above:

Link Layer The link layer transmits data packets, called *frames*, between devices directly connected to the same physical network. The archetypical link layer is *Ethernet* in one of its many physical implementations.

Network Layer Devices connected to different physical networks can communicate through the network layer. An IP packet is sent from one device to another by being wrapped up in a link-layer frame and then being sent either to the recipient if it is connected to the same physical network, or to an intermediate device called a "router". A router that receives a frame first unpacks the IP packet within. If the packet is not addressed to the router itself it decides where to forward the packet to—either another router or the destination device. It re-wraps the packet in another link-layer frame and sends it out the the next link-layer destination. Eventually the packet arrives at its destination.

Transport Layer While the network layer only addresses devices, like computers, the transport layer adds *port numbers* to its communication to help the destination device pass the communication to a particular process. There are two major transport layer protocols: The *transmission control protocol* (*TCP*) implements a virtual connection, taking care of the retransmission of lost or damaged network layer packets and the ordering of packets. The *User Datagram Protocol* (*UDP*) simply sends individual packets, called *datagrams* to a destination process but doesn't provide for a connection or the handling of lost packets.

Application Layer Applications use the transport layer to implement communication between processes on different computers to provide a specific functionality. Applications access the network layer directly when they deal with IP addresses, usually when they try to address their communication peers; this is the one exception to the rule that any layer only communicates with the layers immediately above or below.

A somewhat unusual application layer protocol is the *domain name system* (*DNS*). It provides a translation service turning a host name like `www.example.com` into an IP address and vice versa. Virtually all application programs use this service, so from an application developer's point of view the DNS conceptually belongs to the transport or network layer even though the protocol definition puts it in the application layer.

So how does IPv6 fit in? The figure already explains two essential properties of IPv6.

First of all, IPv6 is a network layer protocol; it doesn't interfere with the transport layer. You may sometimes read about "TCPv6", which doesn't

really exist; usually this means “TCP over IPv6”. More important, since most application software uses the transport layer interface most of the time, it is usually fairly straightforward to make IPv4 applications support IPv6. The majority of work involved deals with the (usually minor) tweaks necessary to support the larger addresses whenever the application needs to deal with addresses directly.

Next, IPv6 runs in parallel with IPv4 even to the point that they “share” a single interface. Legacy systems that need IPv4 continue to work even when IPv6 is enabled; they just require the extra administration effort to maintain them. There won’t be a “great switchover” on a fixed “flag day” that needs to be organized all over the world. Instead, the core strategy to deploy IPv6 in any existing environment is a soft migration, introducing IPv6 in small, easily reversible steps.

2

Preparing for IPv6

Using IPv6 in a Unix network requires a number of straightforward but important preparations.

This chapter presents some suggestions about obtaining globally routed IPv6 addresses, setting up a test environment and a few security precautions. Following that, it explains how to enable and test IPv6 support within the kernel.

2.1 Obtaining Our Own IPv6 Address Prefix

IPv6 addresses are virtually unlimited and we can and should obtain our own `/48` address prefix from our friendly IPv6-enabled ISP.

So the one important step is to contact our ISP and request an IPv6 prefix. But if they turn out to be of the distinctly IPv6-unfriendly kind we still have several options:

1. Find an IPv6-friendly ISP. Especially small ISPs tend to be fairly cooperative to such a request. In the long run this is the one reasonable approach, even though the short-term trouble of switching ISPs may be prohibitive.
2. Find a tunnel provider to connect to. Hexago[1] in Canada and SixXS[2] in the Netherlands are the most widely known; both offer free tunnels to end users at an international scale. Other tunnel providers start to become available, so if we can't get native IPv6 connectivity we are best advised to look for a tunnel provider in our vicinity. While latency doesn't

[1] `http://www.hexago.com/`. At the time of this writing they provide quick and easy access with a minimum of hassle, letting anybody set up a tunnel in less than fifteen minutes.

[2] `http://www.sixxs.net/`. They expect their users to maintain their tunnel up and running 24/7 and won't route any traffic until it has been for an entire week, but in Europe the latency is much better than through Hexago.

exactly improve from using tunnel providers, this is a fairly easy and quick approach to establish basic Internet6 connectivity.
3. If we have a statically assigned globally routed IPv4 address, we can use it to generate our own "6to4 prefix". While 6to4 is not without its problems, it does provide us with a means to connect to the Internet6 even without support from our ISP or a tunnel provider.
4. Use private addresses that won't be routed through the Internet. How they are allocated and used will be explained later on. This is fine if we don't want to connect to the Internet6—and it provides much the same treacherous feeling of security that NAT does. In the long run however, this approach isn't much use since it doesn't let us connect to the Internet6.
5. Just use the address prefix reserved for documentation purposes. This is quite generally *not* a good idea but it turns especially troublesome if we later on try to connect to the Internet6. But as long as we don't try to connect to the Internet6, using it is a reasonable last resort.

Again, we should do ourselves and the entire IPv6 community a favour and first ask our ISP about IPv6 connectivity. Not only will it make it more difficult for them to claim that we "are the very first customer ever to ask for it"—having obtained our own prefix will make it much easier to get IPv6 up and running than any other approach.

2.2 Setting Up Our Test Environment

While we wait for our ISP to assign us an IPv6 prefix, this is the time to set up our environment.

Nobody ever learned how to ride a bicycle just from reading a book about it; very likely you won't learn how to set up and run IPv6 from just reading this book, either. You will need to give it a proper try, and doing so requires a network environment, no matter how small.

2.2.1 Choosing the Hardware

IPv6 doesn't need more resources than IPv4; a few old 486 PCs will do fine. More important is the number of machines and network interfaces. While most examples in this book try to minimize the number of machines needed, dynamic routing simply doesn't make sense with two machines and a single network interface in each.

In addition to the computers themselves we also need some network equipment. Again, this doesn't need to be anything particularly fancy; if all we have is some old Thicknet equipment, that's what we use. There is however one requirement for the networking equipment: We must keep it easily reconfigurable. Having to find the network administrator to re-configure the

VLANs of a big-iron switch or an operator with a key to the networking rack will quickly become exceedingly time-consuming.

My personal favourite however is a virtual test environment; in my case I use VMware, though Xen may soon become an even better alternative. Using these has a number of advantages, from being able to keep the test environment disconnected from anything else to very quick network reconfigurations to a virtually unlimited number of virtual machines that can easily be cloned and taken snapshots of. So if you have a machine with a reasonable amount of RAM (from 1 GB up) this may be the way to go for you, too.

2.2.2 Supplementing the System Installation

All example setups in this book use a fairly minimal installation; if additional packages are necessary to make such an installation work, they will be explicitly mentioned where they are needed.

To simplify life it is useful to prepare a few things right from the start, even if they are not strictly essential.

Choosing the Kernel Especially with Linux it is useful to select a recent kernel version since IPv6 support is still evolving.

Debian Sarge *If at all possible we use a 2.6 series kernel here. All examples in this book refer to it.* [3]

Almost-essential packages With some Unixen we should install a few extra packages right from the start.

While the `bash` shell isn't strictly necessary I personally consider it de facto essential on any system except for very limited appliance-style installations. If you prefer the Korn, C, Z or whichever shell, install that instead.

We will definitely need tools like `ping`, `traceroute`, `netstat` and `route` or their IPv6 counterparts on all systems.

Debian Sarge *The packages `iputils-tracepath` and `iproute` are necessary to use `traceroute6` and the `ip` utility, respectively.*

FreeBSD 6.1 *The `bash` package is located on the second CD-ROM ("Disc 2").*

Solaris 10 *Assuming a "Reduced Networking Core System" we need to install the packages `SUNWgssc`, `SUNWgss` and `SUNWbip` from the installation media for such essential tools as `ping`. The `bash` shell is located in the `SUNWbash` package.* [4]

Man pages and Whatis index If we don't have another machine with these installed, we should make sure we have them on our test system, including an up-to-date Whatis index.

Debian Sarge *The man pages as well as the Whatis index are automatically installed even with a minimal installation. Additionally, installing `debian-reference-en` is a good idea.*

FreeBSD 6.1 *To install the man pages and Whatis index (if we have done a "Minimal" installation) we mount the CD-ROM labelled "Disc 1", change to the* `6.1-RELEASE/manpages` *directory on the CD-ROM and run the* `./install.sh` *script.*

Solaris 10 *We need to install the packages* `SUNWlibC`*,* `SUNWdoc` *and* `SUNWman` *to install the man pages. A subsequent invocation of "*`catman -w`*" updates the Whatis index.* 5

Extended logging It is often helpful to direct all `syslog` messages to a single file. During system setup and debugging I habitually use an entry like

`/etc/syslog.conf`

```
*.debug                    /var/log/debug
```

in `/etc/syslog.conf` on all systems. If you do so too, make sure to create the file with permissions 0600 before restarting the `syslogd`.

Once the system turns productive, don't forget to undo this configuration again, otherwise it will unnecessarily swamp the `/var` file system.

Packet sniffers Sooner or later we need to use a packet sniffer in our environment, either to find a problem or to explore in more detail how IPv6 works.

If we don't have X11 installed we are effectively stuck with `tcpdump`, `snoop` (on Solaris) or possibly `tethereal`. I personally prefer `ethereal`, or `wireshark` as has been recently renamed. All the traces in this book are done with it, but it requires X11 so it isn't generally available.

2.2.3 Backup and Disaster Recovery

If we have to use an IPv4 production system for our experiments, this is an excellent moment to do a full backup and make sure we can actually do a successful disaster recovery.

If we use a dedicated test environment it is also an excellent moment to do a full backup so we can revert our steps if anything goes wrong.

If we use VMware, or Xen, or another sort of virtualized environment, this is the time to explore its snapshot features.

In general we should run a backup or snapshot whenever we reach some usable state. Things *will* go wrong, and chances are we won't be able to do a reliable roll-back by hand.

2.3 Security Precautions

Before we first enable IPv6 on a machine we should make sure we don't open any unexpected security holes to our environment. Aside from any security measures specific to the environment we should consider the following list.

- ☐ If at all possible use a test environment disconnected from production networks and the Internet.
- ☐ You need an IPv6-free test environment. Check with the local network administrators if you are not personally in charge.

If IPv6 is already in use in your environment, don't try to set it up there. Playing around with the IPv6 equivalents of DHCP and dynamic routing in a production environment will cause you as well as your network administrators serious pain.

- ☐ If the system is connected to the Internet, disconnect it if possible. If you have to use 6to4 addresses for the IPv6 setup, it is essential to disconnect until the environment is up and running; otherwise it is possible that you are attacked using 6to4 tunnels across your IPv4 infrastructure.
- ☐ If you can't disconnect from the Internet and are willing to take the extra risk, make sure your firewall (there is one, right?) does *not* pass IPv6. Check its interface configuration for addresses in hexadecimal and for lines containing the string "`inet6`".
- ☐ If you can't block IPv6 on your firewall *you must set up your own, disconnected test-only environment.*

At the time of this writing (early 2006) IPv6 is slowly gaining a reputation for being neglected as a possible security hole in "IPv4-only" networks. Many systems and applications already support IPv6, so once an attacker has got into a network, IPv6 may be used locally to circumvent the existing IPv4 security restrictions.

Enabling IPv6 in an environment doesn't exactly minimize this problem, so please, don't take any unnecessary chances. Even though you may later on realize that some of these precautions are a bit overly restrictive, chances are that you will miss some advanced feature, like one of the tunnel mechanisms, opening a security hole to the environment.

2.4 Kernel IPv6 Support

Finally it is time to enable IPv6 on the test machines and take a quick peek at the relevant kernel configurables.

2.4.1 Enabling IPv6 Within the Kernel

The canonical way to check for IPv6 support is the `ifconfig -a` command. At least for the loopback interface it should show an address line with the address family "`inet6`" and the funny looking address "`::1`".

With FreeBSD 6.1 this already works. With Debian Sarge and Solaris 10 this doesn't suffice for two different reasons: With Debian Sarge we need to

load an additional kernel module while Solaris 10 only "plumbs" interfaces if they are configured. Knowing this it is reasonably straightforward to enable IPv6 temporarily.

Debian Sarge *A simple*

```
# modprobe ipv6
```

does the trick, unless you have built a custom kernel without IPv6 support.

FreeBSD 6.1 *The standard kernel already supports IPv6 and the loopback interface should be up and running without further action. To enable IPv6 on a physical interface* `lnc0` *we need to invoke*

```
# ifconfig lnc0 inet6 up
```

after a reboot.

Solaris 10 *The command*

```
# ifconfig lo0 inet6 plumb up
```

plumbs and enables IPv6 on the loopback interface. 6

Afterwards the `ifconfig -a` output should show an IPv6 address `::1` for the loopback interface until the next reboot.

Enabling IPv6 permanently is just as straightforward.

Debian Sarge *We just add a line*

/etc/modules

```
ipv6
```

in `/etc/modules` *and reboot.*

FreeBSD 6.1 *Adding a line*

/etc/rc.conf

```
ipv6_enable="YES"
```

to `/etc/rc.conf` *permanently enables IPv6 on all interfaces after a reboot or an explicit* `/etc.rc.d/network_ipv6 start`.

Solaris 10 *For an interface* `pcn0`*, creating a file* `/etc/hostname6.pcn0` *enables IPv6 on that interface. As soon as IPv6 is enabled on any interface this way, the startup scripts also enable IPv6 on the loopback interface. For now we create an empty file for at least one physical interface and reboot.* 7

Checking if IPv6 support is permanently enabled is easiest done like this:

- ☐ Reboot the system.
- ☐ Check if `ifconfig -a` lists an `inet6` address of `::1` for the loopback interface.

- ☐ Ping that address; either use the `ping` command (Solaris 10) or the IPv6-specific `ping6` command (Linux, BSD).

If you have built your own custom kernel without IPv6 support you will need to do so again with IPv6 enabled.

Linux *Depending on the kernel version, enable the kernel options "Code maturity level options" → "Prompt for development and/or incomplete code/drivers" and "Device Drivers" → "Networking support" → "Networking options" → "The IPv6 protocol" while doing a* `make menuconfig` *or similar. Alternatively, set the* `CONFIG_IPV6` *parameter in your* `.config` *file with an editor. Afterwards rebuild and reinstall your kernel.*

FreeBSD 6.1 *Re-add the line*

`/sys/i386/conf/CUSTOM`

```
options INET6
```

in the kernel configuration file, reconfigure, rebuild and reinstall your kernel. [8]

At this point IPv6 should be successfully enabled in the kernel.

2.4.2 IPv6-related Kernel Variables

Linux, the BSDs and Solaris offer access to a number of interesting IPv6-related kernel variables.

Debian Sarge *The command*

```
# sysctl -a | egrep ^net.ipv6
```

lists all IPv6-related kernel variables.

FreeBSD 6.1 *Similar to Debian,*

```
# sysctl net.inet6
```

does the job.

Solaris 10 *Finding the relevant kernel variables is a bit more tedious here.*

```
# ndd /dev/⟨device⟩ \?
```

queries all variables defined for the given ⟨*device*⟩*, which is any of* `ip6`*,* `icmp6`*,* `rawip6`*,* `tcp6` *and* `udp6`*. Knowing the variable name and the device, a second invocation*

```
# ndd /dev/⟨device⟩ ⟨variable⟩
```

yields the value of that variable. While this is more tedious to use than the `sysctl` *based queries of Linux and the BSDs, it offers far more information.* [9]

It may be tedious to sift through these variables for hints of a misconfiguration but doing so can be extremely helpful if the system doesn't behave as expected.

2.5 Packet Filter Considerations

Throughout the book there are a number of sections entitled "Packet Filter Considerations". They are generally irrelevant unless we actually set up a packet filter for IPv6. In that case you should read the entire chapter including the packet filter section before you apply it to your system. Otherwise, especially when you read the chapter for the first time, you can safely skip the packet filter sections.

2.5.1 Available Implementations

Packet filter support for IPv6 is generally disappointing: In some cases it simply isn't available at all, in other cases it doesn't support *stateful filtering* (also called *connection tracking*) or some other fundamental filtering criteria. Nevertheless, packet filtering is essential for secure IPv6 operation, especially since we can't use a NAT gateway as a "block everything initiated from outside" catch-all solution.

Packet filters differ in several ways: Some implement IPv4 and IPv6 filtering entirely independent of each other while others use a unified filter framework.

Some have a *first match semantic* where the first matching rule determines if a packet is passed or not while others use a *last match semantic* where the last matching line applies, usually with a `quick` option that provides a first match semantic on individual rules.

Some filters classify packets as incoming or outgoing while others distinguish between packets originating locally, being forwarded and being delivered locally.

While traditional packet filters use a single, linear filtering table with a feature to skip table entries, todays filters usually apply a different syntax with multiple strands of execution. These are called *chains* or *anchors*, depending on the filtering framework.

Debian Sarge *Linux uses a dedicated IPv6 filtering mechanism called* `ip6tables`*. It behaves much like IPv4-only* iptables*, so it uses a first match semantic with a separate "forward" classification and arbitrary filter chains. At the time of this writing (February 2006)* `ip6tables` *doesn't support connection tracking yet but can handle some option headers.*

FreeBSD 6.1 *Currently there are two IPv6-capable packet filters available: The more traditional IPv6-only* `ip6fw` *and the new* `pf` *recently imported from*

the OpenBSD project. Since `ip6fw` *doesn't support connection tracking we will only consider* `pf`*. It uses a last match semantic with a "quick" option, handles both IPv4 and IPv6 and supports connection tracking. It doesn't handle option headers except on an all-or-nothing basis. Considering some regressions from FreeBSD 6.0 to FreeBSD 6.1, IPv6 support in* `pf` *still seems somewhat immature.*

Solaris 10 *There is no IPv6-capable packet filter available.* 10

2.5.2 Basic Configuration

The configuration examples in this book put their emphasis on clarity, not performance. If we needed high performance we would use chains or anchors to minimize the number of rules that must be evaluated for any packet. We would also reorder the rules in such a way that those with the largest number of matches are put towards the beginning of each chain. We would use features like `pf`'s "tables" to speed up lookups of various kinds. We would tune a number of kernel parameters that control the expiration rate of state information, memory allocation within the kernel and similar performance-related attributes.

But this is not a book about advanced packet filtering tricks (which doesn't make sense with todays IPv6 packet filters anyway); neither is it an introduction to packet filtering basics. But if you are reasonably comfortable with configuring packet filters for IPv4, then you should find all the information you need to set up an IPv6 filter as well.

Debian Sarge *The boot scripts don't provide any support to set up IPv6 packet filter rules. Instead, we need to add some* `pre-up` *and* `up` *statements in* `/etc/network/interfaces`*.*

The most simple approach we can possibly come up with is a script that is run before we finally bring up an interface. To start it, we add a line

`/etc/network/interfaces`

```
pre-up /etc/ip6tables.sh
```

to every interface configuration in `/etc/network/interfaces`*. The script we run might look like this for now:*

`/etc/ip6tables.sh`

```
#! /bin/bash
-------------------------------------------------------------------------
/sbin/ip6tables -P INPUT   DROP    || Clean the slate and DROP by default
/sbin/ip6tables -P FORWARD DROP
/sbin/ip6tables -P OUTPUT  DROP
/sbin/ip6tables --flush
/sbin/ip6tables -X
/sbin/ip6tables -P OUTPUT  ACCEPT
```

To install the rules temporarily, we just run the script. A subsequent

```
# ip6tables --list --verbose --numeric
```

will show that we have successfully set up the filter.

Note that the filter rules shown here will silently drop filtered packets. Despite the fact that the `ip6tables` *man page already mentions a* `REJECT` *target this is still being developed and doesn't work at least with Debian and the kernel version it ships with. As we'll see later in section 4.7.4, silently dropping packets in conjunction with IPv6 is a particularly bad habit.*

Additionally, we need to accept output packets by default; for some reason the multicast listener reports will otherwise be filtered even if we let ICMPv6 traffic pass.

FreeBSD 6.1 *To enable the* `pf` *filter at boot time we first add a line*

/etc/rc.conf

```
pf_enable=YES
```

to `/etc/rc.conf`*. Then we add the filter configuration to* `/etc/pf.conf`*. Following the layout required for a* `pf.conf` *file, an initial configuration that doesn't allow any traffic at all may look like this:*

/etc/pf.conf

```
ifs="(lnc0)"                                    || macros and tables
[...]
scrub in all                                    || traffic normalization
[...]
antispoof quick {(lo0) $ifs}                    || anti-spoofing rules
[...]
block return all                                || final catch-all rule
```

This layout follows the ordering required by `pfctl`*. It uses the* `quick` *modifier to use a first match semantic.*

At this point it doesn't allow any traffic at all, but if we add rules before the final `block` *statement, we can enable traffic as we need.*

Notice that the interfaces in the `interfaces` *variable are written in parentheses. Generally,* `pf` *accepts interface names instead of addresses in the filter rules. It substitutes every interface name with a list of the addresses that the interface is configured with. Without parentheses the* `pfctl` *command does the substitution when it loads the filter rules. If the interface configuration changes later on, then we need to re-run* `pfctl` *to update the rules. With parentheses, the packet filter checks the actual interface configuration every time that a packet is run through the filter rule. If an interface configuration changes, then the packet filter will adapt its behaviour accordingly even if we don't re-run* `pfctl`*; we'll see later on why this is necessary.*

Unfortunately, the parentheses notation is apparently broken with FreeBSD 6.1. Regression tests with FreeBSD 6.0 show that at least together with `from` it doesn't seem to work reliably. This leaves us with two choices: Downgrade to 6.0, which works, or omit the parentheses and ensure that we reload the filter rules whenever the interface configuration changes. Throughout the book, we assume that the parentheses work; if necessary, use FreeBSD 6.0 for nodes that need a packet filter.

To enable the filter rules temporarily, we need to run the commands

```
# kldload pf
# pfctl -f /etc/pf.conf
# pfctl -e
```

to load the packet filter kernel module, to load the filter rules and to enable the filter. If we run

```
# pfctl -s rules
```

it displays the currently installed rules. 11

At this point another weakness of todays IPv6 packet filters emerges: Setting up a filter correctly during boot time is non-trivial. An obvious solution would block all traffic on all network interfaces until they are up and running and only then install the final filter rules. But as we'll see in the next chapter, when an interface is first brought up it does a duplicate address detection, so we should really set up a temporary filter configuration that allows just that, then bring up the interface and afterwards install the "final" rules. Unfortunately, todays boot scripts don't do that.

Doing so ourselves will either interfere with the boot scripts or make the packet filter configuration noticeably more complex. If you really worry about this sort of race condition you will find all the information necessary to modify your system accordingly in the following chapters, but otherwise we leave it at this level of imperfection and rather make sure that our filter setup will still be usable with the next system update.

3

IPv6 Address Basics

To set up IPv6 we must understand a few things about IPv6 addresses. This chapter explains the basic IPv6 address architecture.

3.1 Size Matters

IPv6 addresses are 128 bits or 16 bytes[1] long, or four times as long as an IPv4 address. That doesn't sound too exciting in itself, but figure 3.1 gives a first clue that the difference may be vast. To get an even better understanding of the differences between the IPv4 and IPv6 address spaces, it helps to take a look at the numbers involved.

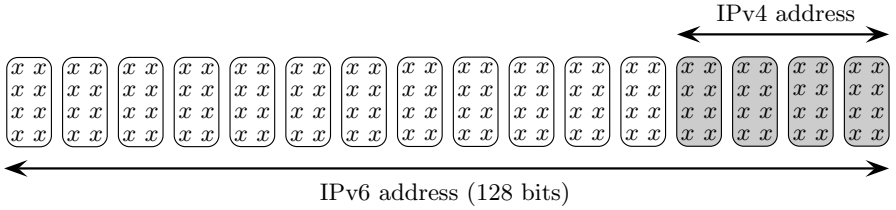

Fig. 3.1. IPv6 addresses compared to IPv4 addresses

As a first step it is obvious to compare the number of IPv4 and IPv6 addresses. There are

$$2^{32} \;=\; 4\,294\,967\,296 \;\approx\; 4.3 \times 10^9$$

[1] Historically, units of eight bits are called *octets* while the smallest addressable unit in a computer is called a *byte*. In the days of the PDP-10, which had nine bits per byte, this distinction was important; today both terms are normally used interchangeably.

IPv4 addresses compared to

$$2^{128} = 340\,282\,366\,920\,938\,463\,463\,374\,607\,431\,768\,211 \approx 3.4 \times 10^{38}$$

IPv6 addresses. So every single IPv4 address is outnumbered by

$$2^{96} = 79\,228\,162\,514\,264\,337\,593\,543\,950\,336 \approx 7.9 \times 10^{28}$$

IPv6 addresses. A more useful comparison puts the number of "sites" that can be connected to the Internet into relation. Assuming a simplified addressing scheme where all sites on the Internet4 are assigned a single "class C"-sized network, i.e. a network with a `/24` prefix, there are

$$2^{24} = 16\,777\,216 \approx 1.7 \times 10^{7}$$

sites that can be connected through the Internet4 (ignoring both sites with other prefix sizes and the fact that not all IPv4 addresses can be used like this). With IPv6, all sites except extremely large ones get assigned a `/48` prefix, so IPv6 can address

$$2^{48} = 281\,474\,976\,710\,656 \approx 2.8 \times 10^{14}$$

sites on the Internet6. This leads to the least impressive but most significant number in this arithmetic exercise: For every IPv4 site there are

$$2^{24} = 16\,777\,216 \approx 1.7 \times 10^{7}\ (= 16\text{M})$$

IPv6 sites addressable. That's like an old-fashioned 1.44 MB floppy disk compared to a data center class 23 TB disk array (which would need to be equipped with a minimum 75 standard 300 GB hard disks).

These numbers are based on a few oversimplifications, but they provide a rough idea how IPv6 resolves the problems and limitations that IPv4 and its insufficient address size impose on the Internet4.

3.2 Address Notation

Different from IPv4 addresses, IPv6 addresses are written in hexadecimal, with pairs of bytes separated by colons. Those pairs of bytes are often called "blocks", even though this term is both unofficial and slightly misleading as "address blocks" are also ranges of addresses. The hexadecimal digits, or half-bytes, are called *nibbles*.

One of my interfaces uses the address

```
fe80:0000:0000:0000:020c:f1ff:fefd:d2be
```

which looks a bit unwieldy. To abbreviate it, within each block leading zeros may be omitted, resulting in the slightly shorter form

```
fe80:0:0:0:20c:f1ff:fefd:d2be
```

Finally, in every address at most one sequence of zero-blocks may be written as two consecutive colons, further collapsing the address to the *compressed form*

```
fe80::20c:f1ff:fefd:d2be
```

which is about as short as we can reasonably expect. Note that a double colon can be used within an address at most once—otherwise the notation would be ambiguous.

In section 2.4.1 we have already seen the address `::1` assigned to the loopback interface. What seemed distinctly odd at that time now expands to the address

```
::1   =   0000:0000:0000:0000:0000:0000:0000:0001
```

finally making sense after all.

Network prefixes are written as with IPv4, using a trailing slash followed by the number of relevant bits in decimal. More precisely, a prefix is written as a full IPv6 address followed by a slash and the number of relevant bits in decimal. This distinction is important in conjunction with the `::` notation; `fe80::f00/64` is equivalent to `fe80::/64`, *not* `fe80:0:0:f00::/64`:

```
fe80::f00      = fe80:0000:0000:0000:0000:0000:0000:0f00
fe80::f00/64   = fe80:0000:0000:0000:****:****:****:****
               = fe80::/64
```

Sometimes an IPv4 address is embedded in the last four bytes of an IPv6 address. These addresses can be written in a *mixed (address) notation* as an IPv6 address with the last two blocks replaced by the IPv4 address, like

```
::ffff:127.0.0.1   =   0000:0000:0000:0000:0000:ffff:7f00:0001
```

(which happens to be what an IPv6-enabled program might see instead of a plain `127.0.0.1` IPv4 address).

IPv6 address notation is defined in RFC 4291 [64], which uses uppercase letters for the hexadecimal digits; implementations however commonly print addresses using lowercase letters and accept uppercase as well as lowercase letters for input.

You may encounter an alternative *base 85 encoding* as of RFC 1924 [38]. While in theory this notation might actually be useful when transmitting IPv6 addresses over limited-bandwidth connections, this RFC is dated as of April 1, 1996 and shouldn't be taken too seriously.

Peter Bieringer has written a tool called `ipv6calc` that converts between various address representations.

Debian Sarge *There is a package on the installation media.*

FreeBSD 6.1 *A package/port is available but doesn't contain the man page.*

Solaris 10 *There are no pre-built binaries available but the sources from* `ftp://ftp.deepspace6.net/pub/ds6/sources/ipv6calc/` *as of version 0.51 build without major problems. They just need to be installed manually.* [12]

3.3 Scopes

RFC 1918 [97] originally reserved the IPv4 address ranges `10.0.0.0/8`, `172.16.0.0/12` and `192.168.0.0/16` for use within "private networks", to be assigned to nodes that don't need global connectivity, like network printers. These address ranges were subsequently abused to provide limited, "one-way" Internet connectivity through NAT gateways, causing an abundance of operational problems along the way.

While IPv6 abolishes the need for NAT, it preserves and extends the original ideas of *address scopes* introduced in RFC 1918.

IPv6 defines a wide range of scopes; most are only used with multicast addresses. Three scopes are especially important because they are defined for all address types:

Global scope Addresses that are routed throughout the entire Internet6 are said to have *global scope*.

Site-local scope Addresses with site-local scope are only routed within a "site". While the term "site" has proven to be too fuzzy to be properly defined, the site-local scope is particularly useful to assign to nodes that shouldn't be reachable from outside a restricted network. These addresses should however *not* be used to implement another generation of NAT-style pseudo-connectivity to the Internet.

Link-local scope This scope doesn't get routed, so it can only be used within a single subnet, often called a *link* in RFCs. It is mostly used for purposes internal to IPv6.

Addresses with a scope larger than link-local can be routed between subnets, so we call them *routable addresses* or *routed addresses*.

The next sections discuss the different address types and the scopes they support.

3.4 Unicast Addresses

Unicast addresses are "normal" addresses, those assigned to a single network interface, like `192.0.2.34` with IPv4.

We have already seen a special unicast address in section 2.4.1: The *loopback address*, `::1`. It may never be assigned to a physical interface and it must always be assigned to the *loopback interface* on any device supporting IPv6.

IPv4 and IPv6 differ in a most fundamental aspect: While IPv4 always assigns only one address to an interface, IPv6 assumes that an interface may have an arbitrary number of addresses assigned.

Most IPv4 implementations have been retrofitted to support multiple addresses per interface, calling them *interface aliases*, *subinterfaces*, or *logical interfaces*. These are often treated as separate *virtual interfaces*, leaving a single *primary address* assigned to the *physical interface*.

IPv6 simply assumes that an arbitrary number of "equal" addresses can be assigned to a single interface. Later on we'll see that this occasionally causes a few minor complications but generally proves very valuable.

Solaris 10 *The Solaris* `ifconfig` *command still uses the "interface alias" notion even with IPv6 addresses. While this can be inconvenient at times, especially when writing portable shell scripts, the implementation complies with the standards.* 13

Just like IPv4, IPv6 addresses consist of a "network part" called *subnet prefix* in IPv6 terminology, and a "host part" called *interface identifier* or *interface ID*. Different from IPv4, the subnet prefix length is always 64 bits for all unicast addresses except those with a `::/3` prefix, so all IPv6 unicast addresses except for the loopback address look like figure 3.2.

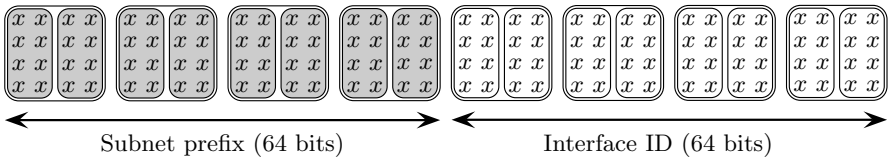

Fig. 3.2. Subnet prefix and interface ID

At this point it is perfectly reasonable to wonder why any single subnet should contain more than four billion (4×10^9) times as many addresses as the entire Internet4. We'll see in section 4.3 that doing so does make sense after all.

One first advantage of this simplification should already be obvious: Abandoning the concept of configurable subnet prefix lengths simplifies network configuration. Using IPv6, misconfigured netmasks are a thing of the past. In the next chapter we'll learn that this simplification has a number of additional, less obvious advantages.

As mentioned before, IPv6 supports unicast addresses with link-local, site-local and global scope.

3.4.1 Link-local Unicast Addresses

Link-local addresses are used for a wide range of IPv6-internal purposes. As soon as an IPv6 interface is enabled, it will automatically acquire a link-local address starting with a prefix `fe80::/64`[2]. Invoking `ifconfig -a` on an IPv6-enabled machine should show such an address for all physical interfaces.

FreeBSD 6.1 *Here even the loopback interface automatically sets up a link-local address. While this isn't required, it doesn't cause any harm, either.*

Solaris 10 *The implementation of link-local addresses shows a harmless bug: The prefix length configured here is `/10` instead of the `/64` as required by RFC 4291 [64, 2.5.1 (p. 8)].* 14

Using `ping6` or, in the case of Solaris, `ping` it is possible to ping other machines within the same subnet.

In general it is necessary to specify the interface to use, though. Because the `fe80::/10` prefix is used on all subnets, it is generally impossible for the kernel to decide which interface to send the ping request to. Depending on the particular implementation, there are various options to the `ping` or `ping6` command.

Debian Sarge *The `ping6` command has an option `-I` ⟨interface⟩ (uppercase letter "I") to specify the interface name.*

FreeBSD 6.1 *Like Debian, `ping6` with FreeBSD supports the `-I` ⟨interface⟩ option. Alternatively it is possible to append a percent sign and the interface name or number to the address, like this:*

```
% ping6 fe80::20c:f1ff:fefd:d2be%lnc0
% ping6 fe80::20c:f1ff:fefd:d2be%1
```

Solaris 10 *Unlike Debian and FreeBSD, the Solaris `ping` uses an option `-i` ⟨interface⟩ (lowercase letter "i") for this purpose.*

Even without an explicit `-i` option a ping to another machine may actually work because Solaris caches the link-local addresses it finds on its attached interfaces. 15

[2] The entire `fe80::/10` prefix is reserved for link-local addresses, even though interfaces are configured with a prefix length of `/64`.

Through the rest of the book we will encounter a variety of similar cases where the command line options differ between Unixen.

The notation shown with FreeBSD is interesting because it hints at some internals of IPv6. Whenever an interface needs to be added to an IPv6 address, this is internally done through a *scope identifier* or *scope ID*, recently renamed *zone index*. Many RFCs use this notation with a percent sign.

- ☐ At this point, check that each machine can ping its own link-local addresses.
- ☐ Next, check that they can ping each other using their link-local addresses.

Explicitly specifying the interface with an address is obviously infeasible for "normal" applications; they would need to implement some sort of logic how to augment an address they obtained from the name services (like DNS) with the proper interface. Fortunately, only link-local addresses need an explicit scope ID to be usable. Applications virtually always use addresses with a larger scope, so options to specify the interface are generally used for IPv6-internal and diagnostic purposes only.

3.4.2 Site-local and Unique-local Unicast Addresses

The next larger scope is called *site-local scope*. Originally an address range was allocated for free use within any given site. Eventually the fear of repeating the same mistakes as with the RFC 1918 private addresses led to the deprecation of this original prefix. It was meant to be replaced by two new prefixes called *unique-local unicast addresses* but until now only one of them has been formally standardized.

This leaves us with three prefixes for site-scoped unicast addresses.

fec0::/10 This is the original *site-local* prefix used for site-local scope. Anybody could use addresses from this pool for local purposes. In September 2004 this prefix was formally deprecated, so it should not be used anymore.

fc00::/8 From this address pool of *unique-local* addresses, `/48` prefixes will be assigned by a central authority. At the time of this writing this authority still needs to be established and a proper standard released.

fd00::/8 This pool of *unique-local* addresses is not centrally managed. Everyone is free to pick a random `/48` prefix for local purposes from this pool. Using a random prefix will ensure that it is statistically very unlikely that two sites will coincidentally use the same prefix.
And no, "randomly" picking `fd00::/48` for your local site is definitely *not* a clever move...

At the time of this writing unique-local addresses are mostly undesirable. They are useful as a fallback during network renumbering events, but since

globally routed addresses are freely available this doesn't justify bothering too much about them at this time, so we defer further discussions on site-scoped unicast addresses until section 16.1.

3.4.3 Global Scope Unicast Addresses

The last scope used with unicast addresses is the *global scope*. Packets sent to such an address will be routed through the entire Internet6.

Global addresses are assigned to sites as `/48` prefixes[3]. The assigned prefix is called the *global routing prefix*. Within a site every subnet is assigned a 16 bit *subnet ID*. Together with the routing prefix the subnet ID forms the subnet prefix. So a globally routed unicast address looks like figure 3.3

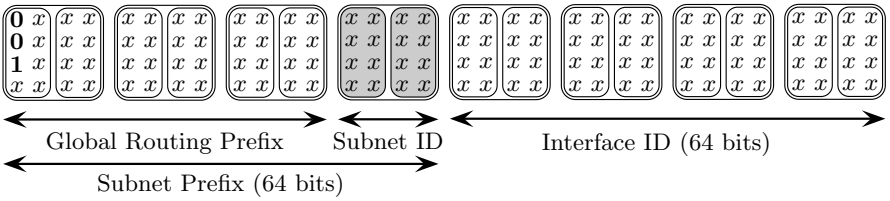

Fig. 3.3. Globally routed unicast addresses

Originally, only the prefix `2000::/3` has been allocated for unicast addresses. Within this prefix some address ranges have been allocated for special purposes.

2001:db8::/32 The APNIC has allocated this prefix for documentation purposes. Throughout this book all example globally routed addresses have this prefix.

2001:db8:fedc::/48 We assume that our (non-existent) provider has assigned us this as our global routing prefix.

2002::/16 These addresses are used for "6to4 tunnels". They provide a configuration-free means to run a tunnel to the Internet6 over IPv4-only networks; they are covered in section 12.2.

3ffe::/16 When IPv6 was still experimental, a global test network called the *6Bone* used addresses from this pool. The 6Bone was disbanded some time ago but you may still occasionally stumble across these addresses.

If your test environment is connected to the Internet4, make sure you filter the `2002::/16` prefix at the IPv6 side and protocol type 41 at the IPv4 side. Otherwise people may be able to work their way through your firewall using 6to4 tunneling.

[3] If your IPv6-incapacitated ISP wants to give you a longer prefix, tell them to read RFC 3177 [72].

In August 2003, RFC 3587 pointed out that additional address ranges may be used for unicast addresses in the future. In the long run we may need to add more unicast prefixes to our configurations, but at the time of this writing `2000::/3` can be considered the only globally routed unicast address range.

With our own `/48` global routing prefix we can set up $2^{16} = 64$k subnets. This may seem excessive to the IPv4-conditioned mind, but for any reasonably sized site this abundance of subnets allows a very fine-grained separation of network segments, mitigating the damage that a rogue machine or inside attacker may cause. If you are a network designer, try to keep in mind that with IPv6 you have a virtually unlimited number of subnets at your disposal.

3.5 Multicast Addresses

From the very beginning multicast support was an integral part of the IPv6 specification. As a consequence, with IPv6, multicasts have finally reached a level of sophistication such that they even make broadcasts obsolete—so IPv6 doesn't support them anymore.

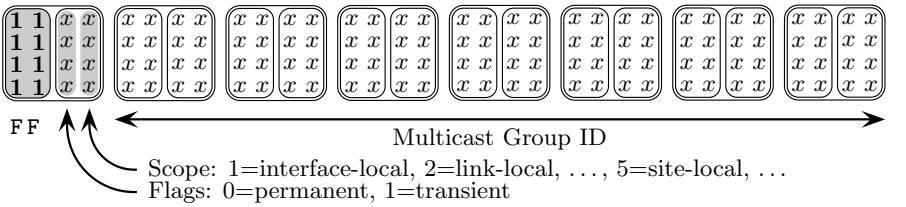

Fig. 3.4. Multicast addresses

IPv6 multicast addresses look like figure 3.4. They start with an `ff00::/8` prefix. The following *flag nibble*, which is either 0 or 1 for now, defines if the address is *permanent* or *transient*. A permanent address is officially assigned for a given purpose by the *Internet Assigned Numbers Authority* (*IANA*) or similar, while transient addresses are assigned locally.

The fourth nibble, called the *scope nibble*, defines the scope of the address. Multicasts support far more different scopes than unicasts; available scope nibble values are

1 interface-local	7 (unnamed)	c (unnamed)
2 link-local	8 organization-local	d (unnamed)
4 admin-local	9 (unnamed)	e global
5 site-local	a (unnamed)	
6 (unnamed)	b (unnamed)	

Only the values 0, 3 and f are reserved and should not be used.

The rest of the address contains the *multicast group ID*. The entire multicast address identifies a *multicast group*. Nodes may *subscribe* to multicast groups to receive the traffic sent to that group.

Of the permanent multicast groups two are particularly useful: The *all nodes link-local multicast group* `ff02::1` and the *all routers link-local multicast group* `ff02::2`.

Using these addresses it is possible to ping all nodes ("devices that speak IPv6") or routers within a subnet, respectively. Since they are link-local addresses, we need to specify the interface again as we did with the link-local unicast addresses.

- ☐ Ping the all nodes multicast address (`ff02::1`) on your subnet. You should receive replies from all IPv6 nodes connected.
- ☐ Now try the all routers multicast address (`ff02::2`). If you receive a reply, alert your network administrator; either there is a rogue IPv6 enabled router on the same subnet or you are about to wreak havoc in an IPv6 enabled network during the experiments following in the next chapter.
  To track down a rogue router continue reading up to section 4.2.1. There you find out how to figure out the Ethernet address of the router.

Multicasts, and especially multicast routing, are surprisingly complex. In chapter 18 we investigate them in far more detail.

3.6 Anycast Addresses

Besides unicasts and multicasts, IPv6 also supports so-called *anycast* packets.

Anycasts are something between a unicast and a multicast packet. They use addresses from the unicast address space and anycast packets are always only sent to one interface, like unicasts. But there are multiple interfaces listening to that address, like multicasts.

Anycasts are still a subject of research. So far, only the *subnet router anycast* address is generally supported; it addresses all routers within a subnet. Its address is the subnet prefix followed by an all-zero interface id, like `2001:db8:1234:5678::` for the subnet `2001:db8:1234:5678::/64`.

- ☐ Pinging this address is possible but you should not receive a reply because you shouldn't have a router in the network yet; instead, depending on the implementation, you may receive an "address unreachable" error or not.
  If you do receive a reply, the same reasoning as with pinging the all-routers link-local address above applies: Either somebody set up a rogue router or your network is already using IPv6.

RFC 2526 reserves some additional interface IDs for anycasts: The interface IDs `dfff:ffff:ffff:ff80`–`dfff:ffff:ffff:ffff` are reserved and shouldn't be used unless assigned by IANA.

With IPv4 a similar effect has been achieved by using the same IPv4 address on different DNS root servers located around the world. Using standard dynamic routing protocols, all requests are automatically directed to the "closest" DNS server using that address.

3.7 Inside IPv6: The IPv6 Headers

Using the `ping` command, or `ping6` on Unixen that use separate programs for IPv4 and IPv6, we can now generate some IPv6 packets to observe with our preferred packet sniffer.

We'll notice a number of "multicast listener" and "neighbor discovery" packets. These are irrelevant for us right now; instead we just look for some "ICMPv6" packets. The Ethernet header doesn't show any surprises, the source and destination addresses are just the Ethernet addresses of the network cards in the nodes that ping each other:

```
Ethernet II, Src: 00:0c:29:2c:b9:14, Dst: 00:0c:29:1a:4f:a1
    Type: IPv6 (0x86dd)
```

Only the Ethernet frame type is slightly interesting: IPv6 packets in Ethernet frames have a frame type of 34525, or 0x86dd.

The IPv6 header is more interesting, because it looks quite different to an IPv4 header.

```
Internet Protocol Version 6
    Version: 6
    Traffic class: 0x00
    Flowlabel: 0x00000
    Payload length: 16
    Next header: ICMPv6 (0x3a)
    Hop limit: 64
    Source address: fe80::20c:29ff:fe2c:b914
    Destination address: fe80::20c:29ff:fe1a:4fa1
```

Like an IPv4 header, the IPv6 header starts with a 4 bit *version* field with the value 6 rather than 4 for IPv4. So even if the Ethernet frame didn't tell us that this was an IPv6 packet, the version field would tell us—for example if we used a link layer other than Ethernet.

Next follows the 8 bit *traffic class* field, the IPv6 equivalent of the IPv4 *type of service* (*TOS*) field. In section 23.1.2 we see how it is used. Until then it is always set to 0, or "default traffic class".

The *flow label* is a 20 bit field that has no equivalent with IPv4. It is meant to allow traffic from one node to another to be tagged for bandwidth allocation

purposes. Section 23.1.1 explains how *integrated services* are expected to use it, but since implementations currently don't support it we will always see it set to 0, which means "not belonging to any particular flow".

The *payload length* contains the length of the data payload following the IPv6 header. It is 2 bytes long and differs from the IPv4 packet length field because its value doesn't include the IPv6 header. Otherwise it serves the same purpose.

The *next header* field is roughly equivalent to the IPv4 *protocol header*. It is a single byte in size and in our example it indicates that an ICMPv6 payload follows the header.

IPv6 calls the IPv4 *time to live* (*TTL*) header the *hop limit* but otherwise doesn't change its functionality. In our example, the packet will be forwarded another 64 hops before an error is returned to the sender.

Finally the source and destination addresses, each 16 bytes long, complete the header.

Compared to IPv4, a number of header fields are missing. The header checksum has been abandoned; IPv6 leaves checksum computation to the link layer for hop-to-hop and the transport layer for end-to-end checksum verification. The fragmentation-related identification, flags and fragmentation offset fields are also gone. If fragmentation occurs, then IPv6 uses a separate option header. The type of this option header would then be stored in the next header field and the option header would again have a next header field that contained the type of data following that header and so on. While IPv4 only introduced option headers when the original header definition proved insufficient, IPv6 makes heavy use of option headers. This reduces the overhead and keeps the design flexible. As an additional consequence, the initial IPv6 header, called the *base header*, has a fixed length, and we don't need a header length field anymore.

After the IPv6 header the ICMP, or rather ICMPv6, payload follows. It looks exactly like its IPv4 equivalent and doesn't offer any new insights.

3.8 Address Allocation Policy and the Routing Table Problem

A huge problem with IPv4 is the number of routes in the *default free zone*, also called *"Internet backbone"* or *"Internet core"*, which contains all routers without a default route to an "uplink" provider . At the time of this writing there are approximately 185000 routes advertised in the default free zone. Since a "core" router must search its routing table for every single packet it wants to forward, this large number generates a most fortunate source of income for high performance router hardware vendors—and a most unfortunate expense for everyone else.

IPv6 deals with this problem using a twofold strategy.

The number of routes in the default free zone is to a large degree caused by awkward address allocations. This makes it difficult to aggregate multiple routes into a larger one. If for example `192.0.2.0/24` and `192.0.3.0/24` were topologically "adjacent", then most core routers could use a single aggregated route for `192.0.2.0/23`.

But IPv4 addresses are allocated permanently and in many "historic" cases without much forethought on route aggregation; after all, *classless inter-domain routing* (*CIDR*) as the means to aggregate routes was only retrofitted to IPv4 in 1992/93 in RFCs 1338 [43] and 1519 [44].

Renumbering an IPv4 network is a lot of work and often requires a substantial downtime, so people will defend their addresses for good reason. In consequence it is difficult at best to convince users and providers to swap their current addresses for more sensible ones (from a routing perspective).

IPv6 solves this problem by providing mechanisms to change addresses on the fly, with a minimum of work and no downtime. In return, addresses are only assigned on a temporary basis. Section 4.3 explains the fundamental mechanisms that we can use to do a renumbering, section 16.5 takes a closer look at the tools we need to manipulate these mechanisms and chapter 24 gives a more strategic overview of a renumbering operation. While the very idea of a network renumbering event makes experienced IPv4 administrators get positively upset, with IPv6 it is rather unspectacular.

Another, possibly even bigger cause of the routing table bloat are "multi-homed sites", sites that have redundant Internet connectivity through multiple Internet service providers. They use *provider-independent addresses*, or *PI addresses* for short; these addresses "belong" to the site and they use the border gateway protocol (BGP) to announce their routes to the entire Internet—by adding yet another route to the core routing tables.

IPv6 uses a fairly radical solution to this problem: There are no PI addresses for IPv6. Everyone short of the top tier network service providers will get their IPv6 addresses from their upstream providers. We will see shortly that the effect of this limitation isn't half as bad as it seems, but if we need to set up redundant network connectivity we need radically new strategies. Section 25.1 shows how to deal with the situation using a tunnel setup.

Both of these approaches are somewhat radical and invariably provoke a severe knee-jerk reflex on experienced IPv4 administrators. But if we assume for now that these things actually work (and they do), then the capability to renumber a network on the fly is an invaluable tool when we need to redesign our network topology—or switch to another ISP if we don't use PI addresses anyway. Using tunnels instead of running our own autonomous system (AS) and BGP routing makes our life not only different but actually easier rather than more difficult. Additionally, the tunnel architecture converges much faster when one of our uplinks fails.

3.9 References

This entire chapter only provides a first introduction to the issues around IPv6 addresses. For a more detailed introduction I consider the second edition of Silvia Hagen's book [54] quite useful; if you understand German, she has also written a similar book [53] in German.

Beyond books the ultimate resource are the official standards, the *Internet RFCs* (for *request for comments*), available from the web page of the *Internet Engineering Task Force* (*IETF*) at `http://www.ietf.org/`.

If you have never read an RFC because you think they are boring (sometimes they are), start with these ones:

RFC 2119 [12] Terms like "must", "should" and "recommended" are frequently used in all RFCs. Their precise meaning is defined in this short but essential RFC.

RFC 1924 [38] This RFC introduces the base 85 address encoding scheme. Technically it isn't relevant, but it shows that even RFCs can be great fun to read.

Essential to the entire address architecture are these:

RFC 4291 [64] The core address architecture specification.
RFC 3587 [65] The global unicast address format.
RFC 3879 [71] The old site-local unicast addresses are deprecated here.
RFC 4193 [66] The new unique-local addresses.
RFC 3177 [72] Address allocation recommendations for global addresses.
RFC 4001 [22] Textual address notation.
RFC 2374 [67] The aggregation of IPv6 routes in the default zone.
RFC 3587 [65] An update to RFC 2374, which renders the overly rigid routing architecture specification obsolete.

The *Internet Assigned Numbers Authority* (*IANA*) maintains a list of assigned addresses at `http://www.iana.org/ipaddress/ip-addresses.htm`.

3.10 Packet Filter Considerations

At this point it is quite feasible to add some general anti-spoofing rules to our packet filter configuration. But we'll see in section 4.2 that we have to let some additional traffic through before we can even test a filter configuration, so we defer the topic of anti-spoofing packet filter rules until section 4.7.

4

Address Configuration

In section 3.4.1 we first saw how to use the `ping` command (or `ping6`, but from now on we'll use both synonymously) to reach other machines via IPv6. But specifying an interface with every command is obviously asking far too much at least from the average user. In this chapter we learn about "proper" addresses without this handicap and how to configure them.

First we assign addresses statically, like many of us still do in the IPv4 world. That done we take a slight detour and explore "neighbor discovery", a mechanism that often goes unnoticed. Then we learn about "stateless autoconfiguration" (SAC), the automatism IPv6 provides for automatic address configuration, which is quite suitably characterized as "DHCP without the pain", and after mixing both configuration styles we run a few tests in our now functional IPv6 subnet. Finally we set up some elementary packet filter rules.

4.1 Static Address Configuration

To get started we first assign static addresses to our machines. This is generally not a good idea, but to get bootstrapped there are no feasible alternatives.

Unless you yourself are the friendly local network administrator, this is the time to make absolutely sure that IPv6 is not yet used in your local network.

Since we don't want to deal with routing yet, we first put all machines in the same subnet (or on the same *link* in official IPv6 terminology). Assuming a site prefix `2001:db8:fedc::/48` we choose two subnet IDs, `abcd` and `cdef`, to get used to the fact that with IPv6 interfaces tend to have multiple addresses. We configure all our addresses with the subnet prefixes `2001:db8:fedc:abcd::/64` and `2001:db8:fedc:cdef::/64`. As interface IDs we just number our hosts starting with `::1`. So our test setup looks like figure 4.1.

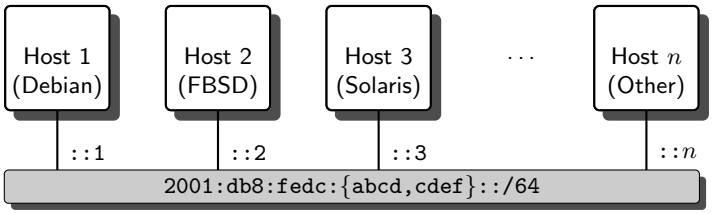

Fig. 4.1. A simple single-subnet test setup

4.1.1 Temporary Configuration

First we configure our interfaces temporarily. The exact syntax of the `ifconfig` call depends on the particular Unix, so you may want to take a look at your local `ifconfig` man page.

Debian Sarge *Linux offers a choice of two commands to configure the interface, `ifconfig` and `ip`. Both configure an interface `eth0` as intended:*

```
# ifconfig eth0 inet6 add 2001:db8:fedc:abcd::1/64
# ifconfig eth0 inet6 add 2001:db8:fedc:cdef::1/64
```

```
# ip addr add 2001:db8:fedc:abcd::1/64 dev eth0
# ip addr add 2001:db8:fedc:cdef::1/64 dev eth0
```

Either one needs an explicit prefix length specified as `/64`, otherwise a nonsensical prefix length will be configured—`/0` with `ifconfig` and `/128` with `ip`.

FreeBSD 6.1 *FreeBSD always assumes a `/64` prefix length so we don't need to specify it for the interface configuration:*

```
# ifconfig lnc0 inet6 2001:db8:fedc:abcd::2
# ifconfig lnc0 inet6 2001:db8:fedc:cdef::2
```

Solaris 10 *We configure the interface `pcn0` from the command line like this:*

```
# ifconfig pcn0 inet6 plumb
# ifconfig pcn0 inet6 addif 2001:db8:fedc:abcd::3/64 up
Created new logical interface pcn0:1
# ifconfig pcn0 inet6 addif 2001:db8:fedc:cdef::3/64 up
Created new logical interface pcn0:2
```

The first line is only necessary if we haven't enabled IPv6 on the interface `pcn0` yet; we can achieve the same effect with an empty file `/etc/hostname6.pcn0` as explained in section 2.4.1. The other two lines add addresses to the interface. Note that it is necessary to specify the prefix length with the address, otherwise a nonsensical `/128` prefix length is used. 16

Next we verify that the configuration is correct. First we check the current interface configuration.

Debian Sarge *We can also view the results with either command:*

```
# ifconfig eth0
eth0      Link encap:Ethernet  HWaddr 00:0C:29:1A:4F:A1
          inet6 addr: 2001:db8:fedc:cdef::1/64 Scope:Global
          inet6 addr: 2001:db8:fedc:abcd::1/64 Scope:Global
          inet6 addr: fe80::20c:29ff:fe1a:4fa1/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:9 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 b)  TX bytes:754 (754.0 b)
          Interrupt:177 Base address:0x1400
```

```
# ip -6 addr show eth0
2: eth0: <BROADCAST,MULTICAST,UP> mtu 1500 qlen 1000
    inet6 2001:db8:fedc:cdef::1/64 scope global
       valid_lft forever preferred_lft forever
    inet6 2001:db8:fedc:abcd::1/64 scope global
       valid_lft forever preferred_lft forever
    inet6 fe80::20c:29ff:fe1a:4fa1/64 scope link
       valid_lft forever preferred_lft forever
```

FreeBSD 6.1 *Unsurprisingly, we can also view the results with* `ifconfig`*:*

```
# ifconfig lnc0 inet6
lnc0: flags=108843<UP,BROADCAST,RUNNING,SIMPLEX,MULTICAST> mtu 1500
        inet6 fe80::20c:29ff:fe3c:e7c5%lnc0 prefixlen 64 scopeid 0x1
        inet6 2001:db8:fedc:abcd::2 prefixlen 64
        inet6 2001:db8:fedc:cdef::2 prefixlen 64
```

Solaris 10 *We can check the configuration with*

```
# ifconfig -a6
lo0: flags=2002000849<UP,LOOPBACK,RUNNING,MULTICAST,IPv6,VIRTUAL> \
                                                    mtu 8252 index 1
        inet6 ::1/128
pcn0: flags=2000841<UP,RUNNING,MULTICAST,IPv6> mtu 1500 index 2
        inet6 fe80::20c:29ff:fec0:463c/10
        ether 0:c:29:c0:46:3c
pcn0:1: flags=2000841<UP,RUNNING,MULTICAST,IPv6> mtu 1500 index 2
        inet6 2001:db8:fedc:abcd::3/64
pcn0:2: flags=2000841<UP,RUNNING,MULTICAST,IPv6> mtu 1500 index 2
        inet6 2001:db8:fedc:cdef::3/64
```

which shows all IPv6 interfaces. The output looks slightly unusual because Solaris creates a logical interface for each address. If we hadn't specified the

prefix length when we configured the interface, then we would see a prefix length of `/128` *here.* 17

Now we can test that the systems can actually reach each other.

- ☐ The interfaces must be physically connected; if they have a "link" LED it must show that they are physically connected to a subnet.
- ☐ They must all be connected to the same subnet.
- ☐ The interfaces must be "up" and "running".
- ☐ All machines must be able to ping themselves.
- ☐ They can ping each other using the configured addresses. (Watch for mistyped addresses if this doesn't work immediately.)

Finally we clean the configuration up again. The easy way to do so is a reboot but if that is undesirable we reverse our steps instead.

Debian Sarge *These are the two variants to undo the configuration:*

```
# ifconfig eth0 inet6 del 2001:db8:fedc:cdef::1/64
# ifconfig eth0 inet6 del 2001:db8:fedc:abcd::1/64
```

```
# ip addr del 2001:db8:fedc:cdef::1/64 dev eth0
# ip addr del 2001:db8:fedc:abcd::1/64 dev eth0
```

Both need the prefix length specified as the example shows, otherwise they will signal an error.

FreeBSD 6.1 *The configuration is reverted like this:*

```
# ifconfig lnc0 inet6 2001:db8:fedc:cdef::2 delete
# ifconfig lnc0 inet6 2001:db8:fedc:abcd::2 delete
```

Solaris 10 *We can undo the configuration changes using the commands*

```
# ifconfig pcn0 inet6 removeif 2001:db8:fedc:cdef::3
# ifconfig pcn0 inet6 removeif 2001:db8:fedc:abcd::3
# ifconfig pcn0 inet6 unplumb
```

Note that different than the configuration above the deconfiguration does not allow the prefix length to be specified. 18

- ☐ At this point another look at the current interface configuration must show that the addresses have been properly purged again.

4.1.2 Persistent Configuration

In the previous section we temporarily configured the interfaces with static IPv6 addresses. A reboot will cause the machines to revert to their initial state, so the obvious next step is to configure the addresses permanently.

Debian Sarge *Like IPv4 addresses, static IPv6 addresses are configured in* `/etc/network/interfaces`*. With multiple addresses this turns out to be slightly messy:*

/etc/network/interfaces

```
auto eth0
iface eth0 inet6 static
    address 2001:db8:fedc:abcd::1
    netmask 64
    up   /sbin/ifconfig eth0 inet6 add 2001:db8:fedc:cdef::1/64
    down /sbin/ifconfig eth0 inet6 del 2001:db8:fedc:cdef::1/64
```

The configuration can then be enabled either with a reboot or doing an

```
# ifdown eth0
# ifconfig eth0 down
# ifup eth0
```

Two things are worth mentioning about the configuration:

- *It is necessary to add the "*`netmask`*" line here, otherwise the interface won't be configured but* `ifup` *will return an excessively informative (and grammatically challenged) "*`Don't seem to be have all the variables for eth0/inet6`*" message.*
- *The* `interfaces` *file can only be tricked into configuring multiple addresses using the* `up` *and* `down` *statements (which may use* `ip` *instead of* `ifconfig` *if you prefer).*

FreeBSD 6.1 *In* `/etc/rc.conf` *the variables starting with "*`ipv6_`*" contain the basic IPv6 configuration:*

/etc/rc.conf

```
ipv6_enable=YES
ipv6_ifconfig_lnc0=2001:db8:fedc:abcd::2
ipv6_ifconfig_lnc0_alias0=2001:db8:fedc:cdef::2
```

The configuration can be enabled doing a

```
# /etc/rc.d/network_ipv6 restart
```

which causes a few harmless error messages but otherwise does the job.

Solaris 10 *We may have already created an empty* `/etc/hostname6.pcn0` *file before. Now we place the static addresses in there like this:*

/etc/hostname6.pcn0

```
addif 2001:db8:fedc:abcd::3/64 up
addif 2001:db8:fedc:cdef::3/64 up
```

To enable the configuration we do a

```
# svcadm restart network/physical
```

or reboot the system.

To check the configuration the checklist on page 38 applies here as well. If feasible it is a good idea to reboot the systems first so we only check the permanent configuration without any interfering temporary changes.

4.2 Inside IPv6: Neighbor Discovery (ND)

At this point we have a chance to take a closer look inside IPv6. This section covers some issues that aren't strictly necessary to set up and operate an IPv6 network but are either interesting or useful if some non-trivial problems occur. As with all sections titled "Inside IPv6", feel free to skip them, especially on first reading.

With the test setup so far it is possible to investigate one of the less obvious new features of IPv6 called *neighbor discovery* (*ND*), as specified in RFC 2461 [91]. While neighbor discovery normally does its business entirely unnoticed, it is both interesting to observe and essential to consider when configuring a packet filter.

4.2.1 Neighbor Solicitations (NS) and Advertisements (NA)

The primary purpose of neighbor discovery is to provide a substitute for the address resolution protocol (ARP).

While ARP is specific to both IPv4 and Ethernet, neighbor discovery is independent of the link-layer protocol. It uses ICMPv6 to provide the functionality of ARP, using multicasts for its purposes.

Figure 4.2 shows how neighbor discovery works. The FreeBSD node from our test setup (the "initiator", `2001:db8:fedc:cdef::2`) wants to ping the Debian node (the "target", `2001:db8:fedc:cdef::1`) but doesn't know its link-layer address.

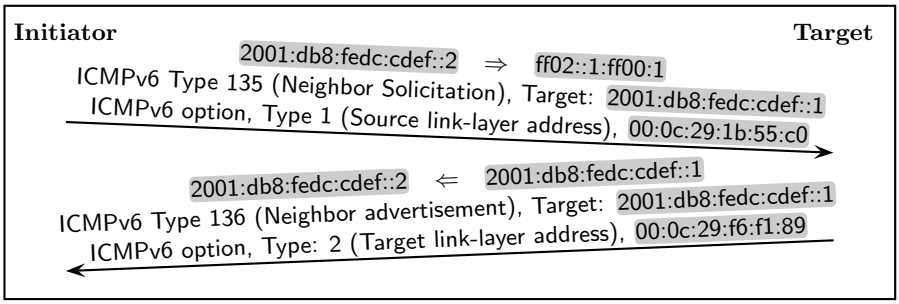

Fig. 4.2. Neighbor discovery as an ARP successor

It does however know its IPv6 address. So it first takes the prefix `ff02::1:ff00:0/104` and appends the last 24 bits of the target IPv6 ad-

dress, together forming the multicast address `ff02::1:ff00:1` in this case.[1] This is the *solicited-node multicast address* associated with the target's IPv6 address. Now it sends its *neighbor solicitation*, an ICMPv6 packet type, to this multicast address, attaching its own link-layer address along the way.

The Debian node listens to this multicast address and receives the neighbor solicitation. It then replies to the initiator's unicast address with an ICMPv6 *neighbor advertisement* packet type, containing its link-layer address.

Afterwards, we can examine the *neighbor discovery cache* to find the associations of IPv6 and link-layer addresses stored there. The IPv6 equivalent of the `arp -a` command as well as its output format depends on the Unix derivative:

Debian Sarge `ip -6 neigh show`
FreeBSD 6.1 `ndp -a`
Solaris 10 `netstat -p -f inet6` 20

4.2.2 Neighbor Unreachability Detection (NUD)

Neighbor discovery packets are also used for another purpose called *neighbor unreachability detection* (*NUD*). It verifies that the cached mapping between an IPv6 and a link-layer address is still valid.

With IPv4, the ARP table entries expire after an implementation-dependent timeout if no packet with matching IPv4 and Ethernet source addresses is received. This approach has two major limitations: The timeout is "dumb" because it doesn't take into consideration information possibly available from the upper protocol layers and communication is checked only in one direction—from the remote to the local machine.

IPv6 and its neighbor unreachability detection mechanism deal with the problem in a more intelligent way. They check that communication between the nodes works not only in one direction but both and it cleverly optimizes the timeout values in the neighbor discovery cache.

In many cases the transport layer protocols, like TCP and to some degree UDP, know if a reply packet can be expected soon. The first optimization called *upper layer positive confirmation* uses this and lowers the timeout whenever a packet is sent out that should cause a reply within a short time.

If the upper layers can't confirm a cache entry, an "initiator" machine actively verifies that a cache entry is still valid by sending a neighbor solicitation to the cached unicast "target" address as shown in figure 4.3. The "target" answers with a matching neighbor advertisement.

Accordingly, neighbor discovery cache entries are in one of these states:

[1] This is an excellent occasion to familiarize yourself with the sometimes slightly confusing IPv6 address notation. Don't worry if it all seems somewhat tedious and error-prone. Eventually it will become more comfortable than decimally written IPv4 addresses.

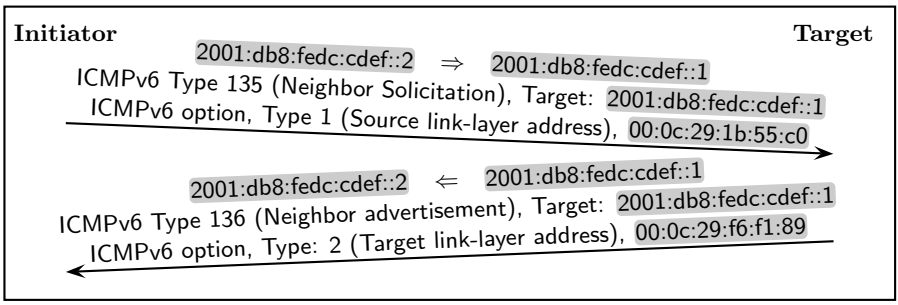

Fig. 4.3. Neighbor unreachability detection

Incomplete Neighbor discovery is currently performed on this entry, i.e. a multicast neighbor discovery packet has been sent but no corresponding advertisement has arrived yet.

Reachable Neighbor discovery has been successful and within the cache time we have received some confirmation that the neighbor is still reachable.

Stale The cache entry has expired but the neighbor wasn't found unreachable yet. At this point no action is taken unless another packet is sent to the neighbor.

Delay The cache entry is outdated as with a stale entry but a packet has been sent to the last known address in the hope that upper layers will shortly confirm the neighbor to be reachable.

Probe A unicast neighbor solicitation has been sent but no answer has arrived yet.

As a consequence, IPv6 reacts far more quickly and reliably to changing network configurations while reducing the traffic overhead in the network.

4.2.3 Duplicate Address Detection (DAD)

The same mechanism is also used when an interface tries to configure a new address, to verify that the intended address is yet unused. Figure 4.4 shows what happens.

There is a difference between "normal" neighbor discovery and duplicate address detection: Because the interface can't use the address until after the duplicate address detection successfully verified that the intended address is available, the addresses on the initiator side differ. The initiator uses the *unspecified address* `::` as the source address of the neighbor solicitation and doesn't send its link-layer address along as an ICMPv6 option. If another machine already uses the address, it must reply to the initiator with a neighbor advertisement using the all-node link-local multicast address `ff02::1` as destination because the initiator cannot be reached through a unicast address yet.

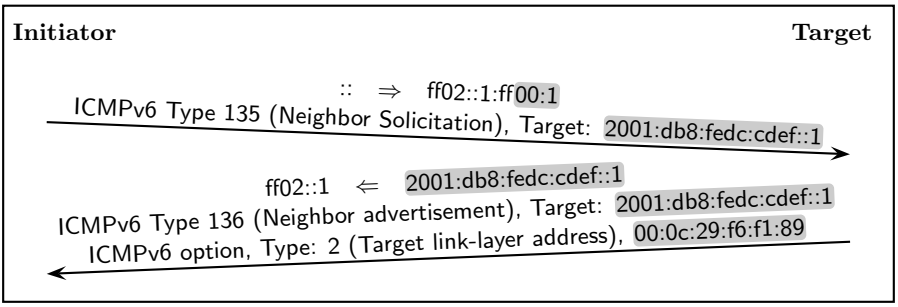

Fig. 4.4. Duplicate address detection

If the initiator doesn't receive a reply, then it assumes that the address is available, subscribes to the solicited-node multicast address and starts to use the address.

4.3 Stateless Address Autoconfiguration (SAC)

Since IPv6 addresses are all too easy to mistype, static address configuration is generally undesirable. IPv6 offers an elegant mechanism called *stateless (address) autoconfiguration* (*SAC*), or *autoconfiguration* for short, to provide for automatic address configuration. It is sometimes characterized as "DHCP without the pain" because it elegantly avoids a range of problems that often occur with DHCP.

4.3.1 The Problems with DHCP

So what's wrong with DHCP? The easy answer is: It is simply too unreliable to be used for "mission-critical" systems. Maybe it is "good enough" for workstations, high performance cluster nodes and the occasional visitor's notebook. But I have yet to see a data center where IP addresses are assigned to servers using DHCP instead of manual, static configuration.

So what *exactly* is wrong with DHCP?

- Since DHCP is a link-layer protocol it can't be routed like ordinary IP traffic. It requires special DHCP routers (called "relays") for every subnet connected. These relays are often run on router hardware, putting an additional burden on them and the network administrators who need to maintain them.
- Large organizations often have distinct staff groups for network and server administration. These need to keep in close sync since all modifications to the network need to be reflected in the DHCP configuration.

- While it isn't impossible to make DHCP servers redundant, doing so either requires fancy cluster setups, which are a problem by themselves, or an abundance of addresses to use distinct address ranges on the redundant DHCP servers. (This is obviously more of a problem for IPv4 than IPv6.)
- DHCP servers need to maintain state; more precisely, they need to keep track of which link-layer address "owns" an IP address and until when. If the "lease database" that keeps track of this state gets damaged, addresses can be assigned to multiple machines. (Again, this problem is worse with IPv4 since it doesn't enforce duplicate address detection.)
- DHCP is very susceptible to rogue DHCP servers. Anybody introducing a misconfigured DHCP server, either out of malice or ignorance, will seriously disrupt network operation—and there have been cases of people installing some "server"operating system on their company notebooks before. (Yet again, this is worse with IPv4 than IPv6: IPv4 will overwrite address configurations while IPv6 only adds addresses to an interface configuration without removing the already configured addresses.)

IPv6 deals with all these problems using a far simpler, more reliable approach. It uses the huge address space in every single subnet to make address collisions next to impossible. The result, called "stateless address autoconfiguration", solves or at least mitigates all the problems DHCP has introduced to the IPv4 world.

4.3.2 Autoconfiguration Concepts

To understand stateless autoconfiguration we need three definitions:

Node A device that supports IPv6.

Router A node that accepts packets on behalf of other nodes, usually to forward them.

Host A node that is not a router.

Throughout the rest of the book, in network diagrams we draw hosts as squares and routers as circles, as in figure 4.5.

Fig. 4.5. Hosts and routers in network diagrams

The distinction between hosts and routers is essential to stateless autoconfiguration because only routers maintain the network prefixes assigned to a subnet. But different than DHCP servers they don't keep track of which

address is assigned to which host or link-layer address. It is entirely up to each host to pick its interface ID. So when a host first enables a network interface, this is what happens:

1. The host picks an interface ID, usually derived from the associated link-layer address. For Ethernet, section 4.5.3 has the details.
2. It creates a link-local address from the prefix `fe80::/64` and the interface ID.
3. It checks if another node is already using that address by doing a duplicate address detection.

So far, nothing unexpected has happened; up to here routers and statically configured nodes behave exactly the same. But the next steps are unique to hosts:

1. The host queries all routers on the subnet with a *router solicitation* (*RS*) about additional prefixes. The routers respond with *solicited router advertisements* (*RA*). These router advertisements contain a list of prefixes allocated to the subnet and an indicator if the router is willing to provide routing services to the connected hosts.
2. For every prefix received the host proceeds as with the link-local prefix and configures another address to the interface.
3. The host keeps listening for updated prefix information and reconfigures its addresses whenever necessary. Routers send these updates as *unsolicited router advertisements* both periodically and whenever the network configuration changes.

This algorithm obviously requires that it is virtually impossible that two hosts try to use the same interface ID. Computing the interface ID from the Ethernet address ensures this, but requires a large interface ID. Since other link-layer protocols may use even longer addresses than Ethernet, the interface ID has been made large enough to hold them. At this point using 64 bits for the "host part" of an address finally starts to make sense after all.

Stateless autoconfiguration provides a host with more information than network prefixes: It tells the host which routers are available on the subnet and are willing to provide routing services. Unlike an IPv4 host, an IPv6 host will not have a single "default route" but a list of "default routers". Whenever it tries to send a packet to an address not directly attached to the subnet, called an *off-link address*, the host will forward the packet to one of the routers and let that router take care of the packet. (This is a slight simplification; section 7.1 explains the details.)

Routers don't use stateless autoconfiguration to receive their addresses; they need to be explicitly configured with addresses and the prefix information to advertise. If they accepted prefixes from other routers on the same subnet, then removing a prefix would require synchronized intervention on all routers at the same time.

Why is stateless autoconfiguration such an improvement over DHCP?

First of all, we neither need another server in every subnet nor a relay agent on a router that isn't designed as a server. Neither do we need to get a server administrator involved with the management of IPv6 addresses, since addresses are now solely managed by routers.

Next, it is trivial to provide all the redundancy wanted by just plugging redundant routers into the network. Since the routers don't need to keep state on which IP address is leased to which link-layer address until when, this is quite straightforward.

Without the need to keep state on the routers, a failing router won't cause any problems. It can be fixed or replaced and brought up without a risk of re-assigning addresses already given to another host.

A rogue router is far less likely than somebody installing a "server" operating system accidentially enabling DHCP. But even if this happens, only *additional* addresses are passed to the hosts. They will still be reachable through their proper addresses. Disconnecting the rogue router will cause the bad addresses to expire and the network to converge to proper operation again.

Finally, stateless autoconfiguration makes it feasible to renumber a network with new prefixes on the fly—chapter 24 explains the procedure in detail.

In the rest of this section we configure nodes to support stateless autoconfiguration. In the most simple case a single router and a single host suffice for some basic experiments. The following examples refer to the slightly more complex setup shown in figure 4.6.

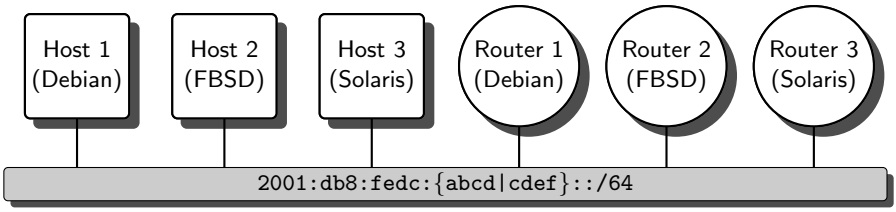

Fig. 4.6. The stateless autoconfiguration test setup

4.3.3 Router Configuration

First we tackle the more complex task and configure the routers as *advertising routers*. While the BSDs and Solaris only require a few lines of configuration, Linux needs an additional software package to be installed

It is possible to configure a router in such a way that it advertises prefixes that it doesn't use itself on its interface; after all, any communication

with these routers can be done using only link-local addresses. But experience shows that doing so is extremely confusing. For this reason some router implementations use the same configuration for interfaces and router advertisements. To do a clean job we make sure that our routers configure addresses with all the prefixes they advertise even on those implementations that use separate configurations for interfaces and router advertisements.

Debian Sarge *First we need to install the* `radvd` *package to turn a node into an advertising router.*

Then we create a file `/etc/radvd.conf` *where we configure the prefixes to advertise on each interface. Since we only have a single interface* `eth0`*, these lines will do:*

/etc/radvd.conf

```
interface eth0                                  || The interface to configure
{
    AdvSendAdvert on;                           || Enable router advertisements
    prefix 2001:db8:fedc:abcd::/64 { };         || The first prefix to advertise
    prefix 2001:db8:fedc:cdef::/64 { };         || The second prefix to advertise
};
```

Unfortunately, the `radvd` *boot script does enable packet forwarding but doesn't configure the kernel to ignore router advertisements from other routers. Not only does this violate the specifications but it is actually very confusing when we need to debug a network. To make the kernel ignore router advertisements, we need to add a line*

/etc/sysctl.conf

```
net.ipv6.conf.default.accept_ra=0
```

to `/etc/sysctl.conf`*.*

To complete the configuration we need to configure the router's interfaces in `/etc/network/interfaces` *as we have already done in section 4.1.2 on page 38.*

Finally, the easiest way to start the `radvd` *and ensure that no addresses are configured from another router's advertisements is a quick reboot.*

FreeBSD 6.1 *FreeBSD keeps all its relevant network configuration in the file* `/etc/rc.conf`*. If we want to use normal interface IDs derived from the Ethernet address, then we use this configuration:*

/etc/rc.conf

```
ipv6_enable=YES
ipv6_gateway_enable=YES
rtadvd_enable=YES
rtadvd_interfaces="lnc0"
ipv6_prefix_lnc0="2001:db8:fedc:abcd 2001:db8:fedc:cdef"
```

The first line enables IPv6 in general, the second enables packet forwarding

and the third starts the router advertisement daemon `rtadvd` *with the fourth line optionally specifying the interfaces that the* `rtadvd` *will service. The last line contains all the prefixes we want to use. Note that we can't use the double colon notation here; the boot scripts can't handle them.*

If we rather want to configure the addresses explicitly to ensure that routers have short addresses, then we might substitute the last line with something like

/etc/rc.conf

```
ipv6_ifconfig_lnc0=2001:db8:fedc:abcd::2
ipv6_ifconfig_lnc0_alias0="2001:db8:fedc:abcd:: anycast"
ipv6_ifconfig_lnc0_alias1=2001:db8:fedc:cdef::2
ipv6_ifconfig_lnc0_alias2="2001:db8:fedc:cdef:: anycast"
```

This configures the addresses fully, using an interface ID of `::2`*. Finally we either need to reboot the router or run the commands*

```
# /etc/rc.d/network_ipv6 restart
# /etc/rc.d/rtadvd start
```

to enable the configuration.

By default, the `rtadvd` *router advertisement daemon reads the addresses configured to all interfaces that it serves and uses the addresses found there for its advertisements.*

Solaris 10 *First we need to install the* `SUNWroute` *package if it isn't installed yet; it contains the* `in.ndpd` *daemon which is responsible for sending router advertisements to a subnet.*

Then we configure the prefixes in `/etc/inet/ndpd.conf`*:*

/etc/inet/ndpd.conf

```
ifdefault AdvSendAdvertisements true
prefix 2001:db8:fedc:abcd::/64 pcn0
prefix 2001:db8:fedc:cdef::/64 pcn0
```

The first line enables router advertisements on all interfaces and the others each assign a prefix to the interface. We enable both packet forwarding and the `in.ndpd` *daemon using the commands*

```
# routeadm -e ipv6-forwarding
# routeadm -u
```

This automatically adds addresses with the configured prefixes to the appropriate interfaces; we still need an empty `/etc/hostname6.pcn0` *file, but we don't configure the addresses there anymore.* 21

Even if we don't have a host to test our router against, we can do some preliminary testing.

- ☐ If possible, reboot the router to ensure that you test the persistent configuration only.

- ☐ Doing an `ifconfig -a` must show that the interfaces are "up" and "running".
- ☐ It must also show that the interfaces have configured the intended addresses.
- ☐ Running either `ps auxw` or `ps -ef` must show a running router advertisement daemon.
- ☐ A packet sniffer must show unsolicited router advertisements every 200 to 600 seconds; the time between router advertisements is randomized to prevent synchronization effects.

For now this is all we need to know about router advertisement configuration. In section 16.5 we take a closer look at the router advertisement daemons and their various tunable parameters.

4.3.4 Host Configuration

Compared to the router configuration, making a host use stateless autoconfiguration is quite straightforward; we don't need to configure anything but the fact that the host should support IPv6.

Debian Sarge *If the interface is also configured for IPv4, nothing needs to be done.*

If however the interface is not used with IPv4, then it is necessary to trick the boot scripts so they enable the interface at all. An entry like

`/etc/network/interfaces`

```
auto eth0
iface eth0 inet manual
    up /sbin/ip -6 link set eth0 up
```

brings the interface up using autoconfiguration.

FreeBSD 6.1 *When we enabled IPv6 with the line*

`/etc/rc.conf`

```
ipv6_enable=YES
```

in `/etc/rc.conf` we already enabled stateless autoconfiguration, turning the machine into a host. Nothing else needs to be done.

Solaris 10 *Similar to FreeBSD, enabling IPv6 on an interface with an empty file `/etc/hostname6.pcn0` already makes the machine support stateless autoconfiguration. Only if we did a "reduced networking core installation" would we need to install the `SUNWroute` package.* 22

Even without a router we can test a host to some degree:

- ☐ First reboot the host if possible.
- ☐ During the reboot use a packet sniffer on a router to verify that the host sends router solicitations to the subnet.

- ☐ Use `ifconfig -a` to check that the interfaces are "up" and "running" and show a link-local address.

In section 4.6 we do some more extensive testing in a subnet with both a host and a router.

4.4 Mixing Static and Automatic Configuration

Sometimes we may want to configure static addresses to a host; maybe we want to advertise these addresses in the DNS and don't want them to be based on the machine's Ethernet hardware, maybe we want to build a high-availability cluster with a virtual IPv6 address.

It is generally advisable to configure these static addresses in addition to autoconfiguration addresses only. Doing so will enable us to reach a machine even if we forgot to change its configuration during a network renumbering.

It is generally possible to disable autoconfiguration entirely. This may however affect the router discovery as well, so we must be prepared to configure our routes, too.

The details again depend on the particular Unix derivative.

Debian Sarge *By default Debian mixes static and autoconfigured addresses. If we add a line*

/etc/sysctl.conf
```
net.ipv6.conf.default.accept_ra=0
```

to `/etc/sysctl.conf`*, then our node won't accept router advertisements anymore. We will need to configure our default routers, too. Alternatively, setting*

/etc/sysctl.conf
```
net.ipv6.conf.default.autoconf=0
```

will disable address autoconfiguration but continue to discover routers through the router advertisements received.

FreeBSD 6.1 *If we configure static addresses, FreeBSD by default disables the autoconfiguration mechanism including default router discovery.*

To re-enable autoconfiguration in the presence of static addresses some minor trickery is necessary. In `/etc/sysctl.conf` *the line*

/etc/sysctl.conf
```
net.inet6.ip6.accept_rtadv=1
```

makes the kernel listen to router advertisements and a line

/etc/rc.local
```
rtsol -a
```

in `/etc/rc.local` *triggers a router solicitation after the static addresses have*

been configured. Alternatively, if the addresses are only configured using the `ipv6_ifconfig_lnc0_alias`⟨*n*⟩ *variables and* `ipv6_ifconfig_lnc0` *remains undefined, autoconfiguration will remain enabled. This latter approach is however undocumented and may fail after a system upgrade if the boot scripts change.*

There is no way to avoid autoconfiguring addresses while still obtaining default routers through router advertisements.

Solaris 10 *Provided that the* `SUNWroute` *package is installed to support stateless autoconfiguration, static and autoconfigured addresses are merged.*

To disable this either uninstall `SUNWroute` *or add a line*

`/etc/inet/ndpd.conf`

```
ifdefault StatelessAddrConf off
```

to `/etc/inet/ndpd.conf`*. This will still configure the prefixes to the interface but show them as "*`NOLOCAL`*", which means "without an address of this prefix".*

There is no way to prevent a host from gathering default routers through router advertisements. 23

4.5 Inside IPv6: Autoconfiguration Details

If we didn't make a mistake with our router and host configuration it should be fairly straightforward to test. But if something went wrong it is sometimes helpful to understand in more detail how autoconfiguration works.

4.5.1 Address States

Addresses on an interface pass through multiple states. If we debug a problem related to addresses, then it is often useful to check that all addresses are in a state that allows for their use.

These are the address states:

Tentative When an interface tries to configure a new address, this address is considered *tentative* until duplicate address detection has successfully assured that no other node is currently using the address. If the duplicate address detection fails, some implementations continue to show this address as "tentative" with their `ifconfig` command and similar.

Duplicate When duplicate address detection failed, showing that another interface already uses the desired address, the address is normally shown as "duplicate".

Valid Once an address has passed the duplicate address detection it becomes valid and can be used. With autoconfigured addresses there are two substates, *preferred* and *deprecated*.

Preferred Autoconfigured addresses that are *preferred* are valid addresses that can be used when a new connection is established. Addresses are only preferred if the associated *preferred lifetime* (*pltime*) is positive. With link-local and statically configured addresses the preferred lifetime is always infinite, otherwise it is set according to the value received by router advertisements.

Deprecated Addresses that can still be used for established connections but shouldn't be used for new ones are called *deprecated*. Their preferred lifetime is zero but their *valid lifetime* (*vltime*) is still positive. The valid lifetime is always at least as large as the preferred lifetime.

Invalid When the valid lifetime drops to zero, the address becomes *invalid* and can't be used anymore.

How to make the interface state and the lifetime counters visible depends on the Unix derivative:

Debian Sarge *We need to use the* `ip` *command here with the options* `addr show`*.*

FreeBSD 6.1 *The additional option* `-L` *extends the output of* `ifconfig` *to include the lifetime values.*

Solaris 10 *There is no documented way to display the lifetime values.* [24]

4.5.2 Router Solicitations (RS) and Advertisements (RA)

Router solicitations and router advertisements are ICMPv6 packets that are sent to the all-routers and all-nodes link-local multicast addresses (`ff02::2` and `ff02::1`), respectively. Figure 4.7 shows an example from the test network.

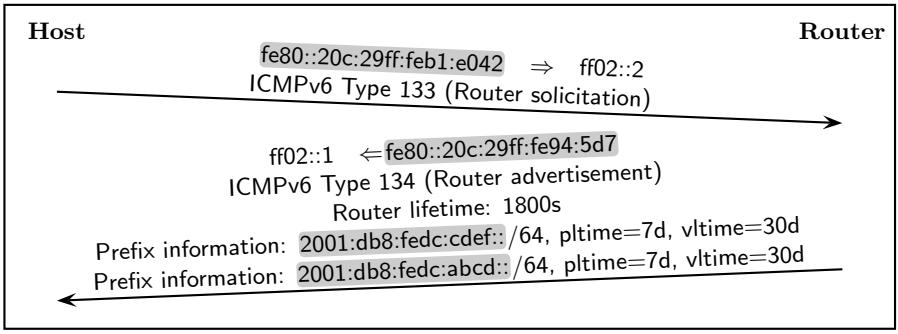

Fig. 4.7. A sample router solicitation/advertisement exchange

The router advertisement contains two important information items: The *router lifetime* holds the number of seconds that the router expects to be

usable as a default router; if it is zero, the router provides address prefixes but cannot be used as a default router. The *prefix information* list contains prefixes, their prefix length and their preferred and valid lifetimes. The default preferred and valid lifetimes are seven and thirty days, respectively. Notice that the router sends its reply to the all-nodes link-local multicast address (`ff02::1`) so all nodes can update their prefix and router data.

Additionally, routers periodically and after changes send *unsolicited router advertisements*. They look exactly the same as replies to router solicitations and are sent at randomized intervals, usually within 200 to 600 seconds.

Finally, when a router is shut down, it is supposed to send another router advertisement with a router lifetime of 0 if possible.

Router advertisements actually contain some more information. Section 16.5 takes a closer look, beyond that see RFC 2461 [91] and RFC 2462 [110] for details.

4.5.3 Ethernet Addresses and Interface IDs

Figure 4.8 shows how a 48 bit Ethernet address is transformed into a 64 bit interface ID. Bit 7 in the first byte of the Ethernet address is called the

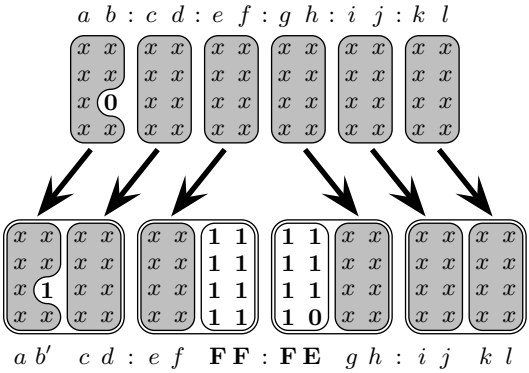

Fig. 4.8. Computing an interface ID from an Ethernet address

global bit and is always 0, indicating that this address is meant to be globally unique. The transformation toggles this bit, which has the great advantage that autoconfigured addresses become tedious to write and leave the "shorter" addresses for static configuration purposes. After three bytes from the Ethernet address, two extra bytes with the fixed values `FF` and `FE` are inserted, followed by the remaining three bytes of the Ethernet address.

It takes a bit of practice to recognize Ethernet addresses and their associated interface IDs. At first glance, the Ethernet address `00:0C:29:1A:4F:A1`

and the interface ID `20c:29ff:fe1a:4fa1` don't look too similar. But with a bit of practice this will soon change.

The transformation is defined in RFC 4291 [64, appendix A]. The resulting 64 bit address format is called *IEEE EUI-64 format*.

4.6 Testing and Debugging

With at least one host and router in a subnet it is possible to do some more testing. This helps to gather some experience dealing with autoconfiguration and related network problems.

Normally, autoconfiguration should work fine unless you mistype an address. Otherwise, problems are best found checking the network stack from the bottom up.

- ☐ All network interfaces and switch ports must show a working link if they have the customary link LEDs.
- ☐ All nodes must be connected to the *same* subnet.
- ☐ All interfaces must be "up" and "running".
- ☐ All hosts must have their interface configured with addresses from all prefixes advertised by the routers. If they don't, either wait for about ten minutes or manually send a router solicitation from one of the hosts; the BSDs have a command `rtsol` for this purpose.
- ☐ All hosts must be able to ping each other with all the advertised prefixes. Again, if this doesn't work, first check if you mistyped an address.
- ☐ All routers must be able to ping each other with the prefixes they both advertise. They must *not* be able to ping each other with prefixes only one of them advertises—section 7.7.6 explains why.

If possible, connect a packet sniffer to the network. While it won't show all traffic in a switched network, it shows at least packets that are sent as multicasts to all connected nodes. If possible, run it on one of the routers; doing so will show more of the relevant packets than running it on a host.

- ☐ Whenever a host boots, the packet sniffer must show the router solicitations it sends as well as the associated router advertisements sent from all routers. Note that we won't see the router solicitations in a switched network with a multicast-aware "smart" switch unless we run the packet sniffer on a router's interface.
- ☐ When a router boots, it sends an unsolicited router advertisement to all nodes. All hosts then configure addresses with the prefixes advertised by the router.
- ☐ Additionally, all routers must send unsolicited router advertisements, usually at intervals of no more than ten minutes.

If you have some time at hand you can do even more testing to watch the behaviour of the stateless autoconfiguration mechanism.

- When all routers advertising a prefix are shut down, all nodes must show the preferred and valid lifetimes of their associated address to be decreasing, eventually changing the address from the preferred to the deprecated and finally to the invalid state.
- When a host is disconnected and plugged into a different subnet, it must obtain new addresses and expire its old addresses. After a reboot it must show only the new addresses, being fully functional again in the new subnet.

Again, this is an excellent occasion to use a packet sniffer to get some practical experience and a feeling for what a working setup as well as some intentionally provoked problems look like.

4.7 Packet Filter Considerations

To set up a basic packet filter, we need to deal with three independent subtasks: We must sanitize "broken" packets as well as we can, we must filter spoofed packets and we must allow some essential ICMPv6. To test if our filter isn't too restrictive, we also allow ICMPv6 echo requests and replies so we can use `ping`.

4.7.1 From Stateless Filtering to Rewriting Filters

The first generation of packet filters only analyzed individual packets. Some IPv6 packet filters are still such *stateless filters*.

More powerful *stateful filters* or *connection tracking filters* analyze packets in the context of a communication flow. While this needs more resources on the filter, it can prevent a range of attacks that can only be detected if a packet is analyzed within the context of the communication flow it belongs to.

Yet more powerful are *rewriting filters* that not only analyze packets but also rewrite part of them to prevent attacks that abuse some implementation bugs on the victim machine, like predictable "random number generators".

Debian Sarge *At this time, the* `ip6tables` *filter is stateless and only partly adapted to the peculiarities of IPv6.*

FreeBSD 6.1 *The* `pf` *filter is a rewriting filter. It generally supports IPv6 but can't filter by most IPv6-specific options. Its stateful filter can't handle multicast traffic properly.*

Solaris 10 *The packet filter shipped with Solaris 10 only filters IPv4.* 25

4.7.2 Packet Sanitation

The more powerful a filter technology is, the less effort it is to filter broken packets. Right now, IPv6 filters are lagging behind their IPv4 counterparts but are slowly catching up.

Debian Sarge *Since* `iptables` *isn't (yet) capable of stateful filtering, it doesn't offer much to sanitize network packets.*

It does filter some IP options, notably routing headers (which are the IPv6 equivalent of IPv4 source routing), though.

FreeBSD 6.1 *Since* `pf` *is stateful and offers some rewriting features, it automatically sanitizes to a degree by itself.*

But it doesn't let us filter by individual IP options; there is a badly documented `allow-opts` *feature which enables a rule to admit "all" IP options while the default behaviour is to block packets with "any" IP option.*

Additionally, the parentheses notation that `pf` *uses to match addresses and prefixes against current interface configurations is at least partially broken with FreeBSD 6.1, so we can't rely on it.* 26

4.7.3 Packet Spoofing (Ingress) Filters

It is good practice to filter incoming packets based on the interface through which they arrive, because this prevents attacks that use spoofed source addresses. But with autoconfiguration this becomes more difficult than with a more static address configuration.

With autoconfiguration, IPv6 addresses may change over time. This poses a problem to some packet filters that only allow configuration with fixed addresses or address prefixes.

Some packet filters support notions like "all addresses configured on interface X" or similar. With others, every time autoconfiguration adds or removes an address the packet filter configuration needs to be updated.

Debian Sarge *Filter rules are "static"; they don't adapt to changing interface configurations.*

FreeBSD 6.1 *By default, filter rules are "static": If an interface name is given as the source or destination, then the configuration line expands to a set of rules with each holding one of the interface's addresses at the time that the configuration is loaded. If however the interface name is put in parentheses, then every time the rule is evaluated the packet is checked against the current interface configuration; again, unfortunately this feature doesn't work properly with FreeBSD 6.1.*

If we use a `pf` *configuration to allow connections from a trusted "inside network" to some untrusted "outside network" only, then packet spoofing filters may even be unnecessary if we filter all traffic statefully: Since only*

trusted parties can create state, spoofing attempts will be blocked because they can't create the necessary state from outside. 27

4.7.4 Essential ICMPv6 Packets

We need to let neighbor discovery packets pass unless we want to use statically configured neighbor caches. For stateless autoconfiguration we also need router solicitations and router advertisements. Finally, we also need multicast listener discovery packets. They use the ICMPv6 types 130–132 and 143.

None of these ICMPv6 packets need to be forwarded, they are only needed within a single subnet. But without these, IPv6 simply won't work.

We must also permit ICMPv6 error packets. Stateful filters usually do so automatically, but stateless filters may require explicit rules to permit these packets. They must also be forwarded on a router.

A server may have multiple addresses and a DNS record for each address, so a properly written client will use each address in turn until it actually reaches the server. If we discarded ICMPv6 error packets, then the clients would wait for a timeout on each address before trying the next. The resulting delay is unacceptable at least if a user is actually waiting for a response.

4.7.5 Sample Filter Configurations

The entire layout of a filter configuration is highly dependent on the features that the particular filter has to offer. That makes the following sample configurations look quite different.

Additionally, there are few well-established approaches to IPv6 packet filtering. The following filter rules are a snapshot of today's still immature state-of-the-art configuration strategies, but it is likely that growing experience and new features in future releases of the available filter implementations will make them insufficient and obsolete in a short time.

Debian Sarge *With the limited filter functionalities of* `ip6tables` *a basic filter setup might look like this. For now we only need to filter incoming packets, but eventually we will also filter outbound and forwarded packets, so we already set up chains for them as well.*

`/etc/ip6tables.sh`

```
#! /bin/bash

INTERFACES="eth0"
ADDR_eth0="2001:db8:fedc:abcd::1 2001:db8:fedc:cdef::1"
```

First we define what our network looks like. These variables are used later on to generate some anti-spoofing rules.

/etc/ip6tables.sh

```
drop ()   { /sbin/ip6tables -j DROP   --append "$@"; }
accept () { /sbin/ip6tables -j ACCEPT --append "$@"; }
chain ()  { /sbin/ip6tables -N "$@"; }
call ()   { chain=$2; srcchain=$1; shift; shift;
            /sbin/ip6tables -j $chain --append $srcchain "$@"; }
ret()     { /sbin/ip6tables -j RETURN --append "$@"; }
```

These shell functions simplify the syntax of the subsequent script. The `drop` *function silently discards a packet; as soon as the* `REJECT` *target works, it should probably be substituted here. The* `accept` *function immediately accepts a packet. The* `chain` *function just creates a new filter chain. With* `call` *we can call another such chain. Finally,* `ret` *ends a filter chain, returning to the calling chain.*

/etc/ip6tables.sh

```
/sbin/ip6tables -P INPUT   DROP
/sbin/ip6tables -P FORWARD DROP
/sbin/ip6tables -P OUTPUT  DROP
/sbin/ip6tables --flush
/sbin/ip6tables -X
```

Before we do anything we clean out the existing configuration.

/etc/ip6tables.sh

```
chain  SANITIZE
call   INPUT SANITIZE
drop   SANITIZE --match ipv6header --header route --soft
ret    SANITIZE
```

Filtering broken packets is extremely limited with `ip6tables`*. We just drop all packets with a routing header, which is the IPv6 equivalent of source routing.*

/etc/ip6tables.sh

```
chain  DESPOOF
call   INPUT DESPOOF
for i in $INTERFACES
do
    eval addrs=\"\$ADDR_$i\"
    for a in $addrs
    do
        ret  DESPOOF -s $a -i lo
        drop DESPOOF -s $a
        drop DESPOOF -s $a/64 -i '!' $i
    done
done
# On routers the next line must be replaced with more specific rules.
ret DESPOOF
```

We drop all incoming packets with one of our addresses as its source address and all packets coming in from the wrong interface. On routers we will later on replace the last line with some more filter rules that must match the local network topology.

/etc/ip6tables.sh

```
chain  CHECKSRC
call   INPUT CHECKSRC
ret    CHECKSRC -s :: -p icmpv6 --icmpv6-type neighbor-solicitation
ret    CHECKSRC -s ::1 -i lo
ret    CHECKSRC -s fe80::/10
ret    CHECKSRC -s fc00::/7
ret    CHECKSRC -s 2000::/3
drop   CHECKSRC
```

Next we check if the source addresses are actually valid. Note that we can't check for anycast addresses here. The first filter rule passes neighbor solicitations from the unspecified address, so duplicate address detection works.

/etc/ip6tables.sh

```
chain  CHECKDST
call   INPUT CHECKDST
ret    CHECKDST -d fe80::/10
ret    CHECKDST -d fc00::/7
ret    CHECKDST -d 2000::/3
ret    CHECKDST -d ff00::/8
drop   CHECKDST
```

Next we check the destination address. Only the addresses listed in these rules are valid destination addresses.

/etc/ip6tables.sh

```
# Deal with ICMPv6
chain  ICMP
call   INPUT ICMP -p icmpv6
call   OUTPUT ICMP -p icmpv6
accept ICMP -p icmpv6 --icmpv6-type destination-unreachable
accept ICMP -p icmpv6 --icmpv6-type packet-too-big
accept ICMP -p icmpv6 --icmpv6-type ttl-exceeded
accept ICMP -p icmpv6 --icmpv6-type parameter-problem
```

The first ICMPv6 types we need to allow deal with various errors. These packet types must also be forwarded on a router.

/etc/ip6tables.sh

```
accept ICMP -p icmpv6 --icmpv6-type neighbour-solicitation
accept ICMP -p icmpv6 --icmpv6-type neighbour-advertisement
```

Without neighbor discovery IPv6 doesn't work too well.

/etc/ip6tables.sh
```
accept ICMP -p icmpv6 --icmpv6-type 130
accept ICMP -p icmpv6 --icmpv6-type 131
accept ICMP -p icmpv6 --icmpv6-type 132
accept ICMP -p icmpv6 --icmpv6-type 143
```

These ICMPv6 types are used for multicast listener discovery. Without them, a smart switch won't send us the multicasts we want and multicast routing won't work.

/etc/ip6tables.sh
```
accept ICMP -p icmpv6 --icmpv6-type router-solicitation
accept ICMP -p icmpv6 --icmpv6-type router-advertisement
```

With router discovery we need to allow router solicitations and router advertisements. If we wanted to be particularly smart we'd allow router solicitations only from hosts to routers and router advertisements only from routers to hosts and the link-local all-node multicast address.

/etc/ip6tables.sh
```
accept ICMP -p icmpv6 --icmpv6-type echo-request
accept ICMP -p icmpv6 --icmpv6-type echo-reply
drop ICMP
```

If we want to use `ping` we need to allow echo requests and echo replies. We drop all other ICMPv6 packets.

/etc/ip6tables.sh
```
accept OUTPUT
```

For now we don't filter outgoing packets.

FreeBSD 6.1 *The `pf` filter is quite powerful. But still we need to do some work ourselves, like checking the source and destination addresses.*

/etc/pf.conf
```
ifs="(lnc0)"
localnets="(lnc0:network) fe80::/10"
```

First we define the interfaces and addresses that we want to filter. Notice the parentheses around the entries: They make the filter use the current addresses of each interface whenever a filter rule is evaluated. Without them, the packet filter only looks up the addresses and prefixes associated with the interfaces when it installs rules, which improves performance but introduces the risk that the packet filter uses outdated addresses if the interface configuration changes. As the warning on page 19 points out, this doesn't work reliably with FreeBSD 6.1.

/etc/pf.conf
```
scrub in all
```

Packet sanitation is extremely simple because `pf` *does a lot of work in its stateful filter engine. And since we don't have much control on what IP options to filter, we can't add any more lines here to filter dangerous options.*

/etc/pf.conf

```
pass in quick on lo0 inet6 from {(lo0) $ifs} to {(lo0) $ifs ff00::/8}
antispoof quick for { (lo0) $ifs }
```

As far as the subnets connected to our node are concerned, anti-spoofing is also quite simple as long as we "trust" our loopback interface. If we wanted to control traffic on the loopback interface, for example to control which local users use which local services, this would become far more complex.

/etc/pf.conf

```
block in inet6 proto icmp6 \
                      from :: to ff02::1:ff00:0/104 \
                           icmp6-type neighbrsol code 0  tag GOODSRC
block in on lo0 inet6 from {(lo0) $ifs} to {(lo0) $ifs}   tag GOODSRC
block in        inet6 from fe80::/10  to {$ifs fe80::/10} tag GOODSRC
block in        inet6 from fc00::/7   to any              tag GOODSRC
# The next line must be replaced on a router.
block in        inet6 from 2000::/3     to any            tag GOODSRC
block in        inet6 from {(lo0) $ifs} to ff01::/16      tag GOODSRC
block in        inet6 from {(lo0) $ifs} to ff11::/16      tag GOODSRC
block in        inet6 from fe80::/10    to ff02::/16      tag GOODSRC
block in        inet6 from fe80::/10    to ff12::/16      tag GOODSRC
block in quick inet6 all ! tagged GOODSRC
```

Now we check if the source addresses are valid. It isn't important if we block or pass packets here; adding the tag is important. Yes, last match semantics are sometimes slightly weird—but they work. The line filtering on `2000::/3` *must be replaced on a router with some more specific rules that match known address prefixes to their proper interfaces. Again, these are mostly necessary only if we want to allow untrusted nodes to connect to us.*

/etc/pf.conf

```
block in        inet6 from {$localnets} to fe80::/10 tag GOODDST
block in        inet6 from any          to fc00::/7  tag GOODDST
block in        inet6 from any          to 2000::/3  tag GOODDST
block in        inet6 from {(lo0) $ifs} to ff01::/16 tag GOODDST
block in        inet6 from {(lo0) $ifs} to ff11::/16 tag GOODDST
block in        inet6 from {$localnets} to ff02::/16 tag GOODDST
block in        inet6 from {$localnets} to ff12::/16 tag GOODDST
block in quick inet6 all ! tagged GOODDST
# Block anycast source addresses (currently infeasible)
```

These rules filter illegal destinations. Yet again, they are mostly necessary only if we want to allow untrusted nodes to connect to us.

/etc/pf.conf

```
# Rules to open certain TCP or UDP ports go here.
block return-icmp6(admin-unr) quick inet6 proto tcp all
block return-icmp6(admin-unr) quick inet6 proto udp all
```

For now we block all TCP and UDP traffic. Later on we put additional rules before these.

/etc/pf.conf

```
pass  quick inet6 proto icmp6 from {$localnets ::} \
                              to {$localnets ff02::1:ff00:0/104} \
                              icmp6-type neighbrsol
pass  quick inet6 proto icmp6 from {$localnets} to {$localnets} \
                              icmp6-type neighbradv
```

We allow all neighbor discovery on the directly attached subnets.

/etc/pf.conf

```
pass  quick inet6 proto icmp6 from {$localnets} to any \
                              icmp6-type groupqry
pass  quick inet6 proto icmp6 from {$localnets} to any \
                              icmp6-type grouprep
pass  quick inet6 proto icmp6 from {$localnets} to any \
                              icmp6-type groupterm
pass  quick inet6 proto icmp6 from {$localnets} to any \
                              icmp6-type 143
```

We also need to allow multicast listener discovery at least when we are in a switched network or want to use multicast routing.

/etc/pf.conf

```
pass quick inet6 proto icmp6 from {$localnets} \
                             to   {$localnets ff02::2} \
                             icmp6-type routersol
pass quick inet6 proto icmp6 from {$localnets} \
                             to   {$localnets ff02::1} \
                             icmp6-type routeradv
```

Router discovery can be controlled more precisely if we distinguish between routers and hosts. If you want, add an `in` *modifier to the first and an* `out` *modifier to the last line for a router or the other way around on a host.*

/etc/pf.conf

```
pass  quick inet6 proto icmp6 all icmp6-type echoreq keep state
# pass  quick inet6 proto icmp6 all icmp6-type echoreq
# pass  quick inet6 proto icmp6 all icmp6-type echorep
```

If we only want to allow ping between unicast addresses, then the first line takes care of everything: Each echo request creates a state that lets the subsequent echo reply pass.

If we also want to ping multicast addresses like `ff02::1` *or* `ff02::2` *to see which nodes are currently up and running, then the stateful filter is reaching its limits and we need to filter the packets in a stateless manner. This is also the reason why we can't filter neighbor or router discovery using connection tracking.*

/etc/pf.conf

```
block return-icmp6(admin-unr) quick inet6 proto icmp6 all
```

For now we block all other ICMPv6 traffic.

/etc/pf.conf

```
block return log all
```

Normally, all packets should hit a matching `quick` *rule before they ever get here. But still, a restrictive catch-all rule at the end is always a good idea.* 28

4.7.6 Testing the Filter Configuration

Checking a packet filter configuration is generally difficult; it is easy to deal with a missing or overly restrictive configuration because whatever we want to do doesn't work, but there is no reliable way to check that a filter configuration isn't too permissive.

Nevertheless, to check that all necessary functionalities are working it helps to use a systematic approach. The following checklist assumes that we have a second node running a packet sniffer in the same subnet, so we can ensure correct behaviour. If the network we sniff is heavily loaded with traffic, then we may need to set filtering rules in the sniffer accordingly; while this introduces another possible source of error it may be our only choice to find the traffic we are actually looking for.

- ☐ Reboot the filter node. On the packet sniffer, watch for duplicate address detection packets and neighbor discovery packets for the all-node link-local multicast address as well as the solicited-node multicast address of the filter node.
- ☐ Stop the filter node again. Configure the second node with the same address as the filter node. Reboot the filter node. The duplicate address detection must mark the address as unavailable.
- ☐ Now disable the address on the second node again and reboot the filter node. When the filter node is up again and has its address configured, add the same address to the second node. The duplicate address detection must now block the address on the second node. Remove the address on the second node again.
- ☐ Check that neighbor solicitation works: Try to ping the second node in the same subnet. The neighbor discovery packets must be visisble

and the neighbor discovery cache on the sending machine must show an entry for the target machine.

- ☐ Try to telnet to a closed port on the remote machine. A “connection refused” error must be returned.
- ☐ If ping packets are allowed, try to ping between the two machines.

5

IPv6 and the Domain Name System (DNS)

Since IPv6 addresses are slightly longer and more cumbersome to type than IPv4 addresses, the next topic to tackle is the domain name system. While this is not exactly the most exciting aspect of IPv6, it will free our hands from typing in IPv6 addresses all the time.

5.1 Getting Started

Before we actually store IPv6 addresses in the DNS we first sort out three fundamental things: We devise a naming standard that lets us use both IPv4 and IPv6 addresses with the least necessary hassle, we set up another test environment and we see how we can avoid using the DNS altogether by storing the addresses in `/etc/hosts` instead.

5.1.1 Naming Conventions

There are multiple established naming standards. Consider our archetypical webserver `www.example.com`. We may add DNS records to its IPv6 addresses like this:

`www.example.com` It is possible to have both IPv4 and IPv6 addresses assigned to a single DNS name. This is essential when a machine must be accessible to users who shouldn't notice if they use IPv4 or IPv6.
The drawback is that it makes it likely to bark up the wrong tree IP-wise: Especially when you start working with IPv6 it is quite likely that you are occasionally trying to solve an IPv4 problem by fixing IPv6 or vice versa.

`www6.example.com` So from a system administrator's point of view it is often more convenient to assign different names to IPv6 addresses. The easiest way to do so is by appending a "6" to the unqualified name.
This approach has the advantage that the names stay fairly short and are easily typed. The main disadvantage is that we need to keep IPv4 and

IPv6 addresses in the same DNS zone. Additionally, a DNS name like `route66.example.com` doesn't lend itself easily to this strategy.

`www.ip6.example.com` With a dedicated subdomain like `ip6.example.com` it is possible to manage all IPv6 addresses in their own DNS zone, effectively decoupling them entirely from the IPv4 world.

The important thing to remember here is that you can assign multiple names to an address as well as multiple addresses to a single name. My personal favourite strategy, again for the example webserver `www.example.com` with the single legacy IPv4 address `192.0.2.80`, the routed IPv6 addresses `2001:db8:fedc:abcd:80::80` and `2001:db8:fedc:cdef:80::80` plus the link-local address `fe80::222:33ff:fe44:5566` looks like this:

`www.example.com`	→	`192.0.2.80`
		`2001:db8:fedc:abcd:80::80`
		`2001:db8:fedc:cdef:80::80`

`www.ip4.example.com`	→	`192.0.2.80`
`www.ip6.example.com`	→	`2001:db8:fedc:abcd:80::80` `2001:db8:fedc:cdef:80::80`
`192.0.2.80`	→	`www.ip4.example.com`
`2001:db8:fedc:abcd:80::80`	→	`www.ip6.example.com`
`2001:db8:fedc:cdef:80::80`	→	`www.ip6.example.com`
`fe80::222:33ff:fe44:5566`	→	`www.ip6ll.example.com`

This turns out to be very useful: We and our users can use the short name if we don't want to worry about the IP version we are currently using. We can also use the long names if we want to use a specific IP version. And a reverse lookup will always yield us a name that includes the IP version, even when we look up the link-local address.

There are however two potential drawbacks: This approach may break some old, badly written programs that do a reverse DNS lookup for "security purposes" and it may confuse ordinary users if they ever see a reverse lookup name saying `www.ip6.example.com` if they actually tried to connect to `www.example.com`.

5.1.2 The DNS Test Setup

Throughout this chapter we use the test setup shown in figure 5.1. The DNS server we build in section 5.2 gets statically configured IPv6 addresses with an interface ID `::1`. For starters, the clients will have statically configured addresses with the two prefixes `2001:db8:fedc:abcd::/64` and `2001:db8:fedc:cdef::/64` and the interface IDs `::11`, `::22`, ... instead of autoconfigured addresses.

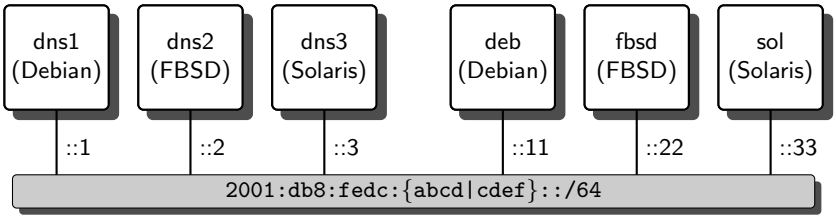

Fig. 5.1. The name system test setup

5.1.3 Local Address Management with `/etc/hosts`

To avoid all the trouble that the DNS introduces, it is sometimes sufficient to maintain the host names and addresses on all machines locally.

Just to see how IPv4 and IPv6 work in parallel, we also configure IPv4 addresses from the `192.0.2.0/24` subnet on the clients, using the last byte of the interface ID as the IPv4 host part.

Following the example from the previous section we can add the necessary data to `/etc/hosts` like this:

`/etc/hosts`

```
# IPv4 addresses
#################
192.0.2.11 deb.ip4.example.com  deb.example.com  deb
192.0.2.22 fbsd.ip4.example.com fbsd.example.com fbsd
192.0.2.33 sol.ip4.example.com  sol.example.com  sol

# IPv6 addresses
#################
2001:db8:fedc:cdef::11   deb.ip6.example.com    deb.example.com  deb
2001:db8:fedc:abcd::11   deb.ip6.example.com    deb.example.com  deb
fe80::20c:29ff:fe73:bb02 deb.ip6ll.example.com

2001:db8:fedc:cdef::22   fbsd.ip6.example.com   fbsd.example.com fbsd
2001:db8:fedc:abcd::22   fbsd.ip6.example.com   fbsd.example.com fbsd
fe80::20c:29ff:fe72:5d4c fbsd.ip6ll.example.com

2001:db8:fedc:cdef::33   sol.ip6.example.com    sol.example.com  sol
2001:db8:fedc:abcd::33   sol.ip6.example.com    sol.example.com  sol
fe80::20c:29ff:fe03:8582 sol.ip6ll.example.com
```

Solaris 10 *Different than Debian and FreeBSD, Solaris expects IP addresses to be placed in* `/etc/inet/ipnodes`*; only IPv4 addresses may also be kept in* `/etc/inet/hosts`*, to which* `/etc/hosts` *is symlinked.*

According to the `ipnodes(4)` man page in this case the entries should be copied to `/etc/inet/ipnodes` as well—which proves not to work: If you try to telnet or ssh (see below) to another machine which has both IPv4 and IPv6 addresses listed in `/etc/inet/ipnodes`, it apparently tries only the addresses belonging to the first IP version listed for the name.

If however you keep IPv4 addresses in `/etc/inet/hosts` and IPv6 addresses in `/etc/inet/ipnodes`, `telnet` and `ssh` will try all addresses they find. 29

We can test with `telnet` if this works: If the remote side doesn't run a telnet server, the client should attempt to use all listed addresses, returning an error notice on every address. The options `-4` and `-6` explicitly force the client to use IPv4 and IPv6, respectively.

Debian Sarge *Without the `-6` option the telnet client will only try the first IPv6 address listed, not all of them.*

Solaris 10 *The telnet client has no `-4` and `-6` options. As a substitute `ssh` with the `-v` option and `-4` or `-6` can be used.* 30

- ☐ Test if telnet works with the `-4` flag. If the remote machine is up and doesn't run a telnet server, the "connection refused" error messages list the IPv4 addresses.
- ☐ Test again with the `-6` flag. Now all and only the IPv6 addresses must be listed.
- ☐ Finally, try again without a `-4` or `-6` flag. This tries all addresses independent of the IP version (but see above for Debian).
- ☐ Depending on the particular Unix, `traceroute6` and/or `ping6` often do a reverse lookup on the addresses they receive replies from. This helps to verify that resolving addresses into names works as expected.

Debian Sarge *Both `ping6` and `traceroute6` do a reverse lookup.*

FreeBSD 6.1 *Only `traceroute6` does a reverse lookup by default; `ping6` supports an extra option `-H` to force a reverse lookup.*

Solaris 10 *The `ping` command does a reverse lookup if the option `-s` is used to provide for a continuous output. The `traceroute` command doesn't do reverse lookups for IPv6 addresses.* 31

In a small network, maintaining an `/etc/hosts` file and distributing it using tools like `scp` or `rsync/ssh` may well be sufficient for a start. Even in larger, production-grade environments it may be useful to maintain the addresses of key servers this way on all the machines in our network just in case.

5.2 IPv6 Addresses in the DNS

IPv6 and the DNS interact with each other in two entirely independent ways. A client that needs to resolve a DNS name into an IP address (or vice versa)

needs to talk to a DNS server, possibly using IPv6 as the underlying protocol and a client may ask the server to resolve a DNS name into an IPv6 address (or vice versa). Both aspects are entirely unrelated.

The canonical DNS implementation in the Unix world is called BIND. It has a number of drawbacks, but so far no alternative has become anywhere near as popular. If you are still unfamiliar with it, see appendix A for an introduction before you read on.

5.2.1 Resolver Configuration

Before we set up a name server we need to configure the resolver library on both the clients as well as the servers. First we tell them to use the DNS.

Debian Sarge, FreeBSD 6.1 *An entry*

/etc/nsswitch.conf

```
hosts: files dns
```

in `/etc/nsswitch.conf` *makes the resolver first look in the local* `/etc/hosts` *file and then ask the name server.*

Solaris 10 *Similarly, two entries*

/etc/nsswitch.conf

```
ipnodes: files dns
hosts:   files dns
```

in `/etc/nsswitch.conf` *tell the Solaris resolver to search* `/etc/inet/ipnodes` *and* `/etc/inet/hosts` *first and then to ask the name server for IPv6 and IPv4 addresses, respectively.* 32

Next we tell the resolver where to look for the DNS server and what default domain to use. For a change, this configuration doesn't differ between the Unixen shown here; we just add our name server address(es) in `/etc/resolv.conf` like this:

/etc/resolv.conf

```
domain example.com.
nameserver 2001:db8:fedc:abcd::1
nameserver 2001:db8:fedc:cdef::1
nameserver 192.0.2.1
```

The last line obviously only makes sense if the particular node is dual-stacked and can reach the IPv4 name server.

At this point it is already possible to test the client configuration.

- ☐ Start a packet sniffer in the network and make it trace DNS traffic (UDP port 53). This requires that the configured name servers are available on the net and answer neighbor solicitations. They don't need to answer the DNS requests at this point.

- ☐ Make the client look for hostnames and/or addresses not locally configured in `/etc/hosts` and such.
- ☐ Check the packet sniffer for matching DNS traffic.

5.2.2 Enabling IPv6 on the DNS Server

Next we need a DNS server. It needs to be reachable at a fixed IP address, so we either configure an existing name server as a statically configured IPv6 host or we set up a new machine accordingly.

With an existing name server the next step is to check that it runs a reasonably recent version of BIND; version 9.2.3 and up should be fine while older versions will cause problems with reverse lookups; see the warning on page 74. Appendix A.2.1 explains how to install a recent BIND version.

Next we need to make our name server listen on an IPv6 socket. With existing servers, we mostly need to add a `listen-on-v6` statement to the `options` section in `named.conf`:

named.conf

```
options {
    [...]
    listen-on-v6 { any; };
    allow-query { "any"; };
    [...]
};
```

The `allow-query` statement is not always necessary, but if it already exists and restricts which addresses may query the name server, then it may need to be updated. Depending on the version of BIND it may be possible to restrict the server to listen on a specific address only, but for our purposes it suffices to allow IPv6 queries on all interfaces. Instead of `any` we may however list network prefixes or such here to restrict access to the name server. This is especially important with BIND versions at least up to version 9.2.4, where `listen-on-v6` doesn't allow us to restrict the name server to individual addresses like at least version 9.3.1 does.

5.2.3 Forwarder Configuration vs. a Fake Root Zone

If we want to use the name server as a *forwarder*, we need to configure the name server accordingly. As with IPv4, this only takes a statement

named.conf

```
options {
    [...]
    forwarders { 192.0.2.1; 2001:db8:fedc::1; };
    [...]
};
```

in the `options` section of `named.conf`. IPv4 and IPv6 addresses can be freely mixed here. If the name server is dual-stacked, it will happily accept DNS queries via IPv6 and forward them to its forwarder via IPv4 and vice versa. If it is not dual-stacked, we must make sure that both servers can reach each other using the same IP version.

Alternatively, we can configure the list of root servers in our name server. Since these are currently all IPv4-only, there are no configuration changes due to IPv6.

If we run in a disconnected environment however, we will want to configure a *fake root zone*. This will ensure that all queries that can't be resolved will immediately be answered with a negative reply; otherwise the resolver library may wait for a timeout, causing considerable delay. The next section explains how to do this together with setting up an "ordinary" forward zone.

5.2.4 Forward Zones on a Primary Server

Next we add data to our DNS server to map domain names into IPv6 addresses. Both IPv4 and IPv6 addresses are stored in the same zone. Assuming that we have a zone "`example.com.`" with its zone data kept in `example.com.fwd`, we just add more resource records to it.

The top of the zone file may look somewhat like this:

`example.com.fwd`

```
$TTL 3600

@ SOA    dns1.example.com. root.dns1.example.com. (1 15m 5m 30d 1h)
  NS     dns1.example.com.
  NS     dns2.example.com.
  NS     dns3.example.com.

localhost A       127.0.0.1
          AAAA    ::1
```

The last line is new and the most interesting one here. It shows that it is perfectly reasonable to have both IPv4 and IPv6 address records for a given name. IPv6 addresses are kept in `AAAA` or "quad A" records that look just like the familiar `A` records used for IPv4.

While it isn't strictly necessary to have a `localhost` entry, it sometimes speeds up the lookups for an unqualified `localhost` name, so adding it here is good practice.

If we don't have a forwarder configuration and want to set up a fake root zone, we must copy the lines so far to another file `root.fwd` in the same directory as `example.com.fwd`.

The only thing missing are the actual DNS data for our six machines. So we add them to the zone file like this:

example.com.fwd

```
dns1       AAAA    2001:db8:fedc:abcd::1
           AAAA    2001:db8:fedc:cdef::1
dns1.ip6   AAAA    2001:db8:fedc:abcd::1
           AAAA    2001:db8:fedc:cdef::1
dns2       AAAA    2001:db8:fedc:abcd::2
           AAAA    2001:db8:fedc:cdef::2
dns2.ip6   AAAA    2001:db8:fedc:abcd::2
           AAAA    2001:db8:fedc:cdef::2
dns3       AAAA    2001:db8:fedc:abcd::3
           AAAA    2001:db8:fedc:cdef::3
dns3.ip6   AAAA    2001:db8:fedc:abcd::3
           AAAA    2001:db8:fedc:cdef::3

deb        AAAA    2001:db8:fedc:abcd::11
           AAAA    2001:db8:fedc:cdef::11
deb.ip6    AAAA    2001:db8:fedc:abcd::11
           AAAA    2001:db8:fedc:cdef::11
fbsd       AAAA    2001:db8:fedc:abcd::22
           AAAA    2001:db8:fedc:cdef::22
fbsd.ip6   AAAA    2001:db8:fedc:abcd::22
           AAAA    2001:db8:fedc:cdef::22
sol        AAAA    2001:db8:fedc:abcd::33
           AAAA    2001:db8:fedc:cdef::33
sol.ip6    AAAA    2001:db8:fedc:abcd::33
           AAAA    2001:db8:fedc:cdef::33
[...]
```

If we don't have a forwarder configuration and want to use a fake root zone, we add

named.conf

```
zone "." { type master; file "root.fwd"; };
```

to our `named.conf`. Finally, we make the `named` reload its configuration or reboot the system.

At this point it is a good idea to ensure that everything works as expected:

- ☐ Check the system log files for suspicious lines containing the string "named".
- ☐ Use `ps` to see if the `named` is running.
- ☐ Use `netstat -a` to see if `named` is listening on IPv6.
- ☐ Check that the name `dns1.example.com` is resolved correctly:

  ```
  dig @::1 dns1.example.com. ANY
  ```

  If you don't use the `ANY` or `AAAA` query type option, `dig` will only look for IPv4 addresses. With `ANY` it will search for both IPv4 and IPv6 addresses, with `AAAA` only for IPv6 addresses.

- ☐ Ask the name server for the entire "`example.com.`" zone with

  ```
  dig @::1 example.com. AXFR
  ```

  and check that the data is correct. If this doesn't work, make sure you are allowed to do a zone transfer from `::1`.
- ☐ Check that forwarding or your fake root zone works with

  ```
  dig @::1 . ANY
  ```

  to retrieve the root zone data that the name server uses.

5.2.5 Reverse Zones on a Primary Server

Now we need to add reverse zones to resolve IPv6 addresses into their corresponding DNS names. Unlike the forward records, reverse records for IPv4 and IPv6 are kept in separate zones. More important, the notation, called *nibble format*, is distinctly ugly, tedious and error-prone.

The domain name notation of the IPv6 address `2001:db8:fedc:abcd::1` looks like this:

1.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.d.c.b.a.c.d.e.f.8.b.d.0.1.0.0.2.ip6.arpa.

Note the reduced font size to fit it onto the page. To make these addresses remotely manageable in a zone file, two tricks are extremely helpful. The zone file for our network prefix `2001:db8:fedc::/48` shows them:

`2001.db8.fedc.rev`

```
$TTL 3600

@ SOA    dns1.example.com. root.dns1.example.com. (1 15m 5m 30d 1h)
NS     dns1.example.com.
NS     dns2.example.com.
NS     dns3.example.com.

;                          1 1 1 1 1 1 1
;       1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6
$ORIGIN d.c.b.a.c.d.e.f.8.b.d.0.1.0.0.2.ip6.arpa.

;                 1 1 1 1 1 1 1
; 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6
1.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0  PTR  dns1.ip6.example.com.
2.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0  PTR  dns2.ip6.example.com.
3.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0  PTR  dns3.ip6.example.com.
1.1.0.0.0.0.0.0.0.0.0.0.0.0.0.0  PTR  deb.ip6.example.com.
2.2.0.0.0.0.0.0.0.0.0.0.0.0.0.0  PTR  fbsd.ip6.example.com.
3.3.0.0.0.0.0.0.0.0.0.0.0.0.0.0  PTR  sol.ip6.example.com.
```

[Continued on next page]

2001.db8.fedc.rev

```
[Continued from previous page]
$ORIGIN f.e.d.c.c.d.e.f.8.b.d.0.1.0.0.2.ip6.arpa.
1.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0  PTR  dns1.ip6.example.com.
2.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0  PTR  dns2.ip6.example.com.
3.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0  PTR  dns3.ip6.example.com.
1.1.0.0.0.0.0.0.0.0.0.0.0.0.0.0  PTR  deb.ip6.example.com.
2.2.0.0.0.0.0.0.0.0.0.0.0.0.0.0  PTR  fbsd.ip6.example.com.
3.3.0.0.0.0.0.0.0.0.0.0.0.0.0.0  PTR  sol.ip6.example.com.
[...]
```

Using a comment "ruler" to keep track of the number of nibbles is probably the only way to reduce the number of cases with the wrong number of nibbles to a reasonable level. The `$ORIGIN` statement is also essential; it defines the "origin", the domain name suffix to append to all unqualified names. Using it properly, like in the example, trims the lines of the zone file to a TTY-compatible width.

Alternatively, Peter Bieringer's `ipv6calc` generates fully qualified domain name representations of IPv6 addresses if we run it as

```
# ipv6calc 2001:db8:fedc:abcd::1 --out revnibbles.arpa
# ipv6calc 2001:db8:fedc:abcd::1 --out revnibbles.int
```

for an address `2001:db8:fedc:abcd::1`.

We make the name server serve the zone from this file by adding

named.conf

```
zone "c.d.e.f.8.b.d.0.1.0.0.2.ip6.arpa." {
    type master;
    file "2001.db8.fedc.rev";
};
```

to our `named.conf`. Finally, we make the name server reload its configuration.

To test it, we proceed as with the forward zones. The `dig` command has an option `-x` that makes it convert the following IP address into the corresponding domain name:

```
% dig @::1 -x 2001:db8:fedc:abcd::1 PTR
```

This tends to make it far easier to query IPv6 addresses.

If you use a `dig` version from a BIND release prior to 9.2.3, this will fail because `dig` will try to look up the address in a different (now deprecated) "bitlabel format". See section 5.2.8 for an explanation.

It is possible to delegate subzones at arbitrary nibble boundaries, but except for large networks that shouldn't be necessary. If you delegate subzones, try to keep all of them at the same nibble boundary to avoid unnecessary confusion.

5.2.6 Secondary Servers

Setting up secondary name servers is no particular problem. The `masters` statement in a zone clause supports IPv6 as well as IPv4 addresses. To the DNS, IPv6 reverse zones are just zones like all others, so they can be replicated like any other zone.

The only thing to watch out for is that the primary and secondary name servers must be able to talk to each other; if one only supports IPv4 and the other only supports IPv6 this won't work.

5.2.7 Testing and Debugging

At this point it is advisable to ensure that the name server is working as expected.

- ☐ Check with `ps` that a `named` process is running.
- ☐ With `netstat -a` check that it listens on IPv6 for incoming requests.
- ☐ Using `dig`, do a zone transfer on all zones you created. Make sure there are no missing trailing dots. In reverse zones, ensure that the number of nibbles is correct—BIND will consider domain names with the wrong number of nibbles perfectly reasonable but the resolver won't find them.
- ☐ If you have any secondary servers, make sure they return the same data as the primary.
- ☐ Again, use `dig` to test individual names.
- ☐ Yet again, use `dig` with the `-x` option to test the reverse resolution of addresses.

Finally, we can check if the resolver library on all nodes can successfully query the DNS servers.

- ☐ On all nodes, including any client, remove all entries from `/etc/hosts` that contain data you want to obtain from the DNS.
- ☐ Check that all nodes can use the DNS names, using `ping` or `telnet` or similar to test the resolver library.
- ☐ Check the log files for any suspicious lines containing the string "named".

5.2.8 Annoying Legacies

As if the DNS wasn't already messy enough, there are some additional historic problems. While up-to-date Unixen won't be affected by these you may run into them either with older releases or with books that are a few years old.

Originally, IPv6 addresses in forward records used the `AAAA` record type that we've already seen and which was introduced in RFC 1886 [108]. In

RFC 2874 [21] `AAAA` records were replaced by `A6` records, which provide an additional feature to copy an address prefix from another forward name, so prefixes could be changed by updating a single resource record. In RFC 3363 [13] the IETF realized that this allows lookup loops and makes it difficult to define a reasonable timeout for DNS queries. So in RFC 3596 [109] the `A6` record type was declared "experimental" again and `AAAA` records were restored as the official record type for IPv6 addresses.

To my knowledge `A6` records have never been used with any resolver library. They are however supported with BIND, so if you really want to be sure that a resolver using them will work, add `A6` records according to the pattern

example.com.fwd

```
www.example.com.   AAAA     2001:db8:fedc:cdef::80
                   A6   0   2001:db8:fedc:cdef::80
```

to your forward zones. Note the additional "`0`" attribute in the `A6` record.

With reverse records the situation is even worse. Originally, as of RFC 1886 again, the nibble format we've seen was used but within the "`ip6.int.`" pseudo-domain. Then in RFC 2672 [19] and RFC 2673 [18] the so-called *bitstring format* or *bitlabel format* was introduced together with the `DNAME` record type. Again, the intention was to simplify DNS reconfiguration when a network was renumbered. To make things even worse, bitstring records were kept in the "`ip6.arpa.`" pseudo-domain. But the same problems as with `A6` records caused another change in standards. In RFC 3363 the problem was analyzed and in RFC 3596 it was finally decided to switch back to the nibble format—but to stick with the "`ip6.arpa.`" pseudo-domain.

Unfortunately, support for all three formats may actually be necessary: The original nibble format in "`ip6.int.`" is still used by some older resolvers; additionally, some up-to-date resolvers first look in "`ip6.arpa.`" and if that fails they also check "`ip6.int.`", so don't be surprised if you see this in a packet sniffer. BIND versions up to 9.2.2 came with a `dig` that used the bitstring format if you used the `-x` option. Only in 9.2.3, which claimed to be a "maintenance release" without new or modified features, this was silently changed so `dig` used the nibble format with "`ip6.arpa.`".

So if you use some older Unix releases with IPv6, first clone your reverse zone data to the old "`ip6.int.`" pseudo-domain. If you also want the bitstring format, first add another zone to your `named.conf`:

named.conf

```
zone "\[x20010db8fedc/48].ip6.arpa." {
     type master;
     file "2001.db8.fedc.bitstring";
};
```

Then create the zone file holding the data. While the SOA and NS records don't change, the PTR records look different than before:

2001.db8.fedc.bitstring

```
[...]
;            11111111112
;  12345678901234567890
\[xabcd0000000000000001/80]  PTR  dns1.example.com.
```

With the bitstring format address parts are written as a hexadecimal number and a decimal number specifying the number of bits. They start with a "`\[x`" followed by the hexadecimal number, a slash, the number of relevant bits in decimal and a "`]`" (without a preceding backslash). Using two bitstrings in the zone name or an `$ORIGIN` statement and as an unqualified name concatenates them.

5.3 Open Issues

At this point the DNS service is functional and relieves us from entering IPv6 addresses. But two issues remain unresolved:

- With IPv4 and DHCP, not only address configurations are distributed but the addresses of important servers, most notably for DNS and the network time protocol (NTP). At this point we still need to configure these by hand on all machines in our network. Chapter 19 explains how to provide the information using a simplified DHCP protocol.
- While stateless autoconfiguration is a very powerful mechanism, it lacks an essential feature: It doesn't update the DNS like a DHCP server does. While this problem is still not fully understood, at least some partial solutions exist to simplify DNS administration. They are the subject of chapter 20.

These issues are a nuisance rather than a pressing fundamental problem that we need to address right away, so we defer them for a while.

5.4 Packet Filter Considerations

Packet filtering interacts with the DNS in two independent ways: It controls which machines may query which name server and it possibly *uses* DNS names in its own configuration. Both of these interactions deserve some consideration.

5.4.1 Filter Rules

As with IPv4, DNS uses UDP port 53 for normal lookups. Depending on the DNS implementation, datagrams may be limited to 512 bytes. DNS will automatically fall back to TCP on port 53 if a DNS message gets too large to

fit inside a datagram. As a special case, zone transfers are always done using TCP.

With IPv4 it is common (though questionable) practice to allow only UDP queries through a packet filter. With IPv6, this becomes a problem for two reasons: The IPv6 addresses make the individual resource records larger and there is a tendency to have more DNS records per domain name. In many cases 512 bytes are still more than enough, but if we run an authoritative name server we should either allow TCP or make sure that all our resource record sets are small enough to fit in a UDP datagram. A forwarding server as well as a client should admit both UDP and TCP on port 53.

Otherwise, the packet filter configuration is quite simple and shouldn't pose any problems. From a packet filter's perspective, DNS is just another application protocol running on port 53. If you don't feel comfortable adding filter rules for DNS, take a peek at section 6.11. It includes DNS along with various other services.

5.4.2 DNS Names in Filter Configurations

Writing packet filter configurations that contain rules for individual nodes leads to a common dilemma: Should we put the address or the name of the individual host into the configuration? Neither approach is without its problems.

Using addresses in the configuration makes it necessary to maintain the addresses of the nodes inside the filter configuration. If an address changes and only the DNS is updated, the filter rules break, possibly resulting in a security hole.

Using "true" DNS names in a packet filter is extremely dangerous since DNS isn't particularly secure. In some cases it is feasible to use either macro definitions at the beginning of the filter configuration or make the packet filter only use `/etc/hosts` for its name resolution and put the addresses of the nodes involved there; either approach will noticeably simplify filter configuration. We might even set up DNSSEC to secure our DNS communications.

But with IPv6, nodes tend to change their addresses more often than with IPv4. Packet filters today don't deal with this too gracefully: They usually do a name lookup only when a filter rule is first installed and afterwards never check if the address changes.

There is no easy catch-all solution to this problem. But if we set up a packet filter on a node just to protect the node itself, we may use the features of the particular filter to decide if a packet is for the node itself or not.

Debian Sarge *The* `ip6tables` *framework distinguishes packets that are sent to the filtering node itself and those that are meant to be forwarded.*

FreeBSD 6.1 *With* `pf` *we can use interface names instead of addresses to select packets to be delivered locally.* 33

Packet filters on routers don't have this option. If we want to keep things reasonably simple, our best bet is a network topology that puts nodes with different security requirements in separate subnets, so routers can filter by interface or subnet. If we can't impose such a topology on an existing network, then we have little choice but to track individual addresses in our filter configurations.

6

Essential Network Services

Even with the DNS up and running a number of essential services are still missing. Before we return to IPv6 proper and set up routing, we make ourselves more comfortable by enabling some services that are commonly used for system administration purposes.

You will notice that this chapter doesn't explain in detail how to install, configure and run the various services. It tells you which packages you need but assumes that you know about the services you want to use and only explains their IPv6 specifics.

6.1 Levels of IPv6 Support

There are basically five different levels of IPv6 support with individual programs:

Distribution level The software as distributed with the operating system (including the BSD ports collection, Solaris "companion CD" and similar) supports IPv6. It works out of the box.

Current source level The current source version available supports IPv6; we need to get the sources, build the software and install it by hand.

Current patch kit level A patch for the current source version exists to add IPv6 support.

Outdated patch kit level For an outdated source version a patch set is available.

No IPv6 support At least we could try to add IPv6 support ourselves...

This chapter focuses on the distribution level support. Only in one case will it point out source level support for essential software that isn't generally available from the distribution. Generally, source level support is acceptable but tedious especially if we want the particular packaging system to support these programs.

Patch sets are generally not a solution—not even current ones. If a security problem becomes known and the main source tree is updated, the patch may break. In this situation we have a choice of either disabling the software or running an insecure, outdated version until an updated patch set becomes available. Neither choice is acceptable in a production setup.

Adding IPv6 support to existing code isn't impossible but can be tedious, especially if the original source code is badly written or maintained only in an ad-hoc style.

6.2 The Inetd Super Daemon

The Inetd family of "super daemons", most notably plain `inetd` and its next-generation equivalent `xinetd`, listens on a configured set of ports, accepts incoming connections and then hands the already established connections to the appropriate "real" servers.

Since Inetd passes already established sockets to the "real" servers, some simple services will automatically support IPv6 when they are started this way. As long as they don't try to do anything IPv4-specific to the socket, like asking for the IPv4 address of the client, they won't even notice that they serve an IPv6 connection.

It is not so much the importance of Inetd as a service that makes us address it as the very first service in this chapter. But in the following section we take a look at some basic network administration tools and there is no better service than Inetd to test them with.

Whether or not the Inetd installed on a system already supports IPv6 depends on the particular Unix as usual.

Debian Sarge *The standard* `inetd` *installed from the* `netkit-inetd` *package doesn't support IPv6. We can replace it with either* `inetutils-inetd`*, which behaves just like* `inetd` *but doesn't bring any IPv6-related documentation, or with* `xinetd`*, which uses a different configuration file syntax.*

FreeBSD 6.1 *The* `inetd` *from the core distribution supports IPv6.*

Solaris 10 *With Solaris 10 the* `inetd` *has been incorporated into the service management facility (smf) framework. It is part of a core installation. For testing purposes the* `SUNWcnsr` *and* `SUNWcnsu` *packages contain the* `echo` *service we use to see how to deal with the IPv6 support.* 34

The simple `echo` service is most commonly used to test the Inetd configuration. It is usually implemented internally within Inetd (with the notable exception of Solaris 10) and is available with both TCP and UDP. Depending on the Inetd implementation we use the configuration differs widely:

Debian Sarge, FreeBSD 6.1 with standard inetd *The* `inetd` *configuration in* `/etc/inetd.conf` *consists of a single line per service. The important part is the third column, specifying the protocol to use. Traditionally it contains either* `tcp` *or* `udp` *to specify TCP or UDP sockets.*

With the arrival of IPv6 the various implementations have seriously diverged with respect to this column. The table below shows the available protocols depending on the Unix.

	IPv4 only	IPv6 only	IPv4 and IPv6
Debian Sarge	`tcp4`, `udp4`	`tcp6`, `udp6`	`tcp`, `udp`
FreeBSD 6.1	`tcp`,`tcp4`, `udp`,`udp4`	`tcp6`, `udp6`	`tcp46`, `udp46`

Most annoyingly, `tcp` *and* `udp` *have different meanings on different implementations. Without general support for* `tcp46` *and* `udp46` *there is no way to configure the* `inetd` *daemon in an implementation-independent manner. To enable the* `echo` *service for both TCP and UDP over IPv6 only we need to change the appropriate lines in* `/etc/inetd.conf` *to read*

`/etc/inetd.conf`

```
echo    stream  tcp6    nowait  root    internal
echo    dgram   udp6    nowait  root    internal
```

At least with IPv6 the configuration syntax is consistent between implementations.

Debian Sarge with xinetd *If we want to use* `xinetd` *rather than* `inetd` *we need to change the configuration in* `/etc/xinetd.d/echo` *like this:*

`/etc/xinetd.d/echo`

```
# default: off
# description: An xinetd internal service which
# echo's characters back to clients.
# This is the tcp version.
service echo
{
        disable         = no                            || Set this!
        type            = INTERNAL
        id              = echo-stream
        socket_type     = stream
        protocol        = tcp
        user            = root
        wait            = no
        flags           = IPv6                          || Set this!
}
```

[Continued on next page]

/etc/xinetd.d/echo

[Continued from previous page]

```
# This is the udp version!
service echo
{
        disable         = no                                    || Set this!
        type            = INTERNAL
        id              = echo-dgram
        socket_type     = dgram
        protocol        = udp
        user            = root
        wait            = yes
        flags           = IPv6                                  || Set this!
}
```

On Debian Sarge the `xinetd` *has an annoying limitation: Even though there is another flag* `IPv4` *it is not possible to enable a service for IPv6 only. We may work around this using the* `bind` *statement, but doing so will cause problems when a network renumbering happens. Alternatively, we can use a line*

/etc/xinetd.d/echo

```
[...]
        no_access       = ::ffff:0:0/96
[...]
```

to block the address range that IPv4 addresses are mapped into; any incoming connection will immediately close again—we'll take a closer look at these addresses in section 16.2. While this still isn't a full replacement for the `Connection refused` *error a client will see if the service is entirely unavailable to IPv4, in many cases this is close enough for all practical purposes. Finally, we may block the port at the packet filter level instead. This involves a bit of extra work but is probably the cleanest solution.*

Solaris 10 *Solaris installs the* `echo` *service as two smf-managed services called* `svc:/network/echo:dgram` *and* `svc:/network/echo:stream`*. First we look at the properties of the TCP-based* `echo:stream` *service:*

```
# inetadm -l echo:stream
SCOPE    NAME=VALUE
         name="echo"
         endpoint_type="stream"
         proto="tcp6"                              || This is the important one
         isrpc=FALSE
         wait=FALSE
         exec="/usr/lib/inet/in.echod -s"
         user="root"
```

[Continued on next page]

```
                    [Continued from previous page]
default  bind_addr=""
default  bind_fail_max=-1
default  bind_fail_interval=-1
default  max_con_rate=-1
default  max_copies=-1
default  con_rate_offline=-1
default  failrate_cnt=40
default  failrate_interval=60
default  inherit_env=TRUE
default  tcp_trace=FALSE
default  tcp_wrappers=FALSE
```

(Afterwards we can do the same for `echo:dgram`*.) The important property here is* `proto`*: it controls the protocol families that the service is made available to. The* `proto` *property may have one of these somewhat misleadingly named values:*

`tcp, udp`	Only support IPv4.
`tcp6, udp6`	Support both IPv4 and IPv6.
`tcp6only, udp6only`	Only support IPv6.

To set the property, we use the `inetadm` *command like*

```
# inetadm -m echo:stream proto=tcp6only
# inetadm -m echo:dgram proto=udp6only
```

and finally enable the service:

```
# inetadm -e echo:dgram
# inetadm -e echo:stream
```

After that we may want to check for the service:

```
# inetadm
ENABLED   STATE          FMRI
[...]
enabled   online         svc:/network/echo:dgram
enabled   online         svc:/network/echo:stream
[...]
# svcs -a
STATE          STIME    FMRI
[...]
online         22:38:04 svc:/network/echo:dgram
online         22:38:06 svc:/network/echo:stream
```

6.3 Basic Debugging—Tools and Procedures

With the `echo` service up and running we can now test some basic network administration tools and procedures.

As a first step we use `ps` to see if the server process is actually running. On System V derived Unixen it usually takes the argument `-e` to list all processes while BSD derived ones use the argument `x` or similar.

The second step uses `netstat` to see if a server socket is open on the port we desire. In all cases, `netstat` produces a list of all sockets—with widely differing output formats and a range of non-standardized options to control its output. The standard options `-a` to list all sockets and `-n` to suppress name resolution are commonly supported but not IPv6-specific. In addition we may use the IPv6-related but not generally standardized options listed in table 6.1 to find the information we want.

Table 6.1. Non-standard options to `netstat`

`-l`	Debian Sarge	List only server sockets.
`-A [inet|inet6]`	Debian Sarge	Select the address family.
`-f [inet|inet6]`	FreeBSD 6.1, Solaris 10	Select the address family.
`-4`	Debian Sarge	Short for `-A inet`.
`-6`	Debian Sarge	Short for `-A inet6`.

Debian Sarge *The options `-4` and `-6` can be combined to list both IPv4 and IPv6 sockets together.*

```
# netstat -l -4 -6
Active Internet connections (only servers)
Proto Recv-Q Send-Q Local Address           Foreign Address  State
tcp        0      0 localhost.localdom:smtp *:*              LISTEN
tcp6       0      0 *:echo                  *:*              LISTEN
udp6       0      0 *:echo                  *:*
```

Alternatively we can use the `lsof` command with the options `-i -P -n`:

```
# lsof -i -P -n
COMMAND PID        USER FD TYPE DEVICE SZ. NODE NAME
exim4  1946 Debian-exim 3u IPv4   3645      TCP 127.0.0.1:25 (LISTEN)
xinetd 2042        root 5u IPv6   4000      TCP *:7 (LISTEN)
xinetd 2042        root 6u IPv6   4001      UDP *:7 (LISTEN)
```

The Linux `netstat` and `lsof` commands both show an ugly and potentially dangerous behaviour in certain cases: It is possible for a server to use IPv6 and to have incoming IPv4 connections mapped into the IPv6 address space using so-called IPv4-mapped IPv6 addresses *like `::ffff:127.0.0.1`. A server doing so should show such an open socket as listening on both IPv4 as well as IPv6. All presented Unixen except Linux do so.*

But Linux does *not*. Especially for those of us using IPv4 only, this may well hide the fact that a server is actually listening on IPv4 as well as IPv6 (where we might accidentially ignore it, leaving us with an unnoticed open socket).

To make things worse, the `netcat6` *package contains a netcat reimplementation that doesn't support port scanning, so we can't even use it for a quick port scan as a workaround. If we have* `nmap` *installed we might consider running*

```
# nmap 2001:db8:fedc:abcd::1 -p 1-65535
```

as a last resort.

FreeBSD 6.1 *Besides the standard* `netstat` *command there is also a non-standard* `sockstat` *command which has the great advantage that it doesn't truncate host names or addresses on display.*

```
# netstat -a -f inet6
Active Internet connections (including servers)
Proto Recv-Q Send-Q  Local Address         Foreign Address  (state)
tcp46      0      0  *.echo                *.*              LISTEN
udp46      0      0  *.echo                *.*
# sockstat -l -4 -6
USER   COMMAND  PID   FD PROTO  LOCAL ADDRESS       FOREIGN ADDRESS
root   inetd    1390  4  tcp46  *:7                 *:*
root   inetd    1390  5  udp46  *:7                 *:*
```

Solaris 10 *The output of the Solaris 10* `netstat` *command is distinctly unbefitting a standard text console at 80 column width—it needs 111 characters per line.*

```
# netstat -a -f inet6
UDP: IPv6
   Local Address     Remote Address     State      If
------------------ ------------------ ---------- -----
[...]
      *.echo                          Idle

TCP: IPv6
   Local Address     Remote Address   [...]   State      If
------------------ ------------------ [...] ----------- -----
[...]
      *.echo            *.*           [...] LISTEN
[...]
```

36

Finally it is a good idea to test the server with a client. If there is a matching client, it may have some "verbose mode" to give us more detailed information about connecting to the server.

If no such client is available, we can use a "generic" client software, like `telnet` or one of the various IPv6-enabled clones of netcat, to access a port directly. All the Unixen shown here come with an IPv6-enabled `telnet` client; we can just use an IPv6 address (or a DNS name resolving to an IPv6 address) as the destination and the port number or service name we want.

The advantages of `telnet` are that it is generally available on all systems and all implementations behave the same. Its disadvantages are that it only supports TCP and that the "network virtual terminal" features may cause unexpected effects.

Netcat on the other hand often needs to be installed on the particular system. Its behaviour and feature set depend on what branch/rewrite we have available. But it also supports UDP and in some cases features a simple port scan to list all open ports.

Independent of the particular service a generic checklist looks like this:

- ☐ Check with `ps` that the server process is running.
- ☐ With `netstat`, ensure that a socket is open for the service.
- ☐ With either an application-specific client or `telnet` or an IPv6-enabled Netcat manually connect to the service.
- ☐ Use a packet sniffer to analyze the network traffic in detail, especially if you need to solve a problem or want to get a more detailed understanding of the service.

6.4 The Secure Shell (OpenSSH)

From a system administrator's point of view, the most important network service is probably the secure shell, usually the OpenSSH implementation. It allows secure remote logins and file transfers to other machines using public key cryptography.

In general, there is nothing special to the installation to make it support IPv6; back in 2003 it was in some cases necessary to compile the OpenSSH from sources (available from `ftp://ftp.openssh.org/pub/OpenSSH/`) to ensure that it supported IPv6, but IPv6 support in software has seriously improved since:

Debian Sarge *The* `ssh` *package contains an IPv6-enabled client and server. The installation scripts ask interactively if we want to run the server.*

FreeBSD 6.1 *The OpenSSH is part of the core system and supports IPv6. The server is enabled in* `/etc/rc.conf`*.*

Solaris 10 *The packages* `SUNWsshcu`*,* `SUNWsshdr`*,* `SUNWsshdu`*,* `SUNWsshr` *and* `SUNWsshu` *contain the IPv6-enabled client and server. Once they are installed we enable the service using* `svcadm`*.* 37

Next we may want to configure OpenSSH. We can use IPv6 addresses with the `ListenAddress` option in `/etc/ssh/sshd_config` just like IPv4 addresses. If we don't change the default configuration, then the `sshd` will listen on both IPv4 and IPv6. (If you need to log in as root, don't forget to set `PermitRootLogin` in `/etc/ssh/sshd_config`.)

Next we ensure that the service is actually running.

- ☐ Check with `ps` that the `sshd` daemon is running.
- ☐ See if a socket is listening on the `ssh` port with IPv6. (Remember the Linux problem with `netstat`.)
- ☐ Use `ssh -v` to connect to the server. The `-v` option provides detailed information about the communication progress. The options `-4` and `-6` explicitly request IPv4 and IPv6, respectively.

There is a minor oddity about `scp`. If we want to copy a file to another machine, something like

```
# scp /etc/shadow nobody@bad.example.com:/tmp
```

works perfectly fine. We won't notice if IPv4 or IPv6 is used (and generally we don't want to care about this). But if we want to use an IPv6 address instead of a DNS name, the colon between the address and the file path becomes ambiguous. We need to specify the address in square brackets, like `[::1]`. Since square brackets are used as globbing metacharacters by most if not all shells, we also need to quote them. So what we want to do looks like

```
# scp /etc/shadow nobody@"[2001:db8:bad::bad:bad]":/tmp
# scp /etc/shadow "nobody@[2001:db8:bad::bad:bad]:/tmp"
# scp /etc/shadow 'nobody@[2001:db8:bad::bad:bad]':/tmp
```

or similar. While this isn't too surprising as long as we are aware of this, almost everybody eventually gets some funny errors because of forgotten quote characters.

6.5 Time Synchronization with the Network Time Protocol (NTP)

Especially in data center environments it is essential to ensure that all machines keep their system clocks synchronized to a defined reference. The standard protocol used for this purpose is the *network time protocol* (*NTP*). It is a powerful and complex peer-to-peer protocol.

At this time there are two major implementations: The traditional reference implementation from the University of Delaware and a new implementation from the OpenBSD project. Since the OpenBSD implementation is less mature and not as widely available we only address the reference implementation here. Its stable releases from version 4.2 on support IPv6.

Depending on the Unix we use, NTP support may need to be explicitly installed. On some legacy systems we may even need to build it from the sources available at `http://ntp.isc.org/bin/view/Main/SoftwareDownloads`.

Debian Sarge *We need the packages* `ntp`*,* `ntp-server` *and* `ntpdate`*. In addition, the package* `ntp-doc` *contains the documentation.*

On Linux the `ntpd` *exhibits an annoying problem: If we have multiple IPv6 addresses configured on our interface, then the server may answer requests with the wrong source address. The client process expects the answer from the same address that it sent the request to, so it won't receive the answer. As a workaround we can configure our time server so that it has only a single address besides its link-local address.*

FreeBSD 6.1 *An IPv6-capable version of the NTP reference implementation is part of the core system.*

Solaris 10 *The standard packages* `SUNWntpr` *and* `SUNWntpu` *don't support IPv6 but a current build of the NTP reference implementation is available from* `http://www.sunfreeware.com/`*. It also needs the* `readline`*,* `ncurses` *and* `libgcc` *packages from there, too. In addition, we will need to create an init script or set up our smf configuration, too. (If you write your own init script, make sure that* `LD_LIBRARY_PATH` *includes* `/usr/sfw/lib`*.)* 38

Configuration is entirely independent of the protocol family. We can specify IPv6 addresses in the config files or as command line parameters like IPv4 addresses. But since the configuration file expects netmasks to be given as an IP address with the relevant bits set to one, netmasks as used with the `restrict` statement don't allow the standard slash notation. Instead we need to specify them like

ntp.conf

```
restrict 2001:db8:fedc:: mask ffff:ffff:ffff:: nomodify
```

Probably the most annoying problem with respect to IPv6 is the fact that the documentation hasn't been updated to cover IPv6 yet.

Once we have the `ntpd` daemon running, we can use `ntpdc` or `ntpq` to see if everything works. There are two minor annoyances you may notice:

- The tabular output of various subcommands has been tuned to fit IPv4 addresses on an 80 column display. These may truncate IPv6 addresses.
- Reverse name resolution doesn't always work.

Otherwise, IPv6 support works without problems.

Checking if the service works mostly follows the generic scheme.

☐ Roughly set the clock manually using `date` if necessary.
☐ Start the NTP service.

- ☐ Check with `ps` to see if the `ntpd` is running.
- ☐ Check with `netstat` to see if the NTP port is open.
- ☐ If you synchronize to another server, wait a few minutes to let the clocks get in sync.
- ☐ Use either `ntpdc` or `ntpq` to see if the service works. Personally I prefer to use the `peers` and `listpeers` subcommands in sequence to get both the current state of the connections and the full name or address of the remote peers:

```
# ntpdc -c peers
     remote           local    st poll reach  delay   offset    disp
==============================================================================
=LOCAL(0)        127.0.0.1     13   64    1 0.00000  0.000000 2.81735
=2001:db8:fedc:a ::            11   64    1 0.00476 65217.349 2.81735
=fbsd.example.co ::            16   64    0 0.00000  0.000000 0.00000
# ntpdc -c listpeers
client    LOCAL(0)
client    2001:db8:fedc:abcd:20c:29ff:fe78:478a
client    fbsd.example.com
```

6.6 Event Logging with Syslog

Another important service in data centers is the network-based event logging. The most common setup consists of a `syslogd` gathering log events locally on every machine and forwarding them to a central "log host". The log host either runs `syslogd` or some monitoring software to receive, store and analyze the incoming events.

IPv6 support within `syslogd` strongly depends on the particular Unix.

Debian Sarge *The standard `syslogd` from the `sysklogd` package doesn't support IPv6. Neither does the alternative `syslog-ng`.*

FreeBSD 6.1 *If logging from remote machines is enabled in `/etc/rc.conf`, both the access restriction option `-a` and the bind-to-address option `-b` of `syslogd` support IPv6 addresses. The options `-4` and `-6` restrict network logging to IPv4 and IPv6, respectively.*

Solaris 10 *By default the `syslogd` accepts log events from other machines over both IPv4 and IPv6.* 39

Remote loghosts can be specified in `/etc/syslog.conf` using either DNS names or IPv6 addresses without further problems.

There is a minor oddity with the `syslog` service: It is UDP based and a sender never receives an acknowledgment from the log host. With IPv6 this leads to a problem if multiple DNS entries exist for the log host: The sender can't find out reliably if a packet was successfully delivered. This problem is best addressed using a single `AAAA` record for a dedicated `loghost` host name.

FreeBSD 6.1 *The option* `-A` *makes a log client send a log message to all addresses that the DNS entry of a log host resolves to. Without* `-A` *the packet is only sent to one of the addresses. While this doesn't entirely solve the issue, this is the best obvious solution short of redesigning the underlying protocol.* 40

Checking the `syslog` service again mostly follows the usual three-step test pattern:

- ☐ Check that the `syslogd` is running.
- ☐ Check that the `syslog` port is open.
- ☐ Use `logger` to send log messages of the appropriate priority to the facility you want to test. If you use network monitoring software in your environment, don't forget to tell the colleagues in the operation center about some impending test messages.
- ☐ Search the log files on the destination machine to see if the messages have arrived.

6.7 E-mail: The Simple Mail Transfer Protocol (SMTP)

Many common mail transfer agents (MTAs) support IPv6 out of the box, so setting up an IPv6-enabled mail server is usually quite simple.

Debian Sarge *The standard mail transfer agent is Exim 4.50 from the* `exim4` *meta-package. If we configure Exim as either directly connected to the Internet or using a smart host, it will automatically support IPv6 unless we specify only IPv4 addresses to listen on for incoming connections. (We can do that either during initial installation or with an explicit* `dpkg-reconfigure exim4-config`*.)*

FreeBSD 6.1 *If we enable incoming mail, Sendmail 8.13.6 from the core system will support both IPv4 and IPv6 without further configuration.*

Solaris 10 *Sendmail 8.13.3 from the packages* `SUNWsndmr` *and* `SUNWsndmu` *supports IPv6 out of the box.* 41

Independently of the particular Unix, most mail transfer agents support IPv6. Some of the more common ones are these:

Exim IPv6 support with Exim is quite mature; it has been available for some time now. There is an oddity about IPv6 addresses in Exim configuration files that you may want to know about: IPv6 addresses are written with double colons, so `::1` is written as `::::1`.
IPv6 support is enabled either by not specifying any listen address at all or by specifying the IPv4 and IPv6 addresses to listen to. To listen on all IPv4 addresses, use `0.0.0.0` and to listen on all IPv6 addresses, use `::::0`.

Postfix Since version 2.2 Postfix officially supports IPv6; there have been patch kits for earlier versions but they use a different configuration syntax than 2.2, so upgrading is probably the best thing to do.
To enable IPv6 support we need to set the `inet_protocols` parameter in `main.cf` either to "`all`", "`ipv6`" or "`ipv4,ipv6`"; otherwise Postfix won't listen on IPv6 interfaces.

qmail There have been sporadic IPv6 patches around, but none of them have been merged into the main source tree. Unless you are willing to spend some time making qmail work with IPv6 and to take the risk of being unable to do a quick upgrade when a security hole is found and fixed in the mainstream source, it is a suboptimal choice if you also want IPv6 support.

Sendmail Similar to Exim, Sendmail has supported IPv6 for some time now. Most systems use m4 macro packages to configure it. Unfortunately these macro packages differ, so if you want to get more fine-grained control on how to tweak the configuration, you need to take a closer look at the particular m4 macros you have.

Testing an e-mail setup again follows the usual strategy. The important issue here is access control lists that restrict relaying functionalities.

- ☐ Use `ps` to see if the mail daemon is running.
- ☐ Check if the daemon actually listens on the interfaces you want.
- ☐ Send mails between machines. Make sure to test both incoming and outgoing directions. If the configurations limit relaying to certain internal addresses or address ranges, make sure to test from corresponding source addresses.

6.8 The World Wide Web: HTTP and HTTPS

IPv6 support with the *hypertext transfer protocol* (*HTTP*) and *secure hypertext transfer protocol* (*HTTPS*) is available with some limitations. Web browsers mostly support IPv6 without problems, but the choice of both web servers and proxies is somewhat limited.

The good news is that the *secure socket layer/transport layer security* (*SSL/TLS*) works independently of the IP version—at least as long as we don't use any certificates issued to a given IP address rather than a DNS name.

6.8.1 IPv6 Addresses in URLs

According to RFC 3986 [8, section 3.2.2], IPv6 addresses in *uniform resource locators* (*URLs*) are written in square brackets as we've already seen with `scp`, like

```
http://[2001:db8:fedc:abcd::80]/index.html
```

As with `scp`, don't forget to put all URLs in quotes if you use them on the command line.

DNS names that resolve to IPv6 addresses are treated just like DNS names that resolve to IPv4 addresses.

6.8.2 Web Browsers

Most web browsers support IPv6: Epiphany, Firefox, Konqueror and Mozilla generally support IPv6.

Depending on the operating system, Lynx has partial IPv6 support—it doesn't allow IPv6 addresses in a URL but handles DNS names that resolve to IPv6 addresses. Wget only supports IPv6 on some platforms, apparently from version 1.10 on. Links doesn't support IPv6 at all.

In more detail, IPv6 support with web browsers looks like this:

Debian Sarge *Wget doesn't support IPv6; all other browsers are available and work with IPv6 as stated above.*

FreeBSD 6.1 *Wget (version 1.10.2) does support IPv6 here. All other browsers are available and support IPv6.*

Solaris 10 *Firefox, Epiphany and Links are unavailable with Solaris. Wget and Lynx don't support IPv6.* 42

6.8.3 The Apache Web Server

With respect to web servers the situation is slightly worse. IPv6 support has been considered a major change in the Apache community; as a consequence, only Apache version 2 has IPv6 support in the main source tree. There have been sporadic patch sets available for some versions of Apache 1.*x*, but at least in an untrusted environment they are no real option.

Configuring the base Apache2 doesn't differ from an IPv4 setup. The `Allow from` and `Deny from` directives accept IPv6 prefixes like they accept IPv4. Only the `Listen` directive differs because it traditionally separates the address and port number with a colon: With `Listen` we must put IPv6 addresses in brackets. Using `::` as the address will make Apache2 listen on all IPv6 interfaces. The default behaviour is equivalent to the setting

`httpd.conf`

```
Listen [::]:80
Listen 0.0.0.0:80
```

Debian Sarge *The configuration file layout doesn't follow the usual standards. Files are mostly kept in* `/etc/apache2/conf.d`*, but the* `Listen` *statements belongs in* `/etc/apache2/ports.conf`*.* 43

Dealing with IPv4-mapped IPv6 addresses is still unsatisfying: There is a compile-time `./configure` option `--enable-v4-mapped` which controls the handling of mapped addresses at compile time. Depending on the particular binary distribution this may have been set during compilation or not. If it was set, then we can control IPv4 and IPv6 support independently using multiple `Listen` directives. Otherwise we are stuck as with `xinetd` on page 84. Again, we have three options: If we use explicit `Listen` directives for addresses individually, then we will run into problems during network renumberings. If we use a `Deny from ::ffff:0:0/96` directive, then connections will still be established but an error message delivered to the client. The cleanest but most complex solution again is a packet filter configuration that rejects incoming IPv4 requests before they actually reach the web server.

Debian Sarge, Solaris 10 *The standard Apache2 package has support for IPv4-mapped IPv6 addresses enabled.*

FreeBSD 6.1 *Contrary to the Apache documentation, FreeBSD as of version 6.1 has also been built with IPv4-mapped IPv6 addresses enabled.* [44]

Testing follows the familiar pattern.

- ☐ Check that there are `httpd` processes running.
- ☐ Check that they listen on port 80 on the interfaces you want.
- ☐ Use your preferred web browser to connect to them.

As far as modules and *CGI* (*common gateway interface*) scripts are concerned, you'll need to figure out if they support IPv6 yourself. Since Apache deals with most network related issues chances are good that your preferred modules and CGI scripts will support IPv6 as well, but I personally recommend strongly to do some thorough testing before you put them into production.

6.8.4 Web Proxies

Unfortunately the most popular web proxy, Squid, doesn't officially support IPv6. There have been patches available for some versions and people have worked on IPv6 support in a development branch, but no IPv6-enabled release has become available yet. So for now, Squid is barely an option.

Some alternatives exist. If we have an Apache2 up and running, we can use its proxy and possibly cache modules. If we need a proxy as a firewall application level gateway, we have a choice of alternate proxies. My personal favourite here is called `ffproxy`. It does neither FTP proxying nor caching but provides extended filtering features and seems to be secure enough to run in a firewall setup without major worries.

To configure Apache2 as a proxy we need to build it with the proxy and optionally the cache module.

Debian Sarge, Solaris 10 *The proxy module works out of the box.*

FreeBSD 6.1 *The binary package is compiled without these modules. We need to re-build the Apache2 from the ports tree with the additional flag* `WITH_PROXY_MODULES` *and possibly with* `WITH_CACHE_MODULES`*.* 45

To enable the proxy module and restrict access to our own network, we add lines like

httpd.conf

```
ProxyRequests On
<Proxy *>
  Order deny,allow
  Allow from 2001:db8:fedc::/48
</Proxy>
ProxyVia On
```

to `httpd.conf`. Restricting access to the proxy isn't mandatory, but running an open proxy is a security risk similar to an open mail relay.

Debian Sarge *Since the configuration is scattered over multiple files in the* `/etc/apache2` *directory, we must copy* `proxy.load` *and* `proxy.conf` *from* `mods-available` *to* `mods-enabled` *and then adjust the proxy configuration in* `mods-enabled/proxy.conf`*.* 46

The lightweight HTTP/HTTPS-only `ffproxy` daemon is available from `http://ffproxy.sourceforge.net/` as source code; the current version is 1.6. It generally compiles and installs cleanly without problems.

FreeBSD 6.1 *The ports/packages collection contains an* `ffproxy` *package so we don't need to compile it ourselves.* 47

As usual we also need to set up our system to start the `ffproxy` command when the system boots, using whatever mechanism the particular Unix provides.

To configure `ffproxy` we first change the line `unrestricted_connect no` to `yes`; otherwise `ffproxy` will only proxy HTTPS requests. In addition we should set `daemonize` to `yes` to daemonize the process after start. Next we need to configure the filters. For simplicity sake, we add a line

/usr/local/share/ffproxy/db/access.ip

```
^2001:db8:fedc:.*
```

to `/usr/local/share/ffproxy/db/access.ip` to allow clients from our own networks to use the proxy. Be careful, this file contains regular expressions rather than address prefixes. After that we start the proxy. Testing again follows the usual pattern:

- ☐ Check with `ps` that `ffproxy` is up and running.
- ☐ Use `netstat` to ensure that it listens (by default on port 8080).

- ☐ My preferred client to test a proxy is `lynx` because it lets me specify the proxy in an environment variable, like

  ```
  # http_proxy=http://proxy:8080/ lynx http://www.example.com/
  ```

  Especially if you want to test access restrictions, make sure you run a client on a machine with a matching IP address to see if access is denied as desired.

6.9 The Network File System (NFS)

Despite its inherent insecurity, the network file system (NFS) is still widely used. It is based on the *remote procedure call* (*RPC*) service, which doesn't support IPv6 on all platforms yet.

Debian Sarge *Linux in general doesn't support RPC over IPv6 yet. This implies that NFS over IPv6 isn't available so far.*

FreeBSD 6.1, Solaris 10 *The portmapper (in both cases called `rpcbind`) supports IPv6 as well as IPv4.* [48]

Besides an IPv6 capable portmapper the only relevant IPv6 issue with NFS are access controls.

FreeBSD 6.1 *In `/etc/exports` we can restrict access to address prefixes like*

/etc/exports

```
/export -network 2001:db8:fedc::/48
```

as well as the customary DNS names.

Solaris 10 *There is no IPv6 equivalent of the usual `rw=192.0.2.0/24` syntax in `/etc/dfs/dfstab`; it is only possible to restrict access by DNS names and domains. For example*

/etc/dfs/dfstab

```
share -F nfs -o rw=-bad.example.com:.example.com /export
```

will grant all machines from the `example.com` domain with the exception of `bad.example.com` read/write access to `/export`.

At first glance the security implications of using a DNS based "authentication" are scary. But considering how insecure NFS generally is, the additional insecurity doesn't really matter that much in most cases. [49]

There is one final bit of information that doesn't directly relate to IPv6 but may be important when testing between different operating systems: Some NFS servers will refuse connections from clients that use a high port number on their side.

FreeBSD 6.1 *FreeBSD servers expect a client to connect from a low port. To change that,* `mountd` *must be started with an additional* `-n` *option. FreeBSD clients will use a low port.*

Solaris 10 *Solaris servers don't care about the port that a client connects from. A Solaris client will use a high port number to connect to a server.* 50

Testing follows the usual pattern but requires some different commands.

- ☐ Check that the NFS-related processes are up and running.
- ☐ Verify that they are listening on the appropriate sockets. Since RPC doesn't bind to fixed port numbers, `netstat` is not too useful here. Use `rpcinfo -s` or `rpcinfo -p` instead. If you want to use a packet sniffer to debug a problem, `rpcinfo` will provide you with the port numbers to watch for.
- ☐ Check the access configuration. Use `showmount -e` here.

  Solaris 10 *Invoking* `share` *without options provides even more information.* 51
- ☐ Try to mount the remote file systems from a client.

6.10 Other Services

A wide range of other applications readily support IPv6. Sometimes it is difficult to tell if they do, or to find an IPv6-enabled implementation of a given protocol.

If we want to know if a given software package supports IPv6, then we should obviously search its documentation. Sometimes we may find an old announcement that a long since released version will be IPv6-enabled, only to find no hint of IPv6 support with this version. Only close scrutiny of the change log shows that IPv6 support has actually been added. This phenomenon has a simple reason: Making software IPv6-capable is usually no big deal, but developers who are not yet familiar with IPv6 don't realize this and expect it to be a huge effort. So there is a spectacular announcement that IPv6 support will be added in the next release when the developers don't yet know what to expect and a minor note in the change log when they have added IPv6 support.

If we don't find anything about IPv6 in the documentation, then it is sometimes helpful to just run the program and use `netstat` to see if it opens an IPv6 socket. This isn't the most reliable strategy, but it is especially helpful if the documentation hasn't been updated yet.

Finally, if we have access to the sources, grepping for the string `INET6` often gives a clue if IPv6 support is at least under way. This is even less reliable than using `netstat` and often time-consuming, but if we really need to know, then it may be worth the effort.

If we don't want to know about a single application or software package, but look for a software that offers a given service or functionality, then it is quite tedious to check all possible candidates this way. The Deepspace6 project at `http://www.deepspace6.net/` offers an extensive overview of the IPv6 support in various software projects, especially with respect to Linux and, to a lesser degree, the BSDs. It is the ultimate resource when looking for a specific service, like an IPv6-enabled NNTP or IMAP server. Occasionally the information there may be slightly outdated, usually claiming that a software isn't IPv6-enabled yet even though its latest release is; please send them an update note if you find such a case.

6.11 Packet Filter Considerations

Since we have already done most of the hard work on packet filter configuration in section 4.7, adding filter rules for services is quite straightforward. We just allow TCP and UDP traffic based on the source and destination port numbers.

Client and server configurations differ noticeably. A client will only allow outgoing connections. A server will always allow incoming connections and in some cases, like web proxies or mail relays, outgoing connections as well. With TCP, we can use the SYN and ACK flags to decide if a connection is to be established. UDP doesn't have flags like that, so we either need to rely on the connection tracking of our packet filter or allow traffic either way. In short, the only major difference between IPv4 and IPv6 packet filter configurations at this point is the limited functionality some packet filters provide, especially as far as stateful filtering is concerned.

Two protocols deserve particular attention: Since RPC and RPC-based services don't run on fixed port numbers, they can't be filtered at the packet level. And since the `syslog` service only sends datagrams from a client to a server, it doesn't really create any state we can use for our filtering purposes.

We can extend the filter configuration in section 4.7 without much trouble to support DNS and the protocols introduced in this chapter.

6.11.1 TCP Services

For "demonstration purposes" we configure our packet filters from section 4.7 to allow TCP traffic on port 22, for the secure shell. We can configure other ports accordingly simply by changing the port number.

Debian Sarge *So far, all TCP packets that made it through the sanitation rules will be discarded by the default rule. To allow TCP connections on port 22 we must filter by the combination of SYN and ACK flags.*

For incoming connections we can use filter rules like

/etc/ip6tables.sh

```
accept INPUT -p tcp --dport 22 --syn
accept INPUT -p tcp --dport 22 --tcp-flags SYN,ACK ACK
```

The first rule allows SYN packets to initiate a connection while the second allows already-established connections (with ACK set but SYN not) to proceed.

Similarly, we can allow outgoing connections with a rule

/etc/ip6tables.sh

```
accept INPUT -p tcp --sport 22 --tcp-flags ACK ACK
```

since all but the first packet (that will be allowed out without extra filtering) will have the ACK flag set.

In both cases we use the fact that `ip6tables` *uses a separate* `FORWARD` *chain for packets that have non-local addresses as both the source and destination. We can simply append the rules to our existing filter configuration script.*

FreeBSD 6.1 *With the stateful filtering features of* `pf` *we only need to filter the initial SYN packet to establish state for the connection.*

For incoming connections we use a rule

/etc/pf.conf

```
pass in quick inet6 proto tcp from any to {$ifs} port 22
                              flags S/SA keep state
```

and for outgoing connections analogously

/etc/pf.conf

```
pass out quick inet6 proto tcp from {$ifs} to any port 22
                               flags S/SA keep state
```

In either case we need to place these lines before the catch-all TCP block rule from section 4.7.5. 52

6.11.2 UDP Services

UDP configuration is similar but can't use the SYN and ACK flags of TCP. We use DNS on port 53 in the configuration example.

Debian Sarge *On a DNS server we need the filter rule*

/etc/ip6tables.sh

```
accept INPUT -p udp --dport 53
```

and on a client the rule

/etc/ip6tables.sh

```
accept INPUT -p udp --sport 53
```

again using the fact that we don't filter outgoing packets. As with TCP, we can simply append these rules to the end of our filter configuration.

FreeBSD 6.1 *Since* `pf` *supports stateful filtering for UDP as well as TCP, the filter rules look quite similar to those for TCP. On a server we need the rule*

/etc/pf.conf

```
pass in quick inet6 proto udp from any to {$ifs} port 53 keep state
```

and on a client the rule

/etc/pf.conf

```
pass out quick inet6 proto udp from {$ifs} to any port 53 keep state
```

As with TCP we must put these rules in front of the catch-all block rule for UDP. 53

6.11.3 Performance Tuning

So far we have blissfully ignored performance issues with our filter rules. There are two simple tricks that are generally useful: We can use entire lists of ports that are treated similarly and we can arrange the rules in an order that minimizes the number of rule evaluations.

Using entire lists of ports in a single filter rule requires a packet filter implementation that can handle such lists. How this is done again depends on the filter implementation.

Debian Sarge *There is a filter module called* `multiport` *that supports lists of up to 15 ports in a single rule.*

FreeBSD 6.1 *The braces notation we use for lists of addresses or interfaces also works with port numbers. It will expand to multiple rules, so it doesn't improve performance but only readability and conciseness of our configuration files. Additionally,* `pf` *supports port ranges, so if a contiguous range of ports is treated the same, then using these ranges can improve performance.* 54

Short of analyzing a packet filter's rule statistics, there are some very simple ways to improve filter performance with respect to the arrangement of filter rules.

With last-match semantics filters, using a "quick" option reduces the number of rule evaluations on virtually all packets.

Filtering TCP before UDP before ICMPv6 usually improves performance because most traffic tends to be TCP.

Using some sort of non-linear rules, like multiple chains with `ip6tables` or anchors with `pf`, can quite drastically reduce the number of rules applied to every packet.

The price for many of these tuning tricks is that most filter rule sets become more difficult to understand and consequently more error-prone. So don't waste your time tuning a packet filter unless you know that the filter has a performance problem and you are willing to take the extra risk of making a mistake.

7

Unicast Routing Basics

There is one feature missing to make our IPv6 setup complete: Routing between multiple subnets. In this chapter we set up routing for unicast packets.

IPv6 routing differs from IPv4 routing in a number of respects. *Classless inter-domain routing* (*CIDR*) was retrofitted to IPv4 when it became obvious that class-based routing was becoming infeasible. IPv6 never had a notion of class-based routing. Similarly, *variable length subnet masks* (*VLSM*) have been added to IPv4 to make better use of the available address space. IPv6 uses a fixed subnet prefix length; this simplifies routing configuration quite noticeably. With IPv4, hosts may find a default router either through DHCP or *ICMP router discovery* as defined in RFC 1256 [37]. IPv6 uses stateless autoconfiguration to discover all routers in a subnet instead.

7.1 Hosts and ICMPv6 Redirects

But how exactly does this work when multiple routers provide different routes? How does a host figure out which router to send a packet to?

Consider the network in figure 7.1. If host 1 wants to send a packet to host 2 or host 3, it will send it to router 1 to take care of the forwarding. As a router, router 1 will know how to deliver the packet either to host 2 or router 2, which in turn delivers it to host 3.

But what happens when host 2 wants to send a packet to host 3? It only has a list of default routers that contains both router 1 and router 2, so there is a 50% chance that it will send the packet to router 1 instead of router 2.

Router 1 knows that it should send the packet to router 2 to forward it to host 3 and it will do so. Router 1 also knows that host 2 and router 2 are connected to the same subnet and therefore future packets from host 2 to host 3 should be sent from host 2 directly to router 2. So it sends an *ICMPv6*

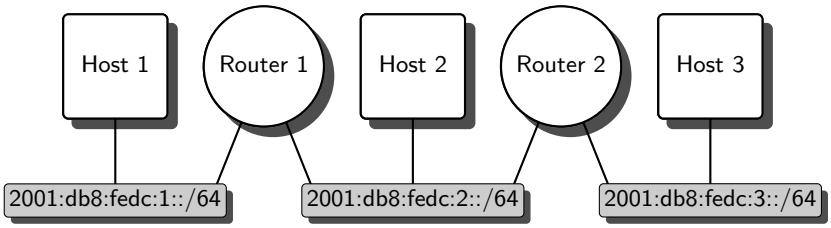

Fig. 7.1. Autoconfiguration and ICMPv6 redirects

redirect message to host 2, telling it to send all future packets to host 3 via router 2.

When host 2 receives the redirect, it stores it in its *destination cache*. The destination cache is effectively a routing table that gathers and updates host routes through redirect messages—in fact, depending on the implementation the destination cache *is* the routing table and the routes discovered are stored there and marked as dynamic. Neighbor unreachability detection ensures that outdated entries don't cause any harm: When the router associated with an entry becomes unavailable, the entry is discarded.

Stateless autoconfiguration and ICMPv6 redirects together provide redundant routing services to hosts without running a dynamic routing service on the hosts involved.

Somewhat surprisingly, there are no timeouts defined for the destination cache entries: A host will discard entries if the cache grows too large, but since stale entries are detected by neighbor unreachability detection it doesn't make sense to discard entries based on a timeout.

7.2 Inside IPv6: ICMPv6 Redirect Protocol Details

ICMPv6 redirects provide a mechanism that lets a router notify a host about a better route to a destination. In our scenario above we have discussed the case that another router is closer to the destination than the one sending the redirect. Additionally, ICMPv6 redirects may be used to notify a host that a destination is "on-link", or directly connected to the same subnet. In a "clean" network these redirects shouldn't occur, but if you have multiple network prefixes configured on a subnet and different hosts for some reason have inconsistent prefix configurations, then these redirects are useful.

Only hosts accept ICMPv6 redirects; routers are explicitly forbidden to listen to them. Otherwise routers could be easily manipulated to forward packets to an attacker instead of the real destination. And if the router accepted ICMPv6 redirects and subsequently sent similar ICMPv6 redirects itself, then a single bad ICMPv6 redirect could result in a self-perpetuating broken routing configuration.

Figure 7.2 shows how redirects are sent. Host 2 from the example sends a packet via router 1 which then decides from its routing table that router 2 was closer to host 3 than router 1. Router 1 still forwards the packet to router 2. Then it sends an ICMPv6 redirect to the host to notify it about the better route via router 2. The exact behaviour is defined in RFC 2461 [91, section 8].

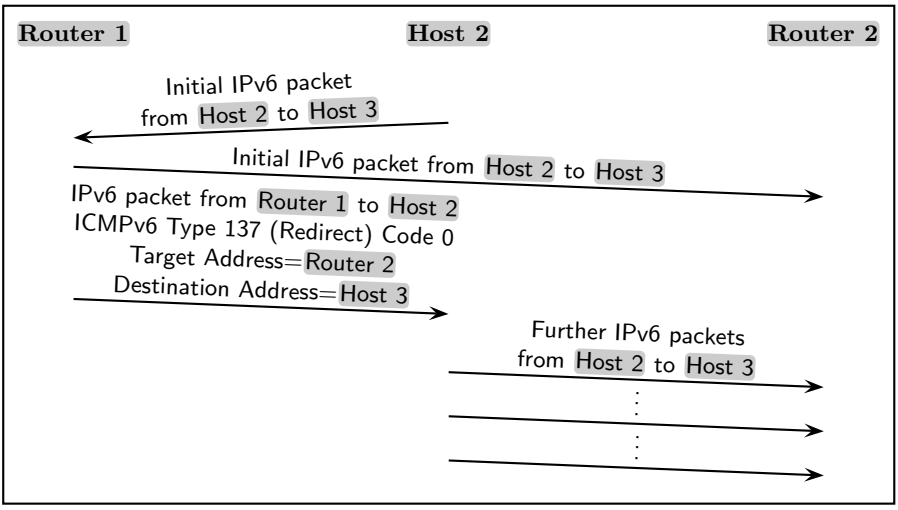

Fig. 7.2. An ICMPv6 redirect in action

The exact format of an ICMPv6 redirect packet is defined in RFC 2461 [91, section 4.5].

The IPv6 header of a redirect must have a link-local address as its source address and a hop limit of 255 when it arrives at the host; otherwise the host must discard the packet. If the hop limit was less, then the ICMPv6 redirect itself may have passed through a router, which doesn't make sense—unless somebody outside tries to send spoofed redirects.

The ICMPv6 packet proper contains these fields:

ICMPv6 type is always 137 (ICMPv6 redirect).

ICMPv6 code is always 0.

Target address is the link-local IPv6 address of the next hop router (router 2) or the destination address if the destination node is on-link.

Destination address is the IPv6 address of the destination.

Target link-layer address Optionally, the packet may contain the link-layer address of the target.

Redirected header The packet must also contain the beginning of the packet that triggered the redirect. This field is sent as an option but must be included in the packet.

To avoid excessive network load, ICMPv6 redirects won't be sent for every ill-routed packet; a host that ignores these redirects could otherwise cause significant network load. RFC 2461 [91, section 8.2] explicitly states that a router must limit the number of redirects it sends. There is no exact specification on *how* to define an acceptable limit, but when we test ICMPv6 redirects we can't expect to see as many redirects as we have sent ill-routed packets.

7.3 Static Routing

As far as routers are concerned, routing hasn't changed much with IPv6. With IPv4, it is easiest to manually set up static routes on all routers. With IPv6, we can do the same.

Assume that we want to add four static routes to a router:

Address/Prefix	Next-hop Router
2001:db8:fedc:aaaa:0ff:0ff:0ff:0ff	2001:db8:fedc:1::1
2001:db8:fedc:bbbb::/64	2001:db8:fedc:1::2
2001:db8::/32	2001:db8:fedc:1::3
default	2001:db8:fedc:1::4

How these routes are configured depends on the Unix in question.

Debian Sarge *The commands* `netstat -rnA inet6`, `route -A inet6` *and* `ip -6 route show` *all display the routing table. We can add routes temporarily like this:*

```
# route -A inet6 add 2001:db8:fedc:aaaa:0ff:0ff:0ff:0ff \
                     gw 2001:db8:fedc:1::1
# route -A inet6 add 2001:db8:fedc:bbbb::/64 gw 2001:db8:fedc:1::2
# route -A inet6 add 2001:db8::/32 gw 2001:db8:fedc:1::3
# route -A inet6 add default gw 2001:db8:fedc:1::4
```

The `route` *command assumes a host route by default, so even for an individual subnet it is necessary to provide the prefix length. Alternatively, we can use* `ip`*:*

```
# ip -6 route add 2001:db8:fedc:aaaa:0ff:0ff:0ff:0ff\
                  via 2001:db8:fedc:1::1
# ip -6 route add 2001:db8:fedc:bbbb::/64 via 2001:db8:fedc:1::2
# ip -6 route add 2001:db8::/16 via 2001:db8:fedc:1::3
# ip -6 route add default via 2001:db8:fedc:1::4
```

If we don't specify a prefix, `ip` *also assumes a host route. To delete a route with both commands we substitute* `add` *with* `del`*.*

If we want to configure the routes permanently, we need to add them to `/etc/network/interfaces`*. Since its syntax only supports the configuration of a default route, we need to add the other routes using* `up` *and* `down` *statements:*

/etc/network/interfaces

```
iface eth0 inet6 static
    address 2001:db8:fedc:1::2
    netmask 64
    gateway 2001:db8:fedc:1::4
    up   /sbin/ip -6 route add 2001:db8:fedc:aaaa:0ff:0ff:0ff:0ff \
                           via 2001:db8:fedc:1::1
    down /sbin/ip -6 route del 2001:db8:fedc:aaaa:0ff:0ff:0ff:0ff \
                           via 2001:db8:fedc:1::1
    up   /sbin/ip -6 route add 2001:db8:fedc:bbbb::/64 \
                           via 2001:db8:fedc:1::2
    down /sbin/ip -6 route del 2001:db8:fedc:bbbb::/64 \
                           via 2001:db8:fedc:1::2
    up   /sbin/ip -6 route add 2001:db8::/16 \
                           via 2001:db8:fedc:1::3
    down /sbin/ip -6 route del 2001:db8::/16 \
                           via 2001:db8:fedc:1::3
```

FreeBSD 6.1 *To display the routing table we use* `netstat -rnf inet6`*. Routes are added with* `route` *using this syntax:*

```
# route add -inet6 2001:db8:fedc:aaaa:0ff:0ff:0ff:0ff \
                   2001:db8:fedc:1::1
# route add -inet6 -net 2001:db8:fedc:bbbb:: 2001:db8:fedc:1::2
# route add -inet6 2001:db8:fedc:bbbb::/64 2001:db8:fedc:1::2
# route add -inet6 2001:db8:/32 2001:db8:fedc:1::3
# route add -inet6 default 2001:db8:fedc:1::4
```

If an address is given, `route` *assumes a host route unless the option* `-net` *is also specified; in this case, it assumes a* `/64` *prefix. So the second and third* `route` *invocations are equivalent. To delete a route, we replace* `add` *with* `delete` *and drop the next-hop router:*

```
# route delete -inet6 2001:db8:fedc:aaaa:0ff:0ff:0ff:0ff
# route delete -inet6 -net 2001:db8:fedc:bbbb::
# route delete -inet6 2001:db8:fedc:bbbb::/64
# route delete -inet6 2001:db8:/32
# route delete -inet6 default
```

We can make the configuration permanent in `/etc/rc.conf`*:*

/etc/rc.conf

```
ipv6_static_routes="route1 route2 route3 default"
ipv6_route_route1="2001:db8:fedc:aaaa:0ff:0ff:0ff:0ff
                   2001:db8:fedc:1::1"
ipv6_route_route2="2001:db8:fedc:bbbb::/64 2001:db8:fedc:1::2"
ipv6_route_route3="2001:db8::/32 2001:db8:fedc:1::3"
ipv6_route_default="default 2001:db8:fedc:1::4"
```

The first line defines a list of names associated with the routes. The following lines contain the arguments to `route` *for each such route.*

Solaris 10 *As with FreeBSD,* `netstat -rnf inet6` *displays the current routing table.*

To add routes, we use the `route` *command:*

```
# route add -inet6 2001:db8:fedc:aaaa:0ff:0ff:0ff:0ff \
                   2001:db8:fedc:1::1
# route add -inet6 2001:db8:fedc:bbbb::/64 2001:db8:fedc:1::2
# route add -inet6 2001:db8:/32 2001:db8:fedc:1::3
# route add -inet6 default 2001:db8:fedc:1::4
```

Different than FreeBSD, the `-net` *and* `-host` *options are parsed but otherwise ignored; we need to specify an explicit prefix length for network routes. Otherwise a* `/128` *prefix length is assumed.*

Routes are deleted using the same syntax but with `delete` *instead of* `add`*; we need to specify the next-hop router even when we remove an existing route.*

Solaris doesn't offer a standardized way to configure static routes permanently; we are expected to use dynamic routing instead. While IPv4 allows us to specify static routes in `/etc/gateways` *when we run dynamic routing, IPv6 doesn't even offer this feature. This leaves us in an awkward situation. If you are really desperate, you might want to extend the smf configuration to provide for static routes; fortunately, a simple but usable solution for both IPv4 and IPv6 has been made available at* `http://www.visi.com/%7Ecpj1/code/static-routes.tar.gz`*. While this is not officially supported by SUN, it may do the trick for your purposes.* [55]

7.4 Dynamic Routing with RIPng

Managing static routing tables tends to be tedious and error-prone, especially since IPv6 addresses are so easy to mistype. With IPv4 the *routing information protocol* (*RIP*) as of RFCs 1058 [60] and 1388 [84] provides us with a means to set up routing tables automatically within a so-called *autonomous system* (*AS*). In RFC 2080 [85] RIP has been modified to support IPv6 addresses, resulting in a routing protocol called *RIPng*.

RIPng has inherited all the limitations of RIP: It supports a maximum of 15 hops between end points; it computes the routes based on hop counts only, ignoring latency, bandwidth or communications costs; and it is slow to discover and cope with failing routers.

But it also inherits all the advantages of RIP: It is generally available, easy to use and needs little resources from the network and routers involved.

If the network topology is reasonably "clean" and the failover requirements are anywhere near "faster than I can change the routing myself", RIPng is

quite useful. Otherwise, OSPF might be the preferred alternative, so we'll take a look at it in chapter 17.

With the exception of most Linux distributions, Unixen normally bring a RIPng daemon with their core system. In consequence, installation and configuration are usually trivial.

Debian Sarge *First we need to install a routing daemon. Since there is no lightweight RIPng-only routing daemon available the best choice is Quagga, a heavyweight but widely used dynamic routing software.*

To enable quagga we first edit `/etc/quagga/daemons` *and modify the lines*

`/etc/quagga/daemons`

```
zebra=yes
ripngd=yes
```

This Debian-specific configuration determines what daemons to run. "Zebra", which is also the name of the predecessor of Quagga, is a daemon that provides a protocol-independent dynamic routing framework, so we need it. Additionally we need support for RIPng, too.

Next we configure the `zebra` *daemon with an empty configuration file* `/etc/quagga/zebra.conf` *and* `ripngd` *with the lines*

`/etc/quagga/ripngd.conf`

```
router ripng
network ::/0
```

The first line enables the RIPng protocol and the second makes the daemon provide announcements on all interfaces. Finally we can start the routing daemons with `/etc/init.d/quagga start`*.*

FreeBSD 6.1 *First we add a line*

`/etc/rc.conf`

```
ipv6_router_enable=YES
```

to `/etc/rc.conf`*. Then we run* `/etc/rc.d/route6d start` *to start the* `route6d` *daemon or reboot the router.*

Solaris 10 *We enable the dynamic routing daemon* `in.ripngd` *with the* `routeadm` *command:*

```
# routeadm -e ipv6-routing
# routeadm -u
```

The first line enables dynamic routing in the boot configuration, the second applies the boot configuration to the running system and saves us a reboot.

7.5 Testing and Debugging

To debug our routing configuration we need to understand that routing isn't inherently symmetric: It is well possible that a packet can be sent from machine A to machine B but not from B to A. And even if both directions work, the packets may travel along different paths. If routing doesn't work, the best bet is to use a tool like `ping6` to generate a stream of packets and a packet sniffer to watch the packets on all routers along the way.

With RIPng it is also important to understand about the timing involved. Every 15–45 seconds a RIPng router will announce its routing table to its peers. The peers check if the announcement contains routes that are shorter than routes they currently use; if so, they update their routing table accordingly. If the router doesn't send an announcement for 180 seconds, its peers assume the router died and drop the routes through it. As soon as they learn about an alternate route they will use this new route.

Depending on the network topology it may take a network several minutes to recover from a router failure. During this time, routing may seem to behave erratically, with `traceroute6` appearing to lose packets at some intermediate router but still reaching the end (because the back routing doesn't work from the intermediate routers) and similar effects. During this time, wait for the network to recover.

To simplify debugging it is often useful to have the routing daemons log all routing table changes:

Debian Sarge/Quagga *We add the lines*

`/etc/quagga/ripng.conf`

```
log file /var/log/ripng.log
debug ripng events
```

to `/etc/quagga/ripng.conf`*. Then we create the file* `/var/log/ripng.log` *and make the* `quagga` *user its owner. Finally, we restart the Quagga daemons.*

FreeBSD 6.1 *The routing daemon needs to be started with the additional option* `-R /var/log/ripng.log`*. We can adapt* `/etc/rc.conf` *accordingly:*

`/etc/rc.conf`

```
ipv6_router_flags="-R /var/log/ripng.log"
```

The `-R` *option makes* `route6d` *log its actions in the specified file. Afterwards we must either restart the routing daemon again or reboot the router.*

Solaris 10 *We must run the commands*

```
# routeadm -s ipv6-routing-daemon-args="-s -v /var/log/ripng.log"
# routeadm -u
```

to enable logging. The `-s` *option to the routing daemon is a default flag that according to the documentation makes even a "single-legged" router send out*

routing information; `-v` *enables event logging and the final argument specifies the log file.* 57

If you want to keep your routing daemons logging all the time, don't forget to rotate the log files occasionally. Otherwise, turn the logging off when you don't need it anymore.

To ensure that the network is working properly, we check it bottom-up again.

- ☐ With `ifconfig`, verify that all interfaces are up, running and configured as intended.
- ☐ If you use static routes, check that the routing tables on all routers show the correct configuration.
- ☐ If you use dynamic routing, check that the routing daemons are running and listening on port 521/UDP.
- ☐ With `ping6` or `traceroute6` check that routing works as expected. Depending on your Unix, use the `-n` option to suppress DNS lookups; otherwise a problem reaching the name server might actually appear like a routing failure towards a completely different destination.
  If you use dynamic routing and run into problems here, check the routing daemon logs and watch the RIPng traffic with a packet sniffer.
- ☐ With dynamic routing, try to bring in a new router. Do its routes propagate? Will shorter routes become known to the other routers?
- ☐ With dynamic routing, bring a router down. If it isn't redundant, do the other routers drop all routes through it? If it is redundant, will they change their routing to a working router?

Again, remember that RIPng doesn't converge too fast. Don't get uneasy, first spend a few minutes watching the routers involved. You should see them converge towards a working routing configuration.

7.6 Inside IPv6: RIPng Protocol Details

RIPng is specified in RFC 2080 [85]. It uses a simple metric of integer numbers from 0 to 15 to measure the "length" of a route. By default, a router will add one to the metric before passing the route on, thus limiting support to networks with a *network diameter*, the maximum number of hops between any two nodes, of 15 hops at most.

There are two types of RIPng messages: Requests and responses. A request may either ask for individual routes or full routing information. A response to a request will only provide the information requested. An unsolicited response announces the entire routing table (but see below on the "split horizon" and "poisoned reverse" features).

RIPng routing daemons listen on the multicast address `ff02::9` as well as their unicast addresses for UDP datagrams on port 521. They will accept

requests from and send solicited responses to non-local addresses for monitoring and debugging purposes. They send unsolicited responses to the multicast group from the link-local addresses of the interfaces only and they only update their routing tables if the message was received that way. When a routing daemon receives an unsolicited response, it updates its routing table if the response informs the routing daemon of a better route. Every time a routing daemon receives a response, it resets the expiration timer of the routes received to 180 seconds.

Routers send *unsolicited responses* with the full routing information every 15–45 seconds; randomization prevents synchronization effects that periodically overload the network.

When a route changes, the routing daemon sends a *triggered update* to all interfaces but the one that the route refers to. This update is delayed a randomized interval ranging from 1 to 5 seconds to prevent excessive network load in certain situations.

To optimize the protocol, the *split horizon algorithm* first eliminates all entries that refer to the interface that the response is sent to; after all, passing data to these destinations shouldn't touch the router sending the response. To speed up convergence, the *poisoned reverse* optimization instead transmits these routes, but with a metric of 16, marking them as unreachable, so they can be removed on the receiving routers before the route's lifetime expires. Both of these optimizations may be explicitly disabled if necessary.

7.7 Routing Architecture Strategies

Generally, IPv4 and IPv6 don't differ too much with regard to routing. But with stateless autoconfiguration a number of things get easier with respect to dynamic routing, so re-thinking an IPv4-oriented routing strategy may be worthwhile.

7.7.1 Basic Considerations

There are some fundamental considerations you probably want to take into account when you manage a routing architecture:

Keep routing symmetric if at all possible. While IP supports asymmetric routing, dealing with it is difficult and tedious. It will make `traceroute6` less useful because you may not receive replies from all intermediate routers and you will need to deal with twice as many routes between any two nodes because you have to consider both directions individually.

Strive for a clean, hierarchical network topology so convergence after a router failure is quick. Avoid "shortcut" links between networks that circumvent the general routing architecture. Finally, keep your network diameter small.

Dynamic routing won't compensate for a messy network topology but will make things worse because convergence becomes excessively slow.

Keep your routing skills up so you can anticipate problems before they occur and quickly deal with them when they actually happen.
Again, dynamic routing doesn't magically substitute an inexperienced network administrator.

Document and understand your network and keep the documentation handy. No matter how experienced you are, if your network has a problem you won't have time to figure out how it is actually set up.

None of these aspects are specific to IPv6, but if you want to use dynamic routing, they are a prerequisite to smooth operations.

7.7.2 Static or Dynamic Routing?

Choosing between static and dynamic routing is more than a dogmatic issue. If you are unfamiliar with dynamic routing concepts and have a "historically grown" network topology that nobody understands anymore, static routing is the way to go. But otherwise, dynamic routing is probably preferable.

Dynamic routing lets us set up redundant networks with minimum hassle as we'll see in the next section. It also simplifies router configuration as we've already seen.

With stateless autoconfiguration and ICMPv6 redirects, hosts already support dynamic routing in a way, even without a local routing daemon that passively listens for route announcements and maintains the routing table. With IPv4 this used to be the reason why dynamic routing was often not used to its full extent: Setting up all hosts to deal with dynamic routing tended to be tedious at best and all too often caused network problems by itself. IPv6 makes dynamic routing feasible in many border cases, so once you have a basic environment with static routing up and running you may want to consider switching to dynamic routing.

7.7.3 Network Redundancy

Consider the simple, non-redundant network in figure 7.3. With dynamic routing we can easily make the backbone network redundant by just adding another interface to each router and connect them to a fallback backbone network.

We can also make the routers redundant. Figure 7.4 shows how to do this. For now ignore the dashed connections between the routers and the backbone networks. All bottom subnets are connected to the two backbones through independent routers. If a router fails, all traffic will be routed through the backbone that the surviving router is connected to. With IPv4, this setup would require a passively listening routing daemon on all hosts connected

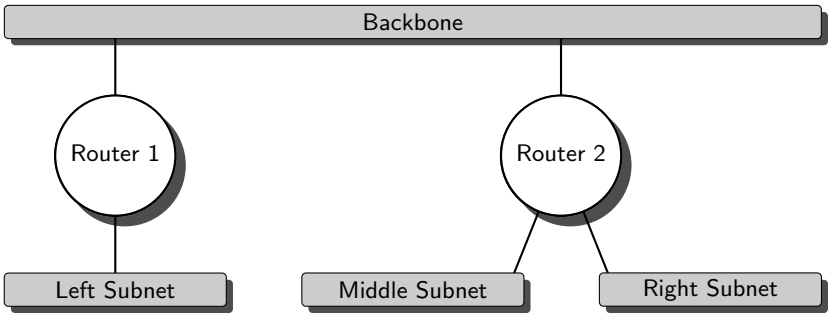

Fig. 7.3. A simple, non-redundant network

to the bottom subnets. With IPv6, autoconfiguration already provides this functionality.

If we also add the dashed connections, we reduce the impact of a router failure on the remaining routers. Assume that router 1a fails. Now all traffic from the middle and right subnet to the left has to pass through router 1b. Without the dashed connections, all traffic also has to pass through router 2b because router 2a can't reach router 1b. With the dashed connections, both router 2a and router 2b can reach router 1b, thus sharing their load over two routers.

We are now missing just one more thing to make our network fully redundant: Redundant bottom subnets. Unfortunately we can't do this with IPv6: If we had redundant bottom subnets and hosts connected to them, each host would need to use the same IP address on both subnets. There are however special network interface cards with redundant physical interfaces and link-

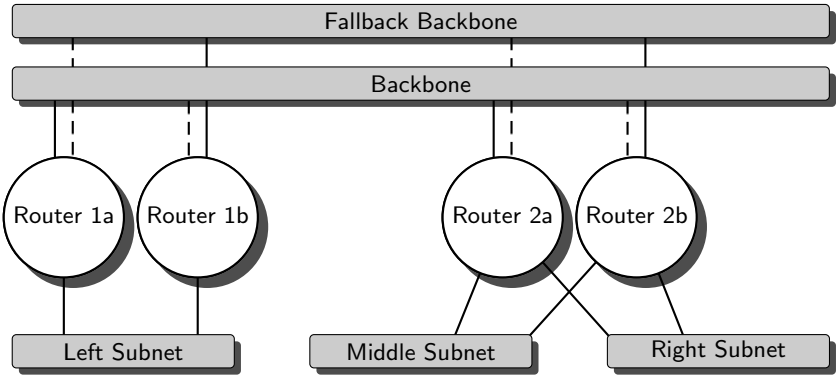

Fig. 7.4. The same network made redundant

layer mechanisms like Solaris' *IP multipathing* (*IPMP*) to achieve redundancy even at the subnet layer.

We could even use IP multipathing and similar to make the backbone redundant. But doing so won't scale while dynamic routing does: If the backbone networks reach their performance limit, dynamic routing lets us simply split them up into multiple subnets.

7.7.4 Router Performance Issues

Performance in routers can actually become a problem with IPv6: Both dedicated hardware routers as well as high-profile server hardware may implement part of their TCP/IP stack in hardware. If we use a server with a *TCP/IP offload engine* (*TOE*) that only implements IPv4, or implements IPv6 only partially, then we may run into performance problems with IPv6 just like the network administrators taking care of big routers.

To deal with that situation we may need to split our networks into smaller segments and/or install additional routers, possibly with less network interfaces. That won't solve all potential performance issues in our network, but in many cases reducing the size of our subnets will alleviate performance issues related to routers.

7.7.5 Performance Issues with ICMPv6 Redirects

Another performance issue related to routing may be caused by ICMPv6 redirects. Figure 7.5 shows how this can happen. The server is receiving con-

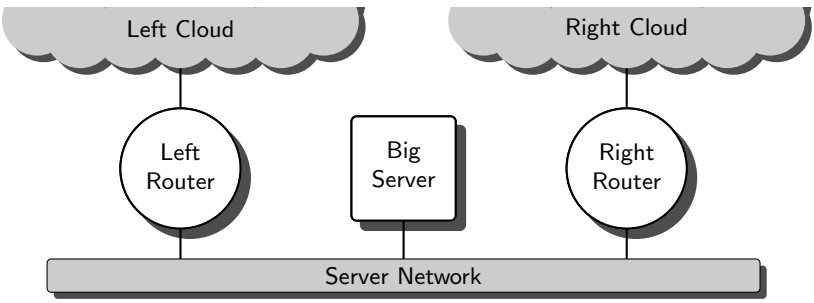

Fig. 7.5. A busy server swamped with ICMPv6 redirects

nections from a large number of machines from both network clouds. Since it will receive routing information only as ICMPv6 redirects, and these redirects are stored as host routes, it will effectively maintain a routing table entry for every single client. This may put a noticeable strain on the server performance.

A network situation like this leaves us several options. We can replace the two routers with a single bigger one connected to both clouds; the host will then send all traffic to that one router and doesn't receive any redirects. We may decide to turn the server into a router; doing so will normally reduce the number of routing table entries quite significantly because it replaces host routes from ICMPv6 redirects with network routes. We may also decide to connect both routers to both network clouds; that will stop the routers from sending ICMPv6 redirects because they can forward packets to any destination. And finally, we may want to separate the routers from the server with an additional router that takes care of the routing on behalf of the server.

7.7.6 Inconsistent Prefix Advertisements

It is possible that multiple routers in the same subnet advertise different network prefixes. This is perfectly legal and in some situations actually quite useful. But it may lead to a phenomenon that is particularly surprising when dynamic routing is also involved.

Figure 7.6 shows such a situation. Both routers advertise different network prefixes to the middle subnet and run a dynamic routing protocol between each other. The left host can ping the right host thanks to dynamic routing.

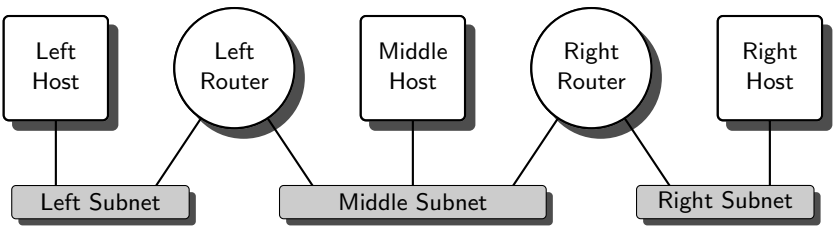

Fig. 7.6. Inconsistent prefix advertisements

The left host can also ping the middle host if it sends it to the address with the prefix that the middle host obtained from the left router. But it *can't* ping the middle host if it uses the prefix from the right router.

Why? When the right router sends a routing announcement to the middle subnet, it doesn't send its entire routing table but first removes all routes that pass through the middle subnet. After all, the routers connected to the middle subnet should already know about them. So the left router will never learn about the network prefix that the right router announces to the middle subnet—remember, routers don't listen to router advertisements. As a consequence, the left router doesn't know how to forward packets sent to addresses with the prefix announced by the right router.

Things get even more surprising if we connect both routers to a backbone network as in figure 7.7. In this case, the left host can reach the middle host,

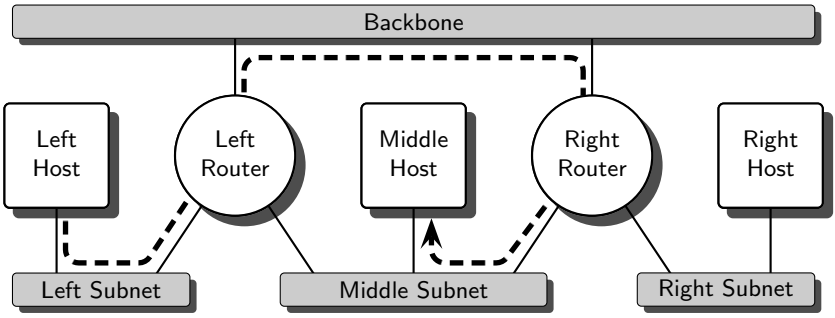

Fig. 7.7. Inconsistent prefix advertisements with a backbone subnet

but the packet travels in an unanticipated and inefficient way: Instead of passing from the left router through the middle subnet to the middle host, it goes through the backbone to the right router and only then through the middle subnet to the middle host.

Now what happened there? The right router sends its routing announcements to the middle subnet as before, so the left router still doesn't know about the additional prefix in the middle subnet. But the right router also sends an announcement to the backbone. This announcement contains a route to the prefix because it is sent to an interface that is different than the one with the announced prefix. Now the left router learns a route to the prefix that passes through the backbone.

When we bring in DNS, then the situation can get even more confusing. If the middle host has a DNS entry for both addresses, then the name server will usually provide the left host with them in a randomized order. In half the cases the connection will work as expected because the left host sends its packets to the address with the prefix obtained from the left router. In half the cases however it will send the packets to the "bad" prefix, causing the effects as mentioned above. Beyond that the address selection algorithms we'll investigate in section 16.4 might also interfere, making problems with inconsistent prefix configurations even more difficult to understand and fix.

There is an obvious conclusion from this surprising behaviour: Keep the network prefixes on all routers in a subnet consistent. As long as you do so, none of these problems will happen. And if they still do happen, check your router configurations for mistyped addresses.

7.7.7 Security Aspects

Dynamic routing is sometimes considered insecure because an attacker may manipulate a router's routing table using bogus routing announcements. If this an issue to you there is effectively only one option to investigate.

Consider a two-tiered network architecture as in figure 7.8. At the bottom tier all hosts are connected to "leaf networks" that only provide direct connectivity between these hosts and to the attached "leaf" routers. Routing in these leaf networks is limited to autoconfiguration and ICMPv6 redirects.

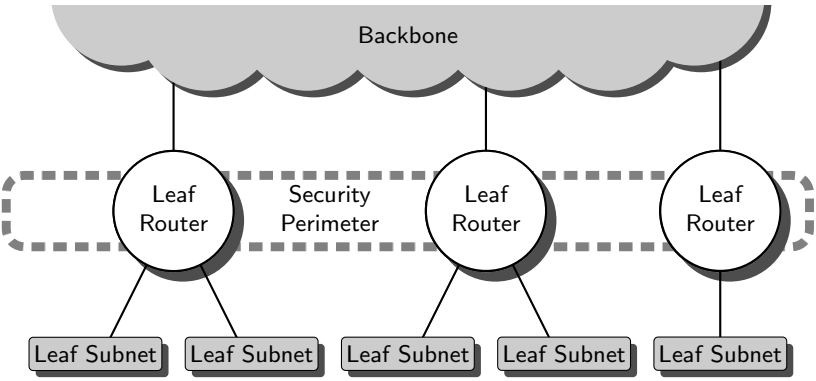

Fig. 7.8. Two-tiered perimeter security topology

Leaf routers won't accept routing announcements from these networks. The top tier consists of "backbone networks". Only routers are permitted to connect to the backbone networks and all dynamic routing happens there. The backbone is considered secure and is possibly protected by packet filters on the leaf routers. All routers connected to the backbone are considered trustworthy, so dynamic routing doesn't need authentication.

So what's wrong with authentication? Well, when RIPv2 was turned into RIPng, people decided to drop the authentication features in RIPv2 and use IPsec for authentication instead. Unfortunately, IPsec implementations today don't support multicast authentication, which leaves us in the unfortunate situation that we can't authenticate a RIPng packet.

7.8 Mixing Static and Dynamic Routing

Mixing static and dynamic routing can be tricky; in many cases it isn't worth the trouble. But there are cases where we need to do it anyway.

If we use dynamic routing but need to connect to another network that doesn't (or vice versa) we will need to set up some "border router" that supports dynamic routing on some interfaces and static routes on others. The same is necessary if we use different routing protocols in different subnets.

Maybe we don't want to use dynamic routing in some networks for security reasons; for example we may have a test environment where inexperienced ju-

nior network administrators are getting to grips with dynamic routing. Again we'll want a "border router" to deal with the situation.

Maybe we cleaned up our network topology except for one particularly nasty kludge that we just can't get rid of right now. We may want to use dynamic routing everywhere except for that kludge.

Finally, we may just feel uncomfortable with an all-or-nothing decision that can't be rolled back in a gradual manner. Personally, this is the most compelling reason to me.

If we need to mix static and dynamic routing, or different dynamic routing protocols, it is best to keep the number of "mixed-mode" or "border" routers as small as possible. It will simplify our work and generally improve the reliability of our network.

We need to find ways to configure static routes on routers that also do dynamic routing. These static routes must be announced through dynamic routing so other dynamic routers will learn about them without being statically configured themselves. How this is done again depends on the particular Unix.

Debian Sarge/Quagga *First we convince the* `ripng` *daemon to announce static routes. We need to add the line*

/etc/quagga/ripngd.conf

```
redistribute static
```

to `ripngd.conf`*. Then we add the actual static routes to* `zebra.conf` *according to the pattern*

/etc/quagga/zebra.conf

```
!              Destination           Gateway
ipv6 route 2001:db8:f00f::/48 2001:db8:fedc:1::1
```

Then we restart the Quagga daemons. That should configure the route in the local routing table and announce the route via dynamic routing.

If you mistype the gateway address in the `ipv6 route` *statement, the address won't be added to your static routing table. So make sure to check the routing table after you add a route this way.*

FreeBSD 6.1 *First we convince the* `route6d` *to announce the static routes it finds in the kernel. To do so we need to add an option* `-s` *to its invocation in* `/etc/rc.conf`*:*

/etc/rc.conf

```
ipv6_router_flags="-s"
```

Then we add the routes just as any ordinary route. The `route6d` *daemon will discover and announce them.*

Solaris 10 *There is no documented way to make* `in.ripngd` *propagate statically configured routes.* 58

7.9 Inside IPv6: Maximum Transmission Unit (MTU) Improvements

A largely unnoticed improvement of IPv6 over IPv4 deals with the way it handles large packets.

Link-layer technologies usually limit the maximum size of a link-layer frame. This maximum size differs between link-layer technologies and may even be configurable.

From the link layer's maximum frame size the *maximum transmission unit* (*MTU*) of a given subnet can be calculated. The MTU defines the maximum size that an IP packet may have to be sent across a given subnet in a single frame. If a packet is larger than the MTU, then it is split into multiple *fragments* that are sent in separate frames.

With IPv4 this is traditionally done in a suboptimal way: If a router receives a large packet through a subnet with a large MTU and then attempts to forward it through another subnet with a smaller MTU, then it will fragment the packet. If another router down the line forwards these fragments through a subnet with an even smaller MTU, then it will fragment the packets again. Since the sender never learns about the fragmentation it will continue to send these awkwardly large packets. Traditional implementations work like this.

Less traditional IPv4 implementations allow the sender to set a flag in the IP header to change this behaviour. If the *don't fragment flag* is set, then a router may send an *ICMPv6 packet too big* notification to the sender instead of fragmenting the packet. The sender can then keep track of the *path MTU*, the minimum of all MTUs between the sender and receiver of a packet, and send fragments at the largest possible size that doesn't get fragmented again. Older routers will however ignore that flag and still fragment packets themselves.

IPv6 always uses this *path MTU (PMTU) discovery* mechanism; routers never fragment packets themselves. This reduces traffic overhead due to repeated fragmentation, optimizes the packet size and and count, and takes some load from the routers involved.

Beyond that, IPv4 MTUs could be ridiculously small—as low as 296 bytes. With IPv6, the MTU of any subnet is at least as large as 1280 bytes.

RFC 1981 [87] and RFC 2460 [24, section 5] contain the definitive specifications.

7.10 Packet Filter Considerations

Routing makes it necessary to reconsider the filter configurations we have developed so far.

We need to revise our anti-spoofing filter rules on nodes with multiple interfaces in general and to make filtering routers deal with packets they are expected to forward.

On hosts as well as routers we may need to permit ICMPv6 redirects in restricted ways; this isn't exactly easy, so we should try to set up a network topology that avoids redirects.

Using packet filters together with dynamic routing is more difficult. There is no universal solution strategy, but with an appropriate network topology packet filters and dynamic routing cooperate at least to some degree.

7.10.1 Source Address Validation (Ingress Filtering)

In section 4.7.5 we have set up some preliminary filter rules to ensure that incoming packets arrive on the "correct" interface. So far we have permitted all traffic from "remote" networks to arrive on any interface, which was fine for a single subnet without forwarding routers.

As soon as we actually route traffic this is insufficient. We need to make sure that traffic from other subnets arrives on the correct interface on all nodes with multiple interfaces, both routers and multi-homed hosts.

Debian Sarge *In section 4.7.5 the* `DESPOOF` *chain ends with a* `ret DESPOOF` *line. We can now replace it with a set of more restrictive rules, tying network prefixes to specific interfaces or sets of interfaces, like*

/etc/ip6tables.sh

```
drop DESPOOF -s 2001:db8:f00::/48 -i '!' eth17
ret DESPOOF -s 2001:db8:f00::/48 -i eth17
[...]
drop DESPOOF -s 2001:db8::/32 -i '!' eth3
ret DESPOOF -s 2001:db8::/32 -i eth3
[...]
ret DESPOOF -s 2000::/3 -i eth2
drop DESPOOF
```

to allow traffic from `2001:db8:f00::/48` *from interface* `eth17` *only, from other addresses with a* `2001:db8::/32` *prefix from interface* `eth3` *only and from other globally routed addresses from interface* `eth2` *only.*

FreeBSD 6.1 *In section 4.7.5 we have used a set of rules to mark packets with a tag* `GOODSRC` *if their source address was acceptable. If we replace the rule*

/etc/pf.conf

```
block in inet6 from 2000::/3 to any tag GOODSRC
```

in `/etc/pf.conf` *with some more specific rules and immediately drop traffic coming in on the wrong interface, then we can filter by proper interfaces like*

/etc/pf.conf

```
block in on lnc17 inet6 from 2001:db8:f00::/48 to any tag GOODSRC
block in quick inet6 from 2001:db8:f00::/48 to any ! tagged GOODSRC
[...]
block in on lnc3 inet6 from 2001:db8::/32 to any tag GOODSRC
block in quick inet6 from 2001:db8::/32 to any ! tagged GOODSRC
[...]
block in on lnc2 inet6 from 2000::/3 to any tag GOODSRC
```

to allow traffic from `2001:db8:f00::/48` from interface `lnc17` only, from other addresses with a `2001:db8::/32` prefix from interface `lnc3` only and from other globally routed addresses from interface `lnc2` only. 59

With both Debian and FreeBSD we should start with the longest prefixes and then work towards shorter prefixes, possibly all the way to `2000::/3` on our upstream interface. It is often possible to reduce the number of rules by using a single rule per prefix—but in the general case, two rules according to the pattern above are necessary.

If we have a redundant network, some addresses may come in from multiple interfaces. In that case, we need to add a line for every interface and prefix combination we want to let through. Otherwise, redundant networks don't pose any additional problems.

7.10.2 Forwarding Filter Rules

Our filter configurations so far only allow traffic either from or to the node running the filter. But on routers we also need to allow packets to be forwarded.

Debian Sarge *The `ip6tables` filter uses a distinct `FORWARD` chain for packets that must be forwarded. This approach is unusual but in many cases very useful. To set up a filter on a forwarding router we run forward traffic through some of the chains we have already defined. The ICMPv6 rules should be more restrictive and only allow traffic that actually needs to be routed; neighbor and router discovery as well as multicast group management packets are inherently non-routeable and should be blocked. A minimal configuration that can be appended at the end of the filter configuration we have set up so far might look like this:*

/etc/ip6tables.sh

```
call   FORWARD SANITIZE
call   FORWARD DESPOOF
call   FORWARD CHECKSRC
call   FORWARD CHECKDST
```

[Continued on next page]

/etc/ip6tables.sh

```
                    [Continued from previous page]
chain  ICMPFWD
accept ICMPFWD -p icmpv6 --icmpv6-type destination-unreachable
accept ICMPFWD -p icmpv6 --icmpv6-type packet-too-big
accept ICMPFWD -p icmpv6 --icmpv6-type ttl-exceeded
accept ICMPFWD -p icmpv6 --icmpv6-type parameter-problem
drop   ICMPFWD
drop   FORWARD
```

Beyond that we can add more ICMPv6 types like echo request/reply, and TCP and UDP traffic as we have done for the `INPUT` *chain before.*

FreeBSD 6.1 *Forwarding doesn't really affect the packet filter configuration except that we need to add rules to allow the traffic we want.*

Stateful filtering establishes two states, one on either interface. So to allow SSH traffic from `2001:db8:fedc:1::/48` *to* `2001:db8:fedc:2::/48`*, we need to add a rule like*

/etc/pf.conf

```
pass quick inet6 proto tcp from 2001:db8:fedc:1::/48 \
                          to 2001:db8:fedc:2::/48 port 22 \
                          flags S/SA keep state
```

without an `in` *modifier. If we only allowed this traffic in, the initial SYN packet wouldn't be allowed to leave our router.* 60

7.10.3 Dealing with ICMPv6 Redirects

As soon as multiple routers with different routes are attached to the same subnet as some hosts, ICMPv6 redirects can occur. If we have such a network topology and need to configure packet filters, we have two choices: Either change the network topology to avoid ICMPv6 redirects or configure our filters to permit them.

Generally, allowing ICMPv6 redirects in security-critical environments is not a particularly good idea, so changing the network topology is actually the preferred approach in this situation. But if we need to filter redirects, doing so is quite straightforward: We just add rules for outgoing redirects on routers and rules for incoming redirects on hosts.

7.10.4 Packet Filters and Dynamic Routing

Configuring packet filters in an environment that relies on dynamic routing is tricky. While it is straightforward to allow RIPng traffic (on port 521/UDP, as explained in section 7.6), dynamically changing routes have some nasty side effects.

If we want to prevent spoofing, we need to know which networks are reachable through which interfaces. If dynamic routing may cause traffic from a given network to arrive through multiple paths, we need to allow traffic from this network from all these interfaces. Doing so may open up the packet filter up to the point that reliable source spoofing prevention becomes impossible.

If we use stateful filters, and dynamic routing may cause the traffic belonging to a connection to be routed through multiple filtering routers, then we may break the connection because the multiple routers can't track the connection. In some cases it may be possible to use whatever high availability mechanisms are available for our packet filters, but generally this makes the filter more complicated. Alternatively, some filters allow a more "relaxed" style of connection tracking, but to some degree this approach defies the very idea of stateful filtering.

In section 7.7.7 we have already seen how we can design a network with a two-tiered topology with perimeter security. If we turn the leaf routers in figure 7.8 into packet filters, then we can safely use dynamic routing on their upper interfaces into the backbone only while we filter all the traffic on their lower interfaces into the leaf subnets. With such a topology we can use the advantages of dynamic routing even in a security-sensitive environment. Depending on the packet filter implementations we use it may even be possible to make the leaf routers redundant by synchronizing their states using some high availability mechanism.

Part II

IPv4/IPv6 Interoperation

8

Interoperation Concepts

As we have already seen in section 1.3, IPv4 and IPv6 operate in parallel without any fundamental problems. But what if we have two nodes that want to communicate with each other but one only supports IPv4 while the other supports only IPv6?

There is no all-embracing solution to this problem. But there are some well-established strategies that serve the purpose in many important cases.

8.1 Dual Stack Configuration and Operation

All interoperation solutions are based on making at least some nodes *dual stacked* so they support both IPv4 and IPv6. In this section we consider a few pitfalls we may encounter when setting up such a node.

Configuring a node for dual stack operation is generally quite simple. IPv4 and IPv6 work independently with the exception that an `ifconfig up` on most Unixen enables both IPv4 and IPv6 at the same time.

Solaris 10 *Interfaces are independently plumbed, configured and enabled.* 61

It is perfectly reasonable to have both `A` and `AAAA` records for the same name in the DNS. So if our web server `www.example.com` is dual-stacked, a resource record set like

example.com.fwd

```
www.example.com. A    192.0.2.80
                 AAAA 2001:db8:fedc:80::80
```

is not only technically legal but generally reasonable.

8.2 Interoperation Problems

At least it would be if there weren't two occasional problems: According to RFC 4074 [88] there is a diminishing number of DNS servers in the wild that return an incorrect error code if we ask for an `AAAA` record and only an `A` record exists and some sloppily IPv6-enabled applications don't attempt to use all addresses that a DNS lookup yields but only the first one. Fortunately, the broken DNS servers are quickly getting updated to behave correctly. The number of badly ported applications has never been too large and knowledge on how to do the modifications properly is quickly spreading in the developer community.

But if we want to make really sure that these problems don't affect us, we should consider using different DNS names for IPv4 and IPv6 addresses.

A more serious problem with dual-stacked systems is non-technical: When we set up nodes to run dual-stacked, then we don't immediately know if they communicate using IPv4 or IPv6. Now if something breaks we may accidentially search for the problem in the wrong place—if the problem happens on the IPv4 side we will look for it on the IPv6 side and vice versa.

To deal with this it takes a bit of experience and preferably a packet sniffer to watch the traffic that is causing the problem. Don't try to outsmart the address selection algorithms; they are reasonably complex and you are best advised to assume that they do a good job. If you really want to know how they work, take a look at section 16.4.

8.3 Dual Stack Everything

When we first try to deploy IPv6 in a production environment, chances are that we can't make any nodes IPv6-only just yet. Instead, all we can do is make everything dual-stacked, watch what "legacy" things still need IPv4, sort them out and eventually, when we have checked that IPv4 simply isn't used anymore, disable IPv4 support at least on the majority of our nodes.

If we manage to do this in a gradual way, slowly deploying IPv6 in small sections of our network environment, this is a quite manageable. Just remember about the problem of barking up the wrong tree mentioned above.

In the long run, permanently dual stacking all machines in our environment won't win us anything but extra work. With some legacy things still requiring IPv4 support, the next step is to make our servers dual-stacked and the majority of our clients either IPv4 only or IPv6 only.

8.4 Dual Stack Servers Only

In general, making a server support both IPv4 and IPv6 is straightforward. We configure both stacks on it, add both `A` and `AAAA` records to the DNS, and if at all necessary make the service listen on both IPv4 and IPv6.

The archetypical dual-stacked service is NFS. If we make an NFS server dual-stacked, then IPv4-only, IPv6-only and dual-stacked clients can use it without problems.

In fact, this is about as far as we can get with NFS. If we have an IPv6-only NFS client and want to connect to an IPv4-only NFS server, we are out of luck. So if we wanted to use an IPv4-only NFS server from the Internet we had no choice but to stick with IPv4.

Fortunately, many of the protocols commonly used on the Internet don't have this limitation. The next few sections explain what we can do to make IPv6-only clients talk to IPv4-only servers and vice versa.

8.5 Connecting to Foreign IPv4-only Servers

When we need to provide access to some IPv4-only servers that are beyond our administrative control, we have two options: Keeping the clients dual-stacked will work but is undesirable. The preferred alternative are devices that map between IPv4 and IPv6 somehow.

These devices can implement the mapping either within the application layer or the network layer. The former are called *application level gateways* (*ALG*), *proxies* or a variety of other names depending on the particular service that is translated. They are often quite simple to set up and configure; we take a look at them in the following chapter 9.

It is also possible to do the mapping within IP. This strategy is somewhat similar to NAT, but noticeably more complex. For services that are not proxy-friendly, using this *protocol translation* is usually the only viable solution, so we deal with it in chapter 10.

8.6 Packet Filter Considerations

Generally packet filters can deal with IPv4 and IPv6 independently of each other. In some cases there are actually independent filters for both protocol families, so dealing with both together requires some work.

Debian Sarge *The packet filters are entirely independent of each other:* `iptables` *filters only IPv4 and* `ip6tables` *only IPv6. Consequently they don't share any configuration. But since both are configured using individual commands it is possible to put their configurations into the same shell script.*

FreeBSD 6.1 *A rule will apply to IPv4 as well as IPv6 unless we explicitly restrict the protocol family using* `inet` *or* `inet6`*.* 62

On routers I usually keep IPv4 and IPv6 separate, because the filter rules are highly dependent on the network topology and the general network configurations are independent of each other. On a dual-stacked hosts it is some-

times more adequate to configure IPv4 and IPv6 together, especially if the same services are used or provided for both IPv4 and IPv6.

Beyond these configuration technicalities there is a fundamental architectural aspect to consider. While it is possible to use both IPv4 and IPv6 in the same subnet, doing so makes packet filter configurations more complex and confusing. Using separate IPv4-only and IPv6-only subnets is slightly more expensive but generally simplifies packet filtering. Since there is little experience with IPv6 packet filter configurations, simplifying the filter configuration is extremely helpful to avoid mistakes.

9

Application Level Gateways

The preferred means to bridge the IPv4–IPv6 gap is an intermediate, dual-stacked machine that provides the connection at the application layer. In many firewall setups, such services already exist for security reasons; they are called *application level gateways* (*ALG*). If we make the machine running the gateway dual-stacked, then we have already solved the problem.

All we need are application level gateways for all the services we want to provide for. Depending on the particular protocol, these gateways are called proxies, gateways, relays, gatekeepers, brokers or whatever else.

9.1 Domain Name Service (DNS)

DNS "proxies" are called "forwarders". If we install our customary BIND on a dual-stacked host and configure it as a standard forwarder, it will serve perfectly well as a proxy between IP versions.

Things will get complicated if we have an IPv4-only name server and IPv6-only clients (or vice versa) and we want the clients to use dynamic DNS updates to update their DNS entries. In theory we might use a dual-stacked secondary DNS server (running a recent version of BIND) and send our DNS updates there, but in practice this setup is tricky at best. In practice, the preferred strategy is to make the primary name server dual-stacked.

9.2 Network Time Protocol (NTP)

The network time protocol inherently supports the concept of proxies; time servers pass their time on to other time servers, thus spreading proper time information all over the net.

In theory, there is one limitation: NTP tracks the number of hops from the reference clock. This metric, called *stratum*, can't be larger than 15. In

practice, if we ever reached this limit, we should reconsider our entire NTP setup.

Another, potentially more relevant problem is the fact that every additional hop makes time synchronization less precise. But again, if this was our problem we should reconsider our NTP setup, get a few GPS receivers, connect some dual-stacked "primary" time servers to them and from there onwards distribute the time information through our net.

9.3 Syslog

It is perfectly possible to make a "loghost" forward log events from a client to another, higher-level "loghost". So any `syslogd` serves as a proxy if we configure it to forward all events to another log server.

In practice, this can be vitally important: If we use monitoring software that doesn't support IPv6 and can't afford to drop it in favour of something IPv6-enabled, a syslog gateway may save the day. This scenario usually happens when the monitoring software in place is an expensive commercial product that has been severely customized to the local needs.

Configuring a syslog proxy it is essential to make sure that there are no forwarding loops; otherwise, the setup is straightforward.

9.4 Simple Mail Transfer Protocol (SMTP)

The archetypical application level gateway is an *SMTP relay*. All mail transfer agents support the relaying of e-mails on behalf of other machines, so all mail transfer agents that support IPv6 can be used as gateways.

The only issue we need to be aware of is not directly IPv6-specific: We must make sure that our gateway can't be abused as an "open relay" to send spam to other machines. Since the access control mechanisms with the most common mail transfer agents are capable of handling both IPv4 and IPv6, this shouldn't pose any challenges beyond those of an "ordinary" configuration in an IPv4-only environment.

9.5 Hypertext Transfer Protocol (HTTP)

We can use both the Apache2 proxy module and `ffproxy` to set up a proxy for HTTP and HTTPS. As with the mail relays, we only need to ensure that the access controls are configured so as to prevent unauthorized use.

9.6 Packet Filter Considerations

Setting up a packet filter on an application gateway doesn't pose any particular challenges; we just need to open the necessary ports for both IPv4 and IPv6.

Gateways are usually IPv6 hosts, not routers, so they should have only a single IPv6 interface. If we want them to connect to separate IPv4-only and IPv6-only subnets, then we should keep in mind that many Unixen don't let us enable IPv6 on individual interfaces. So we should block all IPv6 traffic on interfaces that connect to an IPv4-only interface; otherwise we risk attacks using IPv6 link-local addresses to circumvent the IPv4 filter rules on these interfaces. Similarly, we should block IPv4 traffic on the interfaces connected to IPv6-only subnets.

10

Protocol Translation

We have made all the services in chapter 6 interoperate with IPv4 except for the Secure Shell, which isn't proxyable[1]. We need to use an alternate strategy for it: We translate IPv6 packets into IPv4 packets and vice versa.

10.1 Protocol Translation Concepts

The translation between IPv4 and IPv6 at the network layer is called *protocol translation*. In many respects it is closely related to network address translation (NAT). But instead of rewriting IP addresses and port numbers it replaces entire IP headers.

Similar to NAT, a protocol translator needs to keep state; this makes it difficult at best to make the translator redundant. If the protocol translator fails or the routing changes to send the traffic through another translator, all open connections will die.

Again similar to NAT, if the protocol communicates IP addresses in the data stream, protocol translation won't work unless some application-specific translation is included. The archetypical NAT-incompatible protocol is FTP. It doesn't traverse NAT, so it can't work with protocol translation, either.

Protocol translation is far more complex than NAT. Not only does it replace entire IP headers, but it needs to handle DNS translation as well: If an IPv6-only client wants to connect to an IPv4-only server it will only ask the DNS for IPv6 addresses, or `AAAA` records. These requests must be intercepted, translated into `A` record queries, and the answers from the real name

[1] As a workaround we might set up a dual-stacked machine that users log in to so from there they can log in to the final destination. But in many situations such an intermediate machine poses an undesirable security risk. Aside from that, using port forwarding and some other advanced features of the Secure Shell with this is tedious at best.

server must be translated into matching `AAAA` records that are then sent to the client. `PTR` records must also be treated like this.

There are two different specifications that have been implemented: *Network address translation/protocol translation* (*NAT-PT*) as of RFC 2766 [111] and *transport relay translation* (*TRT*) as of RFC 3142 [74]. Both are based on a specification from RFC 2765 [94] called *stateless IP/ICMP translation* (*SIIT*) which has never been implemented by itself but defines the precise algorithms for transforming the IP packets.

The primary difference between NAT-PT and TRT is the way they deal with the DNS translation. NAT-PT does the translation by itself. This makes it possible to set up the translation bindings before the actual connection is established, so it is even possible to connect from an IPv4 client to an IPv6 server if we have a sufficiently large IPv4 address pool to use for the translation. Then again, this makes NAT-PT a fairly complex, monolithic entity that is difficult to implement properly.

TRT doesn't do the DNS translation itself but leaves it to a translating DNS forwarder. This makes it less complex than NAT-PT but doesn't allow connections from IPv4 to IPv6.

At the time of this writing, there is no working NAT-PT implementation for Unix. Some time ago a project worked on a Linux implementation but it never became stable enough for real use.

The BSD stack brings a TRT implementation called `faith`. Its major drawbacks are that it doesn't run entirely in kernel space, that the associated daemon must be run on a fixed port number and that it only supports TCP; neither UDP nor ICMPv6 (including `ping`) work. In many cases these limitations are acceptable; if not, Cisco has implemented NAT-PT for their IOS.

Another Linux project, pTRTd, tried to implement TRT but again, it didn't get beyond the experimental stage and has apparently been abandoned.

Solaris doesn't support protocol translation at all.

10.2 Setting Up a Protocol Translator

Figure 10.1 shows a simple test setup. The real name server has a zone `example.com` with an entry

`example.com.fwd`

```
v4only.example.com. A 192.0.2.2
```

on the DNS server. We use the prefix `2001:db8:fedc:4444:4444:4444::/96` to map the IPv4 address space into IPv6. Our goal is to connect from the v6-only client to the v4-only server via ssh.

First we set up the protocol translation as such on the translator.

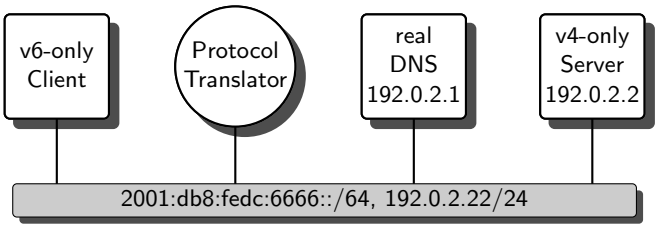

Fig. 10.1. A protocol translation test setup

FreeBSD 6.1 *The TRT implementation consists of a kernel-side interface called* `faith` *and a matching userland daemon* `faithd`*. Together they provide the translation between IPv4 and IPv6. Additionally we need the* trick-or-treat daemon `totd` *to translate DNS records.*

We first configure the protocol translator with an IPv4 address `192.0.2.3`*. Then we set it up as a single-legged advertising router with an IPv6 address* `2001:db8:fedc:6666::1` *so the client will route its IPv6 traffic through it and it has a fixed IPv6 address that we can configure the client's resolver with.*

In `/etc/rc.conf` *we configure the TRT prefix with a line*

/etc/rc.conf
```
ipv6_faith_prefix="2001:db8:fedc:4444:4444:4444::"
```

This automatically enables the kernel side of the TRT implementation after the next reboot. Then we need to start the userland daemon to do the actual translation. To enable TRT translation for the `ssh` *we first need to disable any running* `sshd`*. In the simplest case, we can then run*

```
# faithd ssh
```

either on the command line or in `/etc/rc.local` *to translate every incoming connection on the SSH port, 22/TCP. But if we want to be able to connect to the translator machine itself, we need to tell it what to do when a connection to its own mapped address comes in; in this case we can start the* `sshd` *as if it was started by the* `inetd`*:*

```
# faithd ssh /usr/sbin/sshd sshd -i
```

Finally, if we run an `inetd` *on the translator node, we may let it start the* `faithd` *instead with a line like this in* `/etc/inetd.conf`*:*

/etc/inetd.conf
```
ssh stream tcp6/faith nowait root /usr/sbin/faithd /usr/sbin/sshd -i
```

This is particularly useful if we want to provide translation on a larger number of port numbers since in that case we only run a single `inetd` *process instead of separate* `faithd` *processes for all the port numbers.*

At this point it should be possible to connect from the client to the server if we specify the mapped IPv6 address as the destination:

```
# ssh -v 2001:db8:fedc:4444:4444:4444:192.0.2.2
```

In case this doesn't work, we can usually find the problem like this:

- ☐ *On the translator, first check the IPv4 configuration.*
- ☐ *Now try to ping the v4-only server from the translator via IPv4, specifying its IPv4 address.*
- ☐ *Now do the same with the DNS name instead of the address.*
- ☐ *SSH from the translator to the v4-only server.*
- ☐ *Check that the `faith0` interface is up and running on the translator.*
- ☐ *On the translator, check with `netstat -rnf inet6` that all traffic to the translation prefix is routed through the `faith0` interface.*
- ☐ *Again on the translator, check that the IPv6 interfaces are up, running and configured as expected.*
- ☐ *Check with `sysctl net.inet6.ip6.forwarding` that IPv6 forwarding is enabled.*
- ☐ *Check that the `rtadvd` is running.*
- ☐ *Use `sockstat` to check that either the `faithd` or `inetd` is listening on the `ssh` port (22).*
- ☐ *Use `ssh -v 2001:db8:fedc:4444:4444:4444:192.0.2.2` to see if you can reach the v4-only server via IPv6.*
- ☐ *Check the v6-only client. It should have its interface properly configured and the translator as its only default route.*
- ☐ *Finally, use `ssh 2001:db8:fedc:4444:4444:4444:192.0.2.2` from the client to see if you can reach the v4-only server via IPv6.*

Now we need to address the DNS issue. The DNS translation is done by the trick-or-treat daemon, *`totd`. It is available from the ports/packages collection. We tell it the translation prefix and the address of the real DNS server in `/usr/local/etc/totd.conf`:*

`/usr/local/etc/totd.conf`

```
prefix 2001:db8:fedc:4444:4444:4444::
forwarder 192.0.2.1
```

and set

`/etc/rc.conf`

```
totd_enable=YES
```

in `/etc/rc.conf` to start it on boot. Now we either reboot or run

```
# /usr/local/etc/rc.d/totd.sh start
```

to start the `totd` daemon. On the client we change `/etc/resolv.conf` to refer to the translator node as our name server:

/etc/resolv.conf

```
domain example.com
nameserver 2001:db8:fedc:6666::1
```

At this point it should be possible to connect from the v6-only client to the v4-only server using its DNS name. If there are problems, consider these checks:

- ☐ *On the translator, check that the* `totd` *is actually running.*
- ☐ *Check that it listens on the DNS port (53).*
- ☐ *Check that it provides the properly translated data: Run*

  ```
  dig @::1 v4only.example.com. any
  ```

  on the translator. It must return both the original IPv4 address and the translated IPv6 address `2001:db8:fedc:4444:4444:4444:c000:202` *(*`2001:db8:fedc:4444:4444:4444:192.0.2.2`*).*
- ☐ *Now run* `dig` *again on the client. Omit the* `@::1`*; that way* `dig` *will ask the name server configured in* `/etc/resolv.conf`*, which should be the* `totd` *on the translator. The result should be the same as the previous run on the translator node.*
- ☐ *Finally, do a* `ssh -v v4only` *on the client.*

From this point on, all we need to do to enable protocol translation for additional port numbers is to start additional `faithd` *instances for the ports; we can do so either directly or using the* `inetd`*.* 63

10.3 Operational Issues

A first problem at least with the `faith` implementation are misleading error messages: If we try to `ssh` to a non-existent machine, we won't get a "host unreachable" error or such. Instead, the `faithd` closes the connection when it can't reach the destination machine. Additionally, `ping` doesn't work with `faith`. Both of these phenomena together are extremely misleading even if we know about the setup; a semi-clued user base will quite likely trigger a large number of false alarms if this happens.

Since the protocol translator needs to keep state of all translated connections, it doesn't lend itself to redundant setups. With the `faith` implementation it is impossible to make a translator redundant. Furthermore, all traffic needs to be routed through the translator, so if we use dynamic routing we need to hardwire some route to the translation prefix to the translator.

But the biggest issue with protocol translation is the fact that it interferes with the DNS; if it causes any problems at all, it will effectively render all its DNS clients unusable. And there is another problem with the `totd`:

The `totd` *as of version 1.5.1 has an annoying bug: It doesn't always send a reply from the IP address that it received the request from. If the* `totd` *server has multiple routable addresses configured on the particular interface, the client will ignore the replies because they don't originate from the expected source address. To work around this problem we need to make sure that the translator machine has only a single routable IPv6 address on its interface.*

If performance becomes an issue, it is desirable to split the load over multiple machines. It is straightforward to separate the `totd` daemon from the `faith` translator.

It is even possible to configure multiple `faith` machines with separate prefixes and then configure all the prefixes with the `totd` daemon to distribute the load over multiple `faith` translators. The `totd` will then return an address with a different prefix on every request. If one of the `faith` translators becomes unavailable, this will cause those connection attempts to fail that are directed to the prefix of the failing translator. In consequence, the more `faith` translators exist, the less reliably will translation work.

All these issues show that protocol translation, at least with the implementation we have today, is only a last resort transitional measure and should be avoided when possible.

10.4 Packet Filter Considerations

The packet filter configuration on a protocol translator is quite similar to that of an application level gateway: There is an IPv6 interface that accepts incoming connections and an IPv4 interface, which doesn't have to be different from the IPv6 interface, from which outgoing connections are established.

The difference between a protocol translator and a gateway is that a protocol translator is a router; it forwards packets to the entire translation prefix. So we need filter rules appropriate for a router, not a host.

FreeBSD 6.1 *Filtering on the translation pseudo-interface is unnecessary since all traffic passing through it can be treated on the physical interfaces that it passes through.* 64

In theory it is possible to set up a protocol translator on a "real" router; but doing so tends to make the filter configuration more complex and error-prone. So I personally prefer to keep the protocol translation on a dedicated machine with one or two interfaces.

Part III

Tunnels and Related Topics

11

Tunnel Basics

If we want to connect to the Internet6 but our decidedly IPv6-unfriendly Internet service provider refuses to offer us any direct IPv6 connectivity, then we will most likely use a tunnel to carry our IPv6 traffic over our existing IPv4 connectivity.

But tunneling is a far more versatile concept. It can be used to send traffic over "incompatible" networks, to provide a static IP address to a node or network with changing connectivity, to deal with certain performance problems and to set up redundant network connectivity.

11.1 Concepts and Terminology

A *tunnel* is a mechanism that allows a network layer protocol to use another protocol that is not itself a link-layer protocol as its link-layer connection. In many cases, tunnels are used to connect networks using one network layer protocol through an intermediate network using another network layer protocol.

The network layer protocol used as the link-layer protocol is called the *outer protocol* and the protocol used between the end points is called the *inner protocol*. With the common scenario that we need to pass IPv6 traffic over an IPv4-only intermediate network, IPv4 would be the outer and IPv6 the inner protocol.

A *tunnel node* has an outer protocol configured as a link-layer protocol for an inner protocol. A *tunnel host* provides no forwarding functionality for the inner protocol while a *tunnel router* does. Figure 11.1 shows a tunnel environment with IPv4 as the outer and IPv6 as the inner protocol. The "horseshoe" symbol denotes the tunnel nodes.

To understand how a tunnel works, consider the easiest case first: The two tunnel hosts want to communicate using the inner (IPv6) protocol but are only reachable from each other using the outer (IPv4) protocol. An application on

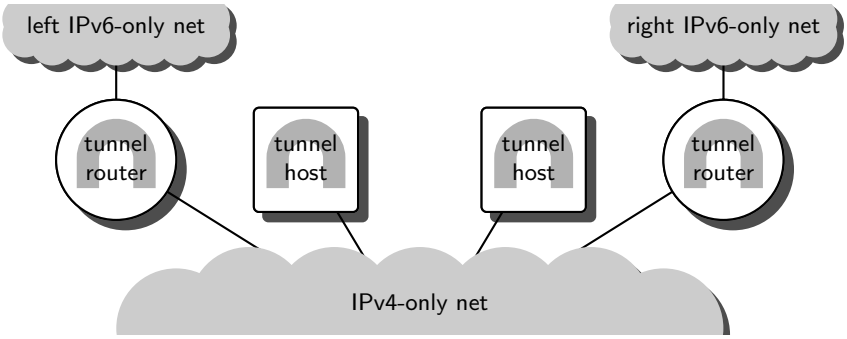

Fig. 11.1. A tunnel environment connecting IPv6 over IPv4

the left tunnel host wants to send some data to another application on the right tunnel host using IPv6. It sends the data through the transport layer to its IPv6 network layer. From there, an IPv6 packet is sent to the apparent link layer; it is put inside an IPv4 packet to the IPv4 address of the right tunnel host and then sent through the IPv4-only net. All intermediate routers will just consider the packet an ordinary IPv4 packet to the right tunnel host and forward it there. The right tunnel host finally receives the packet, recognizes its own IPv4 address as the destination, and unwraps the packet. It finds the IPv6 packet inside and passes it through its IPv6 and TCP implementation to the application.

Tunnel routers behave similarly, but provide tunneling on behalf of other nodes. Assume that a node in the left IPv6-only net wants to send a packet to another node in the right IPv6-only net. It sends a plain IPv6 packet through the IPv6-only net. Eventually, the packet is routed to the left tunnel router. The tunnel router receives it, puts it in an IPv4 packet with the IPv4 address of the right tunnel router as the destination address, and sends it across the IPv4-only net. The right tunnel router eventually receives the packet, unpacks it and passes the IPv6 packet inside on again as a plain IPv6 packet through the right IPv6-only network to the destination host.

Tunnel nodes that put a packet inside an outer layer packet are called *tunnel entry points* while those that unwrap such a tunneled packet are called *tunnel exit points*. In many cases, a tunnel entry point will serve as tunnel exit point for packets travelling in the opposite direction. But as we'll see later on, some tunnels are inherently unidirectional.

11.2 Tunnel Types

IPv6 comes with a daunting choice of different tunnel protocols, some of which are quite similarly named; but "6in4", "6to4" and "6over4" are fundamentally different things. While this is quite confusing at first, most of these tunnels are actually quite useful for one purpose or another.

There are two types of tunnels: *IP-in-IP tunnels* put the tunneled IP packets directly into outer IP packets while other tunnels use more complex mechanisms.

Putting IP packets inside other IP packets has two advantages: It causes very little overhead and in some cases it lets the tunnel entry point deduce the address of the tunnel exit point from the inner destination address.

The other mechanisms provide for a number of features that IP-in-IP tunnels can't possibly offer: They can use non-IP protocols as the outer protocol, offer advanced cryptographic support and penetrate NAT gateways.

11.3 Common Scenarios

There are many different scenarios that make use of tunnels. The most common are these:

If our ISP only provides IPv4 connectivity, then a tunnel to a tunnel service provider will still get us connected to the Internet6 without the need to find a new ISP.

Similarly, if we have multiple IPv6 networks that are only connected to each other via IPv4, then tunnels let us establish IPv6 connectivity between them.

If we need to set up a connection through untrustworthy networks, an encrypting tunnel lets us set up a *virtual private network* (*VPN*).

Beyond that we can use tunnels for some not so obvious purposes:

If we have a high-bandwidth network that uses a hardware IPv4 implementation but needs to deal with IPv6 in software, then we might consider using tunnels to get our IPv6 traffic through that part of the network even if it does support IPv6.

We can use tunnels to set up some "non-standard" routing. This is essential when we want to provide redundant network access to a site, since IPv6 address allocation policies make it virtually impossible to obtain provider-independent address space. Section 25.1 explains how such a setup works.

Some tunnel specifications have been devised that provide IPv6 connectivity to individual hosts even through an IPv4-only network. These should be considered a last resort only. There are virtually no Unix implementations available anyway.

11.4 Operational Issues

Setting up and operating a tunnel opens a number of issues that we need to keep in mind.

Tunnels that need to be explicitly configured can become a serious burden. A single tunnel is not too bad, but if we want to configure tunnels between a dozen tunnel routers, then we need $12 \times 11 = 132$ separate tunnels. This makes self-configuring 6to4 tunnels particularly valuable.

In some cases, broken tunnels are difficult to debug. If we have access to the tunnel entry and exit nodes, then the situation isn't too bad. But if one of the tunnel nodes is outside our administrative control, then things can quickly become tedious and frustrating because we see the tunnel as a broken link-layer connection only.

It is possible to set up *nested tunnels*, like tunneling IPv4 over IPv6 over IPv4. In fact, there are situations where we might have reason to do so. But nested tunnels pose the risk that we route a tunneled packet into the tunnel again. These tunnel loops can cause traffic storms that we must avoid. Fortunately, in many cases a tunnel entry point will refuse to let an already-tunneled packet enter the tunnel unless explicitly configured to do so.

Prepending an additional IP header to a packet will make it larger. The maximum transmission unit (MTU) of a tunnel is therefore smaller than that of the actual link layer. Thanks to the mandatory path MTU discovery this usually won't affect network traffic too much. But there have been cases of "telnet works but FTP doesn't" that were eventually traced back to a bad MTU configuration.

Finally, tunnels can seriously distort the topology of a network. Tunnels will always be considered direct link-layer connections between the tunnel end points. Without proper planning and documentation tunnels will make networks unintelligible to both us as well as dynamic routing, which needs to be configured properly to reflect the actual "distance" between the tunnel endpoints, otherwise it will send traffic through a tunnel even though a better route exists.

11.5 Security Considerations

From a security perspective, tunnels are a mixed blessing.

There is an increasing number of attacks on assumedly "IPv4-only" networks that don't filter tunneled packets properly. If a packet filter doesn't stop tunneled traffic, then some combination of tunnel mechanisms may be used to send traffic past an "IPv4-only" packet filter.

Attackers also use readily accessible tunnel entry points to obscure the source of their attack. RFC 3964 [99, section 4.3] explains some of these mechanisms.

Setting up a packet filter to deal with tunnels isn't exactly trivial either.

On the positive side, tunnels allow us to implement the encryption of traffic across untrustworthy networks at a limited number of central points.

11.6 Choosing the Proper Tunnel

With all the different tunnel types available it is sometimes a daunting task to select the proper one for any given job. Fortunately, some useful rules of thumb exist for many common situations.

For a moderate number of fixed tunnels, using configured tunnels as of section 12.1.2 is usually the best approach.

When the number of tunnels gets larger or the tunnels change often, 6to4 tunnels with our own border routers as of section 12.2 eventually become a better alternative.

To encrypt traffic across untrusted networks both IPsec in tunnel mode as of chapter 21 and OpenVPN as of section 13.3 are feasible options.

OpenVPN is also useful to set up a tunnel through an IPv4 NAT gateway. Without it, we need to set up the tunnel node on the NAT gateway itself, configure reverse NAT or use a NAT-capable tunnel broker service.

For a small site there are two options: If we want to use a tunnel service provider, then we need to use whatever they offer; that's usually either a configured tunnel or a semi-standardized NAT-capable tunnel and a *tunnel broker* protocol to adjust for dynamically assigned IPv4 addresses, as section 14.1 explains. Without a dedicated tunnel service provider, a 6to4 tunnel is another option at least for non-critical operations if we have a statically assigned IPv4 address and a router that can be configured to run as a 6to4 border router.

To deal with performance bottlenecks that are specific to IPv6, our best option may be configured tunnels. Configured and 6in6 tunnels are useful to establish some "non-standard" routing, like setting up a fallback route through a second provider. Section 12.3 has the details on tunnels over IPv6.

Finally, if we need to set up a tunnel with a Cisco router as a peer, *generic routing encapsulation* (*GRE*) may be the only choice; while GRE is less efficient due to its extra overhead compared to a configured tunnel, it is widely used by the Cisco community. Section 13.1 explains the details.

12

IP-in-IP Encapsulation

The tunnels introduced in this chapter *encapsulate* IP in IP using special protocol types in the protocol (IPv4) or next header (IPv6) field. Figure 12.1 shows how a plain, unencapsulated IPv6 packet in an Ethernet frame is encapsulated in an IPv4 packet (omitting the trailing Ethernet checksum).

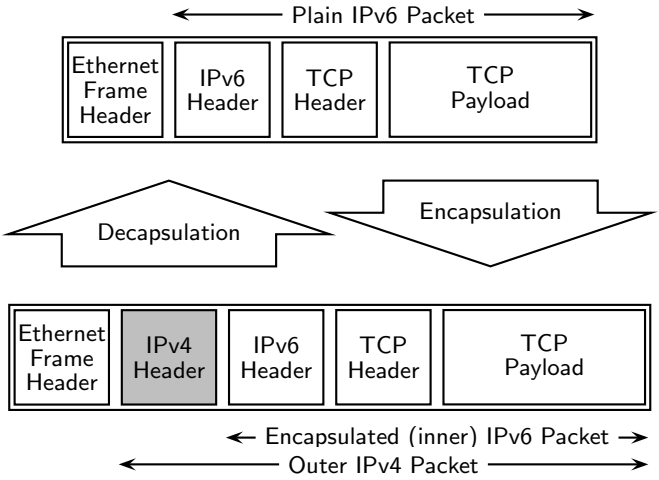

Fig. 12.1. An IPv6-in-IPv4 packet

The outer IP header treats the inner IP packet just like any ordinary transport layer payload. Only the protocol type in the outer IP header identifies the inner packet as IPv4 (protocol type 4) or IPv6 (protocol type 41).

With IP-in-IP encapsulation it is perfectly reasonable to tunnel IPv4 over IPv4, or IPv6 over IPv6. In fact, some implementations provide a single consistent framework and interface for all IP-in-IP tunnels.

From a conceptual point of view, encapsulation adds an intermediate *tunnel layer* to the TCP/IP stack of a tunnel node, as figure 12.2 shows. This layer isn't always used; to the network layer it appears as just another link layer while the link layers consider it part of the network layer.

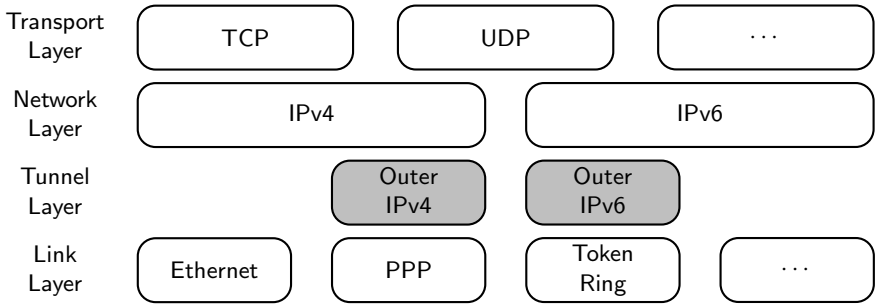

Fig. 12.2. The TCP/IP stack with encapsulation

From an implementor's point of view the tunnel layer is part of the network layer, if only because the link layer can't distinguish encapsulated from unencapsulated packets and therefore can't tell where to pass an incoming packet up to. But considering the packet format, the notion of a tunnel layer suits our purposes quite well.

Besides the IP versions of the inner and outer IP headers, IP-in-IP encapsulation mechanisms differ in the way the tunnel entry point determines the address of the tunnel exit point.

12.1 Configured and Automatic (6in4) Tunnels

Originally, two encapsulation tunnels for *IPv6-in-IPv4 encapsulation*, or *6in4 encapsulation* for short, were defined in RFC 2893 [47] and its predecessor RFC 1933 [46]: Configured and automatic tunnels. Their successor, RFC 4213 [95] recently declared automatic tunnels obsolete and changed the way tunnel endpoints configure their link-local addresses.

Configured tunnels are widely used since the bandwidth overhead they require is very small. Automatic tunnels are formally obsolete but still available with many implementations; they are mostly relevant when configuring a packet filter to protect from attacks based on these automatic tunnels.

Both tunnels were originally considered unidirectional. It is possible and reasonable to use an automatic tunnel in one direction and a configured one in the other. Without automatic tunnels this doesn't make sense anymore, so configured tunnels are generally considered bidirectional for most purposes.

12.1.1 The Link-local Address Problem

Some implementations are still based on the old RFCs, causing an annoying but otherwise harmless interoperability problem.

The old RFC 2893 [47, section 3.7] specifies that the link-local inner address of a tunnel interface should be `fe80::`⟨*outer address*⟩. RFC 4213 [95, section 3.7] changes this and just demands that the link-local address be chosen in a way that ensures it to be different from the address used by the remote peer.

The implementations differ significantly in the way they deal with the inner link-local addresses.

Debian Sarge *This implementation uses link-local addresses according to the stricter RFC 2893 specification. It doesn't assume that its peer does the same and therefore conforms to RFC 4213 as well.*

FreeBSD 6.1 *The* `gif` *interface looks up the interface that is configured with the outer tunnel address and re-uses its link-local address as the inner tunnel address. This is in compliance with RFC 4213 but violates the old RFC 2893.*

Solaris 10 *The behaviour is strictly according to RFC 2893. The link-local address is derived from the outer tunnel address. Additionally, Solaris assumes the peer to conform to RFC 2893 too and only configures a host route to the link-local address derived from the peer's outer address.* 65

This causes an interoperability problem between FreeBSD 6.1 and Solaris 10. To work around it we need to make all tunnel nodes conform to a single standard. Preferably, we want Solaris 10 to set up a network route for the entire link-local prefix `fe80::/10` on its tunnel interface. Alternatively we can force FreeBSD 6.1 to use link-local addresses that conform to RFC 2893. We'll see how to set up either workaround below.

12.1.2 Configured Tunnels

Configured tunnels are virtual point-to-point links between machines. The tunnel entry point needs to be configured with the IPv4 address of the tunnel exit point. Additionally, if we want to reach more than the tunnel exit point through a tunnel, we need to set up our routing to use the tunnel for the addresses we want to reach through it.

As a special case it is possible to set up a *default (configured) tunnel*, a configured tunnel with a default route directing all IPv6 traffic through that tunnel.

Configured tunnels are specified in RFC 2893 [47, section 4] and RFC 4213 [95, section 3].

Figure 12.3 shows a common test setup. The upper left network simulates the Internet6 while the right network represents a local network connected to the Internet6 through a tunnel. The two tunnel routers just differ in their routing configuration: The left tunnel router only routes traffic to the right upper subnet through the tunnel while the right tunnel router sends all non-local traffic through the tunnel.

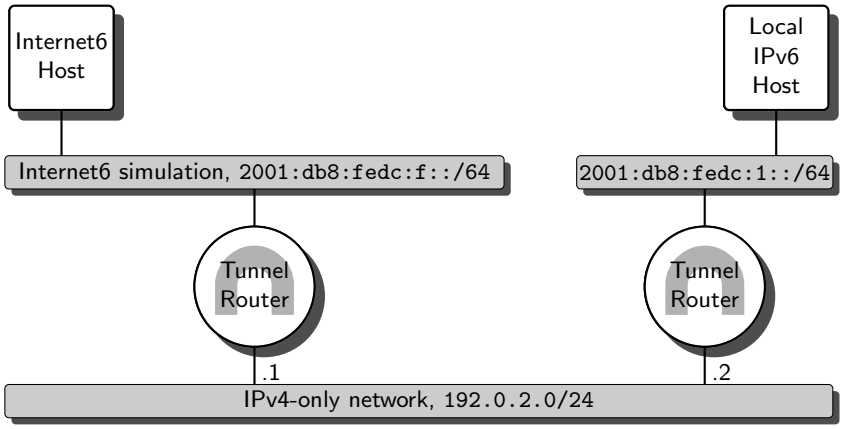

Fig. 12.3. Using a configured tunnel over an IPv4-only network

To configure the tunnel routers we first set up IPv4 on their bottom interfaces and IPv6 including packet forwarding and a router advertisement daemon on their top interfaces.

- ☐ The tunnel routers must be able to ping each other using IPv4.
- ☐ The hosts must be able to ping the router connected to their subnet via IPv6.
- ☐ The routers must have a router advertisement daemon running.
- ☐ The hosts must have a default IPv6 route to their router.
- ☐ IPv6 packet forwarding must be enabled on the routers.

Next we configure the tunnel interface on each router.

Debian Sarge *The tunnel interfaces are called* `sit`⟨n⟩*. By default,* `sit0` *is used for automatic tunnels, so we set up* `sit1` *for the tunnel. In theory,* `ip` *lets us name the tunnel interfaces anything we like; in practice, doing so tends to be confusing rather than helpful at least in textbook setups. So we just stick with the more traditional approach and call the interfaces* `sit`⟨n⟩*.*

There are multiple ways to configure a tunnel manually. The preferred method uses `ip` *like*

```
# ip tunnel add sit1 mode sit remote 192.0.2.2 ttl 64
# ip link set dev sit1 up
```

on the left tunnel router and similarly with `remote 192.0.2.1` *on the right tunnel router. This creates a tunnel interface* `sit1`*, enables IPv6-in-IPv4 tunneling on it and configures a remote peer. Additionally it is advisable to set the TTL explicitly as the example shows. Linux otherwise uses the hop limit or TTL from the inner packet and copies it to the outer packet; this may be useful in some situations but is generally considered troublesome rather than useful—it will break* `traceroute`*, for example.*

In addition we may also provide the arguments `local 192.0.2.1` *to the first line to restrict incoming packets to a single local address. If the tunnel router has multiple interfaces, then doing so is a good idea.*

At this point our left tunnel router will automatically configure the link-local address `fe80::192.0.2.1`*. If we want to use more than link-local addresses on our tunnels, we can configure additional static addresses on them as usual:*

```
# ip -6 addr add 2001:db8:fedc:4646:1::1/64 dev sit1
```

This isn't strictly necessary but sometimes comes in handy when we want to ping the tunnel exit point.

In `/etc/network/interfaces` *we can make the same configuration permanent like this:*

/etc/network/interfaces

```
auto sit1
iface sit1 inet6 v4tunnel
    endpoint 192.0.2.2
    local 192.0.2.1                    || Optional
    address 2001:db8:fedc:4646::1      || Optional
    netmask 64                         || Only needed with previous line
```

After a reboot or an `ifup sit1` *the tunnel should be configured correctly.*

FreeBSD 6.1 *All encapsulation except for 6to4 tunnels is done with* `gif`$\langle n \rangle$ *interfaces—the "generic tunnel interfaces". With these we must supply both the local and remote outer addresses when we configure them.*

To set up the tunnel on the left router temporarily,

```
# ifconfig gif0 create tunnel 192.0.2.1 192.0.2.2 up
```

creates the tunnel interface `gif0`*, configures its local and remote outer address and brings the interface up. If we want to add more than the link-local inner addresses that are automatically configured, we do an*

```
# ifconfig gif0 inet6 2001:db8:fedc:4646::1 up
```

as with any ordinary interface.

To configure the left router permanently, we can set the equivalent variables in `/etc/rc.conf` *like*

/etc/rc.conf

```
gif_interfaces="gif0"
gifconfig_gif0="192.0.2.1 192.0.2.2"
ipv6_ifconfig_gif0="2001:db8:fedc:4646::1"          || Optional
```

The first line defines a list of `gif` *interfaces to configure, the second sets the outer tunnel addresses for* `gif0` *and the third assigns it an optional static IPv6 address.*

To enable the configuration we must either reboot or do a

```
# /etc/rc.d/netif restart
# /etc/rc.d/network_ipv6 restart
```

to re-read `/etc/rc.conf`*.*

As section 12.1.1 already explained, FreeBSD uses link-local addresses that may cause problems with older tunnel implementations. Most notably, Solaris will configure the wrong link-local address for a FreeBSD tunnel peer, making the tunnel unusable. To work around this problem, we can run the script

/etc/rc.local

```
#! /bin/sh

gif_interfaces="`ifconfig -a | sed '/^gif[0-9]*:/!d;s/:.*//'`"

for g in $gif_interfaces
do
    localaddr="`ifconfig $g | sed '/tunnel/!d;s/.* inet //;s/ .*//'`"
    badlinklocal="`ifconfig $g \
                  | sed '/inet6 fe80/!d;s/.*inet6 //;s/%.*//'`"
    for bad in $badlinklocal
    do
        /sbin/ifconfig $g inet6 $bad delete
    done
    /sbin/ifconfig $g inet6 fe80::$localaddr
done
```

after we have set up the outer tunnel addresses; it will remove all link-local addresses first and then add one conforming to the old RFC 2893 standard, thus making the interface conform to both the old and new specifications. To fix the configuration after boot we can run the script from `/etc/rc.local`*.*

Solaris 10 *Similar to FreeBSD, Solaris brings a tunnel implementation that provides for arbitrary IP-in-IP tunneling. Different than FreeBSD the interfaces are named depending on the type of tunnel. For a configured tunnel, we need an* `ip.tun`$\langle n \rangle$ *interface.*

On the left tunnel router the command

```
# ifconfig ip.tun0 inet6 plumb tsrc 192.0.2.1 tdst 192.0.2.2 up
```

will create ("plumb") the tunnel interface `ip.tun0` *as an IPv6-in-IPv4 interface, set the addresses of the tunnel end points and enable the interface.*

If we also want to configure a routable IPv6 address on the tunnel interface, then we need to know both the local as well as the remote address. Then an

```
# ifconfig ip.tun0 inet6 addif 2001:db8:fedc:4646::1 \
          2001:db8:fedc:4646::2 up
# route add -inet6 fe80::/10 -iface fe80::192.0.2.1        || Optional
```

will configure `2001:db8:fedc:4646::1` *and* `2001:db8:fedc:4646::2` *as the local and remote IPv6 address, respectively. The optional last line makes the tunnel conform to RFC 4213.*

If we create a file `/etc/hostname6.ip.tun0` *with the content*

/etc/hostname6.ip.tun0

```
tsrc 192.0.2.1 tdst 192.0.2.2 up
addif 2001:db8:fedc:4646::1 2001:db8:fedc:4646::2 up      || Optional
```

we make the configuration permanent. To enable the configuration we either run `svcadm restart network/initial` *or reboot. Note that this permanent configuration doesn't set up the extra route to add RFC 4213 compatibility; we'll deal with additional routes in the next section.* 66

At this point the tunnel routers should be able to reach each other using IPv6 while only IPv4 traffic is seen in the bottom subnet.

- ☐ The tunnel interfaces must be up and running.
- ☐ They must be configured with the proper IPv4 address of the remote tunnel router.
- ☐ The tunnel routers must be able to ping each other on their IPv6 addresses.

At this point we should probably make sure that there is actually only IPv4 traffic in the bottom subnet. If we start a packet sniffer like `ethereal` on the bottom subnet we will see something surprising in its packet summary buffer:

Ethereal Summary Buffer

```
 No. Time     Source          Destination     Proto. Info
   1 0.000000 fe80::c000:202  fe80::c000:201  ICMPv6 Echo request
   2 0.006032 fe80::c000:201  fe80::c000:202  ICMPv6 Echo reply
   3 1.023251 fe80::c000:202  fe80::c000:201  ICMPv6 Echo request
   4 1.023713 fe80::c000:201  fe80::c000:202  ICMPv6 Echo reply
[...]
```

So according to the packet sniffer there are IPv6 packets in our IPv4-only subnet. Even if we split the subnet and put an IPv4-only router in between we see the same phenomenon. The explanation for this surprising behaviour is as simple as unexpected: If we look at the packet details we'll see that they are indeed properly encapsulated:

Ethereal Packet Details

```
Frame 1 (90 bytes on wire, 90 bytes captured)
Ethernet II, Src: 00:0c:29:78:6c:26, Dst: 00:0c:29:ae:8c:c0
Internet Protocol, Src Addr: 192.0.2.2 (192.0.2.2),
Dst Addr: 192.0.2.1 (192.0.2.1)
Internet Protocol Version 6
Internet Control Message Protocol v6
[...]
```

Ethereal displays the *inner* addresses in its summary. This isn't precisely what we would expect in this context, but for many other purposes this behaviour is preferable over displaying the outer addresses.

12.1.3 Routing Through a Tunnel

Next we configure the tunnel routers to forward traffic between the upper IPv6 subnets. As usual, we have a choice between static and dynamic routing.

Static routing through tunnels works without surprises except that tunnels share a peculiarity with all point-to-point links: They are configured with an *interface route* that usually requires us to specify an interface name rather than a next-hop address.

In our example we configure the left tunnel router with a network route for `2001:db8:fedc:1::/64` and the right tunnel router with a default route through the tunnel.

Debian Sarge *We can set the route using either* `ip` *or* `route`*. Both support a* `dev` *keyword that lets us specify the interface instead of the IP address of the tunnel peer. With* `ip` *we add the routes like this:*

```
# ip -6 route add 2001:db8:fedc:1::/64 dev sit1      || Left Router
# ip -6 route add default dev sit1                   || Right Router
```

Similarly, `route` *does the same with a different syntax:*

```
# route -A inet6 add 2001:db8:fedc:1::/64 dev sit1   || Left Router
# route -A inet6 add default dev sit1                || Right Router
```

To remove a route again, both with `ip` *and* `route` *we just replace the* `add` *keyword with* `del`*.*

Section 7.3 already explained the limitations of `/etc/network/interfaces` *with respect to static routing. On the right tunnel router we can use a* `gateway fe80::192.0.2.1` *statement for the* `sit1` *interface. Otherwise we need to use the commands from above in* `up` *and* `down` *statements on the tunnel interface.*

FreeBSD 6.1 *We configure our routes with* `route`*, using the* `-iface` *option:*

```
# route add -inet6 2001:db8:fedc:1::/64 -iface gif0  || Left Router
# route add -inet6 default -iface gif0               || Right Router
```

To remove a route again we substitute `add` *with* `delete`*. Additionally, we don't need to provide the* `-iface gif0` *when we delete a route.*

To make the configuration permanent we add the lines

/etc/rc.conf

```
ipv6_static_routes="tunnel0"
ipv6_route_tunnel0="2001:db8:fedc:1::/64 -iface gif0"
```

on the left router. On the right router we can specify `default` *instead of the network prefix in the second line.*

Solaris 10 *The* `route` *command has an* `-iface` *option similar to FreeBSD. It is used differently, though: Instead of specifying an interface name we pass it the local inner IP address of the tunnel interface:*

```
# route add -inet6 2001:db8:fedc:1::/64 \
                   -iface fe80::192.0.2.1            || Left Router
# route add -inet6 default -iface fe80::192.0.2.2    || Right Router
```

To remove a route again we use the keyword `delete` *instead of* `add`*.*

As already explained in section 7.3, Solaris doesn't really support static routes in its boot configuration. If we really need to set up static routes, then the script mentioned there also supports the `-iface` *syntax.*

If we want to make our tunnel RFC 4213 compliant, then we need to add the route mentioned above. The command

```
# route add -inet6 fe80::/10 -iface fe80::192.0.2.1
```

ensures that arbitrary link-local addresses are reachable through the tunnel. 67

Now our tunnel should be up and running. We test it just like any routing setup:

- ☐ The tunnel routers have routing table entries configured for the remote networks.
- ☐ An IPv6 `ping6` from one of the IPv6-only hosts to the other works.

If we segment the bottom IPv4-only network and make the traffic between the two tunnel routers pass through an IPv4-only router, then we can observe why tunnels are called tunnels: Neither `traceroute6` nor the hop count of `ping6` show the intermediate IPv4 routers. Packets behave like electrons in some quantum physics experiment: They "tunnel" through the IPv4 network without showing a sign of actually "passing through the IPv4 medium".

With dynamic routing we need to make sure that link-local addresses work across the tunnel. If they do, dynamic routing works without problems either and we can let it take care of the routing tables.

The only potential problem that we may run into has to do with the way packets are "tunneled": A tunnel will appear to dynamic routing like a direct connection between the tunnel endpoints, no matter how many outer

protocol layer hops are actually between them. This can seriously confuse dynamic routing because it breaks the way metrics are used to choose the shortest route. To deal with this situation the best strategy is a network topology without redundant connections of which one route is tunneled and the other is not.

In some cases it is however appropriate to meddle with the routing metrics to fit the actual network topology. We'll see in section 25.1 how to use dynamic routing and tunnels to set up redundant Internet connectivity.

12.1.4 Automatic Tunnels

Automatic tunnels provide a very simple means to set up a tunnel host as an exit point. By themselves they only allow tunnel hosts to communicate with each other, which isn't too exciting. But if the tunnel host has a configured tunnel in the opposite direction, possibly with a default route through that tunnel, then automatic tunnels can be useful.

Automatic tunnels have the advantage that they don't need to be configured. But they are limited to tunnel hosts, so they have been made obsolete by 6to4 tunnels. In fact, some IPv6 implementations come without support for automatic tunnels.

The advantage that automatic tunnels don't need any configuration is also a security concern: An assumedly IPv4-only machine can in some cases be reached through this if no adequate filters are in place.

Tunnel hosts that are exit points of automatic tunnels use special *IPv4-compatible addresses*: If they have the IPv4 address `192.0.2.130`, then they also have the IPv4-compatible address `::192.0.2.130`. A tunnel entry point can therefore derive the IPv4 address of the destination tunnel host from its IPv6 address. These addresses are sometimes confused with IPv4-mapped IPv6 addresses (`::ffff:0:0/96`) used for dual-stacked servers, but both are entirely unrelated.

Automatic tunnels are specified in RFC 2893 [47, section 5] and declared obsolete in RFC 4213 [95, section 8].

Support for automatic tunnels varies in different Unixen.

Debian Sarge *The `sit0` interface implements automatic tunneling. It lets us set up tunnel hosts as well as tunnel routers.*

FreeBSD 6.1 *Automatic tunnels are not supported.*

Solaris 10 *The `ip.atun0` interface implements automatic tunneling for tunnel hosts only. Short of setting up individual tunnels for all associated tunnel hosts there is no support for setting up a tunnel router.* 68

Considering their limited availability, the fact that they are declared obsolete, the general availability of 6to4 tunnels as a far more versatile successor and their security issues, automatic tunnels are best avoided.

12.1.5 Security Considerations

Configured and automatic tunnels raise some security concerns. Most notably, automatic tunnels have gained a reputation as an easy way to send IPv6 traffic past IPv4-only packet filters that don't filter by protocol type.

Additionally, tunnels may be used to hide the actual source of a packet: If an attacker creates an IPv6 packet with a fake IPv6 source address and then encapsulates it using its real IPv4 address as source, this packet will pass all filters that detect bad source addresses on the way to the tunnel exit point. There it will be decapsulated and then sent to the final recipient. Due to the decapsulation the recipient can't identify the original sender.

In section 12.6 we take a look at how to configure packet filters to prevent these attacks.

If we need to make a tunnel trustworthy, then we should use proper cryptographic means. We can use IPsec (see chapter 21) or OpenVPN (see section 13.3) to provide the protection we need for our particular purposes.

12.2 6to4 Tunnels

The 6to4 tunnels specified in RFC 3056 [15] manage to combine the advantages of configured and automatic tunnels: The inner IPv6 address contains the IPv4 address of the tunnel exit point but the destination node may be different from the tunnel exit point.

6to4 uses addresses with the special unicast prefix `2002::/16`. To every globally routed unicast IPv4 address it assigns the global routing prefix `2002:`⟨*IPv4 address*⟩`::/48`. Traffic to these addresses may be routed normally; the prefix was taken from the standard `2000::/3` address range exactly for this purpose. But it may also be encapsulated and tunneled to the IPv4 address embedded in the IPv6 address.

Note that 6to4 doesn't work with the private IPv4 addresses reserved in RFC 1918 [97], `10.0.0.0/8`, `172.16.0.0/12` and `192.168.0.0/16`. Using them would introduce all the problems they cause to the IPv4 world into the IPv6 world. Similarly, 6to4 can't use multicast addresses or the loopback address as the embedded IPv4 address.

Since the embedded IPv4 address is put somewhere inside the IPv6 address we need to convert the decimal IPv4 address into a hexadecimal prefix—something like `2002:192.0.2.1::1` or such won't work. On most Unixen, a simple script like

mk6to4

```
#! /bin/sh
echo "$1" | tr '.' ' ' \
| ( read a b c d;
    printf "2002:%x:%x\n" $(($a*256+$b)) $(($c*256+$d)); )
```

takes care of turning an IPv4 address into the associated 6to4 prefix; just run it with an IPv4 address as its argument and it returns the IPv6 prefix.

12.2.1 6to4 Tunnel Hosts

To get started we first set up 6to4 tunnels between dual-stacked end hosts. Figure 12.4 shows our test environment. While it isn't particularly useful in itself, it is both the most simple configuration possible and an ideal basis for the "real" setups in the following sections.

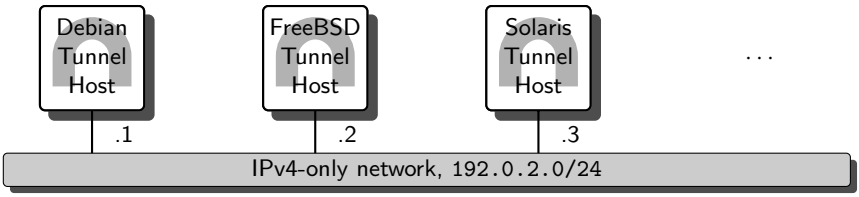

Fig. 12.4. 6to4 tunneling between hosts

Debian Sarge *The preferred method to configure the tunnel temporarily uses the* `ip` *command. 6to4 tunnels again use the* `sit`$\langle n \rangle$ *interfaces.*

```
# ip tunnel add sit1 mode sit local 192.0.2.1 ttl 64
# ip link set dev sit1 up
# ip -6 addr add 2002:c000:201::1/16 dev sit1
```

These commands first create a tunnel interface named `sit1` *with a local outer address* `192.0.2.1`*, then bring the interface up and finally configure its inner IPv6 address. Again, it is advisable but not mandatory to set an explicit TTL in the first line.*

The same configuration in `/etc/network/interfaces` *looks like this:*

/etc/network/interfaces

```
auto sit1
iface sit1 inet6 v4tunnel
        address 2002:c000:201::1
        netmask 16
        endpoint any
        local 192.0.2.1
```

Since `ip` *lets us name our interfaces anything we like, using* `sit1` *for the interface name isn't mandatory. We might as well call it* `6to4` *or whatever; doing so tends to be confusing however, so I personally prefer to use* `sit1` *which is easily identified as an encapsulating tunnel interface.*

FreeBSD 6.1 *6to4 tunnels are implemented as* `stf0` *(for six-to-four) interfaces. Using a single* `ifconfig` *invocation we can create, configure and enable such an interface:*

```
# ifconfig stf0 create inet6 2002:c000:202::1 prefixlen 16 up
```

We don't need to configure the outer address explicitly because the 6to4 address implicitly contains it.

We make the same configuration permanent in `/etc/rc.conf` *by just specifying the IPv4 address to use:*

/etc/rc.conf

```
stf_interface_ipv4addr=192.0.2.2
```

(along the usual `ipv6_enable=YES`*) to set up 6to4 support permanently.*

The FreeBSD implementation allows only a single `stf` *interface with only a single 6to4 address assigned to that interface. While this is technically considered a bug, it usually doesn't affect practical use.*

Solaris 10 *The 6to4 tunnel interfaces are called* `ip.6to4tun`$\langle n \rangle$*. To set up the interface we only need the outer IPv4 address:*

```
# ifconfig ip.6to4tun0 inet6 plumb tsrc 192.0.2.3 up
```

plumbs, configures and enables the interface.

By creating a file `/etc/hostname6.ip.6to4tun0` *with the contents*

/etc/hostname6.ip.6to4tun0

```
tsrc 192.0.2.3 up
```

we make the configuration permanent. 69

The configurations for Debian Sarge and FreeBSD 6.1 show a surprising prefix length on the tunnel: In section 3.4 we saw that all subnet prefixes must be 64 bits long and now we set the prefix length to 16. But tunnel interfaces are not regular interfaces and the 6to4 prefix isn't a subnet; since we can reach the entire `2002::/16` range through our 6to4 tunnel interface this prefix length is actually correct. Solaris deals with this differently: It sets up an interface route for `2002::/16` through the tunnel interface.

All implementations set the inner address on the tunnel interface to `2002:`$\langle$*IPv4 address*$\rangle$`::1`. While this isn't mandatory, it is a widely established practice that we shouldn't deviate from without compelling reasons.

Before we connect machines behind tunnel routers, we make sure that the tunnel hosts so far actually work as expected.

- ☐ Ensure that the tunnel hosts can reach each other via IPv4.
- ☐ Check that the tunnel interfaces are up and running.
- ☐ Verify that they use the proper IPv4 addresses as their outer tunnel addresses.

- ☐ Their inner tunnel addresses must be set to `2002:`⟨*IPv4 address*⟩`::1` unless explicitly configured otherwise.
- ☐ The routing tables must show an interface route directing `2002::/16` through the tunnel interface.
- ☐ The tunnel hosts must be able to ping each other using their 6to4 addresses `2002:`⟨*IPv4 address*⟩`::1`.

12.2.2 Tunnels Between 6to4 Sites

The next step extends our existing setup such that we replace our tunnel hosts with tunnel routers, called *6to4 border routers*, or *6to4 routers* for short, and connect additional hosts behind them. The border routers are configured to talk to each other but don't provide connectivity to non-6to4 addresses yet. Figure 12.5 shows the extended setup.

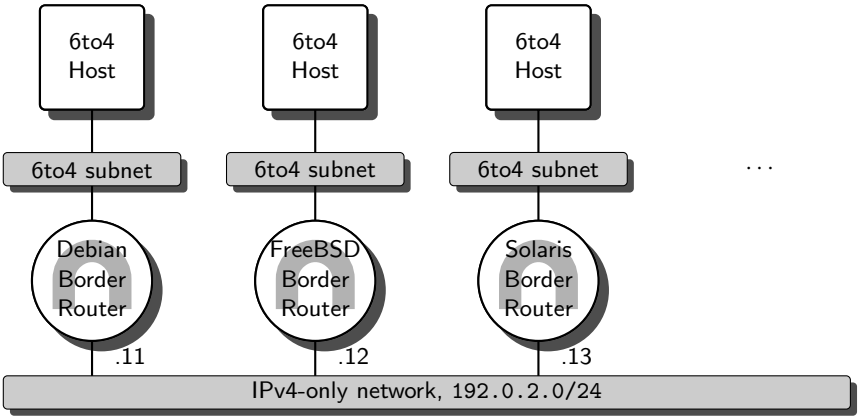

Fig. 12.5. 6to4 tunneling between 6to4 sites

First of all we need to replace our tunnel hosts with border routers and set up the upper "6to4 subnets" as well as the "6to4 hosts". A *6to4 host* is a host that happens to have an IPv6 address from the `2002::/16` address pool; otherwise it is just a plain host without any tunneling functionality (as opposed to a "6to4 *tunnel* host"). Similarly, the 6to4 subnets in the diagram are subnets where a router advertises a prefix from the 6to4 address range. A border router is a standard advertising router with a 6to4 tunnel interface, so there are two feasible lines of approach to set this up: Take our tunnel hosts and make them advertising routers or take an advertising router and add a tunnel interface. Either way will create a working border router.

In a test environment I prefer to set up an advertising router, make sure it works as expected on the "local" interface as an advertising router, and then

set up the tunneling. That way all problems will be limited to my own local environment rather than be leaking outside. But if we were going to establish 6to4 addresses in an existing environment, then it might be preferable to set up the tunnel first to make sure that the 6to4 addresses work correctly before we start to advertise them in our subnets. For now we use the first approach.

When we configure the upper interface of a border router, we need to choose a subnet prefix for the 6to4 subnet. In theory, any prefix from the global routing prefix `2002:`⟨*IPv4 address*⟩`::/48` associated with our outer tunnel address will do; only the one we configured the tunnel interface with is already used. We choose the first available one, `2002:`⟨*IPv4 address*⟩`:1::/64`. We now configure each border router as an advertising router as we have already done in section 4.3.3.

- ☐ At this point the 6to4 hosts must be able to ping "their" tunnel router using the 6to4 addresses.

Next we configure the IPv4 interface and the tunnel interface on all border routers as we have done in the previous section. The tunnel should now work as expected. Think about it: The sending host has a default route to its border router. The border router examines the destination address and realizes that it is a 6to4 address. It sends it through its tunnel interface, where it is encapsulated and passed to the IPv4 address of the receiver's tunnel router. The receiver's tunnel router decapsulates the packet, realizes that its destination is on its upper subnet and delivers it to the receiving 6to4 host.

The final test looks like this:

- ☐ The 6to4 hosts must be able to ping each other.
- ☐ A packet sniffer in the IPv4-only network must show encapsulated packets only.

The last check is actually important: As long as we use a single subnet for our IPv4-only network, we can't suppress native IPv6 traffic there. If we run a dynamic routing protocol on the border routers, then we might accidentially route native IPv6 traffic through the assumedly IPv4-only subnet, so we need to check this. Alternatively, we can split the bottom subnet and put IPv4-only routers between the tunnel routers.

12.2.3 Tunnels Between 6to4 and Native IPv6 Sites

Now we connect 6to4 sites to *native IPv6 sites*, i.e. those that don't use 6to4 addresses. To do so we need a *6to4 relay router*, a router that has both 6to4 and native IPv6 connectivity.

Figure 12.6 shows a simple test environment. The border router to the left connects the left 6to4 subnet to the IPv4 network. The relay router to

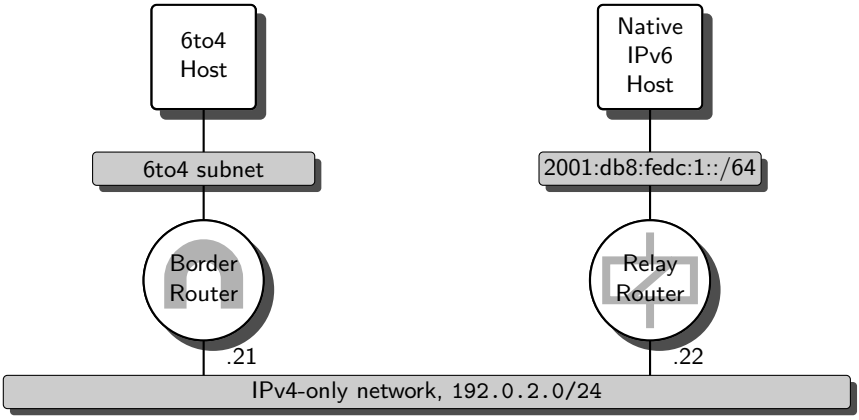

Fig. 12.6. Connecting 6to4 sites and native IPv6 sites

the right connects the right non-6to4 IPv6 subnet to all 6to4 sites through the IPv4 network.

The left border router is set up as in the previous section. Additionally we configure a route for the prefix `2001:db8:fedc:1::/64` via the relay router.

Debian Sarge *Without any obvious reason we need to configure the route using the IPv4-compatible address of the relay router instead of its 6to4 address. The command*

```
# ip -6 route add 2001:db8:fedc:1::/64 via ::192.0.2.22
```

configures the route temporarily. To make the configuration permanent we must use `up` *and* `down` *statements in* `/etc/network/interfaces` *again.*

FreeBSD 6.1 *We can add a route temporarily like*

```
# route add -inet6 2001:db8:fedc:1::/64 2002:c000:216::1
```

or set up the same route permanently in `/etc/rc.conf` *as explained in section 7.3.*

Solaris 10 *For security reasons that we'll investigate in section 12.2.7 support for 6to4 relays is severely restricted. We can only configure one relay router; doing so will set our IPv6 default route to this relay router.*

To enable the relay temporarily, we do a

```
# 6to4relay -e -a 192.0.2.22
```

to enable 6to4 relaying and to set the address of the relay router to use.

To make the configuration permanent, we add the lines

/etc/default/inetinit

```
ACCEPT6TO4RELAY=YES
RELAY6TO4ADDR=192.0.2.22
```

to `/etc/default/inetinit`. 70

The relay router configuration is almost exactly the same as a border router. We only change its configuration for the upper interface: Instead of using a 6to4 prefix we use a native IPv6 prefix here.

Solaris 10 *Due to the security concerns mentioned above, Solaris 10 implements such restrictive sanity checks that it is impossible to use Solaris 10 on a relay router.* 71

At this point it should be possible to ping between the two end hosts. If not, try to watch the ping packets with a packet sniffer in all the subnets.

- ☐ Ping the right, native host from the left, 6to4 host.
- ☐ With a packet sniffer watch the echo requests travel the subnets.
- ☐ Watch the echo replies.

At this point it is necessary to configure every border router with the addresses of the relay routers to every native IPv6 site. Effectively, we haven't won much compared to configured tunnels yet. The next section shows how to simplify the configuration again.

12.2.4 Connecting to the Internet6: Default Relay Routers

Assuming that all native IPv6 sites are connected to the Internet6, there is little reason to set up individual routes for all these sites. Instead, a border router should be able to reach the entire native IPv6 address space through a single relay router which is connected to the Internet6; we just configure the border router with a default route to the relay router.

From a configuration point of view the situation looks much like figure 12.7: Our border router just needs a default route directed towards the relay router.

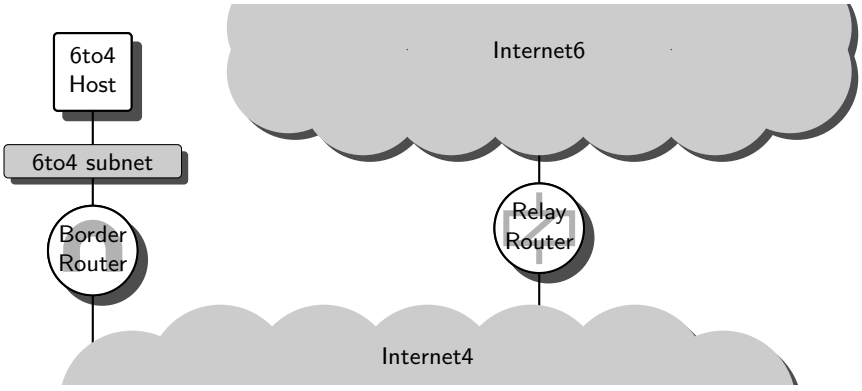

Fig. 12.7. 6to4 default relay routers—simplified case

The relay router doesn't need any configuration at all to reach any border routers because it can extract their IPv4 addresses from the IPv6 packets it forwards to them.

12.2.5 Public Relay Routers

To make life even easier, a number of *public relay routers* exist all over the Internet. They all use the same IP address `192.88.99.1` or `2002:c058:6301::`. Thanks to dynamic routing all traffic to that address will be directed to the "closest" public relay router and we don't have to worry where exactly that public relay is.

This trick, so-called "IPv4 anycast", has been used for some time with some of the DNS root servers; it does have its problems, but in many cases it works well. The end result is that we can get full Internet6 access if we just set up a border router with a default route aimed at `2002:c058:6301::`. In fact, some Unixen use this address by default.

To set up a border router that uses the next public relay router we do this:

Debian Sarge *We need to set our default route here, because there is no default setting. The configuration in* `/etc/network/interfaces` *must include at least the lines*

`/etc/network/interfaces`

```
auto sit1
iface sit1 inet6 v4tunnel
        gateway ::192.88.99.1
        netmask 16
        endpoint any
        local 192.0.2.1
```

if `sit1` *is the tunnel interface and* `192.0.2.1` *is the IPv4 address of our border router.*

FreeBSD 6.1 *The essential configuration in* `/etc/rc.conf` *is*

`/etc/rc.conf`

```
stf_interface_ipv4addr=192.0.2.1
ipv6_defaultrouter=2002:c058:6301::
```

assuming `192.0.2.1` *is the IPv4 address we use for our border router.*

Solaris 10 *To make the configuration permanent, we set up the tunnel device in* `/etc/hostname6.ip.6to4tun0` *with a line*

`/etc/hostname6.ip.6to4tun0`

```
tsrc 192.0.2.1 up
```

if the tunnel interface has the IPv4 address `192.0.2.1` *and add a line*

/etc/defaults/inetinit

```
ACCEPT6TO4RELAY=YES
```

to `/etc/defaults/inetinit` *to enable 6to4 relay support.* 72

In summary, if we have a globally routed static IPv4 address, then we automatically have an IPv6 `/48` prefix that we can use to connect to the Internet6 with a minimal effort.

Setting up a public relay router is far more demanding. In theory, a public relay router does little more than the relay routers we have set up in section 12.2.3. In practice the "IPv4 anycast" part is quite demanding. The anycast address must be announced through dynamic routing, usually with the border gateway protocol (BGP). A public relay router must also announce a route to `2002::/16` on its native IPv6 interface. If the relay router has a problem, then the routing must be updated so traffic is directed to other relay routers as quickly as possible. To detect a failing relay we need a reliable monitoring setup. If we only want to provide the relay service to certain IPv4 networks, then we must limit the distribution of our routing announcements accordingly. If anything goes wrong, there may be a huge number of people getting seriously annoyed; most particularly, announcing the anycast address to machines that are not actually allowed to use the relay (or the `2002::/16` prefix to users that you don't want to provide service for) will effectively disconnect these users from "their" 6to4 relay. If this affects other people's business, prepare for expensive lawsuits.

RFC 3068 [69] allocates the `192.88.99.1` address for the public relay routers. It also has all the ugly details about public relays in general and how to set them up properly.

12.2.6 Operational Issues

As exciting as 6to4 is, there are operational issues that are worth some consideration before we decide to use 6to4 tunnels.

With the "IPv4 anycast" address we can reach the "nearest" public relay—if everything works as expected. But in practice public relays have some problems:

A broken public relay will shadow the working second-to-next relay, making it inaccessible through the anycast address. All fallback scenarios for 6to4 rely on the assumption that the relay operators don't make the mistake of announcing a broken relay with their dynamic routing. A wide range of things may go wrong on the relay operator's side, leaving us almost entirely helpless.

It isn't even trivial to figure out who is operating our next public relay (but `traceroute` will give us some clues), how to contact them if things go wrong or who to ask if they intend to provide their relay service permanently; we might do some initial latency checks before we set up a 6to4 environment

only to find out later on that "our" relay was a temporary test setup by the local university and we now have our traffic routed around half the world to the "next" relay, which also turns out to be unstable at best.

This makes 6to4 public relays effectively useless for production-grade purposes unless we make our IPv4 provider support 6to4 tunneling. But then, if we have a dedicated 6to4 tunnel provider, we might as well configure the unicast address(es) of their relay(s) on our border router(s).

Even if we decide to use a configured relay router as we have done in section 12.2.4, using 6to4 to connect to the Internet6 has an inherent problem. In figure 12.7 on page 165 we have seen a simplified setup with a single border router and a single relay, but reality looks more like figure 12.8. A ping

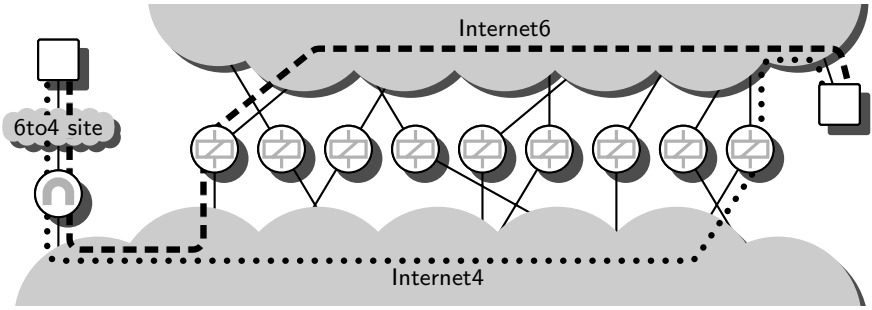

Fig. 12.8. 6to4 default relay routers getting out of control

request from the left host travels along the dashed line while a reply follows the dotted line. Asymmetric routing in general tends to make life more difficult than symmetric routing, but in this case three effects become particularly important: If we are operating the 6to4 site but not the right host and its network setup, then we have no idea what relay router the right host uses and the number of IPv6 hops from left to right is noticeably larger than from right to left, which tends to cause some confusion. If we can't ping the right host from the left host, then we don't have any idea if the problem is caused by our border router, our provider's relay router or the relay router that the right host uses. With asymmetric routing, tools like `traceroute` lose much of their usefulness; the return packets that are generated within the Internet6 cloud will all travel entirely different paths, so a missing reply doesn't tell us anything about where a problem really occurs.

A 6to4 border router needs a globally routed IPv4 address. Whenever that address changes we need to distribute a new prefix through our network. Small sites that use a semi-permanent Internet connection with a dynamically assigned IPv4 address, like many DSL customers, are forced to do a "hard" renumbering without a grace period to deprecate the old address. While this may be feasible in some cases, in most it is a serious problem. Effectively,

6to4 needs statically assigned globally routed IPv4 addresses for its border routers.

So what are 6to4 tunnels good for?

If we just want to connect multiple locations which only have Internet4 connectivity, then most concerns except for the need for static IPv4 addresses become irrelevant. This scenario is particularly useful in environments that are highly distributed with many small sites or locations, like a company network with a large number of local offices. For scenarios like these, 6to4 tunneling may be quite appropriate.

If we only want to play around with IPv6, don't have any native Internet6 connectivity and don't want to bother setting up a tunnel to a tunnel provider, then 6to4 can be an option. If the next public relay router works reasonably well, then performance may be better than a configured tunnel to a tunnel provider far away.

Finally, 6to4 works without support from our local ISP. If we are stuck with an IPv6-unfriendly ISP, using 6to4 will still gain us Internet6 access even without support from that ISP. This wins us time to find a better ISP as well as it puts pressure on our ISP, especially if we don't pay by traffic volume like they do: Eventually they may realize that they can save money providing direct Internet6 connectivity because they don't have to pay their upstream providers for the extra traffic caused by the encapsulation headers.

In summary, relay routers, and especially public relay routers using the 6to4 anycast address, are potentially troublesome and shouldn't be considered for production-grade environments. For experimenting they are a useful tool. 6to4 tunnels between 6to4 sites save us a lot of configuration effort especially with larger environments and are not affected by the problems that relay routers introduce.

12.2.7 Security Considerations

6to4 tunnels raise a number of security problems.

From a 6to4 site operator's point of view the most significant problems are those we have already seen with configured and automatic tunneling in section 12.1.5: 6to4 tunnels may be used to hide the actual source of a packet, possibly for denial-of-service or reflection denial-of-service attacks.

The public relay router anycast mechanism may be used to redirect traffic to a relay router controlled by an eavesdropper or to disrupt the service. The problems mentioned in the previous section may be caused on purpose.

In general, 6to4 tunneling should not be used without proper packet filtering as explained in section 12.6.

For additional information, check RFC 3964 [99]. It presents an overview of the security implications of 6to4 tunnels in general.

12.3 Tunneling Over IPv6 Networks

RFC 2473 [17] defines a standardized way to encapsulate traffic over IPv6 networks in a way very similar to the configured tunnels in section 12.1.2; in fact, some implementations use a generic tunneling interface for all IP-in-IP encapsulation.

12.3.1 IPv4-in-IPv6 (4in6) Encapsulation

While it may seem a bit early to consider encapsulating IPv4 in IPv6, many implementations already support this. The protocol type in the next header field is 4 for "encapsulated IPv4" as explained in RFC 1853 [101].

To configure a tunnel we first need to set up IPv6 connectivity between the tunnel endpoints. Then we create and configure the tunnel pseudo-interface. In the example configuration in figure 12.9 the left router's addresses are `2001:db8:fedc:1::1` and `10.0.0.1` as shown in the configurations below; the right router's addresses are `2001:db8:fedc:1::2` and `10.0.0.2` as the outer and inner tunnel addresses, respectively. The right router should be configured the same as the examples below with the local and remote addresses swapped.

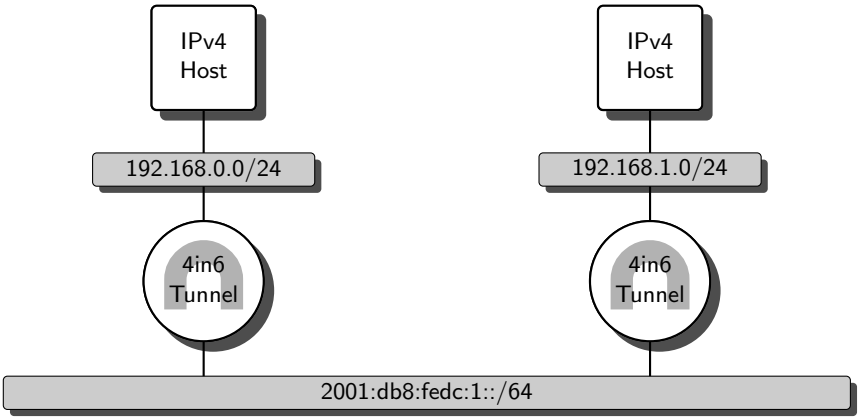

Fig. 12.9. A test setup tunneling IPv4 in IPv6

Debian Sarge *At this time, there is no implementation for IPv4-in-IPv6 encapsulation.*

FreeBSD 6.1 *The* `gif`$\langle n \rangle$ *interface supports arbitrary IP-in-IP encapsulation including IPv4-in-IPv6. To configure a tunnel endpoint temporarily,*

```
# ifconfig gif0 create tunnel \
                2001:db8:fedc:1::1 2001:db8:fedc:1::2 up
# ifconfig gif0 10.0.0.1 10.0.0.2
```

creates the interface, configures the outer addresses and enables the tunnel in the first line and then configures the inner addresses in the second.

Unfortunately the boot scripts can't deal with this setup properly, so it is easiest to put these commands in `/etc/rc.local` *to make the configuration permanent.*

Solaris 10 *The tunnel interfaces called* `ip6.tun`$\langle n \rangle$ *provide for encapsulation with IPv6 as the outer protocol. To configure a tunnel temporarily,*

```
# ifconfig ip6.tun0 plumb \
           tsrc 2001:db8:fedc:1::1 tdst 2001:db8:fedc:1::2 \
           10.0.0.1 10.0.0.2 up
```

creates the interface, sets the outer and inner addresses and then brings up the interface.

The same configuration made permanent needs the lines

`/etc/hostname.ip6.tun0`

```
tsrc 2001:db8:fedc:1::1 tdst 2001:db8:fedc:1::2
10.0.0.1 10.0.0.2 up
```

in `/etc/hostname.ip6.tun0`*.* 73

If we just want to set up a tunnel host, this is all it takes. The checklist looks much like for a configured tunnel:

- ☐ Both nodes must be able to ping each other using IPv6.
- ☐ Their tunnel interfaces must be up and running.
- ☐ They must have the proper IPv6 addresses for their outer tunnel addresses.
- ☐ The inner IPv4 addresses must be set correctly.
- ☐ The nodes must be able to ping each other's IPv4 address.
- ☐ A packet sniffer must show that the packets between the tunnel nodes are actually encapsulated.

If we want to provide a tunnel service to other nodes, then we need to make our tunnel node an IPv4 router. In theory, doing so doesn't pose any problems: We set up the local IPv4 configuration, enable packet forwarding and configure our IPv4 routes, either statically or using a dynamic routing protocol.

Surprisingly enough this causes an unexpected little problem at least with Solaris and FreeBSD (and possibly other Unixen as well): The boot scripts don't configure static routes or default routes through a tunnel interface, apparently because the routes are configured before the tunnels are set up.

FreeBSD 6.1 *Since we already need to set up our tunnels in* `/etc/rc.local` *or similar, it is easiest to set the static routes there, too. For a default route in our example a line*

`/etc/rc.local`

```
/sbin/route add default 10.0.0.2
```

in `/etc/rc.local` *works as expected.*

Solaris 10 *Unfortunately, an entry in* `/etc/defaultrouter` *doesn't work. We can either use the workaround mentioned in section 7.3 or use dynamic routing across the tunnel.* 74

Dynamic IPv4 routing across these tunnels works without problems. To make sure that our setup works as expected, we first check that the tunnel routers can reach each other using IPv4. Then we do some more checks:

- ☐ If we use dynamic routing, on both tunnel routers `ps` must show a running routing daemon.
- ☐ The routers must have appropriate routing table entries for the network(s) they reach through the tunnel. (With dynamic routing it may take some time for these to appear—up to 45 seconds per hop in the peer network.)
- ☐ Nodes in the networks behind the tunnel routers must be able to ping each other across the tunnel.

12.3.2 IPv6 in IPv6 (6in6) Encapsulation

RFC 2473 [17] also defines how to encapsulate IPv6 in IPv6. While this may appear useless at first glance, it actually comes in very handy when we need to set up redundant Internet6 connectivity as we'll see in section 25.1.

In theory, configuration is almost exactly the same as in the previous section except that the inner addresses are now also IPv6 addresses instead of IPv4 addresses. In practice, keeping the inner and outer addresses and interfaces from interfering with each other can be tricky especially when dynamic routing comes in.

Before we set up a tunnel we need to configure the direct IPv6 connectivity between the tunnel nodes. For the time being we use static routing here; without extra configuration dynamic routing mistakes a tunnel for a direct link and behaves suboptimal at best.

Eventually we'll try to use dynamic routing through the tunnel. This forces us to put the routers in separate subnets connected by another router that doesn't run a dynamic routing protocol; otherwise the tunnel routers may use dynamic routing between each other and route the traffic to each other directly instead of using the tunnel. Figure 12.10 shows a minimal test setup.

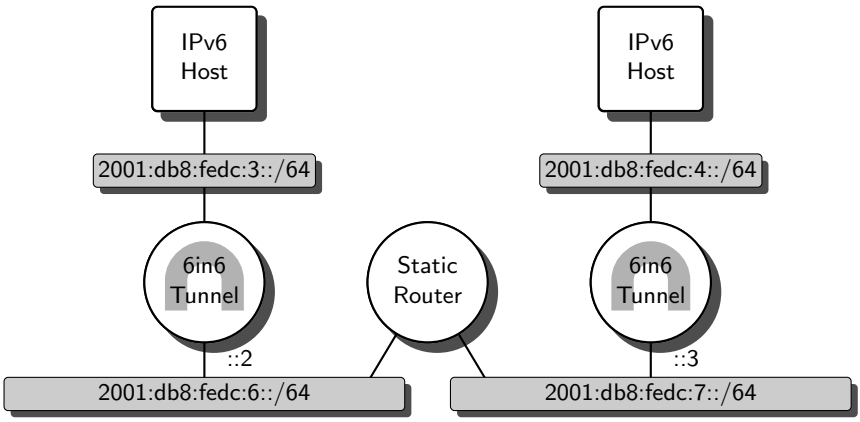

Fig. 12.10. A simple 6in6 tunnel environment

Before we can set up the tunnel we need to configure the static router and tunnel routers.

- ☐ The tunnel routers must be able to ping each other using their addresses from the lower networks.

Next we set up the tunnel interfaces of the tunnel routers. Since we want to use static routes first, we configure the tunnel interfaces with explicit inner addresses `2001:db8:fedc:66::2` and `2001:db8:fedc:66::3`. For the left tunnel router the configuration looks like this:

Debian Sarge *There is a kernel module for 6in6 tunnels but the userland tools to configure it are still missing from the distribution.*

FreeBSD 6.1 *To configure the interface `gif0` temporarily, the commands*

```
# ifconfig gif0 create
# ifconfig gif0 inet6 tunnel 2001:db8:fedc:6::2 2001:db8:fedc:7::3
# ifconfig gif0 inet6 2001:db8:fedc:66::2 up
```

create the interface, set the outer tunnel addresses, configure the local inner address of the tunnel and bring the interface up.

For a permanent configuration we need to run the same commands from a custom boot script or put them in `/etc/rc.local` again.

Solaris 10 *We can configure the interface `ip6.tun0` temporarily with the commands*

```
# ifconfig ip6.tun0 inet6 plumb
# ifconfig ip6.tun0 inet6 tsrc 2001:db8:fedc:6::2 \
                          tdst 2001:db8:fedc:7::3
# ifconfig ip6.tun0 inet6 2001:db8:fedc:66::2 \
                          2001:db8:fedc:66::3 up
```

to create the interface, set the outer and inner tunnel addresses and bring the interface up. The line

/etc/hostname6.ip6.tun0

```
tsrc 2001:db8:fedc:6::2 tdst 2001:db8:fedc:7::3 \
2001:db8:fedc:66::2 2001:db8:fedc:66::3 up
```

in /etc/hostname6.ip6.tun0 establishes the same configuration permanently. 75

Analogously we must configure the right tunnel.

- ☐ At this point we must be able to ping the tunnel nodes across the tunnel.
- ☐ A packet sniffer must show that the traffic through the bottom subnets is actually encapsulated.

If our tunnel nodes weren't routers but hosts, the configuration would be complete now. But since they are set up as routers, we need to add static routes to them for each opposing upper network.

FreeBSD 6.1 *We set the route using the* `route` *command like*

```
# route add -inet6 2001:db8:fedc:4::/64 -iface gif0
```

either temporarily or permanently from our boot script (or `/etc/rc.local`*). Since the* `gif0` *interface won't be up until after the standard boot scripts have run, putting a static route in* `/etc/rc.conf` *unfortunately doesn't work even if we specified the peer address instead of the tunnel interface as the next hop.*

Solaris 10 *The command*

```
# route add -inet6 2001:db8:fedc:4::/64 -iface 2001:db8:fedc:66::2
```

sets up the route temporarily. To make it permanent we must use the workaround from section 7.3 again. 76

We give the tunnel setup a final check:

- ☐ Nodes from behind the tunnel must be able to ping each other.
- ☐ A packet sniffer in the bottom subnets must show that the traffic between the tunnel endpoints is encapsulated.

If we want to use dynamic routing across the tunnel, then we need to configure it slightly differently. The RIPng specification insists that all RIPng traffic originates from link-local addresses, so we need to use link-local addresses on the tunnel interfaces.

Solaris 10 *This leads to a problem similar to the one we have already encountered with configured tunnels in section 12.1.1: The configuration computes the link-local address of the peer from its outer tunnel address by combining the* `fe80::/64` *prefix with the interface ID of the outer address. Other*

implementations, like FreeBSD, use a different link-local address. We can work around this problem by setting a route for `fe80::/10` through the tunnel as we have done in section 12.1.2. (77)

Again we first configure the bottom interfaces of the tunnel routers so that they can reach each other without the tunnel.

Then we set up the tunnel. This time however we use link-local addresses as the inner tunnel addresses.

FreeBSD 6.1 *If we bring up the tunnel interface `gif0` with*

```
# ifconfig gif0 create inet6 \
          tunnel 2001:db8:fedc:6::2 2001:db8:fedc:7::3 up
```

either manually or from `/etc/rc.local`, then `ifconfig` will configure the tunnel with a link-local address.

Solaris 10 *The command*

```
# ifconfig ip6.tun0 inet6 plumb \
          tsrc 2001:db8:fedc:6::2 tdst 2001:db8:fedc:7::3 up
```

plumbs the tunnel, configures the outer tunnel addresses and enables it. The equivalent permanent setup looks like

`/etc/hostname6.ip6.tun0`

```
tsrc 2001:db8:fedc:6::2 tdst 2001:db8:fedc:7::3 up
```

in `/etc/hostname6.ip6.tun0`. If the tunnel peer uses a link-local address different than the one Solaris expects, we also need to set an interface route for `fe80::/10` through the tunnel as we did in section 12.1.2. (78)

With the tunnel set up and running we need to enable or start the dynamic routing as explained in section 7.4. If we have done a temporary configuration, we may also need to restart the routing daemon to find the new interface.

After the dynamic routing has had some time to update the routing tables, it should be possible to use the tunnel.

- ☐ The dynamic routing daemons must be running.
- ☐ All of the previous checks for static routes must be met.

At this point we still don't run dynamic routing between the tunnel routers and the middle non-tunnel router. In theory, using dynamic routing both through the tunnel and the bottom subnets is perfectly reasonable; in practice it causes unanticipated effects. Consider the slightly larger example in figure 12.11: If we want to send a packet from any router to the right tunnel router it should travel straight to the right. But dynamic routing detects a "direct link" between the two tunnel routers, so routers 1 through 4 will actually send their packets left to the left tunnel router: Dynamic routing assumes that the shortest path for the packet from router 4 to the right tunnel router

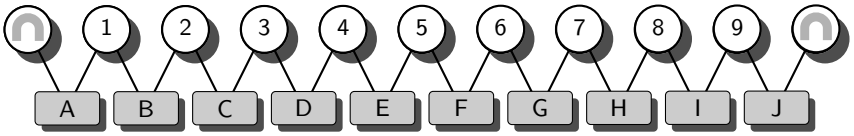

Fig. 12.11. A tunnel environment that confuses dynamic routing

takes five hops, through the subnets D, C, B, A and the tunnel, which appears to the dynamic routing one hop shorter than the untunneled path through the subnets E, F, G, H, I and J.

Some dynamic routing implementations let us set a metric for every connection. With RIPng we might want to set the distance metric between the tunnel routers to the number of intermediate subnets that the packet is tunneled through, in our example 10 hops. Doing so the dynamic routing algorithm would behave reasonably again, at least as long as the "untunneled" distance between the tunnel routers doesn't change.

Unfortunately, lightweight RIPng implementations don't offer this feature. And even then, with a maximum metric of 15 hops RIPng quickly reaches its limits if we start to manipulate the distance metrics. Without some more powerful routing protocols, using dynamic routing both through a tunnel and between the tunnel routers is usually a bad idea. Chapter 17 explains how to use OSPF with IPv6; for this purpose it is a far superior choice.

Finally, some any-in-IPv6 tunnel implementations have another surprising feature to offer: It is sometimes possible to use the same tunnel for IPv4 and IPv6 traffic. But using a single tunnel for both IPv4 and IPv6 as the inner protocol generally doesn't offer any advantages over using two separate tunnels and tends to confuse things quite noticeably, so this is a feature better to avoid.

12.4 6over4 Tunnels

RFC 2529 [14] specifies another encapsulating tunnel: *6over4 tunnels*, not to be confused with 6to4 tunnels. At this time, implementations are rare to non-existent at least in the Unix world. For completeness sake, and for those who need to set up packet filters to protect networks with 6over4-capable machines, we should still take a quick look at this tunneling mechanism.

6over4 tunnels treat the outer IPv4 protocol as a multicast-capable link layer. The IPv6 link-local address associated with an IPv4 "link-layer" address `192.0.2.1` is `fe80::192.0.2.1`. Multicast IPv6 addresses are mapped to `239.192.0.0/16` with the last two bytes of the IPv6 multicast address forming the last two bytes of the IPv4 multicast address, too.

Since 6over4 assumes IPv4 to be a multicast-capable link layer, the underlying IPv4 infrastructure must provide for IPv4 multicast routing. This alone is probably reason enough why 6over4 never became popular enough to be widely implemented.

12.5 The Intra-site Automatic Tunnel Addressing Protocol (ISATAP)

RFC 4214 defines the *intra-site automatic tunnel addressing protocol* (*ISATAP*). It allows sparsely distributed IPv6 hosts in an IPv4-only infrastructure to communicate with each other as well as the Internet6.

ISATAP is available with current versions of Microsoft Windows but isn't generally available to the Unix world.

Similar to 6over4 it uses IPv4 as a link layer. But different than 6over4 it doesn't depend on IPv4 multicast routing. Without multicast routing, neither neighbor discovery nor router discovery work as before.

ISATAP uses special interface IDs of the form `0:5efe:`⟨*IPv4 address*⟩ and `200:5efe:`⟨*IPv4 address*⟩. With these interface IDs, neighbor discovery becomes unnecessary because the link layer (aka. IPv4) address of any ISATAP node can be extracted from its IPv6 address.

Replacing router discovery with a multicast-independent mechanism is more difficult. ISATAP uses a "potential router list" that must be initialized somehow using external means; these may be manual configuration, a DNS entry like `isatap.example.com` or some DHCP extension. Using this list, ISATAP sends unicast router solicitations to all the candidate routers.

The purpose of ISATAP is to provide IPv6 connectivity to individual nodes in a non-IPv6 environment. Today this scenario appears to be somewhat unrealistic; a small number of individual nodes could be taken care of using configured tunnels. When the number of IPv6 nodes reaches the point where configured tunnels become a burden, then it is normally time to deploy native IPv6 support.

12.6 Packet Filter Considerations

Packet filters and tunnels don't generally mix well. Configurations tend to be complex, difficult to comprehend and therefore prone to misconfiguration. Beyond that some inherently unsolvable problems exist that we must deal with at the architectural level.

12.6.1 Fundamental Problems

For a packet filter, tunnels cause a disturbing number of serious problems. In some cases it may be possible to mitigate them, but there is no general solution.

Tunnels make it easy to spoof addresses. When a tunnel exit point decapsulates a tunneled packet, it discards the outer header of the packet. An attacker may use some source address in the outer header to pass the packet through a packet filter that checks the still encapsulated packet. Once the packet is decapsulated this address is lost, so subsequent packet filters can't ensure that the inner and outer addresses correspond.

Even if we had a (currently non-existent) packet filter implementation that let us check on the inner and outer addresses, an attacker might use nested tunnels to circumvent even these checks—a packet filter would need a means to deal with non-constant nesting levels, possibly with a mixture of IPv4 and IPv6 headers.

If we have an entire tunnel router "farm", repeatedly encapsulated packets can be sent to one tunnel router, be decapsulated there for the first time, then on to another tunnel router and so on until they actually arrive at the final victim node. Tracing back such a packet will be extremely tedious.

A major job of packet filters is to ensure that a packet with a given source address arrives through the correct interface; with tunnels an attacker can circumvent standard routing mechanisms, and therefore anti-spoofing ingress filters. The problems are similar to source routing attacks with IPv4.

With automatic tunnels, 6to4, 6over4 and ISATAP there is always a chance that a misconfigured client accepts these packets. These packets can't be filtered except on the tunnel exit point or by denying all 6in4 traffic.

12.6.2 Manageable Special Cases

In some special cases it is possible to combine packet filters and tunnels in a reasonable way.

Nested tunnels should be avoided whenever possible. There are very few situations that put them to good use, and most of these could be dealt with more appropriately with a sanitized network topology.

If we only use a tunnel to connect to the Internet6 through a tunnel service provider, then packet filter configurations are reasonably straightforward: We simply won't trust anything coming in through the tunnel, much like we don't trust anything coming in from our untunneled Internet4 uplink.

In a more generalized way we can use packet filters and tunnels if we don't trust anything coming out of a tunnel interface—or don't trust it more than the networks we allow tunneled traffic to come in from.

The opposite also holds true: If we only allow tunnels from trustworthy interfaces, then we can trust the tunneled traffic as much as we trust all of these interfaces together.

If however we use "trusted" and "untrusted" interfaces and tunnels together on a single router, then all bets are effectively off; todays packet filters simply can't cope with such a situation. The only thing we can do is split the tunnels over multiple filtering routers, each dealing with only a single "security domain". With encapsulating tunnels, cryptographic authentication is not a solution to this problem since encapsulation doesn't provide authentication.

To deal with the "configuration-free" encapsulation mechanisms, i.e. automatic tunnels, 6to4, 6over4 and ISATAP, the only viable strategy is to deny all encapsulation in all networks except between the tunnel nodes. Even then, these tunnel mechanisms may be used to attack a node connected to the same subnet as the attacker, but filtering encapsulated packets on a router at least prevents remote attacks. A network topology that restricts tunnels to a minimal number of subnets and separates ordinary workstation-style hosts from these subnets will protect from these attacks. To deal with local attacks, all nodes that are not set up as a tunnel exit point must locally block encapsulated traffic.

12.6.3 Configurations

Showing full example configurations for all the tunnel types we have seen in this chapter is infeasible. Instead we take a look at a number of criteria to filter for and how to configure a packet filter for each of these criteria.

One of the good things about encapsulation and packet filters is that a default "deny all" rule will block encapsulated packets unless we explicitly permit them.

To deal with encapsulated packets on a packet filter we need to allow traffic by the protocol type in the IP header.

Debian Sarge *The* `iptables` *and* `ip6tables` *commands both support an option* `--protocol` ⟨*proto*⟩*, or* `-p` ⟨*proto*⟩ *for short. With* `-p ipv6` *or* `-p 41` *we filter for IPv6 as the inner protocol. Similarly, with* `-p ipencap` *or* `-p 4` *we can filter for IPv4 as the inner protocol.*

If the packet filter just passes the traffic to a separate tunnel node, then we just need to add matching filter rules to the `FORWARD` *chain.*

If the packet filter is configured on the tunnel node itself, then things get slightly confusing. The encapsulated packets bear a local address, so they are run through the `INPUT` *and* `OUTPUT` *chains. The unencapsulated packets are only run through the* `INPUT` *and* `OUTPUT` *chains if they bear a local address, but through the* `FORWARD` *chain if the tunnel node is a tunnel router and the unencapsulated packets are forwarded. If you think about it for a moment you'll see the logic behind this.*

FreeBSD 6.1 *The* `pf` *configuration supports a filter option* `proto` ⟨*proto*⟩ *that filters the inner protocol. We can use* `proto ipv6` *or* `proto 41` *to filter IPv6 and* `proto ipencap` *or* `proto 4` *to filter IPv4 as the inner protocol.* 79

Generally, unencapsulated packets go in and out of the tunnel interface and encapsulated packets only pass through the physical interface(s).

We can filter configured, automatic and 6to4 tunnels this way. They only differ in the addresses they use, so we just need to configure the source and destination addresses accordingly. At this time it isn't feasible to filter 6over4 or ISATAP; if we allow 6in4 encapsulation, then we let these through, too.

Filtering 4in6 and 6in6 tunnels follows the same pattern as configured tunnels, only the protocol type (with 4in6) and the outer addresses differ.

13

Other Tunneling Methods

Besides encapsulating tunnels a range of more general tunneling mechanisms exist.

The generic routing encapsulation (GRE) is a tunnel protocol popular with some dedicated router hardware. The IETF has recently released the specification of a UDP based tunnel protocol called Teredo. OpenVPN is a VPN implementation that has become quite popular recently; besides its cryptographic features it provides a way to tunnel IPv6 through NAT gateways.

13.1 GRE

RFC 2784 [40] specifies the *generic routing encapsulation* (*GRE*), a tunneling mechanism that tunnels "everything over everything"—not limited to the IP world. This extremely generic tunneling protocol inserts an extra eight bytes between the outer network layer header and the tunneled packet. With IP as the outer protocol the protocol version (with IPv4) or next header field (with IPv6) is set to 47.

Otherwise GRE behaves just like a generic IP-in-IP encapsulation. The reason to use it, in spite of the extra overhead, is usually a Cisco router at the other end; the Cisco GRE implementation allegedly has better performance than their configured tunnels.

Debian Sarge *The undocumented* `gre`$\langle n \rangle$ *interfaces implement GRE tunnels. We can configure them using the* `ip` *like a configured tunnel if we substitute the* `mode sit` *option with* `mode gre`.

There is no support in `/etc/network/interfaces` *for GRE tunnels, so we must supply our own boot script or use* `up` *and* `down` *statements to configure the tunnel.*

The interfaces don't configure link-local addresses, so we must assign them addresses ourselves.

FreeBSD 6.1 *The GRE interfaces are called* `gre`⟨*n*⟩*. They are configured mostly like* `gif` *interfaces. There is a minor difference, though: When configuring the inner addresses, the local address needs an explicit* `/128` *prefix length if it is an IPv6 address. So the commands to set up an IPv6-GRE-IPv4 tunnel, to tunnel IPv6 over IPv4 using GRE, the necessary commands to set up the tunnel are*

```
# ifconfig gre0 create tunnel 192.0.2.1 192.0.2.129
# ifconfig gre0 inet6 2001:db8:fedc:6666::1/128 \
                      2001:db8:fedc:6666::2 up
```

This creates the tunnel interface, configures the outer addresses (using IPv4 addresses `192.0.2.1` *for the local and* `192.0.2.129` *for the remote address here), then configures the inner IPv6 addresses (*`2001:db8:fedc:6666::1` *for the local and* `2001:db8:fedc:6666::2` *for the remote side) and brings the interface up.*

If no inner addresses are configured, link-local addresses are automatically assigned. If we want to run dynamic routing over the tunnel, then we may decide not to configure any explicit addresses.

There is no support for `gre` *interfaces in* `/etc/rc.conf` *so all configuration must be done in* `/etc/rc.local` *or a custom boot script.*

Tunneling doesn't work if the tunnel is between two nodes in the same subnet at least if IPv6 is tunneled over IPv4. This bug is probably irrelevant in most cases but turned up when I was experimenting with this tunnel type.

Solaris 10 *There is no GRE implementation available.* 80

13.2 Teredo

Another tunnel protocol that is designed to penetrate NAT gateways as a "last resort" is *Teredo*. The official specification was only recently released as RFC 4380 [70]. Microsoft has previously shipped their current Windows with an implementation based on a draft of the RFC. The *Miredo* project has implemented Teredo for Linux and the BSDs; the distributions don't include a package, but `http://www.simphalempin.com/dev/miredo/` has both Debian packages and sources.

Teredo is UDP based, using port number 3544. Implementations based on the draft use the preliminary `3ffe:831F::/32` address prefix allocated to Microsoft; RFC 4380 officially allocated the prefix `2001::/32` for Teredo.

A Teredo client will initially contact a Teredo server to obtain its IPv6 address. Embedded within this address is the IPv4 address of the Teredo server. If Teredo clients want to reach each other, they query the Teredo server for the current address of their peer. Depending on the kind of NAT

used, the Teredo server will contact the destination client and make it initiate a connection to set up the necessary NAT state.

Teredo clients and native IPv6 nodes can communicate with each other using Teredo relays. These relays have both IPv4 and native IPv6 connectivity and announce a route to the Teredo prefix on their IPv6 interface.

The problem that Teredo addresses is fairly uncommon today: Small sites that are stuck behind a non-IPv6 NAT border router are easier connected using OpenVPN behind that border router or a NAT-penetrating tunnel broker as we'll see in section 14.1. Sites reaching a size where performance becomes enough of an issue to prevent the use of OpenVPN can set up border routers to get IPv6 connectivity either natively or through configured or 6to4 tunnels.

13.3 OpenVPN

Probably the most unusual tunnel mechanism in this chapter is *OpenVPN*: It isn't really meant as a tunnel protocol as such but as a *virtual private network* (*VPN*) security solution. It runs in user space, not within the kernel, so its performance can't compete with kernel-based tunnels like encapsulation or GRE. Since it tunnels IPv6 over UDP, it bloats the tunnel packets more than the encapsulating tunnels. But these handicaps let it operate through NAT gateways and make it portable, so it works on a wide range of operating systems. These two features make it a unique tool, even though its IPv6 support is still limited.

OpenVPN traditionally runs as a peer-to-peer protocol with two equally configured peers talking to each other. Since OpenVPN 2.0 there has been support for a "server mode" where a single server operating on a single port waits for an arbitrary number of clients to initiate a tunnel; but this operation mode doesn't support IPv6, so it is currently of little value in an IPv6 scenario.

Throughout this section we talk about a "peer-to-peer configuration" if both tunnel nodes are able to send an initial packet to the other peer. If a NAT gateway is placed between them, then only the node "behind" the NAT gateway can initiate a tunnel; we call this a "client/server configuration". The "server" is the node which can be initially reached from a "client" and the "client" is the node behind the NAT gateway that must initiate the tunnel setup. The client may be located behind a NAT gateway since the initial tunnel setup packet will also set up the necessary translation state on the NAT gateway(s) between the client and server.

OpenVPN runs on a configurable UDP or TCP port, by default on 1194/UDP. OpenVPN can tunnel either as an IP layer router or as a link-layer bridge. It supports data compression using the LZO library. Besides statically configured shared secrets it offers SSL/TLS based public key authentication and strong encryption.

Installing OpenVPN on most Unixen is quite straightforward. If it isn't included as a binary package, building it from source poses no real problem. The sources are freely available from `http://www.openvpn.net/`.

Debian Sarge *The package* `openvpn` *contains OpenVPN 2.0. The boot script expects the configuration file to be* `/etc/openvpn/openvpn.conf`*.*

FreeBSD 6.1 *The ports and packages collections bring OpenVPN 2.0.5. To enable OpenVPN we add a line*

`/etc/rc.conf`

```
openvpn_enable=YES
```

to `/etc/rc.conf`*. As usual with FreeBSD ports/packages, the configuration file is in* `/usr/local/etc/openvpn/openvpn.conf`*.*

Solaris 10 *The tunnel device available for Solaris 10 doesn't support IPv6, so OpenVPN doesn't either.* 81

To test OpenVPN we need a test environment with a NAT gateway. Figure 13.1 shows the setup that we use in the following examples.

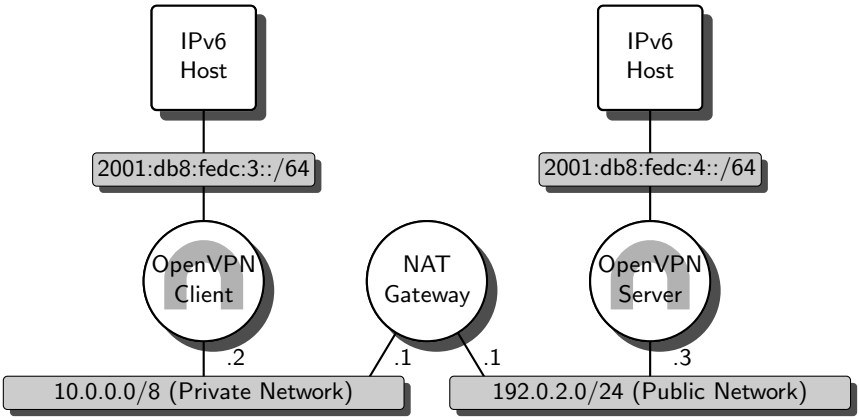

Fig. 13.1. OpenVPN in a NAT environment

Before we set up the OpenVPN tunnel we configure the tunnel routers as advertising routers on their upper interfaces. We also configure the bottom interfaces of the routers using the IPv4 addresses shown in the diagram. Finally we set up the NAT gateway with its right interface as its public/NAT interface.

To make sure that the configuration so far works as expected we run a few tests:

- ☐ The IPv6 hosts must receive their subnet prefixes from the routers.

- ☐ IPv6 forwarding must be enabled on the tunnel routers.
- ☐ The bottom interfaces of the tunnel routers must be up, running and configured with their IPv4 addresses.
- ☐ An IPv4 ping from the left tunnel router to the right must work.
- ☐ An IPv4 ping from the right tunnel router to the left must *not* work; otherwise the NAT gateway doesn't do any address translation but forwards the traffic unchanged.

Now we need to configure OpenVPN to set up the tunnel. In the simplest case, the lines

openvpn.conf

```
daemon                          || Make OpenVPN run as a daemon
dev tun                         || Use IP tunnel, not Ethernet tunnel
tun-ipv6                        || Support IPv6 tunneling
remote 192.0.2.3                || Address or DNS name of peer
up "sh ${config%.conf}.up"      || Run script to configure interface
```

in `openvpn.conf` do the job. While the first three lines are self-explaining, the last two deserve some more consideration: The `remote` statement defines the address of the peer and optionally the port number to use and the `up` statement runs a script after we have brought up the interface.

If we operate without a NAT gateway in between, either peer can initiate a tunnel setup. In that case we would configure each with the IP address or DNS name of the other as with the `remote` line shown above. With that we are settled. But as soon as NAT comes in, things become more difficult. On the client we need a configuration like

openvpn.conf

```
remote 192.0.2.3                || Address of peer; must be configured on client
ping 30                         || Ping peer, to maintain NAT state
```

in `openvpn.conf`. Here we need the client, that is the peer "behind" the NAT gateways, to initiate the tunnel. We do so by specifying the address of the server as before. Additionally we must keep the tunnel "alive" by continuously sending packets before the NAT gateways drop the address mapping. To do so we make the client ping the server every thirty seconds. On the server side we just omit the `remote` statement so it will wait for an incoming packet from the client.

OpenVPN doesn't configure the tunnel interface by itself, so we need to do it with a little script that is run whenever a tunnel is established. The cryptic `up` statement above does exactly this. It executes a shell script named and located like the configuration file but with the suffix `.up` instead of `.conf`. The script looks different, depending on the particular Unix and our preference for static or dynamic routing. OpenVPN sets some environment variables before it starts the script; in these scripts we only use `$dev` which contains the name of the tunnel device.

Debian Sarge *Linux expects us to supply an explicit IPv6 address to the tunnel interface. While that is fine with static routing, it leaves us with a minor problem if we need a link-local address for RIPng. If we run a script like*

openvpn.up

```
localaddr="‘ip -6 addr show eth0
           | sed ’/inet6 fe80::/!d;s/.*inet6 fe80::/fe80::/;sX/.*XX’
           | head -n 1‘"
/sbin/ip -6 link set $dev up
/sbin/ip -6 addr add $localaddr dev $dev
/sbin/sysctl -w net.ipv6.conf.$dev.use_tempaddr=-1
/sbin/ip -6 route add default dev $dev
```

from `openvpn.up`*, then we just “recycle” the link-local address of* `eth0`*. If we want to configure the local address ourselves, then we can simply remove the first three lines and assign the local address we prefer to* `localaddr`*. The* `sysctl` *invocation reduces the (harmless) error messages the kernel occasionally logs about the EUI64 identifier of the tunnel interface. Finally we can append some statements that set up static routes as in the last line.*

FreeBSD 6.1 *We can simply bring up the interface without an explicitly configured IPv6 address using the commands*

openvpn.up

```
/sbin/ifconfig $dev up
/sbin/route delete inet6 default || true
/sbin/route add -inet6 default -iface $dev
```

in `openvpn.up`*. The first line brings the interface up and the optional second and third line set up a static default route.* 82

At this point our tunnel through the NAT gateway should be functional.

- ☐ The tunnel nodes have their tunnel interfaces up, running and configured with the expected local addresses.
- ☐ The tunnel nodes are able to ping each other using their inner tunnel addresses.
- ☐ A packet sniffer shows the periodic ping packets from the client to the server.
- ☐ The hosts behind the tunnel can ping each other.
- ☐ Even after several minutes of inactivity the server can ping the client without prior traffic.

If a server needs to support multiple clients, then we need to do a bit more configuration. On the server side we need to run a separate `openvpn` instance for every client. Every instance needs to run on a different port number. For the server the additional configuration option can be added to the configuration file using a line

openvpn.conf

```
port 6666
```

or with the command line option `--port 6666`. The client needs an extended format of the `remote` specification like

openvpn.conf

```
remote 192.0.2.3 6666
```

or an additional command line argument `--remote 192.0.2.3 6666` to contact the server on port 6666.

This setup is still tedious; with IPv4, OpenVPN provides a more comfortable and efficient way to set up servers, but with IPv6 we are currently forced to have the server run a separate daemon for every client.

OpenVPN has a number of additional features—most notably the encryption of traffic. Since these are independent of IPv6 we won't get into the details. The OpenVPN home page at `http://www.openvpn.net/` has plenty of documentation for more "mainstream" usage.

At this time, OpenVPN has a number of limitations with regard to IPv6: It doesn't configure the interfaces automatically as it does for IPv4, we can't set up a "big server" that deals with an arbitrary number of clients on a single UDP port and we can't make Solaris run IPv6 over an OpenVPN tunnel. But in many practical cases OpenVPN provides us with the NAT-penetrating tunnel protocol we may need.

13.4 Packet Filter Considerations

The problems and strategies we have seen in section 12.6 for encapsulating tunnels generally apply to the tunnels in this chapter, too.

We can filter for GRE using the protocol type patterns as with encapsulation. The protocol type for GRE is 47 or `gre`. With OpenVPN and Teredo we can filter by UDP port number; Teredo uses port 3544 and OpenVPN whatever it is configured to use, or 1194 if we don't explicitly configure a port number.

These non-encapsulating tunnels have a major problem, though: We can't filter by the inner protocol type. If we put the packet filter on the tunnel exit node, then we can filter the traffic coming out of the tunnel interface, so this case can be dealt with. But if we have a separate packet filtering router between the tunnel nodes, then we can't configure it to allow "only IPv6 tunneled over GRE" or similar.

14

Advanced Tunneling Issues

So far we have avoided a number of more advanced issues related to tunneling in general rather than a single tunneling mechanism. In this chapter we address these more fundamental issues.

First we take a look at tunnel brokers, which automate the configuration and management of tunnels. Then we address the interaction between tunnels and NAT gateways and how to make the two cooperate. A major risk when using tunnels are tunnel loops that cause a network meltdown. We take a closer look at the problem and how to mitigate it. Next we tweak some of the more advanced tunable parameters to make our tunnels work better. Finally we consider some problems caused by mixing tunneled and native IPv6 connectivity.

14.1 Tunnel Brokers

Today a number of *tunnel service providers* exist. They provide IPv6 connectivity through tunnels. Since most of their customers are stuck not only behind NAT gateways but have their IPv4 provider dynamically assign them their IPv4 address, these tunnel providers have since developed tools to maintain tunnels automatically. These *tunnel brokers* manage the configuration of existing tunnel protocols so they don't need to be handled manually. Additionally, tunnel service providers have come up with some additional NAT-capable tunnel mechanisms of their own.

So what do these tunnel brokers actually do? RFC 3053 [28] explains the concepts in detail. For our purposes, consider an example scenario: Imagine myself on a business trip doing IPv6 trainings as usual. In the evening I sit down in my hotel room to do some serious Web surfing. My Internet access is limited to IPv4 through a NAT gateway of some kind that I don't have any control over.

When I start my tunnel broker client, it connects to my tunnel service provider's tunnel broker server using TCP over IPv4. Through some authentication mechanism I first authenticate myself. Next the client and server check what sort of connectivity I have. Is there a NAT gateway in between or do I have a globally routed IPv4 address? Then they choose the best kind of tunnel mechanism for me to use—either a configured 6in4 tunnel or something else that can penetrate the NAT gateway. The tunnel broker client then configures my end of the tunnel while the server sets up the tunnel endpoint at the far end. Additionally the tunnel broker server may tell me my IPv6 address or `/48` prefix. Once the tunnel is established I have set up my IPv6 connectivity, including a globally routed address or prefix. As long as I keep the client running, the client and server will monitor the tunnel and automatically start over if it breaks down—for example if I move from my room to the breakfast lounge (it must have been a long night again), which may be covered by a different WLAN access point.

Two of the more innovative tunnel service providers around the globe are SixXS in the Netherlands (`http://www.sixxs.net/`) and Hexago in Canada (`http://www.hexago.com/`). Both provide free tunnel services to everybody. Rather than waiting for a protocol specification they have both come up with an implementation of the concepts from RFC 3053 and are now working on putting them through the standardization process.

Besides the actual tunnel broker functionality both implement NAT-penetrating tunnel protocols. Both support some IPv6-in-UDP-in-IPv4 protocol that is very much like a configured 6in4 tunnel except that it uses UDP/IPv4 as the outer tunnel protocol. Both provide a free, open-source implementation of their tunnel broker client.

In a way, tunnel brokers put another abstraction layer on top of the tunnel mechanisms we've seen so far. They take care of authentication and handle changing IPv4 addresses gracefully, which has so far forced us to use OpenVPN with all its shortcomings. Instead of forcing us to decide on the most appropriate tunnel mechanism they take care of this automatically.

Even though the standardization of tunnel brokers is lagging behind, using them today is perfectly feasible especially for small to medium sized sites if no native IPv6 connectivity is available.

If tunnel broker services become more widely available around the world, then they may eventually provide a service with all the advantages but none of the disadvantages of 6to4 public relays.

14.2 Tunnels and NAT Gateways

So far we have considered the relationship of tunnels and NAT gateways based on one question only: Is the tunnel protocol capable of passing through an existing NAT gateway? But if we have some control over the NAT gateway, then some interesting options become available.

14.2.1 Strategies

If the NAT gateway is beyond our administrative control, then there is nothing we can do except to use a tunnel protocol that is capable of penetrating a NAT gateway.

If we manage the NAT gateway ourselves, then we may find two options feasible: We may be able to put the tunnel endpoint on the NAT gateway itself or we may use *reverse NAT*, also called *packet redirection* or *destination NAT* (*DNAT*). If either of these options is available, then we can use NAT-incompatible tunnel protocols in spite of the NAT.

Putting the tunnel endpoint on the NAT gateway has some advantages: It is cheap because we only need a single device—this saves both money on hardware and time on system administration effort. It is also easy to configure because we don't need to set up packet redirection.

But it also has some drawbacks: Putting a tunnel endpoint on a NAT gateway adds to its workload. As soon as security becomes an issue we also need to configure a packet filter on the NAT gateway. With an outside interface, a NAT pseudo-interface, a tunnel pseudo-interface and an interface towards the internal networks, this is somewhat demanding. If we need cascaded protection, the combined NAT gateway and tunnel endpoint becomes a single point of failure.

Using reverse NAT has the opposite properties: It is more expensive because it needs two devices that must be bought and administrated. Setting up the redirection is usually complex because it needs to be done within packet filter configurations.

But except for the redirection configuration it keeps the NAT gateway simple, so packet filtering on the NAT gateway stays straightforward, too. The tunnel end point can be placed in a DMZ, so multi-level security architectures are feasible.

Tunneling mechanisms that embed an IPv4 address in an IPv6 address usually won't be able to penetrate a NAT gateway using redirection; the NAT gateway will break the relationship between the IPv4 and IPv6 address. In theory it may be possible to use such a tunnel anyway, but in practice implementations usually won't support this kind of kludge. With configured, GRE or OpenVPN tunnels it is however possible to use reverse NAT.

14.2.2 Configurations

Figure 14.1 shows the relevant nodes in an environment using a tunnel through a NAT gateway. We configure the tunnel nodes as usual, with one major exception: If the remote tunnel node is located behind a NAT gateway doing redirection, then we need to configure the NAT gateway's public address as the remote outer address. In the diagram, this means that the right tunnel node will configure `192.0.2.1` as its peer, not `10.0.0.2`.

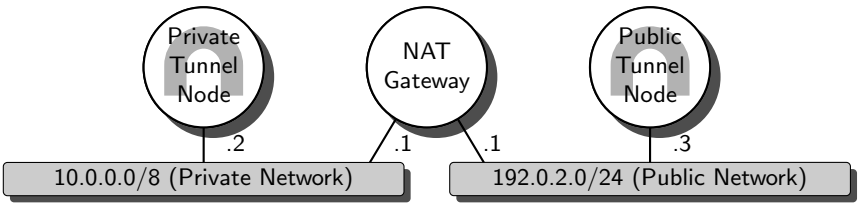

Fig. 14.1. Tunnels in a NAT environment

The NAT gateway needs to support redirection. In the most relevant cases, configured and GRE tunnels, we need to do redirection by protocol type, not by TCP or UDP port numbers as usual. How to do this depends on the NAT gateway implementation. The example configurations below assume a configured tunnel; for GRE, the protocol type used must be 47, or `gre`, instead of 41, or `ipv6`.

Debian Sarge *Using* `iptables` *we can enable NAT in general using the commands*

```
# modprobe iptable_nat
# iptables -t nat -A POSTROUTING -s 10.0.0.0/8 -j SNAT --to
192.0.2.1
```

This takes care of all traffic travelling from the left to the right tunnel node, implementing NAT in general. If the left tunnel node initiates the traffic, this is all it takes. But if the right tunnel node may also initiate traffic, then we need an additional command

```
# iptables -t nat -A PREROUTING -s 192.0.2.2 -d 192.0.2.1 \
          --proto ipv6 -j DNAT --to 10.0.0.2
```

to enable reverse NAT for 6in4 traffic.

As usual, if we want to make this configuration permanent, we use an `up` *statement in* `/etc/network/interfaces`*.*

FreeBSD 6.1 *The two most relevant NAT implementations are* `natd`*, which is used together with the* `ipfw` *packet filter, and the NAT implementation within the* `pf` *packet filter.*

The `natd` *has an option* `-redirect_proto` *(note the underscore) that we can use to set up a redirection based on protocol type. The appropriate lines in* `/etc/rc.conf` *for our examples are*

/etc/rc.conf

```
natd_enable=YES
natd_interface=lnc1
natd_flags="-redirect_proto ipv6 10.0.0.2"
```

Alternatively, using a `pf` *rule*

/etc/pf.conf

```
rdr on lnc1 proto ipv6 -> 10.0.0.2
```

achieves the same. If we want to configure filter rules here as well, then we only need to set up rules for the already translated addresses, like

/etc/pf.conf

```
pass quick proto ipv6 from 192.0.2.2 to 10.0.0.2
pass quick proto ipv6 from 10.0.0.2  to 192.0.2.2
```

83

14.3 Nested Tunnels and Tunnel Loops

A potentially fatal problem with tunnels are routing loops that send an already tunneled packet into the same tunnel again.

14.3.1 Network Meltdown from a Tunnel Loop

A "normal" routing loop is fairly harmless: A packet will be sent through the loop until its time to live (IPv4) or hop limit (IPv6) counter reaches zero. Then an ICMPv6 error is sent back to the sender. If this ICMPv6 error gets caught in the same or another loop, it will be discarded as soon as the TTL or hop limit expires.

With tunnels the situation is different: When a packet enters a tunnel, a new header with a "recharged" TTL/hop limit is prepended. If the tunneled packet somehow loops back to the tunnel entry without being untunneled first, then it will be sent through the tunnel again—with its TTL/hop limit reset yet again.

Debian Sarge *The Linux kernel behaves differently if we don't set an explicit TTL or hop limit: It copies the TTL or hop limit from the inner packet. This is useful in some cases, but generally it is advisable to set an explicit TTL.* 84

During this process the packet grows with every new encapsulation. Eventually it is too large to pass through the tunnel since it exceeds the tunnel's maximum transmission unit (MTU). At this point the tunnel entry point behaves depending on the IP version of the outer header: With IPv4 it may fragment the packet and send the fragments through the tunnel as two separate packets, to enter the loop again. With IPv6, and IPv4 packets that have the don't fragment flag set, it sends an ICMPv6 "too big" message to the sender which then fragments the packet itself and re-sends the fragments.

In either case the result is a *network meltdown*—the network is saturated from the traffic by the single initial packet.

14.3.2 Tunnel Loop Causes

In the simplest case a tunnel loop is caused by some misconfiguration, either of a new tunnel or some changed static routes. If this happens, then whoever did the configuration will usually find out quickly what happened, break the loop and restore the network to normal operation.

With dynamic routing the situation is different: Whenever the routing changes, some temporary inconsistencies occur until the routing converges to a fully functional state again. As long as routers can still communicate with each other, this will eventually solve the problem. But with a network meltdown it is possible that routers can't communicate with each other anymore because the very network they want to use to do so is saturated, so the attached nodes and switches discard packets. At this point, manual intervention is required. If the situation is really bad, then it may be necessary to access some network components physically, which again may take some time if no system administrator is currently near the network.

14.3.3 Preventing Tunnel Loops

To prevent a meltdown due to a tunnel loop, we need to prevent already tunneled packets to enter a tunnel again.

For tunneling anything over IPv6, RFC 2473 [17, section 4] introduces an optional *tunnel encapsulation limit* option header that limits the nesting level of IPv6. RFC 1853 [101, section 3.3] devises a heuristic to detect simple tunnel loops. With all other tunnels we need to take care of the situation ourselves.

For all practical purposes we can't rely on any tunnel implementation to be smart enough to prevent tunnel loops. This leaves us with four options: Use only tunnels with distinct inner and outer protocol, like only IPv4 as outer and only IPv6 as inner protocol, prevent tunnel loops ourselves using a proper ingress filter on the tunnel interface, use only static routing in the tunnel environment or just hope for the best.

Setting up an ingress filter on the tunnel interface is fairly simple if we want to avoid passing already tunneled packets through it. But when it comes to limiting the nesting depth of tunnels, there is no packet filter available today with a filter syntax that lets us express such a restriction.

FreeBSD 6.1 *In theory the* `gif` *interfaces use a* `sysctl` *variable called* `net.link.gif.max_nesting` *that lets us specify the nesting depth as long as we limit ourselves to* `gif` *tunnels. As of today, setting it to 0 will unfortunately disable the interface while setting it to anything larger than 0 it apparently enables nested tunnels without restricting the nesting level.*

Similarly, the `gre` *interfaces use a variable* `net.link.gre.max_nesting`*.*

Using static routes in a tunnel environment is a feasible alternative. As long as our tunnel is our "default router" and we simply set the default routes of all other routers towards it, this will effectively solve our problems.

The situation gets complicated as soon as we want to use dynamic routing on the "untunneled" side of the tunnel but static routes for the tunnel path. Such a setup becomes necessary if we need redundant tunnels and tunnel routers. There is no catch-all solution to this problem, but as always with dynamic routing, keeping the network topology clean is an essential measure. Beyond that it helps to set up the tunnel routers with two separate interfaces, one "inside" and the other to pass the tunneled traffic through. We need to configure the routing daemon on the tunnel router not to announce the routes it is directly connected to on the tunnel interface but only the routes beyond its tunnel peer; this isn't always possible with the lightweight RIPng distributed with most Unixen. But as soon as we install and run a full routing framework like Quagga we can work around this problem as well.

Setting up a dynamic router for these purposes isn't trivial. Chapter 17 addresses the more advanced aspects of dynamic routing that we need in a situation like this.

14.4 Tunnel Parameter Tuning

There are two tunable parameters available with many tunnels that deserve some attention. They aren't essential for a tunnel to work, but there are cases where a tunnel will work better if these two parameters are adjusted to the particular environment.

14.4.1 The Maximum Transmission Unit (MTU)

According to section 7.9, the maximum transmission unit (MTU) of an interface defines how large the payload of its link-layer frame may be. When an IP packet exceeds this size, it is split into fragments that are sent separately to the destination address. There they are reassembled.

How the fragmentation works depends on the IP version used as the inner protocol. IPv4 fragmentation is done at the tunnel entry point while IPv6 notifies the original sender to use smaller fragments using path MTU discovery.

In either case it is desirable that a packet is not fragmented multiple times; fragments of fragments of fragments cause severe overhead on both the intermediate routers as well as the tunnel endpoints.

If we know that a link with an unexpectedly small MTU exists along the tunnel path, then we may want to set the tunnel MTU accordingly so no packets get fragmented. It may take some experimenting on the tunnel endpoints to ensure that no fragmented packets occur. Using brute force we can use a packet sniffer to watch for fragmented tunnel packets coming in while we gradually reduce the tunnel MTU.

How we can set the tunnel MTU depends on the particular Unix. Note that IPv6 needs an MTU of at least 1280 bytes while IPv4 allows far smaller MTUs. Ethernet has an MTU of 1500 bytes unless it supports *jumbo frames* which may exceed 8 kbytes in size.

Debian Sarge *Using the generally preferred* `ip` *command,*

```
# ip link show
```

displays the MTUs of all interfaces and

```
# ip link set sit1 mtu 1400
```

adjusts it to our needs.

FreeBSD 6.1, Solaris 10 *The* `ifconfig` *command displays the MTU together with other information about the system's interfaces. The syntax*

```
# ifconfig gif0 inet6 mtu 1400
```

sets the MTU. 86

There are cases where MTU tuning reduces the traffic overhead as well as the workload on the routers dealing with the fragmentation quite noticeably.

14.4.2 Hop Limit and Time to Live (TTL) Parameters

Another tunnel parameter that we may want to tune is the IPv6 *hop limit* or its IPv4 equivalent, the *time to live* (*TTL*) field. It limits the maximum number of hops to the tunnel exit node.

When a tunnel entry point sends a packet through the tunnel, it prepends another IP header. This IP header has its hop limit/TTL field initialized with a configurable value. Depending on the particular Unix, the hop limit may be set either globally or for individual tunnels; implementations differ widely in the way they handle the hop limit.

Debian Sarge *The* `ip` *command shows the TTL with the command line arguments*

```
# ip tunnel show
```

and sets it according to the pattern

```
# ip tunnel change sit1 ttl 20
```

for every tunnel interface individually.

As we have already seen in section 12.1.2, Linux will copy the TTL or hop limit from the inner packet if we don't set it explicitly. This behaviour limits the effects of a tunnel loop to some degree but breaks applications that explicitly control the TTL or hop limit.

FreeBSD 6.1 *There are two* `sysctl` *variables called* `net.inet.ip.gifttl` *and* `net.inet6.ip6.gifhlim` *that set the IPv4 TTL and IPv6 hop limit globally for all* `gif` *interfaces. There is no documented way to set the TTL for 6to4 or GRE tunnels.*

Solaris 10 *The* `ifconfig` *command shows the TTL or hop limit when we display the interface configuration. To change it,*

```
# ifconfig ip.tun0 inet6 thoplimit 10
```

sets the TTL/hop limit for an individual interface and IP version. 87

Lowering the TTL/hop limit can to some degree reduce the risk of tunnel loops caused by dynamic routing: If the route between the tunnel nodes gets longer than usual due to some temporary routing changes, then the tunnel temporarily breaks down, hopefully before a tunnel loop causes a meltdown. If the tunnel end points are very far apart, we may need to increase the hop limit accordingly.

14.5 Mixing Tunnels and Native Connectivity

In theory it is possible to mix tunneled and untunneled connectivity at a site. Doing so isn't quite as easy as it seems, though.

Why should anyone actually do this? In some cases we may want to improve our traffic to sites that are currently close by with respect to the IPv4 network topology but far away as far as IPv6 connectivity is concerned. In particular, using a 6to4 border router to speed up traffic to 6to4 sites while we already have native IPv6 connectivity is worth some thought. Additionally, we might use tunnels for a fallback uplink.

So what sort of problems do we have to anticipate if we set up mixed connectivity?

We need to be extremely careful if we use dynamic routing. And since dynamic routing is necessary if we want to make our network connectivity redundant, we can't just use static routes instead. The main risk is that we might accidentially announce "short" routes through our site even though we don't have much bandwidth. This problem is not so much related to tunnels but dynamic routing in general, so we defer its discussion until chapter 17, where we get into dynamic routing in more detail, and section 25.1, where we consider ways to set up redundant network connectivity.

Using multiple connections makes packet filters noticeably more complex to configure. Stateful filtering becomes extremely difficult at best if dynamic routing redirects traffic arbitrarily around the filters. If we run multiple different tunnel types on a single tunnel router, packet filtering also becomes difficult—in particular, mixing 6to4 and automatic tunnels is known to cause serious problems; RFC 3964 [99, section 6.1] explains why.

None of these problems are reason enough not to use multiple tunnels, or mixing tunneled and native IPv6 connectivity. But we need to be aware of the potential problems when we do so.

15

The Point-to-Point Protocol (PPP)

Technically, the *point-to-point protocol* (*PPP*) is not a tunnel protocol but a point-to-point link-layer protocol originally intended for serial lines. In practice, some implementations also support TCP or even UDP instead of serial lines, making it possible to use PPP as a tunneling mechanism. More important however, handling PPP is very similar to configured tunnels, so it makes sense to take a look at it together with the tunnel protocols we have seen so far.

In 1996, RFC 2023 [58] first extended PPP to support IPv6. Two years later its successor RFC 2472 [59] changed a few details; it still represents the official specification for IPv6 over PPP.

15.1 Implementations and Installation

Today there are two widely used PPP implementations: The *kernel PPP* implementation by Paul Mackerras is mostly kernel-based and only uses a userspace `pppd` to set up a connection while the *userland PPP* implementation called `ppp` by Toshiharu Ohno uses a generic tunnel interface and runs entirely in userspace. In theory both support IPv6, but in practice some ports don't.

Debian Sarge *The kernel PPP implementation included in a core installation does support IPv6.*

FreeBSD 6.1 *Both PPP implementations are available within a core installation, but the kernel PPP implementation doesn't support IPv6.*

Solaris 10 *The kernel PPP implementation supports IPv6. The distribution media contain it in the packages* `SUNWpppd`*,* `SUNWpppdr` *and* `SUNWpppdu`*. After the packages are installed the kernel must be reconfigured to enable the PPP device driver.* 88

15.2 Basic Configuration

For demonstration purposes we use a very simple setup; figure 15.1 shows the layout. To keep the configuration simple we use null modems as links, configure the nodes as peers so either one can initiate a PPP session and don't authenticate the links.

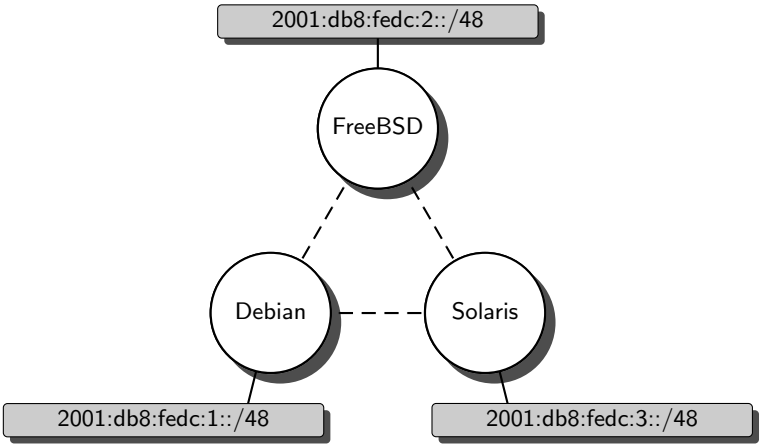

Fig. 15.1. The PPP test setup

Debian Sarge *The configuration in* `/etc/ppp/options.server` *needs at least the lines*

`/etc/ppp/options.server`

```
lock          || Use UUCP-style lock files
noauth        || Don't expect peer to authenticate itself
ipv6 ,        || Don't configure addresses manually
noip          || Disable IPv4 (optional)
noccp         || With FreeBSD as peer: Avoid incompatibility problem
nodetach      || We want to start this via /etc/inittab—see below
```

To start PPP once we we can run the command

```
# pppd /dev/ttyS0 file /etc/ppp/options.server
```

from a shell.

Apparently a bug in the implementation breaks the `persist` *option: When the connection shuts down and then comes up again,* `pppd` *can't re-initialize the serial line. This makes the* `persist` *option useless at least in the configuration shown here.*

As a workaround we can use the `nodetach` *option in the configuration and a line in* `/etc/inittab` *like*

/etc/inittab

```
T0:23:respawn:/usr/sbin/pppd /dev/ttyS0 file /etc/ppp/options.server
T1:23:respawn:/usr/sbin/pppd /dev/ttyS1 file /etc/ppp/options.server
```

for every serial interface. This will restart the `pppd` *whenever it terminates.*

FreeBSD 6.1 *For our purposes most default settings are sufficient, so we just need the lines*

/etc/ppp/ppp.conf

```
default:
    set openmode active          || Don't wait for peer to initiate session
    disable ipcp                 || Disable IPv4 (optional)
serial0:                         || We call this configuration serial0
    set device /dev/cuad0        || Use this serial device
serial1:                         || ... and the same for the second device
    set device /dev/cuad1
```

in `/etc/ppp/ppp.conf` *to configure the setup. Afterwards we can start* `ppp` *for the lines either from the command line using*

```
# ppp -dedicated serial0
# ppp -dedicated serial1
```

or using the same commands from `/etc/rc.local`*. If we had only a single serial line we could also set some variables in* `/etc/rc.conf` *to achieve the same result—the details are explained in* `/etc/defaults/rc.conf`*.*

Solaris 10 *A minimal configuration in* `/etc/ppp/options.server` *needs the lines*

/etc/ppp/options.server

```
lock            || Use UUCP-style lock files
persist         || Keep running forever
ipv6 ,          || Don't configure addresses manually
noip            || Disable IPv4 (optional)
noccp           || With FreeBSD as peer: Avoid incompatibility problem
```

Following the online documentation from SUN[1] *we can start the interface using a script* `/etc/ppp/demand` *with the contents*

/etc/ppp/demand

```
#! /bin/sh

if [ -f /var/run/ppp-demand.pid ] &&
   /usr/bin/kill -s 0 `/bin/cat /var/run/ppp-demand.pid`
```

[Continued on next page]

[1] See `http://docs.sun.com/app/docs/doc/816-4555/6maoquihv?a=view`.

/etc/ppp/demand

[Continued from previous page]

```
then
    :
else # All pppd invocations go here.
    /usr/bin/pppd /dev/cua/a file /etc/ppp/options.server
    /usr/bin/pppd /dev/cua/b file /etc/ppp/options.server
fi
```

After running

```
# /etc/init.d/pppd start
```

or a quick reboot PPP automatically starts. 89

Once the connection is set up on both sides, we should check it. If the interface doesn't come up as expected, then logging the *link control protocol* (*LCP*) and the *IPv6 control protocol* (*IPV6CP*) often proves helpful.

- ☐ The interfaces on both sides must be up, running and showing a link-local address.
- ☐ Pinging the all-node link-local multicast address `ff02::1` across the link must show replies from both endpoints.

15.3 Adding Routable Addresses and Static Routes

So far we have only set up link-local addresses. IPv4 over PPP offers a feature that one peer can tell the other what IP address to use; this is commonly used by ISPs to assign their customers a dynamically chosen IPv4 address. For IPv6 no equivalent feature is available. Since IPv6 addresses are usually statically assigned, we can and must configure both peers with their own IPv6 address.

The `ipv6` option we have used with `pppd` in the previous section only lets us specify the local and remote interface ID, but no complete addresses. The user PPP implementation doesn't even let us specify an interface ID, let alone a full address. To work around this we need to use some scripts that are run once the PPP session is up and running.

We can also use the same scripts for another, independent purpose: To set up static routes when an interface comes up.

Debian Sarge *If either of the two scripts `/etc/ppp/ipv6-up.local` and `/etc/ppp/ipv6-down.local` exists and is executable, then it is run whenever the connection is started or terminated, respectively. The environment variables `PPP_IFACE`, `PPP_TTY`, `PPP_SPEED`, `PPP_LOCAL`, `PPP_REMOTE` and `PPP_IPPARAM` contain the interface name, the TTY device name, the TTY*

speed, local and remote link-local addresses and an optional parameter that we can configure with an `ipparam` *statement in our configuration.*

A script to set up additional addresses and routes may look like this:

/etc/ppp/ipv6-up.local

```
#! /bin/sh
case $PPP_IFACE in
    ppp0) /sbin/ip -6 addr add dev ppp0 2001:db8:fedc:12::1/64
          /sbin/ip -6 route add dev ppp0 2001:db8:fedc:2::/64
          ;;
    ppp1) /sbin/ip -6 addr add dev ppp1 2001:db8:fedc:13::1/64
          /sbin/ip -6 route add dev ppp1 2001:db8:fedc:3::/64
          ;;
esac
```

Unfortunately, the name of the peer is unavailable. If we are running an ISP, then this is most unfortunate because we can't just configure our PPP interfaces by using the authenticated name of the peer. In section 15.6 we take a look at this problem.

FreeBSD 6.1 *To run a script after the the connection has come up, we first need a configuration file* `/etc/ppp/ppp.linkup` *with the contents*

/etc/ppp/ppp.linkup

```
MYADDR6:
  shell /etc/ppp/ipv6-up LABEL INTERFACE
```

Whenever an interface comes up, this configuration runs the shell script `/etc/ppp/ipv6-up` *with the label from the configuration in* `/etc/ppp.conf` *and the interface name as its command line parameters.*

The `/etc/ppp/ipv6-up` *script then configures the additional addresses and routes:*

/etc/ppp/ipv6-up

```
#! /bin/sh
case $1 in
    serial0) /sbin/ifconfig $2 inet6 2001:db8:fedc:23::2 up
             /sbin/route add -inet6 2001:db8:fedc:3::/64 -iface $2
             ;;
    serial1) /sbin/ifconfig $2 inet6 2001:db8:fedc:12::2 up
             /sbin/route add -inet6 2001:db8:fedc:1::/64 -iface $2
             ;;
```

While the additional configuration file `ppp.linkup` *makes the configuration somewhat more tedious than the* `pppd` *implementation, it provides far more data to the shell script. This makes it a superior choice when setting up a dial-in server.*

Solaris 10 *The two scripts* `/etc/ppp/ipv6-up` *and* `/etc/ppp/ipv6-down` *are run whenever a PPP link goes up or down, respectively.* `Pppd` *passes them*

the interface name, TTY device name, TTY speed, local and remote link-local address and possibly some additional configurable string as parameters.

A sample `/etc/ppp/ipv6-up` *script may look like this:*

/etc/ppp/ipv6-up

```
#! /bin/sh
case $1 in
sppp0) /sbin/ifconfig sppp0 inet6 addif \
                      2001:db8:fedc:13::3 2001:db8:fedc:13::1 up
       /sbin/route add -inet6 2001:db8:fedc:1::/64 \
                    -iface 2001:db8:fedc:13::3
       ;;
sppp1) /sbin/ifconfig sppp1 inet6 addif \
                      2001:db8:fedc:23::3 2001:db8:fedc:23::2 up
       /sbin/route add -inet6 2001:db8:fedc:2::/64 \
                    -iface 2001:db8:fedc:23::3
       ;;
esac
```

Solaris assumes PPP interfaces to be point-to-point interfaces and it expects us to provide both the local and remote address when we configure such a point-to-point interface. This forces us to keep track of our peers' IP addresses when we write such a script.

Solaris doesn't remove the subinterfaces when an interface goes down, so we need to clean them up in `/etc/ppp/ipv6-down`*. A simple catch-all solution is*

/etc/ppp/ipv6-down

```
#! /bin/sh
/sbin/ifconfig $1:1 inet6 unplumb
```

90

Checking the results of the script is fairly obvious. If the script is executable and all external commands and filenames used within are either given as absolute pathnames or within the (usually quite restrictive) `PATH`, then problems are quite unlikely.

- ☐ After the interface has come up, check that the static addresses and/or routes are configured as expected.

15.4 Dynamic Routing Across PPP Links

Setting up routable addresses and static routes on a PPP interface is tedious, so dynamic routing across a PPP link may be a useful alternative.

On all the Unixen shown, RIPng works without problems across PPP links. The configuration doesn't differ from an "ordinary" RIPng setup. RIPng

uses link-local addresses between routers, so we don't even need to configure routable addresses on the PPP interfaces.

The only problem with dynamic routing across PPP links isn't IPv6-specific: In scenarios like a dial-up ISP serving a large number of customers, dynamic routing can be used to divert traffic from one customer to another. Generally speaking, if the PPP link connects two different security domains, then dynamic routing is usually a bad idea.

To check the configuration, the same procedures as with "normal" dynamic routing apply. A quick check looks like this:

- ☐ Check that the interfaces are up and running.
- ☐ Wait at least 45 seconds.
- ☐ Check the routing tables for routes that have been set up through the PPP link.

15.5 PPP and Autoconfiguration

So far we have only used PPP between routers. But what if we connect a host with a router using PPP? In that case we may want to support stateless autoconfiguration across the PPP link.

In most cases, using autoconfiguration on a PPP link doesn't pose any problems. But some implementations consider point-to-point links unfit for autoconfiguration and refuse to support it.

Solaris 10 *There is no support for autoconfiguration over a PPP link.* 91

Setting up a host doesn't differ from the configuration we have used in section 15.2 except that we apply it to a host instead of a router.

Configuring a router is slightly more demanding: As in section 15.3 we need to configure a static address on it when the interface comes up. We also need to convince the router advertisement daemon to advertise the proper prefix on the link.

Debian Sarge *Unfortunately* `radvd` *doesn't deal with dynamically appearing and disappearing interfaces. This leaves us two options: Either restart a single, long-running* `radvd` *whenever a PPP link comes up or start a separate* `radvd` *with its own configuration for every PPP link.*

Using a single `radvd` *for all interfaces we first need to add interface configurations like*

/etc/radvd.conf

```
interface ppp0 {
    AdvSendAdvert on;
    IgnoreIfMissing on;                    || Don't forget this; see below.
    prefix 2001:db8:fedc:fed1::/64 ;
};
```

to `/etc/radvd.conf`*. The* `IgnoreIfMissing` *statement ensures that* `radvd` *ignores all unavailable interfaces. Additionally we need to restart* `radvd` *whenever a PPP link comes up. To do so we add a line*

`/etc/ppp/ipv6-up`

```
/etc/init.d/radvd restart
```

to the `/etc/ppp/ipv6-up` *script.*

This approach has the major disadvantage that it assigns a fixed prefix to a PPP interface, which is undesirable on a dial-in server with multiple clients. To work around this we need to know who our peer is, so we can assign it the prefix it needs. Unfortunately, `pppd` *doesn't pass the peer name to the* `ipv6-up` *script, so this isn't really feasible.*

FreeBSD 6.1 *If we configure* `rtadvd` *with the interfaces we want it to serve, then we will get the occasional harmless error message while a link is down, but otherwise the autoconfiguration will work across the PPP link.*

Since `rtadvd` *by default advertises the prefixes that a given interface is configured with, it handles PPP interfaces that change their prefixes over time correctly. This is particularly important if we set up a dial-in server that doesn't assign fixed interfaces to its peers.* 92

Checking autoconfiguration across PPP links doesn't differ much from checking it over regular subnets. A few differences do exist however:

- ☐ If multiple hosts connect to a server, check with each host that it receives the expected prefix or prefixes and configures its interface accordingly. Make double sure that the prefixes are correct for each host.
- ☐ Check that the advertising daemon doesn't quit when a PPP interface goes down.

15.6 Beyond a Single Interface: Operational Issues

So far we have happily ignored a number of issues that make real-world PPP configuration such a pain.

We have skipped modem chat scripts because they are independent of the IP protocol that we use. We have also skipped authentication, which is also independent of the IP protocol we use. We have so far assumed that a given PPP interface is associated with a single, fixed peer. This is fine if we are a leaf site connecting to an ISP via PPP—unless we are also running the server at the ISP side.

But what if we have a single PPP server with a number of serial lines that users connect to? The association between the serial line and PPP interface isn't fixed, so even if we have dedicated lines to our users we never know who is connected to which PPP interface. With a dial-in modem pool or similar

we don't even know who is connecting to which line. The same applies to a *PPP over Ethernet* (*PPPoE*) setup, where the number of "lines" isn't even fixed.

To work around this, we need our peers to authenticate. All PPP implementations support this without problems. But we also need access to our peer's name when we bring up an interface. This turns out to be a problem.

Debian Sarge, Solaris 10 *The* `pppd` *doesn't pass the peer's name to the* `ipv6-up` *script. This makes it effectively impossible to use* `pppd` *on a production-grade PPP server.*

FreeBSD 6.1 *In section 15.3 we have used* `/etc/ppp/ppp.linkup` *to run a shell script when an interface comes up. If we extend it to pass an additional parameter* `USER` *to the* `ipv6-up` *script, then we can configure the PPP interface depending on the connected user.* 93

15.7 Packet Filter Considerations

PPP dynamically creates and configures its interfaces, so we need to set up our packet filters accordingly.

If we set up multiple PPP interfaces, then the same problems as in the previous section occur. To work around them we must dynamically update the filter rules whenever an interface comes up. Fortunately all packet filter implementations shown let us set up filter rules on interfaces that don't yet exist.

Both PPP implementations only run the configuration scripts after the interface is up and running. This is unfortunate if we need to set up different filter rules for different peers, for example if we want to set up anti-spoofing rules. In this case we need to set up the packet filters to deny all traffic with the possible exception of neighbor and router discovery packets before we first bring up an interface. The script must then add custom filter rules for the particular peer before it may set up static addresses or routes. When the connection shuts down, we need to remove the filter rules again.

Debian Sarge *The script* `/etc/ppp/ipv6-down` *is run whenever a link shuts down. It behaves like its* `ipv6-up` *peer.*

FreeBSD 6.1 *From* `/etc/ppp/ppp.linkdown` *we can invoke a script when a link shuts down the way we run a script from* `ppp.linkup` *when we bring a link up.* 94

This still leaves us with a potentially dangerous situation: If the shutdown script fails, then it may leave the filter configuration open to attacks by the peer connecting next to the serial line.

Part IV

Additional Base Features

16

More on Addresses

Chapter 3 provided all the information necessary to get IPv6 up and running. But there is more to IPv6 addresses than we have seen to far. This chapter covers a number of not so essential aspects concerning IPv6 addresses as such.

16.1 Site-local and Unique-local Addresses

In section 3.4.2 we introduced site-local and unique-local unicast addresses. Until now they haven't been particularly exciting, but they are quite useful as a fallback during network renumberings.

16.1.1 From Site-local to Unique-local Addresses

Originally, the IPv6 address architecture standards (RFCs 1884 [61], 2373 [62] and 3513 [63]) defined the address range `fec0::/10` as "site-local" unicast addresses. They were similar to the private IPv4 addresses defined in RFC 1918 [97] (`10.0.0.0/8`, `172.16.0.0/12` and `192.168.0.0/24`) and anybody was free to use them for internal purposes as long as they were only used inside a local network cloud.

Experience has shown that this approach introduces a number of problems. RFC 3879 [71] pointed out two core causes: Address ambiguity, or multiple machines using the same address, and an ill-defined concept of "site". Problems related to the "site" concept are mostly a matter of interpretation of the term "site" in a particular context. But even if your network might be considered a "site" by whatever definition, the more serious problems related to the ambiguity of addresses remain. Some of them, like the trouble of setting up "multi-sited routers", can be trivially solved by not using site-local addresses for inter-site or global purposes—like NAT in the IPv4 world. But site-local addresses that leak into dynamic routing tables and the DNS are more serious. To solve these problems it was necessary to make even private addresses unique.

Discussions sprang up to devise an address range for private purposes where addresses were not ambiguous; they just wouldn't be globally routed. Originally, it was planned to use the `fc00::/8` address range to assign `/48` prefixes by a central authority and `fd00::/8` to pick random `/48` prefixes without central management, thus making them unique only by probabilistic standards. Eventually, RFC 4193 [66] defined the `fd00::/8` prefix accordingly. Until now, there has been neither an official standard nor a central management authority for the `fc00::/8` address range. RFC 4291 [64], the successor of RFC 3513, formally declares the old site-local prefix `fec0::/10` obsolete.

Throughout this book, we call both site-local and unique-local addresses *site-scoped addresses*.

So what exactly is the difference between the old `fec0::/10` and the new `fd00::/8` prefix? With `fec0::/10`, the majority of people used "short" prefixes like `fec0:0:0::/48` for their internal purposes, causing these addresses to be ambiguous. The new `fd00::/8` is used differently: Everybody is free to choose a *randomly chosen* `/48` prefix from this range. Everybody choosing a prefix randomly can be confident that nobody else uses this prefix, which effectively solves the ambiguity problem.

Everybody "randomly" picking the `fd00:0:0::/48` prefix will join the group of administrators that perpetuate the address ambiguity problems among themselves.

16.1.2 What is a "Site"?

There have been numerous discussions about the definition of the term "site". Instead of attempting to define the term, it has proven more useful to list a number of criteria that make a network a "non-site".

1. If its routing and DNS aren't managed by a single administrative authority, it isn't a site.
2. If it isn't contiguous, it isn't a site.
3. If it contains "sub-sites", it isn't a site.
4. If it is part of a "super-site" then it isn't a site.
5. If it overlaps with another "site", then it isn't a site.
6. If it "shares" a node with another "site", then it isn't a site. Nodes never belong to more than one site.

If any of these criteria applies, using site-scoped addresses will cause trouble. So don't use them and don't consider your network a site.

16.1.3 When to Use Unique-local Addresses

What are site-scoped addresses good for? First of all, don't use them for non-local purposes. IPv6 addresses aren't scarce, so there is no need to abuse them for NAT and similar kludges.

In virtually all cases you want to get globally routed addresses if you have any Internet6 connectivity at all. But since your upstream provider may occasionally require you to renumber your network, consider using them as a fallback especially during a renumbering event.

Remember, non-routeability of these addresses was never intended as a security feature. If you misconfigure one of your routers, your machines may still be reachable from the outside using exactly these addresses.

But still, putting all network printers in a subnet that doesn't have global addresses won't exactly hurt either.

16.1.4 Routing Configuration

Generally, unique-local addresses can be used with dynamic routing like global addresses. But on border routers we need to stop local routes from leaking to the outside. To prevent the propagation of unique-local addresses we configure our router like this:

Debian Sarge/Quagga *Adding the line*

/etc/quagga/ripngd.conf

```
distribute-list global-only out eth0
```

immediately after the `router` *statement and*

/etc/quagga/ripngd.conf

```
ipv6 access-list global-only permit 2000::/3
ipv6 access-list global-only deny any
```

at the end restricts all routing announcements to global addresses on interface `eth0`*.*

FreeBSD 6.1 *Invoking* `route6d` *with the option* `-O 2000::/3,lnc0` *limits interface* `lnc0` *to announcing global routes. To configure this permanently, we can add the option to the* `ipv6_router_flags` *variable in* `/etc/rc.conf`*.*

Solaris *There seems to be no way to make the* `in.ripngd` *restrict the routes it propagates.* 95

16.1.5 DNS Setups

Even more serious than leaking routes are DNS servers that return site-scoped addresses to the outside; connections may be established more slowly because an outside client may first try to use a site-scoped address to reach a server inside before falling back to a global address.

To work around this problem we need two name servers: One providing the full data to our internal machines and another only delivering global addresses to requests from outside. While this "split namespace" approach is admittedly expensive it is a well-established design used in many larger organizations.

With a bit of shell scripting it is generally quite simple to filter out all internal data from the internal name server and copy only the relevant data to the externally visible one.

16.2 IPv4-mapped IPv6 Addresses

In section 6.3 we first encountered *IPv4-mapped IPv6 addresses*, or *mapped addresses* for short. Now we take a closer look at them.

16.2.1 Making an IPv6 Server Support IPv4

According to RFC 3493 [45, section 3.7], mapped addresses allow a server process to serve both IPv4 and IPv6 clients through the same IPv6 socket.

When the node that hosts such a server receives an IPv4 packet at the port that the server listens on, then the kernel translates the IPv4 addresses into IPv6 addresses by appending the 32 bit IPv4 addresses to the IPv6 prefix `0:0:0:0:ffff::/96`. Using these it passes the packet through the transport layer to the server process, which considers them ordinary IPv6 traffic. When the server sends data back, the network layer extracts the embedded IPv4 addresses again and sends out the data as IPv4 packets.

Figure 16.1 shows an example. The client on the left-hand side with an IPv4 address `192.0.2.99` tries to reach the server on the right-hand side at its IPv4 address `192.0.2.1`, for example at port 80 (HTTP) to get a web page. It obtains a TCP socket from the kernel and opens a connection to the server. The client side kernel wraps up the TCP communication in IPv4 packets with a source address `192.0.2.99` and a destination address `192.0.2.1` and sends them across the network. The server kernel receives the packets. Looking

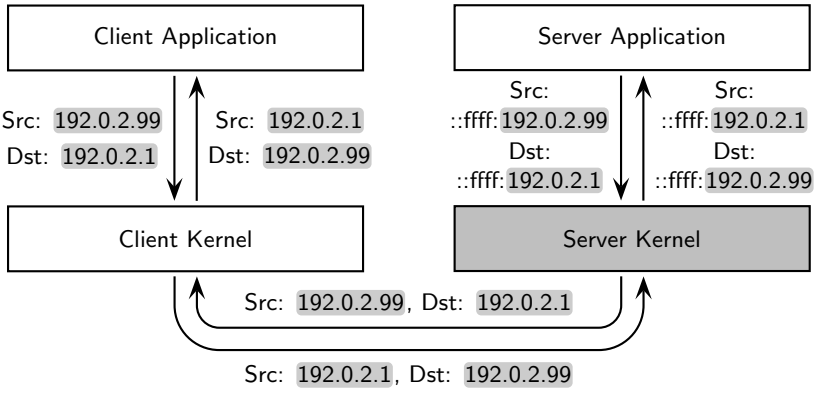

Fig. 16.1. IPv4-mapped IPv6 addresses in action

up the port number it realizes that no IPv4 server socket listens on that port but an IPv6 socket with enabled IPv4-mapped IPv6 support does. It changes the address information attached to the packets to contain the local address `::ffff:192.0.2.1` and the remote address `::ffff:192.0.2.99` and passes them through the transport layer to the server application. When the server application sends data back it will use these addresses as source and destination address, respectively. The server kernel intercepts them and turns them back to the IPv4 addresses they contain before passing any packets to the network.

While this description glosses over some interesting details concerning the communication between the network and transport layers within the server kernel, it explains how these addresses work.

From a programmer's point of view, using mapped addresses simplifies server programming quite noticeably. When a server first starts to listen on a port for incoming packets, it just needs to tell the operating system that it wants to support mapped addresses. From that point on it only deals with IPv6-based connections and doesn't need to bother about IPv4 anymore; if it wants, it can still figure out which connections are using mapped addresses. But otherwise it simply provides its service to both IPv4 and IPv6 without any extra effort.

16.2.2 Operational Aspects

From an operational point of view, mapped addresses are occasionally more troublesome.

We have already seen in section 6.3 that the `netstat` and `lsof` commands with Linux won't list server sockets that have enabled mapped addresses as open IPv4 sockets. A system administrator who doesn't care about IPv6 may thus never realize that some service is available via IPv4. Jeroen Massar recently proposed a patch to `netstat` to change this behaviour but it remains to be seen if and when this patch will make its way into the regular source base.

According to RFC 3493 [45, section 5.3] mapped addresses must not be enabled by default. Many implementations provide a global switch to control this behaviour. What's worse, some of them have mapped addresses enabled by default. With applications that expect the operating system to conform to the standards and don't explicitly disable mapped addresses this can cause serious problems. Simply setting the system-wide switch to disable mapped addresses by default may break any packages that implicitly rely on mapped addresses to be enabled.

Debian Sarge *By default, mapped addresses are enabled. There is a* `sysctl` *variable that controls this behaviour. The command*

```
# sysctl -w net.ipv6.bindv6only=1
```

or the equivalent setting in `/etc/sysctl.conf` *disables mapped addresses as the default setting.*

FreeBSD 6.1 *As the standard requires, mapped addresses are disabled by default. We can change this behaviour through the* `sysctl` *variable called* `net.inet6.ip6.v6only`*, which is set to 1 by default.*

Solaris 10 *Mapped addresses are enabled by default. There is no documented way to disable this behaviour.* 96

When configuring a packet filter, mapped addresses should always be filtered; they are only used between the kernel and local application processes.

16.3 Dynamically Changing Interface IDs

Using a globally unique interface ID poses a security risk: If a node keeps moving between networks, then a strategically placed attacker is able to track its movements within the network topology just by tracking the interface ID.

16.3.1 The "Road Warrior" Problem

Consider the average "road warrior" as an example. He travels from one place to another, using whatever network connectivity available wherever he is. He keeps however reading his e-mails using IMAP to his home server, or a public mail service provider.

Now assume an attacker to be located close to this mail server. Even when the mail traffic is encrypted, it is possible for the attacker to analyze traffic by interface IDs. If the attacker figures out the interface ID of the road warrior, then he can always discover the location of the road warrior by simply looking up the last packet with the matching interface ID; the subnet prefix tells down to the subnet where the road warrior has been last seen, network-wise.

16.3.2 Temporary Addresses

To solve this particular issue, RFC 3041 [90] devises an optional mechanism to provide for periodically changing interface IDs that make this kind of tracking infeasible. The RFC is titled "Privacy Extensions for Stateless Address Autoconfiguration in IPv6", therefore this mechanism is often referred to as *privacy extensions*. The addresses are often called *temporary addresses*. They are configured in addition to the addresses configured via autoconfiguration.

Note that changing interface IDs don't solve the similar problem that small sites and home users have with regard to their network prefix. Making that prefix change to hide from what site certain traffic originates is an entirely different issue; either use anonymizing proxies for your traffic or have your ISP dynamically assign you a different prefix every time you connect to the Internet6.

In more detail, this is how temporary addresses work: When an interface is first enabled, it does the usual autoconfiguration. It then creates a first random interface ID; this interface ID isn't globally unique so, according to the interface ID format explained in section 4.5.3, the global bit (bit 7) in the first byte is always set to zero. This is repeated for a maximum of five times if duplicate address detection indicates that the intended address is already in use.

The preferred and valid lifetimes are initially set to some configurable values (usually one day and one week, respectively) minus a random value of up to 10 minutes. Router advertisements will lower these values, but not increase them. When the preferred lifetime expires and the address becomes deprecated, then a new temporary address is created.

In theory it is possible that a temporary address is chosen that "belongs" to another machine which has been manually configured but is down when the duplicate address detection checks the address. This "problem" is mostly academic, but consider putting statically configured hosts and those using temporary addresses in separate subnets.

For the full specification, see RFC 3041.

16.3.3 Performance Considerations

What are reasonable values for the lifetime configurables? Since they are lowered by a random value of up to 10 minutes, the original values must obviously be larger than that. But there is another issue that we need to take into consideration: The number of addresses configured at a time.

Neighbor discovery requires for every address that we also subscribe to the associated solicited-node multicast address. Since multicasts are usually filtered in hardware both within a link-layer switch and the network card we must take care not to subscribe to too many such multicast addresses: If the hardware can't handle the number of subscriptions itself, then we receive all packets in the subnet and the kernel needs to do the filtering. Depending on the network traffic in the subnet this can be a major performance hit.

In addition, some implementations have a statically configurable limit imposed on the number of unicast addresses simultaneously configured on an interface.

Most applications expect addresses to be static, so we can't make the valid lifetime arbitrarily small; otherwise applications will break when an address becomes invalid and the connections that still use it fail.

The number of configured interfaces is effectively dependent on the number of advertised prefixes and the ratio of valid and preferred lifetime settings. Assuming that we know the number of prefixes $\#p$, the time we need to keep a deprecated address valid t_d and the maximum preferred lifetime t_p, we can compute the maximum number of addresses on the interface $\#a$ as

$$\#a \quad = \quad \#p \left(\frac{t_d}{t_p - 600s} + 2 \right)$$

including the link-local address. If we want to minimize the preferred lifetime setting based on a fixed maximum number of addresses on an interface, this is equivalent to

$$t_p = \frac{t_d}{\frac{\#a}{\#p} - 2} + 600s$$

and the valid lifetime setting t_v is trivially $t_v = t_p + t_d + 600s$, the sum of the preferred lifetime setting, the required deprecated lifetime and the ten minutes that may be randomly subtracted from the preferred lifetime.

If you try to minimize the preferred lifetime setting, keep in mind that you may need to configure additional prefixes temporarily when you do a network renumbering.

In general it is probably best to set the preferred lifetime to at least two hours. But even then you will probably have to lower the valid lifetime setting to a level that may upset a number of applications. Making the valid lifetime setting more than five times as large as the preferred lifetime setting is likely to make life difficult just from the number of addresses configured at any time.

Finally, if it doesn't seem to make sense to you to optimize these settings, just leave them as they are. Or don't use temporary addresses at all—this is an optional feature, not a mandatory one.

16.3.4 Configuration and Operation

How to check, enable and configure temporary addresses depends on the particular Unix again.

Debian Sarge *There is no standardized way to configure temporary addresses. Provided that the* `ipv6` *module has been loaded via* `/etc/modules` *before the interface is configured, the lines*

`/etc/network/interfaces`

```
auto eth0
iface eth0 inet manual
    up /sbin/ip -6 link set eth0 up
    up /sbin/sysctl -w net.ipv6.conf.eth0.use_tempaddr=1
    # Change and enable the next lines to configure the lifetimes
    # up /sbin/sysctl -w net.ipv6.conf.eth0.temp_prefered_lft=86400
    # up /sbin/sysctl -w net.ipv6.conf.eth0.temp_valid_lft=604800
```

in `/etc/network/interfaces` *do the trick; the first* `sysctl` *variable enables temporary addresses while the second and third set the preferred and valid lifetimes in seconds, respectively. The lifetime values shown here are the defaults (one day and one week). Note that "prefered" in* `temp_prefered_lft` *is written with a single "r" instead of the more customary "rr".*

The temporary addresses can be observed using either `ifconfig` *or* `ip`*. Since* `ifconfig` *doesn't show the lifetimes,* `ip` *is strongly preferred. Temporary addresses are marked as "secondary". Doing an* `ip -6 addr show eth0` *reveals a minor problem: The addresses are listed multiple times.*

FreeBSD 6.1 *Temporary addresses are controlled via* `sysctl`*. To enable temporary addresses, we put these lines into* `/etc/sysctl.conf`*:*

/etc/sysctl.conf

```
net.inet6.ip6.use_tempaddr=1          || Enable temporary addresses
net.inet6.ip6.temppltime=86400        || Default value (1 day)
net.inet6.ip6.tempvltime=604800       || Default value (1 week)
net.inet6.ip6.prefer_tempaddr=0       || Default value (don't prefer)
```

The preferred and valid lifetimes can be adjusted as needed. Setting the `prefer_tempaddr` *variable to* `1` *will change the system behaviour so that outgoing packets will use a temporary address by default. As we will see below, this behaviour may break applications.*

There is no way to control this behaviour for individual interfaces.

Temporary addresses are displayed using `ifconfig -a`*. They are marked as "temporary". The additional option* `-L` *displays the lifetimes as well.*

Solaris 10 *The* `in.ndpd` *takes care of temporary address handling. It expects its configuration in* `/etc/inet/ndpd.conf`*. We can either enable temporary addresses by interface or like this*

/etc/inet/ndpd.conf

```
ifdefault TmpAddrsEnabled true
ifdefault TmpValidLifetime 86400          || Default value (1 day)
ifdefault TmpPreferredLifetime 604800     || Default value (1 week)
```

for all interfaces. We can check the addresses configured using `ifconfig` *where they are tagged as "temporary" but as already explained in section 4.5.1 there is no way to display the current lifetimes.* 97

Since temporary addresses are meant to improve privacy, it doesn't make sense to put them into the DNS—at least not to make them globally visible. If we have a name server that only serves local clients, then it might make sense to put the addresses there for debugging purposes, but so far no readily available tools exist to do so.

Dealing with addresses that can't be resolved to host names by the DNS may complicate troubleshooting some problems. What's worse, the more often these addresses change, the harder it is to track down the machine causing a problem. It remains to be seen how troublesome temporary addresses will eventually prove.

16.3.5 Using Temporary Addresses

According to RFC 3484, temporary addresses must only be used if the program explicitly requests them. If they were used automatically, applications

that assume addresses to be fixed or expect a working reverse DNS entry would break.

At the time of this writing no standardized way exists to let an application make the operating system use a temporary address on its outgoing connections. A working draft exists, but it is neither formally agreed upon nor implemented yet. Applications might enumerate all of a systems addresses and filter by the global bit to look for temporary address themselves, but doing so is tedious.

As a workaround to this dilemma, some Unix implementations provide a global setting that overrides the default behaviour of using permanent addresses. Using this may cause problems, but if we really need temporary addresses, then this is most likely the way to go.

Debian Sarge *Linux prefers temporary over permanent addresses on interface* `eth0` *if we set the* `sysctl` *variable* `net.ipv6.conf.eth0.use_tempaddr` *to* `2` *instead of* `1`.

FreeBSD 6.1 *The* `sysctl` *variable* `net.inet6.ip6.prefer_tempaddr` *controls the use of temporary addresses. It defaults to* `0`*, but if we set it to* `1`*, FreeBSD will prefer to use temporary addresses for outgoing traffic.* 98

16.4 Address Selection Algorithms

Since IPv6 interfaces usually have multiple addresses assigned to them, it becomes non-trivial to choose both the local source address as well as the "best" remote destination address for a packet. RFC 3484 [26] considers this problem.

Address selection happens in two different situations: The kernel picks a source address for a packet when it needs to send a packet to a given destination and the sending application didn't provide an explicit source address; from a coder's point of view, the application didn't call `bind(2)` to set the source address.

When an application wants to send a packet to a destination, then it needs to specify the destination address. If it uses the resolver library to resolve a domain name into an address, then the resolver returns an entire list of candidate addresses; the program uses the `getaddrinfo(3)` library function to request this list. The application should then try to use each of these candidate destination addresses in turn until it succeeds in sending the packet. To minimize the number of unsuccessful attempts, the resolver sorts the list of addresses such that the ones most likely to succeed will appear first in the list. It is however left to the application to use the list in this order; if it wants to rearrange the order or only uses part of the addresses, doing so is perfectly valid.

16.4.1 The Address Selection Policy Table

Both algorithms use a *policy table* to choose the best address. The default policy table as of RFC 3484 looks like table 16.1. For a given address we search

Table 16.1. The default address selection policy table (from RFC 3484 [26])

Prefix	Precedence	Label
`::1/128`	50	0
`::/0`	40	1
`2002::/16`	30	2
`::/96`	20	3
`::ffff:0:0/96`	10	4

for the longest matching prefix in the first column as we would in a routing table. The second column then yields a *precedence value* for the address and the third a *label*. These two values are then used by the algorithms later on.

It may be slightly surprising to see the IPv4-mapped addresses in this table. But `getaddrinfo(3)` can be asked to return both IPv4 and IPv6 addresses, so it does make sense to list them here. IPv4 addresses will be considered along with IPv6 addresses using their IPv4-mapped IPv6 representation.

16.4.2 Source Address Selection

The source selection algorithm computes a single "best matching" source address for a given destination address. It starts with all local addresses as candidate source addresses and then filters them according to these (slightly simplified) rules.

1. If the destination address is also a candidate source address, discard all other candidates.
2. Discard all candidates with inappropriate scope.
3. Discard deprecated addresses if preferred ones are still available.
4. With mobile IPv6, discard "care-of addresses" (we'll learn about mobile IPv6 in chapter 22).
5. If a candidate address is assigned to the interface that we'd reach the destination through, then discard all candidates assigned to other interfaces.
6. If a candidate exists that is matching the same label in the policy table as the destination address, then discard all candidates with a different label.
7. Prefer permanent ("public") addresses over temporary "privacy extension" addresses unless the application demands otherwise.
8. Prefer addresses that share the longest prefix with the destination.
9. Choose one of the remaining addresses in whatever implementation-dependent way.

From a system administrator's point of view we should only need to influence this algorithm through the labels in the policy table, but as we have seen in the previous section, some implementations let us interfere with the choice of temporary over permanent addresses on a global basis.

16.4.3 Destination Address Ordering

The destination address selection algorithm behaves differently in that it doesn't return a single address but rather sorts the list of destination addresses.

Before the ordering algorithm starts it first obtains an unsorted list of destination addresses from the resolver library. For each destination address it computes the associated source address using the algorithm from the previous section. Then it sorts the destination addresses according to these criteria:

1. Prefer addresses that have a usable source address over those that don't or are otherwise unreachable, for example when an interface is down or no usable route to them exists.
2. Prefer addresses that have the same scope as their associated source address.
3. Prefer destination addresses with a source address that is preferred over those with a source address that is deprecated.
4. Prefer "home addresses" over "care-of addresses"; again, this refers to mobile IPv6.
5. Prefer addresses where the destination address and its source address have the same label in the address selection policy table.
6. Prefer destination addresses with a high precedence in the policy table.
7. Prefer native addresses over tunnel addresses like 6to4 addresses.
8. Prefer addresses with a small scope.
9. Prefer addresses with a long common prefix between the destination address and the associated source address.
10. Prefer addresses in the order of the original list returned from the resolver.

Together these criteria let the resolver library sort all destination addresses in a fully deterministic order.

16.4.4 Tuning the Policy Table

In most cases the default policy table is perfectly reasonable and shouldn't be modified. But in some cases it may be useful to tune it to a local network's peculiarities.

If we use multiple prefixes in our local site, we may want to give all of them a high precedence and a common label to ensure that traffic within our site uses these addresses. If we want to prefer IPv4 over IPv6, then we increase the precedence of the IPv4-mapped addresses.

How the table is manipulated as usual depends on the particular implementation.

Debian Sarge *There is no documented way to manipulate the default policy table.*

FreeBSD 6.1 *The command* `ip6addrctl` *controls the policy table similar to a routing table or packet filter rule set.*

Using it without arguments displays the current policy table. To add an entry,

```
# ip6addrctl add 2002:c000:205::/48 45 999
```

associates the prefix `2002:c000:205::/48` *with the precedence 45 and the label 999 while*

```
# ip6addrctl delete 2002:c000:205::/48
```

deletes the line again. Additionally,

```
# ip6addrctl flush
```

deletes the entire table and

```
# ip6addrctl install /etc/ip6addrctl.conf
```

installs a configuration table from the file `/etc/ip6addrctl.conf`*.*

If we want to install a custom configuration from boot, we put it into `/etc/ip6addrctl.conf` *and set the variable* `ip6addrctl_enable=YES` *in* `/etc/rc.conf`*.*

Solaris 10 *The command* `ipaddrsel` *without arguments displays the policy table. Doing so reveals that Solaris doesn't use numeric values for the labels but arbitrary strings.*

To change the table we need to read it from a configuration file like

```
# ipaddrsel -f /etc/inet/ipaddrsel.conf
```

which loads it from `/etc/inet/ipaddrsel.conf`*, which is the configuration file used at boot. Finally, we can run*

```
# ipaddrsel -d
```

to revert to the default policy table.

99

But since the policy table needs to be changed on all machines individually, doing so is an expensive effort that should be avoided if possible.

16.5 Stateless Autoconfiguration Tuning

In section 4.3 we have seen how stateless autoconfiguration provides hosts with all the information they need to configure their addresses and default

routers. So far we have assumed that the router advertisement daemon uses reasonable defaults for a number of parameters. Now we take a closer look at some of its tunables.

The exact meaning of the individual tunables is defined in RFCs 2461 [91] and 2462 [110].

As usual, different Unixen have different advertisement daemons that are configured differently.

Debian Sarge *The* `radvd` *needs a configuration file* `/etc/radvd.conf`*. All configuration is specific to individual interfaces. We have already seen examples in section 4.3.3, like*

/etc/radvd.conf

```
interface eth0 {
     AdvSendAdvert on;
     prefix 2001:db8:fedc:abcd::/64 { };
};
```

Interface-specific parameters are configured like the `AdvSendAdvert` *flags above. For individual prefixes we can set more parameters in the braces following the prefix; so far the default settings were sufficient so the braces after the* `prefix` *statement were empty.*

FreeBSD 6.1 *So far we ran the* `rtadvd` *daemon without explicit configuration. Without a configuration file it simply assumes that all prefixes configured on a router's interfaces are to be advertised and all tunables are set to their default values; we just controlled the list of interfaces it serves on the command line, but even that isn't strictly necessary. If we create a configuration file* `/etc/rtadvd.conf`*, then we can control the tunable parameters with it. The* `rtadvd` *will still use the prefix information it obtains from the interface configurations unless we start it with the additional option* `-s` *to disable this behaviour.*

The configuration uses `termcap` *syntax, so this is probably the perfect occasion to point out that the Linux* `radvd` *is also available from the ports collection. As a starting configuration we explicitly set all tunables to their default settings like this:*

/etc/rtadvd.conf

```
default:\
    :chlim#64:raflags#0:rltime#1800:rtime#0:retrans#0:\
    :pinfoflags="la":vltime#2592000:pltime#604800:mtu#0:
lnc0:\
    :addr="2001:db8:fedc:abcd::":prefixlen#64:tc=default:
```

For every interface there exists a single configuration line. Continuations are indicated by a trailing backslash. Individual parameters are separated by colons. Empty parameters are ignored, so after a continuation we can start a

line with white space and another colon to make the configuration more readable. Numeric values are assigned using a hash mark "`#`" and strings using an equal sign "`=`". Strings that contain colons must be quoted in double quotes. The special parameter `tc` defines an entry that contains default settings to be inherited; it must be the last parameter in a configuration.

Solaris 10 *The `in.ndpd` is unusual because it implements router discovery for both hosts and routers as well as the RFC 3041 privacy extensions. If no configuration file `/etc/inet/ndpd.conf` exists, is empty or doesn't set the `AdvSendAdvertisements` flag on an interface, then the node will be considered a host.*

Configuration is done per interface or per prefix. Additionally, default settings for all interfaces and prefixes can be defined. Every prefix or interface is configured in a single configuration line; a trailing backslash indicates that the next line is a continuation of the current. Prefix configurations start with `prefix 2001:db8:fedc:abcd::/64` or `prefixdefault` for the default prefix configuration. Interface configurations similarly start with `if pcn0` or `ifdefault` for the default interface configuration. 100

16.5.1 Tuning the Advertising Interval

Advertising routers periodically send unsolicited router advertisements. To avoid synchronization effects, the time between advertisements is randomly chosen between a minimum and maximum value. In highly volatile environments it may be useful to shorten these intervals if hosts are often connected to different environments—in theory hosts should send router solicitations whenever their network connectivity changes, but in practice it may be difficult to detect a change in network connectivity on certain media or with some device drivers.

According to RFC 2461 [91, section 6.2.1] the maximum interval (MaxRtrAdvInterval) between unsolicited advertisements defaults to 600 seconds. It must be within the range from 4 to 1800 seconds. The minimum interval (MinRtrAdvInterval) defaults to 0.33×MaxRtrAdvInterval, or 200 seconds with the default maximum interval. Its permissible range is from 3 seconds to 0.75×MaxRtrAdvInterval.

Debian Sarge *The intervals can be configured as*

`/etc/radvd.conf`

```
interface eth0 {
     MaxRtrAdvInterval 30
     MinRtrAdvInterval 5
     [...]
};
```

FreeBSD 6.1 *The parameters are called `mininterval` and `maxinterval`. We can be set them like*

/etc/rtadvd.conf

```
default:\
        :mininterval#5:maxinterval#30:
```

or similar.

Solaris 10 *The intervals are set per interface using a configuration like*

/etc/inet/ndpd.conf

```
ifdefault MaxRtrAdvInterval 30 MinRtrAdvInterval 5
```

101

By default the minimum interval is set to 0.33×MaxRtrAdvInterval. If we set the maximum interval to anything less than 10 seconds, then the minimum interval will default to less than 3 seconds, so we must also set the minimum interval to an acceptable value. Depending on the implementation the daemon will otherwise refuse to start (Debian Sarge, FreeBSD 6.1) or violate the standard (Solaris 10).

Testing the configuration is fairly straightforward:

- ☐ Make sure that the router advertisement daemon is actually running.
- ☐ With a packet sniffer watch for the router advertisements and how often they are sent.

16.5.2 Per-interface Information

Router advertisements contain a variety of information that we can configure to our needs. Some of them relate to an individual interface and others to the prefixes advertised. First we consider the interface-specific fields.

The most important is the *router lifetime*, which defines for how much longer a router is willing to serve as a default router. Setting the lifetime to zero tells the hosts not to use the router as a default router at all—when we built our single-legged routers in section 4.3.3 we should have set this to zero. When a router is shut down it should send a final router advertisement with this value set to zero. Permissible non-zero values are from the maximum router advertisement interval up to 9000 seconds, or 2.5 hours. The router lifetime defaults to three times the maximum router advertisement interval.

RFC 4191 [27] introduced the 2 bit *router priority* field. It defaults to 0, or medium priority, and can be changed to 1 for high or −1 for low priority.

We can also set the *current hop limit* field and the *link MTU* option. A host will initialize the hop limit of an outgoing packet with the current hop limit, or to its default value (usually 64) if the router sets this field to zero. On link layers that have a variable MTU, the link MTU option is used to set the MTU on the attached interface.

In rare cases it may also be interesting to distribute non-standard timing parameters for neighbor discovery and neighbor unreachability detection. The *reachable time* field sets the time (in milliseconds) that a neighbor discovery cache entry stays "reachable". The *retransmit timer*, also in milliseconds, specifies the time between consecutive neighbor discovery messages during address resolution and neighbor discovery. Both may be set to zero to indicate that they don't contain any valid data.

The *managed flag* and *other stateful configuration flag* are used with DHCPv6 and covered in chapter 19. Finally, RFC 3775 [75] introduced a *home agent flag* that identifies a router as a mobile IPv6 home agent; we'll take a look at mobile IPv6 in chapter 22.

Debian Sarge *We configure the interface-specific fields like this:*

/etc/radvd.conf

```
interface eth0 {
     AdvSendAdvert true;
     AdvDefaultLifetime 9000;          || Router lifetime (seconds)
     AdvDefaultPreference medium;      || Router priority; also "low" or "high"
     AdvCurHopLimit 96;                || Current hop limit
     AdvLinkMTU 1400;                  || Link MTU
     AdvReachableTime 3000;            || Reachable time (milliseconds)
     AdvRetransTimer 50;               || Retransmit timer (milliseconds)
     [...]
}
```

FreeBSD 6.1 *A sample configuration might look like this:*

/etc/rtadvd.conf

```
lnc0:\
        :rltime#9000:\                 || Router lifetime (seconds)
        :raflags="":\                  || Router flags; see below
        :chlim#96:\                    || Current hop limit
        :mtu#1480:\                    || Link MTU
        :rtime#3000:\                  || Reachable time (milliseconds)
        :retrans#50:                   || Retransmit timer (milliseconds)
```

The `raflags` *parameter is a string that may contain either a "*`h`*" or "*`l`*" character to indicate high or low router preference; it may also contain additional letters related to DHCPv6.*

Solaris 10 *The configuration is similar:*

/etc/inet/ndpd.conf

```
if pcn0 \
    AdvSendAdvertisements true \
    AdvDefaultLifetime 9000 \       || Router lifetime (seconds)
    AdvCurHopLimit 96 \             || Current hop limit
    AdvLinkMTU 1480 \               || Link MTU
    AdvReachableTime 3000 \         || Reachable time (milliseconds)
    AdvRetransTimer 50              || Retransmit timer (milliseconds)
```

Solaris doesn't yet support the router priority flags, which were added to the specifications only recently. 102

Testing the configuration is quite simple if we have a packet sniffer in the subnet.

- ☐ Start the packet sniffer.
- ☐ Start the router advertisement daemon.
- ☐ Check that the router advertisements show the expected properties.

16.5.3 Subnet Prefix Information

Beyond the interface-specific information, router advertisements contain some information about every subnet prefix they advertise.

Obviously, the prefix itself must be included. Together with the prefix, the prefix length is sent; even though RFC 4291 [64] requires the prefix length of every subnet to be 64 bits, this is explicitly included.

We have already learned about the preferred and valid lifetimes and their role with stateless autoconfiguration.

Additionally, three flags are available. The *on-link flag* (sometimes called *L-bit*) indicates that all addresses with the given prefix are connected to the subnet; it is normally set to 1. The *autonomous flag* (also called *A-bit*) is normally set to 1 to mark the prefix as suitable for stateless autoconfiguration, which is called "autonomous" address configuration in RFC 2461 [91].

The last one, called *M-bit*, is used for mobile IPv6.

The flags are usually not relevant. But network renumberings make it necessary to set the preferred and valid lifetimes. According to RFC 2461 [91, section 6.2.1] implementations must provide the means to specify the lifetimes both as a time relative to the moment when the advertisement is sent, like "two hours", and an absolute time by when the lifetime expires, like "10/10/2006 0:00 GMT". In practice, not all implementations let us configure the lifetimes as absolute values.

Debian Sarge *The configuration for a single prefix looks like this:*

/etc/radvd.conf

```
interface eth0 {
    AdvSendAdvert on;
    prefix 2001:db8:fedc:abcd::/64 {
        AdvPreferredLifetime 0;        || Default value is 7 days
        AdvValidLifetime 600;          || Default value is 30 days
        AdvOnLink on;                  || Default setting; could also be "off"
        AdvAutonomous on;              || Default setting; could also be "off"
    }
}
```

There is no way to set fixed expiration dates for the preferred and valid lifetimes.

FreeBSD 6.1 *The configuration for a single prefix looks like this:*

/etc/rtadvd.conf

```
lnc0:\
    :addr="2001:db8:fedc:abcd::":\|| The first prefix...
    :prefixlen#64:\              || ... and its length
    :pltime#7200:\               || The pltime (default 30 days)
    :pltimedecr:\                || Decrement pltime
    :vltime#10800:\              || The vltime (default 7 days)
    :vltimedecr:\                || Decrement vltime
    :pinfoflags="la":            || On-link/autonomous flags (default)
```

The meaning of the `addr`, `prefixlen`, `pltime` *and* `vltime` *parameters should be obvious. The Boolean* `pltimedecr` *and* `vltimedecr` *parameters indicate that the preferred and valid lifetimes should be decremented so they expire after the time defined in* `pltime` *and* `vltime`*, respectively. The* `pinfoflags` *parameter contains the on-link ("l") and autonomous ("a") flags; these are set by default.*

To add another prefix with another set of parameters, we can append them to the interface configuration. The parameter names are appended a number, so for the second prefix we define its address in `addr2` *instead of* `addr`*.*

Solaris 10 *The prefix configuration looks mostly as expected:*

/etc/inet/ndpd.conf

```
prefix 2001:db8:fedc:abcd::/64 pcn0 \          || The prefix and interface
    AdvPreferredLifetime 7200 \                || The preferred lifetime
    AdvPreferredExpiration "2011-11-11 11:11" \|| An absolute pltime
    AdvValidLifetime 10800 \                   || The valid lifetime
    AdvValidExpiration "2011-11-22 22:22" \    || An absolute vltime
    AdvOnLinkFlag on \                         || On-link flag
    AdvAutonomousFlag on                       || Autonomous flag
```

We can specify the preferred and valid lifetimes as relative to the time that the advertising daemon sends its advertisement with the `AdvPreferredLifetime` *and* `AdvPreferredLifetime` *parameters. Alternatively we can set absolute*

lifetimes using `AdvPreferredExpiration` *and* `AdvValidExpiration` *as the example shows. If these absolute deadlines exist, then they take preference over their relative counterparts.* (103)

We can test the prefix advertisements with a packet sniffer much like we checked the correct handling of interface-specific parameters.

- ☐ Start the packet sniffer.
- ☐ Start the router advertisement daemon.
- ☐ Check that the router advertisements show the expected properties.

16.5.4 Expiring a Prefix From a Subnet

The most common reason to meddle with the data that routers advertise is a network renumbering. We'll take a closer look at the entire procedure in chapter 24, right now we just make the router advertisement daemon expire a prefix from a subnet.

We can't just remove the prefix from the router; if we did this, then the router wouldn't know that the subnet is directly reachable and therefore couldn't deliver packets with a destination address from the prefix.

If we just set the preferred and valid lifetimes to zero, then according to what we've seen so far all hosts will mark the prefix as invalid and not use it anymore even for existing connections. An attacker connected to the subnet could therefore run a very simple denial of service attack by just sending router advertisements for the prefixes with zero lifetimes.

To avoid these attacks, RFC 2462 [110, section 5.5.3] demands that a host will behave like this only if the router advertisement was authenticated with an IPsec AH header—which isn't available for multicasts. Otherwise a host will lower the valid lifetime it stores for the prefix to no less than two hours.

It is however possible to set the preferred lifetime to zero with an appropriate router advertisement. While this might still be used for a less threatening denial of service attack under certain conditions, quickly deprecating an address is desirable when we need to fix a problem on short notice.

So unless we set up an IPsec framework that supports multicast authentication, expiring a prefix from a subnet takes at least two hours, but as long as we have another preferred prefix available we can quickly switch over to it. The deprecated prefix won't be used anymore except for existing connections.

In the previous section we have seen that some of the advertising daemons don't implement absolute expiration dates properly. The easiest strategy to deal with the situation is this:

1. To expire a prefix from the subnet we first set the preferred lifetime in the advertisement daemon's configuration to zero and restart the daemon.

2. We need to figure out for how long we need to keep a deprecated prefix before we can turn it invalid. This depends on the applications we use as well as the willingness of our upstream provider to keep the prefix routed to us. If we aren't in a rush, keeping the prefix deprecated for a week is in many cases desirable. We need to keep it for at least two hours, unless we use IPsec authentication in the subnet.
3. At least two hours before we finally want to invalidate the prefix we set its valid lifetime to zero and restart the advertising router.
4. When the address is finally expired, we remove it entirely from the router configuration.

To avoid serious problems if anything goes wrong, three measures have proven particularly useful:

- An alternate prefix, either the successor of the one we expire, or a unique-local prefix, should be configured. All nodes must be reachable through this alternate prefix.
- Every change to the router configuration must be visible in the advertisement packages. To avoid surprises it is extremely helpful to observe them in a packet sniffer.
- After every step the hosts must show an updated interface configuration. Some advertising routers don't immediately send an advertisement when they are restarted, so we must wait a while for the daemon to send an updated advertisement.

16.6 The Router Renumbering Protocol

RFC 2894 [20] defines the *router renumbering protocol*. Using site-local multicasts it provides a means to distribute new prefixes throughout a site without configuring all routers involved.

In theory, the protocol promises to be a very valuable tool. In practice, implementations are scarce at best and the idea of a user mistake, bug or denial-of-service attack using a renumbering message rendering an entire site's network out of order deters experienced administrators from using the protocol.

In the long run it is quite likely that the renumbering protocol will prove useful. Using an automatism to do some tedious and error-prone task, like changing prefix configurations in routers, can actually prevent a number of stupid mistakes. As soon as IPv6 support becomes commonplace enough that ISPs provide for the needs of end customers without network skills, the protocol will quite likely prove extremely useful. But for now router renumbering is little more than a nice idea that nobody really wants to try except possibly in a pure test environment.

17

Advanced Routing with Quagga

In chapter 7 we have taken a first look at various essential routing topics. We have used RIPng to maintain our routing tables with minimal effort. As a bonus, in a redundant network topology RIPng takes care of failing routes faster than we can do manually. But RIPng has its drawbacks: It converges too slowly to make an existing TCP connection survive a router failure, it only supports a very primitive routing metric and it doesn't scale too well in a large environment.

In this chapter we take a look at some of the more advanced features that a more powerful RIPng implementation and the OSPF routing protocol offer. While these features come at the price of running a fairly heavyweight routing protocol framework, there are many situations that simply don't leave us another choice.

Running routing protocols like OSPF has long been the domain of dedicated hardware routers. Using Unix for this purpose we are pushing it to the limit. So we restrict ourselves to some scenarios that actually occur with Unix based routers and skip those that should be left to dedicated routers.

17.1 The Quagga Routing Framework

The most popular routing framework in the Unix world is called *Quagga*, a derivative of *GNU Zebra*. Since it is powerful, complex and in many ways quite "un-Unix-ish", we take a look at it in general before we actually try to use it for some more advanced routing setups.

17.1.1 Features and Peculiarities

Compared to the RIPng daemons for FreeBSD and Solaris, Quagga is huge and complex. It provides a very flexible framework for dynamic routing, supporting multiple routing protocols at the same time. It even passes routing information between different routing protocols.

To do so, Quagga consists of a central framework daemon called `zebra`, which takes care of statically configured routes and operates as a mediator between the system's routing table and the other daemons which implement the individual routing protocols.

Quagga needs to have full control of the interface and routing configurations. It is generally a bad idea to interfere with it through the system's own network configuration boot scripts and related tools. It is up to us to choose a strategy to configure our network: If we don't configure the network in the system's boot configuration but leave the work entirely to Quagga, then we may have problems with some unrelated services that are started before Quagga. Alternatively we can manually keep the boot configuration in sync with the Quagga configuration to avoid these problems. If we accidentially let them run out of sync however, then Quagga may be affected.

As a general rule of thumb, Quagga is best run on a dedicated machine with no other services (short of the usual SSH daemon) and the network configuration left entirely to Quagga.

Another peculiarity of the Quagga framework is the way that the configuration is handled. In section 7.4 we configured Quagga on Debian by editing some configuration files. That worked fine for a minimalistic RIPng setup, but as soon as we want some more advanced configurations this will become rather troublesome. The canonical way to deal with Quagga configurations looks quite similar to the way that Cisco administrators handle their routers: There is a command line interface called the *virtual terminal interface* (*VTY*) for every daemon in the framework that we can connect to using `telnet`. This interface lets us manipulate the configuration on the fly without the need to restart a daemon after we've changed the configuration, which is a substantial advantage if we deal with large routing tables.

But a command line interface also has a number of drawbacks: Its handling is un-Unix-ish and and doesn't mix too well with established administration tools and procedures. It doesn't support comments. Yes, if we edit the files that Quagga uses to store its configuration by hand, then we can add comments. But if we change the configuration through the virtual terminal interface and save it to the file, then we overwrite all comments. The Quagga daemons expect the configuration in these files to be arranged in a specific, but undocumented, order. So writing configurations by hand is often quite tedious and error-prone, simply because the necessary documentation is missing. This can be particularly dangerous if we try to generate the configuration for a number of routers using scripts. Some routing protocols expect that all routers involved use the same settings for some parameters. If we maintain these parameters on all routers manually, then we are bound to make occasional mistakes that cause individual routers to fail. If we use a script to generate the necessary configuration, then a mistake in that script, possibly due to the missing documentation, may take all routers down.

A final peculiarity is the way how Quagga deals with non-existent interfaces. If we try to configure a non-existent interface, Quagga will actually accept that configuration without a hint that the interface doesn't exist. This can be quite unnerving if we just mistyped the interface name, but it has a huge advantage: We can set up an interface configuration before the interface actually exists. This frequently happens with tunnels and PPP interfaces. And if an interface unexpectedly goes down, then Quagga will remove the "static" routes associated with it and generally behave quite smart about the missing interface.

17.1.2 Supported Routing Protocols

In section 7.4 we have already used Quagga in a very simple setup to support RIPng on Debian Sarge. The Quagga RIPng daemon `ripngd` offers a wide range of fine-grained controls to adjust its behaviour.

If RIPng doesn't provide the features we need, Quagga also offers an OSPFv3 daemon, somewhat misleadingly called `ospf6d`. OSPFv3 is more complex to deal with, but also far more powerful than RIPng. It is the most important reason to use Quagga with IPv6.

RIPng and OSPFv3 are both *interior gateway protocols* (*IGPs*): They only operate in a contiguous set of subnets with well-defined borders called an *autonomous system* (*AS*). Neither RIPng nor OSPFv3 scale too well: RIPng is limited by its metric to networks with a diameter of 15 hops while OSPFv3 requires every router to keep track of all network links and routers within the autonomous system.

Between autonomous systems *exterior gateway protocols* (*EGPs*) are used. The de facto standard today is called the *border gateway protocol* (*BGP*). Its multi-protocol extensions provide for IPv6.

Originally, only ISPs with their own autonomous system needed to run BGP. But when users grew more and more dependent on their Internet connectivity and demanded redundant network uplinks, IPv4 had little to offer except to assign these users their own autonomous systems and provider-independent addresses and let them run their own BGP routers. Fortunately, IPv6 offers a simpler way to provide redundant network connectivity—section 25.1 explains the details.

17.1.3 Installing Quagga

To set up a Quagga-based router we start with a base installation. We don't need to configure the network since we'll take care of this through Quagga later on. Next we need to install the Quagga software and enable it. Most Unixen ship with a pre-compiled Quagga package, which is quite helpful because the source code from `http://www.quagga.net/` doesn't compile out of the box on some Unixen.

Debian Sarge *The distribution media include a* `quagga` *package containing version 0.98.3.*

FreeBSD 6.1 *Quagga version 0.98.3 is available from the ports/packages collection.*

Solaris 10 *As of today the Quagga source code doesn't compile on Solaris 10 and the Zebra binaries included on the installation media have been compiled without IPv6 support.*

Short of debugging the Quagga source code to make it compile on Solaris 10 we have two options: We can search `http://www.quagga.net/` *for a usable binary release or we can build Zebra from the* `SUNWzebraS` *source package on the installation media.*

Since we can't rely on the availability of binary packages on the Web, we install basic development environment from the packages `SUNWbzip`, `SUNWgzip`, `SUNWhea`, `SUNWsrh`, `SUNWcpp`, `SUNWsprot`, `SUNWtoo`, `SUNWarc`, `SUNWlibmr`, `SUNWlibm`, `SUNWbtool`, `SUNWgcmn`, `SUNWgccruntime` *and* `SUNWgcc` *and the Zebra source code from* `SUNWzebraS`*. Then we rebuild Zebra with the commands*

```
# cd /usr/sfw/share/src/zebra-*
# PATH=/bin:/sbin:/usr/sbin:/usr/xpg4/bin:/usr/ccs/bin:/usr/sfw/bin
# export PATH
# ./configure --prefix=/usr/sfw --sysconfdir=/etc/sfw/zebra \
              --localstatedir=/var/run --enable-ipv6 \
              --enable-vtysh
# make
# chmod 755 install-sh
# make install
```

104

Next we need to preconfigure the Quagga daemons so that they start and open the configuration port. As a first step we need to create the configuration files `zebra.conf`, `ripngd.conf`, `ospf6d.conf` and such for all daemons that we want to use. They should all contain a single line

`zebra.conf, ripngd.conf, ospf6d.conf, ...`

```
password xyzzy
```

that holds the password that admits access to the virtual terminal interface.

Just for the records: Don't use "xyzzy" as your password. It is only an example password and you are expected to choose your own. (As if you didn't know...)

Depending on the particular Unix and compilation, these files go in a different directory:

Debian Sarge *The directory is* `/etc/quagga`*.*

FreeBSD 6.1 *The directory is* `/usr/local/etc/quagga`*.*

Solaris 10 with Zebra *According to the* `--sysconfdir` *option we've used with the* `configure` *script the directory is* `/etc/sfw/zebra`*; the binary distribution also uses this directory.* 105

Now we need to enable forwarding and start Quagga. Again, different Unixen use different ways to start the daemons:

Debian Sarge *A line*

/etc/sysctl.conf

```
net/ipv6/conf/all/forwarding=1
```

in `/etc/sysctl.conf` *enables forwarding after the next reboot.*

The file `/etc/quagga/daemons` *contains the list of all Quagga daemons we want to use. For now it suffices to change the lines for* `zebra` *and* `ripngd` *to*

/etc/quagga/daemons

```
zebra=yes
ripngd=yes
```

Don't remove the remaining lines, otherwise the boot script will complain about them.

Additionally we probably want the virtual terminal interfaces of all daemons to be accessible using the IPv6 loopback interface only. The file `/etc/quagga/debian.conf` *contains the options that the boot script starts the daemons with. We should change all occurrences of "*`-A 127.0.0.1`*" to "*`-A ::1`*".*

FreeBSD 6.1 *In* `/etc/rc.conf` *we need the lines*

/etc/rc.conf

```
ipv6_enable=YES
ipv6_gateway_enable=YES
quagga_enable=YES
quagga_daemons="zebra ripngd"
quagga_flags="--daemon -A ::1"
```

to enable IPv6 in general, turn on forwarding and start the `zebra` *and* `ripngd` *daemons.*

Solaris 10 with Zebra *First we need a script to start and stop the daemons. We could take the scripts from the binary Zebra package and make them support the* `ripngd` *and* `ospf6d` *daemons, but since these scripts only run a single routing protocol, they are probably not worth the effort. So instead we write our own script* `/etc/sfw/zebra/startstop.sh`*:*

/etc/sfw/zebra/startstop.sh

```
#! /bin/sh
case "$1" in
    start) shift
           for daemon in zebra "$@"
           do
               /usr/sfw/sbin/$daemon --daemon -A ::1
           done
           ;;
    stop)  for daemon in bgpd ripngd ospf6d ripd ospfd zebra
           do
               pkill -x $daemon
           done
           ;;
    *)     echo "Usage: $0 start|stop <daemons...>"
           exit 1
           ;;
esac
```

Next we use `routeadm` *to set up the routing services.*

```
# chmod 755 /etc/sfw/zebra/startstop.sh
# routeadm -s ipv6-routing-daemon=/etc/sfw/zebra/startstop.sh
# routeadm -s ipv6-routing-daemon-args="start ripngd"
# routeadm -s ipv6-routing-stop-cmd="/etc/sfw/zebra/startstop.sh\
                                     stop"
# routeadm -e ipv6-forwarding
# routeadm -e ipv6-routing
```

To make the virtual terminal interface accessible by name we finally add the lines

/etc/services

```
zebra  2601/tcp
ripd   2602/tcp
ripngd 2603/tcp
ospfd  2604/tcp
bgpd   2605/tcp
ospf6d 2606/tcp
```

to `/etc/services`. 106

To ensure that our boot configuration is correct we should now reboot the router. We can check that everything works as expected:

- ☐ Forwarding must be enabled. On systems with `sysctl` check it with

  ```
  # sysctl -a | grep forward
  ```

  and on Solaris 10 use

  ```
  # ndd /dev/ip6 ip6_forwarding
  ```

- ☐ Use `ps` to verify that the `zebra` and `ripngd` daemons are running.
- ☐ Ensure with `telnet` that the virtual terminal interfaces work:

```
# telnet ::1 zebra
[...]
Password: xyzzy
router-1> quit
# telnet ::1 ripngd
[...]
Password: xyzzy
router-1> quit
```

- ☐ Finally use `netstat` to check that the virtual terminal interfaces only listen on the IPv6 loopback interface.

At this point we have a fully functional but yet unconfigured router at hand.

17.1.4 Using the Virtual Terminal Interface

To configure our router we first telnet to the VTY interface of the `zebra` daemon:

```
# telnet ::1 zebra
[...]
Password: xyzzy
router-1>
```

This is the *unprivileged mode* of the VTY. It behaves quite similarly to the Cisco IOS interface, so if you are familiar with IOS, then the Quagga VTY will appear quite familiar to you. If you are not, then there are a few things you will want to know: Any time that you press the '?' key you will be shown a list of all possible commands or parameters you can use. If you have already typed part of a command, then '?' will show all possible completions. If you hit the tab key instead, the VTY will complete the command if the part you typed is already unambiguous. But even if you don't type the tab key the VTY will recognize and accept the command; the command `list` shows all the possible commands currently available. So typing `s in` is automatically interpreted as `show interface` and lists the current interface configuration. Since we are still in unprivileged mode, there is little harm we can do trying out all commands except for `enable`—you may want to take a look around by yourself. To leave the VTY, use the command `quit`.

When we enter `enable` (or `en` for short) in unprivileged mode, then the VTY switches to *privileged mode* (you may also hear the term *enable(d) mode*). The command prompt changes from '`>`' to '`#`' and both '`?`' and `list` show a larger choice of commands to issue. The first one to try is `show running-config`, which displays the current configuration of the `zebra` daemon.

From privileged mode we can enter *configuration mode* using the command `configure terminal`, which doesn't configure the terminal settings

but reads configuration changes from the terminal. In configuration mode we can use the `hostname` command to set the hostname (which should really be "routername") used within Zebra:

```
# telnet ::1 zebra
[...]
Password: xyzzy
amnesiac> enable
amnesiac# configure terminal
amnesiac(config)# hostname router-1
```

At this point we have only changed the *running configuration* that the daemon currently uses. To make the change permanent we need to copy it to the *startup configuration* which is kept in the configuration file `zebra.conf`:

```
router-1(config)# write file
Configuration saved to /etc/quagga/zebra.conf
router-1(config)# quit
router-1# quit
```

If we want to undo the hostname configuration, then we can enter configuration mode mode again and with a command

```
router-1(config)# no hostname
```

remove the hostname configuration from the running configuration. The command prefix `no` lets us remove most configuration lines from the configuration.

17.1.5 Interface and Static Route Configurations

We still haven't configured our network interfaces yet. Assume that we have a Linux router with interfaces `eth0` and `eth1` and want to assign them the addresses `2001:db8:fedc:1::1` and `2001:db8:fedc:2::1`, respectively. We first need to enter configuration mode again. From there, we can select each interface in turn using the `interface` command, set its configuration and leave the interface configuration again:

```
router-1> enable
router-1# configure terminal
router-1(config)# interface eth0
router-1(config-if)# no ipv6 nd suppress-ra
router-1(config-if)# ipv6 address 2001:db8:fedc:1::1/64
router-1(config-if)# quit
router-1(config)# interface eth1
router-1(config-if)# ipv6 address 2001:db8:fedc:2::1/64
router-1(config-if)# quit
router-1(config)# write file
router-1(config)# quit
router-1# quit
```

We should quickly check that the configuration is correct:

- ☐ Using `ifconfig -a` or `ip -6 addr show` from the command line verify that the `zebra` daemon has set the interface configuration correctly.

Now we may want to add a static route, say to `2001:db8:fedc:3::/64` with the next-hop router at `2001:db8:fedc:2::2`. Again we enter configuration mode as before, set the route using the command

```
router-1(config)# ipv6 route 2001:db8:fedc:3::/64 2001:db8:fedc:2::2
```

and then save the configuration (which we won't show in the console transcripts anymore). To check that the route is correctly set, we leave the VTY and use the operating system's features to display the routing table.

Solaris 10 with Zebra *Different to the other Unixen we need to configure the interfaces using empty* `/etc/hostname6.`⟨*interface*⟩ *files because Solaris needs to plumb the interfaces before Zebra can detect and configure them.* 107

17.1.6 Router Advertisements

The `zebra` daemon also offers router advertisements. We must configure them for every interface individually, which gives us some fine-grained control but makes configuration tedious on routers with many interfaces. Additionally, we need to configure the prefixes to advertise; they are not derived from the addresses configured to the interface.

To enable router advertisements on interface `eth0` on our router from the previous section, we first go into configuration mode and then enter the commands

```
router-1(config)# interface eth0
router-1(config-if)# ipv6 nd prefix 2001:db8:fedc:1::/64
router-1(config-if)# no ipv6 nd suppress-ra
```

Afterwards we proceed accordingly for interface `eth1` and save the configuration. To test the configuration we set up a host in each subnet and verify that it receives the correct prefix and router information.

Solaris 10 with Zebra *The Zebra framework doesn't support router advertisements, so we need to use the* `in.ndpd` *daemon for them.* 108

17.1.7 Debugging Capabilities

Before we engage the first dynamic routing protocol, we finally take a look at the debugging capabilities of the Quagga framework.

First we need to decide what debug information we are interested in. The `zebra` daemon lets us log internal events, the communication between `zebra`

and the kernel, and packets that `zebra` sends or receives. These categories are similar to the facility parameter in `syslog` messages. Additionally, all messages have a `syslog`-style priority level; we can choose from what level on we want to log messages. Finally we need to tell `zebra` where to send the debug messages. Similar to Cisco routers we can log everything to the VTY, but we can also send the messages to a log file—which some Cisco routers can't because they don't have a hard disk.

If we want to log all messages related to the communication with the kernel at a priority level of "informational" and above to a file `/tmp/zebra.log`, then we enter configuration mode and use the commands

```
router-1(config)# debug zebra kernel
router-1(config)# log file /tmp/zebra.log informational
```

to set the debugging configuration as we need.

The other Quagga daemons also have similar debugging capabilities that we can use in the same way.

17.2 RIPng Revisited

In section 17.1.3 we have already enabled the `ripngd` daemon. Different than the lightweight RIPng implementations we have already seen with FreeBSD and Solaris, `ripngd` needs some basic configuration before it starts to propagate routes.

17.2.1 Enabling RIPng Support with Quagga

To configure the `ripngd` daemon we need to telnet to its own port, which is different from the one `zebra` uses:

```
# telnet ::1 ripngd
```

The interface looks quite similar to the one we have seen with `zebra`. Again we need to enter configuration mode. If we just want to run RIPng on all interfaces, then we first enable the RIPng routing protocol using the `router ripng` command, make it run on all interfaces with the `network` command and redistribute all connected subnet prefixes.

```
router-1(config)# router ripng
router-1(config-router)# network ::/0
router-1(config-router)# redistribute connected
```

The `redistribute` command offers us some control of which kind of routes to distribute. Besides directly connected routes we can choose to include static routes, routes found in the kernel from elsewhere, or routes received via OSPFv3 or BGP.

Now we should check that routes propagate as expected. Using the generic `netstat -r` command or similar will show us the kernel's routing table. But Quagga offers a more useful alternative with the `show ipv6 ripng` command:

```
router-1> show ipv6 ripng
Codes: R - RIPng, C - connected, S - Static, O - OSPF, B - BGP
Sub-codes:
      (n) - normal, (s) - static, (d) - default, (r) - redistribute,
      (i) - interface, (a/S) - aggregated/Suppressed

   Network      Next Hop                      Via     Metric Tag Time
C(i) 2001:db8:fedc:1::/64
                  ::                         self       1    0
C(i) 2001:db8:fedc:2::/64
                  ::                         self       1    0
R(n) 2001:db8:fedc:3::/64
                 fe80::20c:29ff:fed9:3d85    eth1       2    0  02:54
```

The last entry in this example configuration shows that the `ripngd` installed a route to `2001:db8:fedc:3::/64` through `fe80::20c:29ff:fed9:3d85` on interface `eth0`, which has a RIPng metric of 2 and will expire in 2:54 minutes.

17.2.2 Limited Route Distribution

So far we haven't achieved anything that the lightweight RIPng daemons we have used in section 7.4 can't do. But Quagga offers us a number of features way beyond the capabilities of the lightweight RIPng daemons.

If we want to use a leaf/backbone topology as in section 7.7.7, then Quagga starts to become interesting: It lets us restrict dynamic routing to individual interfaces and network prefixes.

```
router-1(config)# router ripng
router-1(config-router)# network eth0
router-1(config-router)# network 2001:db8:fedc::/48
router-1(config-router)# redistribute connected
```

This configuration will support RIPng only on interface `eth0` and all other interfaces that are configured with an address from the `2001:db8:fedc::/48` range. For the leaf/backbone topology we can simply enable RIPng on the backbone interfaces only.

In some cases we may want a router to receive routing information on an interface but not to send out its own routing table on that interface. We can configure such a *passive interface* similarly to a normal interface:

```
router-1(config)# router ripng
router-1(config-router)# passive-interface eth1
```

Passive interfaces are particularly useful if we want to avoid routing traffic

through a *multi-homed host*, a node with multiple interfaces that doesn't forward any packets.

Finally, *access lists* offer control of what routes to propagate. Assume that we want RIPng to propagate only local routes from the `2001:db8:fedc::/48` prefix and a default route. Then we first set up an access list we call `ourfilter` using the commands

```
router-1(config)# ipv6 access-list ourfilter permit
                                        2001:db8:fedc::/48
router-1(config)# ipv6 access-list ourfilter permit ::/0 exact-match
router-1(config)# ipv6 access-list ourfilter deny ::/0
```

Access lists follow first-match semantics, so this access list will permit all prefixes from the `2001:db8:fedc::/48` range and a default route. All other prefixes will be blocked.

Now we set up the router to use our access list. We can apply it to incoming or outgoing RIPng responses, to an individual interface or in general. For our purposes

```
router-1(config)# router ripng
router-1(config-router)# distribute-list ourfilter in
```

is perfectly sufficient; it applies the filter to incoming RIPng responses on all interfaces. If we wanted to filter outbound RIPng responses we'd use `out` instead of `in` and if we wanted to apply the filter only to an individual interface we'd append the interface names.

17.2.3 Metric Tuning

Towards the end of section 12.3.2 we considered a tunnel scenario that made it necessary to manipulate the metric associated with an interface. Quagga offers us several ways to do that.

The `redistribute` command has an option `metric` $\langle n \rangle$ that sets the metric of all matching routes. For example,

```
router-1(config)# router ripng
router-1(config-router)# redistribute connected metric 2
router-1(config-router)# redistribute static metric 5
router-1(config-router)# redistribute kernel metric 8
```

will redistribute all directly connected prefixes with a metric of 2, all static routes with a metric of 5 and all routes found in the kernel routing table with a metric of 8.

But in more complex cases, like the tunnel scenario mentioned, this isn't enough. What we want here is a way to add some extra "metric penalty" to individual interfaces and address ranges. The `offset-list` command offers us both.

Assume that we want to add an extra metric of 4 on all routes from the metric range 2001:db8:ffff::/48 we receive on interface eth1. We set up an access list far-away and then apply the offset-list command to it:

```
router-1(config)# ipv6 access-list far-away permit
                                          2001:db8:ffff::/48
router-1(config)# ipv6 access-list far-away deny any
router-1(config)# router ripng
router-1(config-router)# offset-list far-away in 5 eth1
```

If we omit the interface name in the last line, then the offset will be applied to all packets that match the far-away access list. If we set up an access list that applies to ::/0, then we can use offset-list based on the interface only. Finally, it is also possible to apply an offset-list on an outbound interface if we use the keyword out instead of in.

17.2.4 Route Aggregation

Quagga has an undocumented feature that lets us aggregate multiple routes within a router. Experiments show that we can configure the router with a set of aggregatable addresses like this:

```
router-1(config)# router ripng
router-1(config-router)# aggregate-address 2001:db8:fedc:0::/63
```

This will tell the router to merge the prefixes 2001:db8:fedc:0::/64 and 2001:db8:fedc:1::/64 to a single prefix 2001:db8:fedc:0::/63. Multiple aggregate-address commands may be given. With such a configuration, a router will announce a route to the aggregated prefix with a metric of 1 as soon as it obtains a route to any of the non-aggregated /64 prefixes.

This feature is useful if we have multiple network clouds that run RIPng and only connect to each other through a few border routers. If we use route aggregation on these border routers, then a router in one cloud will only see an aggregate route to all subnets in the other clouds. This will reduce the workload on all routers involved. Additionally, it allows each cloud to have a network diameter of 14 hops and therefore lets us use RIPng even in fairly large networks.

But route aggregation with RIPng is risky: A RIPng router can't know if it has a route to all subnet prefixes within an aggregate prefix. Using route aggregation may cause a router to announce a route to a subnet that it doesn't actually have, and at a metric that may let it shadow an available route elsewhere. Besides, using an undocumented feature should always be a last resort only.

17.2.5 Non-standard Timing Parameters

It is possible to change the default timing parameters of the Quagga RIPng implementation. This violates the standard, either increases the convergence

time or the RIPng network traffic and causes instable routing if the parameters aren't configured the same on all routers in a subnet. Still, certain exceptional situations may force us to use this feature.

There are three timing parameters that we can tune: The interval at which the router sends its route announcements, the time after which a received route expires and the time after which an expired route is entirely forgotten.

The command `timers basic` lets us set these parameters. To make RIPng converge as fast as reasonably possible without regard for the traffic involved, we can set these parameters using the command

```
router-1(config)# router ripng
router-1(config-router)# timers basic 1 5 20
```

to send announcements every second, expire received routes after five seconds and drop expired routes after twenty seconds.

If we accidentially connect a router with default timer settings to the same subnet as another router with these custom settings, then both together may disrupt dynamic routing within the entire network cloud: Whenever the "default" router sends an unsolicited response, the "fast" router receives it, realizes that it found a new route, and sends triggered updates to all its other interfaces. Five seconds later this new route expires, so the "fast" router sends out another triggered update that marks the route as unreachable again. When the "default" router sends another unsolicited response, then the same sequence of events repeats.

So again, meddling with these parameters puts the entire dynamic routing infrastructure at risk. Don't to it unless you absolutely have to.

17.3 Open Shortest Path First (OSPF), version 3

Even though the previous sections show that RIPng can be coerced to do a number of advanced tricks if the implementation supports them, they also show that we are pushing its limits. If we really need these tricks to make RIPng do what we want, then we are quite likely trying to use the wrong tool for the job. The *open shortest path first, version 3* (*OSPFv3* or *OSPF*) routing protocol is more complex to configure and operate than RIPng in a simple, straightforward environment, but in a complex setup it quickly becomes the preferred routing protocol because it offers a number of features we may need.

OSPFv3, as specified in RFC 2740 [16], is a straightforward port of OSPFv2, an IPv4-only routing protocol specified in RFC 2328 [89].

17.3.1 Features and Limitations

OSPF has some major advantages over RIPng: It recovers from network failures within a few seconds rather than several minutes, offers a powerful

metric, supports networks with large diameters and lets us aggregate subnet prefixes into larger routing prefixes.

It also has a number of disadvantages: OSPF is a very complex protocol and is therefore more susceptible to faulty implementations. Different than RIPng it needs some configuration before it can actually run on a router. And even though it does support networks with large diameters, it doesn't scale too well simply because it needs far more resources than RIPng.

The Quagga implementation of OSPF for IPv6 has at least two more drawbacks: According to its documentation, at least up to version 0.98.3 there is no support for prefix aggregation through so-called "areas" and a router can't run multiple instances of OSPF at the same time.

Solaris 10 with Zebra *Apparently the* `ospf6d` *daemon doesn't work—it fails to establish an adjacency with its peers. So for the rest of this chapter we ignore this implementation.* 109

17.3.2 Basic Concepts

OSPF is an interior routing protocol like RIPng, so it runs within a well-defined contiguous network cloud; the OSPF specifications frequently assume that we run a single OSPF instance throughout an entire autonomous system. All OSPF routers within this network cloud keep track of the entire network topology and the *link state* of all routers within the cloud. In OSPF terminology, a *link* is not a subnet as in IPv6 terminology, but a connection between two routers or a router and an attached subnet. As soon as a router learns that a link has changed, it can then quickly recompute its routing table.

Link state changes are quickly announced through the entire cloud by *flooding*: A router that learns about a link state change will immediately send a *link state advertisement* (*LSA*) to all other routers that it is directly connected to except for the one that it received the change from. To avoid a network storm, routers that receive a change that they already know about will ignore that change.

These two properties make OSPF so powerful: An OSPF router always has very up-to-date information about the state of the entire network and uses this information to set up a matching routing table. RIPng on the other hand takes three minutes to decide that a neighboring router has become unreachable and then only propagates this information at an average 3 seconds per hop.

17.3.3 Essential Configuration

The OSPFv3 daemon within the Quagga framework is named `ospf6d`. To enable it we first create a configuration file `ospf6d.conf` in the Quagga configuration directory that holds the usual password line:

ospf6d.conf

```
password xyzzy
```

(Again, replace "xyzzy" with your own password.) As with `ripngd` we need this line so we can access the `ospf6d` daemon through the VTY interface. Next we start the daemon. Again we follow the same procedure as with `ripngd`:

Debian Sarge *In `/etc/quagga/daemons` we must change the line related to `ospf6d`. Additionally we turn off RIPng support at least for now:*

/etc/quagga/daemons

```
zebra=yes
ripngd=no
ospf6d=yes
[...]
```

Again, the boot script will complain if we remove any daemon from this list completely, so we must only change these three lines.

FreeBSD 6.1 *To enable the `ospf6d` and disable `ripngd` again, we set the variable `quagga_daemons` in `/etc/rc.conf` accordingly:*

/etc/rc.conf

```
quagga_daemons="zebra ospf6d"
```

110

Now we either reboot and make sure that our router starts up properly, or restart the Quagga daemons manually. Afterwards we configure the `zebra` daemon as before, assigning addresses to interfaces and enabling router advertisements if necessary.

Within an OSPF cloud every router must be assigned a unique *router ID* to identify it. This router ID is a 32 bit integer that "happens to be written in dotted-quad notation like an IPv4 address". For obvious reasons IPv4 network administrators routinely assign a router one of its IPv4 addresses as the router ID. Unfortunately OSPFv3 still uses the same format for the router ID, so we can't just use the interface ID of an interface as the router ID. Instead, we must assign the router ID manually.

Additionally we need to assign all interfaces to an *area*, which is a sub-cloud within our OSPF cloud. Areas are identified by another 32 bit integer that again "happens to be written in dotted-quad notation like an IPv4 address". We'll take a closer look at OSPF areas in section 17.3.9. For now we assign all interfaces to the so-called *backbone area* with the special area ID `0.0.0.0`.

To configure a router we telnet to its `ospf6d` port and use the VTY interface of the `ospf6d`:

```
# telnet ::1 ospf6d
[...]
Password: xyzzy
router-1> enable
router-1# configure terminal
router-1(config)# router ospf6
router-1(config-ospf6)# router-id 255.0.0.1
router-1(config-ospf6)# interface eth0 area 0.0.0.0
router-1(config-ospf6)# interface eth1 area 0.0.0.0
router-1(config-ospf6)# interface eth2 area 0.0.0.0
```

At this point our OSPF router is ready to run.

17.3.4 A Simple Test Setup

Now that we have seen how to configure an OSPF router it is time to set up a test environment that lets us experiment with OSPF. The examples in the next few sections refer to the environment shown in figure 17.1. It models

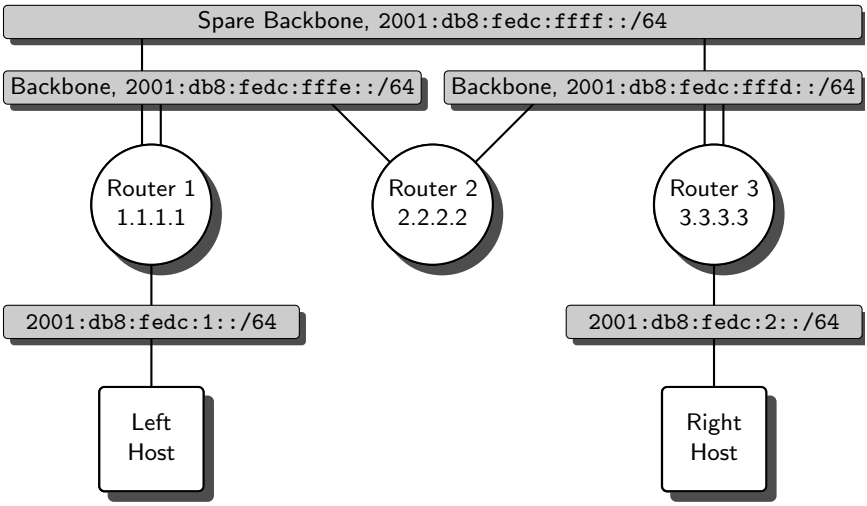

Fig. 17.1. A simple OSPF test setup

a small part of a standard two-tiered network topology with a redundant backbone at the top and two leaf networks towards the bottom. The backbone consists of a single "spare backbone" subnet and a "main backbone" with two subnets and router 2 connecting these subnets. To each leaf subnet connects a host. The routers connected to the leaf subnets provide them with router advertisements.

Note that the backbone subnets don't need any subnet prefixes assigned to them. Routers only communicate using multicasts and their link-local

addresses, so it isn't strictly necessary to assign any subnet prefixes here. Without them, reaching router 2 from one of the hosts is impossible, however. This can be considered a security feature, but it can also be a serious problem if the router needs remote attendance.

Debian Sarge *We need to assign an address to every router interface, even those connected to the backbone, otherwise the router won't set up a connection to its peers.* 111

We should now do a few rudimentary checks to make sure that our test setup works as expected:

- ☐ All routers must run `zebra` and `ospf6d` processes.
- ☐ Check with `ifconfig` or `ip` that all routers have the correct interface configuration.
- ☐ Check that router advertisements work correctly: Both hosts must show a routable address on their interface.
- ☐ On all routers check the routing tables with `netstat -r` or `ip -6 route show`: They must show a route for every subnet in the environment.
- ☐ The two hosts must be able to ping each other.

17.3.5 Understanding OSPF Status Information

The `ospf6d` daemon offers a huge variety of status information that we can use to debug routing problems. Taking a look at them also helps us to understand how OSPF works.

First we take a look at an individual router's view of the OSPF cloud. On router 3, which happens to run FreeBSD, we list its immediate neighbors:

```
router3# show ipv6 ospf6 neighbor
Neighbor ID   Pri   DeadTime   State/IfState   Duration   I/F[State]
1.1.1.1         1   00:00:37    Full/BDR       00:04:08   lnc0[DR]
2.2.2.2         1   00:00:32    Full/DR        00:03:34   lnc1[BDR]
```

The first column shows the router ID of the neighbor listed. The third column "DeadTime" shows for how much longer our router assumes its neighbor to be reachable. By default, OSPF routers "ping" each other every ten seconds and assume their peer dead if they don't receive such a "ping" within forty seconds. These timings are freely adjustable, so we'll take a closer look at them in section 17.3.6. The "State/IfState" column in the example states that the connection to the other routers is fully established.

The "BDR" and "DR" strings indicate the role of the neighbor on the particular subnet. A *designated router* (*DR*) for a subnet is "assuming responsibility" for that subnet: It assigns the subnet a "name", consisting of the designated router's router ID and a index number called *link state ID* which

uniquely identifies the attached interface on the designated router. The designated router is also responsible for announcing the existence of the subnet to the OSPF cloud. The *backup designated router* (*BDR*) assumes the role of designated router if the original designated router fails. All routers that are neither designated router nor backup designated router are usually called *DR-other* in status displays. The second column, labeled "Pri" for *priority*, shows the priority of the neighbor to become designated router. It is an 8 bit unsigned integer and defaults to 1. The higher the number, the more eligible is the neighbor to become designated router.

The "Duration" column in the neighbor display shows for how long the neighbor has been "talking" to our router. In OSPF terminology, it shows for how long an *adjacency* has been established, or for how long they have been exchanging OSPF routing information.

Finally, the "I/F[State]" shows from which interface our router reaches the neighbor and what our own role as a designated router is.

Next we ask our router about its view of the OSPF cloud. More precisely, we ask it to show us the shortest paths to all other routers. Every router computes these shortest paths in a *shortest path first tree* (*SPF tree*).

```
router3# show ipv6 ospf6 spf tree
+-3.3.3.3 [0]
  +-2.2.2.2 Net-ID: 0.0.0.2 [1]
  | +-2.2.2.2 [1]
  +-3.3.3.3 Net-ID: 0.0.0.1 [1]
    +-1.1.1.1 [1]
      +-1.1.1.1 Net-ID: 0.0.0.3 [2]
```

The first line of output shows our router as the root of the SPF tree. It has the router ID `3.3.3.3` and is directly reachable from itself. The second line shows that we are connected to a subnet that has router `2.2.2.2` as its designated router and `0.0.0.2` as its link state ID—this is the right main backbone subnet. Through this subnet we can reach router 2, identified by its router ID of `2.2.2.2`, as the third line shows. The fourth line shows we are also connected to another subnet that we are the designated router of and which we assigned the link state ID `0.0.0.1`; this is the spare backbone. Through the spare backbone we can reach router `1.1.1.1` and through it we can reach another subnet—the left leaf subnet with the `2001:db8:fedc:1::/64` prefix. The numbers in brackets are the distance between our router and the element of the SPF tree measured in the OSPF cost metric that we'll investigate in section 17.3.8.

What's surprising is that the subnet lines don't show the associated address prefixes, too. This would be quite helpful, but OSPF simply doesn't work that way: Our router knows that router 1 can deliver packets to the prefix `2001:db8:fedc:1::/64`, but it doesn't know through which interface router 1 can do so. This distinction makes OSPF immune to problems caused by

inconsistent router prefix configurations on the routers; the problems we have analyzed for RIPng in section 7.7.6 are simply non-existent with OSPF.

Additionally, there is no entry for the leaf subnet below our router 3. Directly connected subnets simply don't show up.

So how does our router generate the routing table from the SPF tree? First it needs to know about all prefixes available in the area. In a moment we'll see how our router finds out about them. For now we just assume that somehow it knows of all other routers and which prefixes they have a direct connection to. From this our router generates a list of all prefixes and the routers attached to each. For every prefix it can then look up the attached routers in the SPF tree and choose the closest one. Also from the SPF tree it deduces the shortest path to that router. Knowing that, it adds a routing table entry to the prefix with the next router being the first hop on the path to the router connecting to the prefix. We can take a look at the result:

```
router3# show ipv6 ospf6 route
*N IA 2001:db8:fedc:1::/64     fe80::20c:29ff:fe63:a804 lnc0 00:08:48
*N IA 2001:db8:fedc:2::/64     ::1                         0 02:18:23
*N IA 2001:db8:fedc:fffd::/64  ::                       lnc1 02:18:18
*N IA 2001:db8:fedc:fffe::/64  fe80::20c:29ff:fe63:a804 lnc0 00:46:50
                               fe80::20c:29ff:fef1:c04b lnc1
*N IA 2001:db8:fedc:ffff::/64  ::                       lnc0 00:46:50
```

The first column here marks all active routes with `*N`. Next follows the abbreviation `IA`, which stands for *intra-area route*; these routes are within the area that the interface connects to. Then follow the routed prefix and the address of the next-hop router. There is a minor surprise here: The second line shows `::1` as the next-hop router to the directly attached subnet `2001:db8:fedc:2::/64` but the third and last line show `::` as the next-hop router to the two backbone subnets that we attach to. The difference between them is that the bottom subnet is only reachable through our router 3, so it appears differently in the routing table. Following the next-hop router we find the interface that connects us to the next-hop router and how long the route has been active.

Still we want to learn about the prefixes within an area. The command `show ipv6 ospf6 database intra-prefix detail` presents us a list of all known prefixes within an area. Beyond that, a wide range of `show ipv6 ospf6 ...` commands provide us with information about the OSPF cloud in an almost arbitrary level of detail.

17.3.6 Timing Considerations

Next we should take a look at the way OSPF works, why it handles network failures so much faster than RIPng and how we can improve its responsiveness in various cases.

When an OSPF router attaches to a subnet, it sends a *hello packet* to that subnet. This packet will be received by all other OSPF routers on that subnet. They will send the router a number of packets containing various *link state advertisements* (*LSA*s) that contain all the information it needs to know about the subnet, the routers connected and the OSPF cloud in general. There is no way to speed up this event. In fact, it does take some time to complete.

Similarly, when the network changes, the routers that detect the change will send new link state advertisements to their peers. These LSAs only contain information about the change that occurred. The neighbors will forward these LSAs to their peers, eventually flooding them to all routers that need to know about the change. Again, there are no tunable parameters that control the speed of the flooding. But flooding is a mechanism that runs as fast as the LSAs can be flooded through the network, so the only ways to speed it up are faster networks or a network topology that reduces the maximum time that a packet needs to travel from one "end" of the network to the other.

When a router receives an LSA, it has to recompute its routing table. Again it does this at the greatest possible speed, so there are no tunable parameters. If we need to speed the recomputation up, then we need to minimize the routing table. Using few large subnets is obviously a dubious strategy. Aggregating multiple routes is more feasible; section 17.3.9 explains how we can use multiple OSPF areas to do so.

But still, there are two configurable parameters that control the performance of OSPF in a certain situation: OSPF will only send out those LSAs when it detects a change in the network topology—most notably the failure of a link. In some cases the operating system will notify the OSPF daemon as soon as a link becomes unavailable, so the LSAs are sent out immediately. But in other cases, like a router connected to another through an Ethernet subnet, neither the operating system nor the OSPF daemon notice right away if a peer suddenly dies. In this particular case the "ping" packets we've mentioned before become crucially important. They are just *hello packets* again and are by default sent every 10 seconds by every router. If a router hasn't received a hello packet from a peer for a configurable time, by default 40 seconds, then the router assumes its peer to be unreachable. While 40 seconds is far better than the 180 seconds timeout that RIPng uses, it is still long enough to make impatient users call us up. So we may want to sacrifice a bit of bandwidth and send hello packets more often so that we can detect a failing link faster.

The two parameters are called *hello interval* and *router dead interval*. According to RFC 2328 [89, section 10.5] these two parameters are embedded in the hello packets; a router receiving a hello packet with parameters different than its own will discard the packet. So it is mandatory to keep these parameters the same on all router interfaces connected to the same subnet.

Configuring the parameters is quite straightforward: We just set them on a per-interface basis using the commands

```
router3(config)# interface lnc0
router3(config-if)# ipv6 ospf6 hello-interval 1
router3(config-if)# ipv6 ospf6 dead-interval 4
```

to minimize the time that a dead route stays undetected. Using an even shorter dead interval setting increases the risk that dropped or broken hello packets will cause sporadic route updates.

The performance impact of such a configuration is often acceptable. OSPF won't send any LSA updates unless the network has changed, so we only need to worry about the extra network traffic caused by the hello packets. Only if we use low-bandwidth WAN connections should we be careful about these settings. But these are usually point-to-point links that use some sort of carrier signal or similar, so a failure will be detected by the operating system and more frequent hello packets are unnecessary anyway. So throughout the rest of this chapter we'll assume a hello interval of 1 second and a dead interval of 4 seconds on all interfaces.

RIPng enthusiasts might refer to section 17.2.5 and point out that it is possible to manipulate similar parameters with RIPng, too. That is correct at least with Quagga, but doing so violates the RIPng standard and tends to cause problems because inconsistent configurations are not detected by the protocol. In contrast, the OSPF design explicitly supports timing parameter tuning.

17.3.7 Failover Tests

Next we can use our test environment to gather some experience with the way OSPF behaves when a network failure occurs.

There are several ways to observe the `ospf6d` daemon at work. From the outside we can use `ping` and `traceroute` between the hosts to see how the hop count or the route of a packet itself change. We can also use the various debugging features of `ospf6d` to show us what's happening; unfortunately, log files tend to get quite cluttered with arbitrarily detailed information. We can watch the routing table change from the perspective of the operating system. Finally we can use a packet sniffer to watch the OSPF protocol at work—if we are willing to learn about the OSPF protocol internals.

We also have two fundamentally different ways to cause a network failure: We can shut down individual interfaces through the `zebra` daemon and the `shutdown` interface command, or we can just start to unplug individual cables or entire switches or routers. Shutting down an interface cleanly will cause an immediate routing update while the less graceful failures show how the dead router discovery mechanism works.

I personally like to use `ping` during these tests; its output shows how many packets were lost during a failover, the hop count indicates if the route is optimal at least in simple cases, and if the sender receives any network errors

during the transition period I can also see them. If I also run `traceroute` before and after the failover, then I can compare any changes in routing.

If you have a chance to do so, then this is an excellent moment to set up a test environment, break things and watch how OSPF deals with network failures. Take a look at the debugging facilities and make yourself comfortable with them.

17.3.8 The Cost Metric

Our test environment still has a problem: Traffic between router 1 and router 3 is routed through the "spare backbone" instead of the segmented "main backbone". If the "spare backbone" happens to be an unmanaged 100BaseTX switch while the "main backbone" is a managed 1000BaseFX switch with a routing engine that implements our router 2, then we may actually want to route the traffic through router 2 rather than down the direct "spare backbone" link. So we need to adjust the OSPF routing metric to these assumed properties of our network topology.

OSPF supports the notion of a *cost* associated with every router interface. The cost is a dimensionless 16 bit integer. Whenever a path is computed from one router to another, the sum of all costs assigned to the outgoing router interfaces on that path is minimized.

By convention, the `ospf6d` uses 100 Mbit/s divided by the bandwidth of the link as the cost. Ten years ago, when 100 Mbit/s was about as fast as any commercially available link was, that convention was quite useful: A 10 Mbit/s link had a cost of 10 while a 100 Mbit/s link had a cost of 1. Today this convention is useless: A 1 Gbit/s link would have a cost of 0.1, which we can't reasonably represent as an integer.

In many cases it is appropriate to pick the maximum bandwidth of all links in our network cloud and define it as a cost of 1. Dividing the maximum bandwidth by the bandwidth of any other link will yield that link's cost.

Consider our example network. If our two main backbone segments were Gigabit Ethernets while our spare backbone and the leaf subnets only ran at 100 Mbit/s, then we would assign the main backbone subnets a cost of 1 and the other subnets a cost of 10.

On our preferred router 3 we configure the cost of every interface. If `lnc0` is the interface to the spare backbone, `lnc1` connects to the main backbone segment and `lnc2` attaches to the bottom leaf subnet, then we configure them like this:

```
router3(config)# interface lnc0
router3(config-if)# ipv6 ospf6 cost 10
router3(config-if)# exit
```

[Continued on next page]

```
                    [Continued from previous page]
router3(config)# interface lnc1
router3(config-if)# ipv6 ospf6 cost 1
router3(config-if)# exit
router3(config)# interface lnc2
router3(config-if)# ipv6 ospf6 cost 10
router3(config-if)# exit
```

Similarly we configure router 1. On router 2 we should check the cost settings, but they should already be set to 1. Now we can test if the routing avoids the slow spare backbone:

- ☐ A `traceroute` from one host to the other should show four hops, the three routers and the receiving host.
- ☐ A look at the routing tables of all routers should show that they route all traffic through the main backbone segments.

The OSPF cost metric is well suited to point-to-point links with different bandwidths in either direction, like ADSL—we just need to assign different costs to the connecting interfaces.

Another feature that OSPF supports is *equal-cost multipath routing*: If two routes at the same cost exist, then OSPF will distribute traffic between the two. Quagga supports this feature if the underlying operating system can handle multiple routes to the same destination.

The OSPF metric does however have a limitation: It is inherently latency-oriented. The costs on a route accumulate and OSPF minimizes the total cost along a path. That is perfectly reasonable if we want to optimize latency, because latency also accumulates. But if we are more interested in high bandwidth, then the cost metric is inadequate: For maximum bandwidth we choose a path that consists of high bandwidth links only, no matter how many hops that path has. If we need bandwidth rather than latency, for example for a network connecting the servers in a data center to the backup tape library, then the OSPF cost metric is a problem rather than a solution.

17.3.9 Scalability, OSPF Areas and Route Aggregation

Another major problem with OSPF is the fact that it doesn't scale. When a network grows, the SPF tree on every router grows. The database storing all the LSAs on each router grows, too. The number of LSA updates will grow because there are more routers, links and prefixes that need to be announced. Eventually, the OSPF daemon will require noticeable CPU and memory resources to do its job. Finally, the routing table also grows, increasing the delay of every single packet even if the network doesn't change at all.

In section 17.2.4 we coerced RIPng to use aggregated routes in a somewhat haphazard way to work around this problem. OSPF offers a different strategy: It lets us divide our entire OSPF cloud into multiple *OSPF areas*.

Areas are contiguous sub-clouds within an OSPF cloud. Every area is assigned an *area ID*, again a 32 bit integer "written like an IPv4 address". The special area ID `0.0.0.0` that we have used in our configurations so far is reserved for the *backbone area*. All non-backbone areas attach directly to the backbone area. Every subnet and every router interface belong to exactly one area. Routers with interfaces in different areas are called *area border routers* (*ABRs*).

According to the documentation of Quagga 0.98.3, support for OSPF areas is still missing in the `ospf6d` *daemon. While the following experiments show that this isn't quite true, support for areas is still definitely incomplete. Don't use it in production environments.*

Now assume that we want to split our test environment into multiple areas. We could turn routers 1 and 3 into area border routers and replace the bottom leaf subnets with something more complex, say two subnets and a router each. The result would be something like figure 17.2.

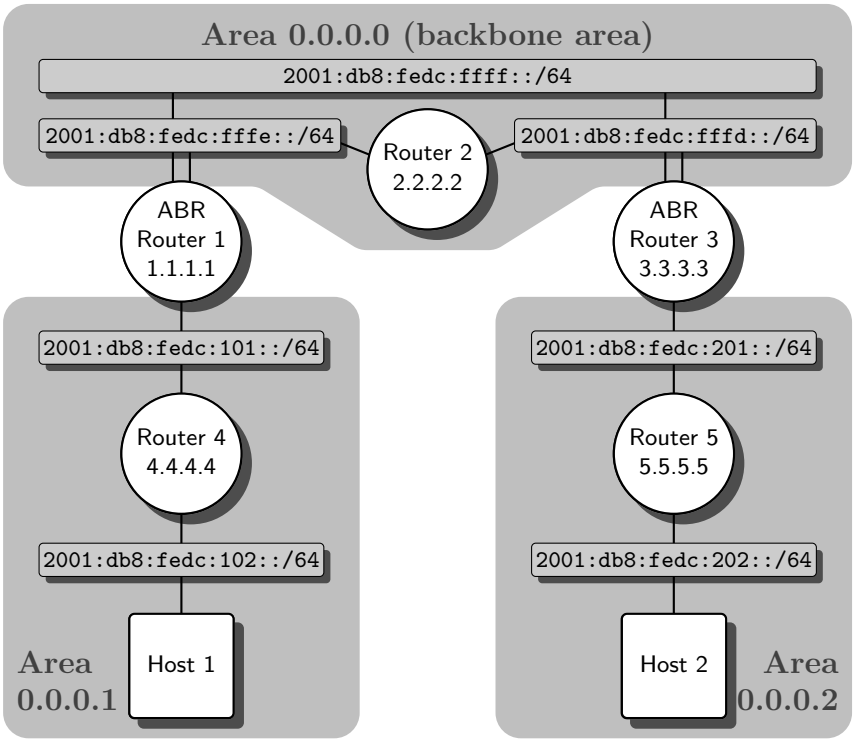

Fig. 17.2. A multi-area OSPF cloud

So what do we have to do? First we add the extra routers and subnets according to figure 17.2. We also need to change the prefixes within the subnets in areas `0.0.0.1` and `0.0.0.2`—otherwise we can't aggregate the prefixes in the areas to a single shorter prefix. At this point we should check that our configuration—without multiple areas—works as expected by pinging one host from the other.

Now we configure all router interfaces within our non-backbone areas with the same area ID. On router 1, which happens to run Debian Sarge and connects to the `2001:db8:fedc:101::/64` network with interface `eth2`, we change the area associated with the interface using the commands

```
router1(config)# router ospf6
router1(config-ospf6)# no interface eth2 area 0.0.0.0
router1(config-ospf6)# interface eth2 area 0.0.0.1
router1(config-ospf6)# exit
```

Similarly we need to change the settings for all interfaces on router 4. Now the non-backbone area `0.0.0.1` should be up and running. First we test that our network is still functional:

- ☐ The hosts must still be able to ping each other.

Now we take a look at router 2. We start with a look at its SPF tree:

```
router2\# show ipv6 ospf6 spf tree
+-2.2.2.2 [0]
   +-2.2.2.2 Net-ID: 0.0.0.1 [1]
   |  +-1.1.1.1 [1]
   +-2.2.2.2 Net-ID: 0.0.0.2 [1]
      +-3.3.3.3 [1]
         +-3.3.3.3 Net-ID: 0.0.0.1 [11]
         +-3.3.3.3 Net-ID: 0.0.0.3 [11]
            +-5.5.5.5 [11]
```

Unsurprisingly, there is no entry for router 4: It belongs to another area, so it doesn't show up in the SPF tree. Then how does router 2 get a route to the subnets in area `0.0.0.1`? We take a look at the routing table.

```
router2# show ipv6 ospf6 route
*N IE 2001:db8:fedc:101::/64  fe80::20c:29ff:fe41:d647  lnc0 00:49:37
*N IE 2001:db8:fedc:102::/64  fe80::20c:29ff:fe41:d647  lnc0 00:49:37
*N IA 2001:db8:fedc:201::/64  fe80::20c:29ff:fe83:a8e5  lnc1 01:07:26
[...]
```

So what has changed? There are still two separate routes for the prefixes, so we obviously haven't achieved any route aggregation yet. But the second column looks slightly different than before: Instead of `IA` for an intra-area route we see the routes marked as `IE`, for *inter-area route*. So far, everything seems to work fine. Now we should set up the second non-backbone area, `0.0.0.2`, accordingly.

Setting up multiple areas has reduced the size of the SPF tree in the routers. But still, the size of the routing table hasn't changed because there was no route aggregation. Unfortunately, route aggregation support is still missing in the `ospf6d` daemon. The configuration syntax seems to be there, because analogously to OSPFv2 we can issue the command

```
router3(config)# router ospf6
router3(config-ospf6)# area 0.0.0.2 range 2001:db8:fedc:200::/56
```

which should make our area border router 3 announce an aggregated route to the `2001:db8:fedc:2::/56` prefix. But the necessary functionality is still missing; the command doesn't show any effect.

17.3.10 Other OSPF Features and Further Reading

OSPF has a number of additional features to offer. Most of them relate to areas, which Quagga doesn't fully support yet: *stub areas*, *not so stubby areas* (*NSSAs*), *virtual links* and different *metric types* for aggregated routes. Another key feature that is currently missing from Quagga is support for multiple OSPF instances on a single router; this feature is necessary to set up gateways between separate OSPF clouds without running BGP between them.

For more information on the internals of the OSPF protocol, Silvia Hagen's books [53, 54] offer quite useful introductions. Beyond that, the official standards are RFC 2328 [89] for OSPFv2 and RFC 2740 [16] for the IPv6-related changes from OSPFv2 to OSPFv3.

Finally, the Texinfo documentation for the Quagga framework explains some more configuration issues; it is available with the source code.

17.3.11 Operational Issues

As we have seen, setting up an OSPF cloud is reasonably straightforward as long as we know what we are doing and the network topology isn't too distorted. But still, there are some pitfalls to avoid.

OSPF is a resource hog, and it doesn't scale. So we should keep a close eye on the CPU and memory usage on our OSPF routers, preferably through a centralized monitoring facility. If the load gets considerable, then it is time to either replace the router hardware or split off some additional areas (as soon as Quagga supports them).

OSPF only transmits changes in network topology. If a change gets lost, then OSPF may not converge towards a reasonable configuration. RIPng is more forgiving in this respect.

There is one misconfiguration that OSPF really can't handle: multiple routers with the same router ID. My personal recommendation is to assign a new router ID to every router right after purchase and never change that router ID.

Within an area OSPF can't aggregate prefixes. So assigning network prefixes doesn't need any particular forethought. But it helps to assign short prefixes to all areas, so aggregation between areas works well. As a general rule of thumb I suggest splitting the subnet ID into two 8-bit parts and assigning the first 8 bits to the area.

When an area gets too large, first choose the prefix for the new area, then do a renumbering of the subnets that are to move, and then move the prefixes to that new area. Remember, renumbering isn't too much of an issue with IPv6.

Finally, like RIPng OSPFv3 doesn't have any authentication because it is supposed to rely on IPsec. But since it also uses multicasts, which IPsec implementations don't support, OSPFv3 should only be run in subnets that we consider trustworthy; the two-tiered network topology in section 7.7.7 is as useful for OSPF as it is for RIP.

17.4 Beyond RIP and OSPF

While RIPng and OSPFv3 are the most important IPv6 routing protocols with respect to Unix, there are some other protocols that also deserve to be mentioned. As a border case, Quagga also supports BGP-4 and its IPv6 extensions. Additionally, Quagga has a number of interesting features unrelated to IPv6. This section provides a quick overview.

17.4.1 The Border Gateway Protocol (BGP)

The most important IPv6 routing protocol besides RIPng and OSPFv3 is the *border gateway protocol, version 4* (*BGP-4*, or *BGP* for short). It is the exterior gateway protocol commonly used between autonomous systems.

BGP was originally devised as an exterior routing protocol for IPv4 in RFC 1771 [96], but its *multiprotocol extensions* as of RFC 2858 [7] support a wide range of other protocols. RFC 2545 [86] defines in detail how to make BGP carry IPv6 routing information.

BGP only runs between peers, called *BGP speakers*, across TCP connections. The network clouds that it connects, the autonomous systems, are identified by a uniquely assigned 16 bit integer *autonomous system number* (*ASN*). Within each autonomous system an interior routing protocol ensures that all routers within the autonomous system can reach each other. BGP treats an autonomous system much like RIP or OSPF treat a single router: It just detects the connections between autonomous systems and computes an "exterior routing table" called a *routing information base* (*RIB*) between these autonomous systems. From this routing information base it sets up the actual routes between BGP routers in different autonomous systems and passes routing data to the interior gateway protocol to disseminate within its autonomous system.

The BGP protocol is fairly simple, at least compared to OSPF. It uses a distance vector protocol like RIP, so it scales better than OSPF. It doesn't periodically send the entire routing information base across a connection but instead it transmits only changes. BGP detects and avoids routing loops; this allows for fast propagation of routing changes.

BGP supports a number of different *policies* that offer fine-grained control of the routing decisions it makes. These policies, together with the sheer size of the routing tables in the *default free zone*, the part of the Internet that doesn't have a default route, and the possible implications of a misconfiguration, make BGP administration so critical.

Even though Quagga implements BGP and its IPv6 extensions, using it for BGP is fairly uncommon. In general, BGP is best left to dedicated router equipment.

17.4.2 Other Routing Protocols

Two other interior routing protocols that support IPv6 are IS-IS and EIGRP. They are of little relevance in a Unix context, but still deserve a short introduction.

The *intermediate system to intermediate system intra-domain routing exchange protocol* (*IS-IS*) is an OSI routing protocol that provides interior routing services for both the TCP/IP and ISO/OSI stack. Quagga has a very early but yet undocumented implementation. Work is still in progress to add IPv6 support to the protocol.

The *enhanced interior gateway protocol* (*EIGRP*) is an interior routing protocol developed by Cisco Systems. They recently released IPv6 support with EIGRP. Unix implementations are not available.

17.4.3 IPv6-independent Quagga Features

Throughout this chapter we have focused on the IPv6-related features of the Quagga framework. We have ignored some general features that are not directly related to IPv6, but are often quite useful.

When we set the password for the VTY interface, we used the most straightforward approach. Quagga supports a number of more advanced features, from assigning a separate password for the privileged mode to storing passwords in an encrypted manner.

The `vtysh` tool provides access to all daemons through a single interface. It can be configured in such a way that it grants users access to the system based on their user ID, making the password verification obsolete. Additionally, `vtysh` is useful in shell scripts that dynamically update a router's configuration.

If a routing daemon dies unexpectedly, then a wide range of problems can occur. If the situation is really bad, then we can't reach the router remotely

anymore and need to get physical access to fix the problem. To deal with this situation, the `watchquagga` tool serves as a watchdog process that restarts Quagga daemons that have died.

17.5 Packet Filter Considerations

Introducing routing protocols besides RIPng doesn't change the basic packet filtering issues from section 7.10. But some details are worth a closer look.

The OSPF multicast group IDs are `ff02::5` for all OSPF routers and `ff02::6` for the designated router and backup designated router. OSPF doesn't use UDP, or TCP, as its transport layer protocol. Instead, OSPF is its own transport layer protocol, using protocol number 89. We can use the same packet filter configuration syntax as for encapsulating tunnels to filter by protocol number or name, so section 12.6.3 applies to OSPF filtering as well.

BGP filtering is simpler: It uses TCP on port 179, so filtering BGP connections is quite straightforward. Keep in mind that BGP expects all BGP routers within an autonomous system to be fully meshed; this affects the packet filter configuration as much as the `bgpd` configuration.

18

Multicasts Beyond the Link-local Scope

Even though we have used multicast packets for a wide number of purposes by now, we have mostly ignored them. The occasional "multicast listener" packets we may have seen in our packet sniffer were annoying rather than interesting since, at least within a single subnet, multicasts just work without any problems. But still, the way multicasts work is quite interesting and, as soon as we start to route multicast packets, surprisingly complex. So in this chapter we take a closer look at them.

We start with a few tools that help us to deal with multicast problems and then look inside the "multicast listener" packets. Once we have the tools and knowledge we need, we tackle the two major approaches to multicast routing and learn about some problems that make multicast routing quite challenging. This leads us to a multicast extension that makes multicast routing scale even to Internet dimensions. Finally, we analyze the operational issues that may arise with the use of large-scale multicast routing and take a look at the necessary packet filter configurations.

18.1 A Closer Look at Multicasts

In this section we gather a toolkit to analyze, understand and debug multicasts. We start with a few new terms used with multicasts, then investigate some helpful diagnostic tools and procedures and finally take a closer look at the way multicast listeners communicate with multicast routers.

18.1.1 Terminology

In section 3.5 we have already seen that a *multicast address* consists of a `ff00::/8` prefix followed by a *flag nibble*, a *scope nibble* and finally a *multicast group ID* of 112 bits. The multicast address identifies a *multicast group*; both terms are often used synonymously, though the term multicast group is also used for all the *multicast listeners*, the nodes that have sent a multicast listener

report to signal that they want to receive packets sent to the multicast address. Listeners are also called *receivers* or *multicast group members*. A new listener is said to *join* or *subscribe to* a multicast group and a listener that stops listening *leaves* the multicast group.

Nodes that send packets to a multicast group are called *senders*; they are identified by the source address of the packets they send to the multicast group, so the term is slightly imprecise with respect to nodes that have configured multiple addresses to a single interface.

Multicast routers receive multicast groups that they don't necessarily listen to themselves and forward them towards actual listeners. The interface that they receive a multicast packet on is called the *upstream* or *incoming interface*, all interfaces that they send the packet to are called *downstream* or *outgoing interfaces*. The incoming interface depends on the source of the packet.

Multicast routers maintain *multicast routing tables* that keep track of the outgoing interface for all multicast packets. In the most simple case they store a list of interfaces for every multicast group while in more complex situations they store a list of interfaces for pairs of senders and multicast groups. These *source-specific multicast routes* are often written as (S, G), where S is the sender address and G the multicast address. If the table entry doesn't contain a specific sender address, this is called a *wildcard route*, written as $(*, G)$.

18.1.2 Multicast Diagnostics

When we try to solve multicast-related network problems, then usually we first take a closer look at the receiving node. Generally, the `netstat` command supports a `-g` option that displays all the multicast groups that a node receives. Its output format looks differently on the various Unixen and in some cases it is possible to get more information using less portable commands.

Debian Sarge *Both* `netstat` *and* `ip` *show us the currently subscribed multicast groups:*

```
debian-host:/tmp# netstat -g -n
IPv6/IPv4 Group Memberships
Interface       RefCnt Group
--------------- ------ ---------------------
lo              1      224.0.0.1
lo              1      ff02::1
eth0            1      ff02::1:ff00:0
eth0            2      ff02::1:ff65:23de
eth0            1      ff02::1
```

The `RefCnt` *column shows the number of sockets that listen to the multicast group.*

```
debian-host:/tmp# ip -6 maddr show
1:      lo
        inet6 ff02::1
2:      eth0
        inet6 ff02::1:ff00:0
        inet6 ff02::1:ff65:23de users 2
        inet6 ff02::1
```

Here only those addresses that have multiple sockets listening to them display the number of sockets as `users`*, as the second to last line shows.*

FreeBSD 6.1 *The* `netstat -g` *output here includes not only the group memberships, but also the multicast routing table, which should be empty right now.*

```
freebsd-host# netstat -g
[...]
IPv4/IPv6 Multicast Group Memberships
Group                Gateway           Netif
ff02:1::2:e1c4:d5c8  33:33:e1:c4:d5:c8  lnc0
ff02:1::1            33:33:00:00:00:01  lnc0
ff02:1::1:ff5b:915b  33:33:ff:5b:91:5b  lnc0
224.0.0.1            <none>            lo0
ff01::1              <none>            lo0
ff02:3::2:e1c4:d5c8  <none>            lo0
ff02:3::1            <none>            lo0
ff02:3::1:ff00:1     <none>            lo0
```

The display here embeds the scope ID, which identifies the interface, in the second block of the multicast address, so `ff02:1::1` *really means* `ff02::1%1`*.*

Alternatively, the FreeBSD-specific command `ifmcstat` *displays similar information in a different format.*

Solaris 10 *We can use the* `-f inet6` *option of* `netstat` *to list only IPv6 multicast groups:*

```
# netstat -f inet6 -g
Group Memberships: IPv6
If       Group                     RefCnt
------ -------------------------- ------
lo0    ff02::1:ff00:1                  1
lo0    ff02::1                         1
pcn0   ff02::202                       1
pcn0   ff02::1:ffa4:2ce6               2
pcn0   ff02::1                         3
```

112

So checking which multicast groups a node receives poses no problem as long as we have access to the node itself.

Especially when we test multicast routing we need a tool to make a node join a given multicast address. There is no standard tool available for this; however, from my home page (`http://www.benedikt-stockebrand.net/`) there is a simple tool available to do exactly that. It is called `mcjoin` and takes a single multicast address as a parameter. When it starts, it joins the multicast group so that we can ping the node at that multicast address.

Testing from another node is just as easy. The `ping` command usually serves well if we want to reach all nodes that receive a multicast group. Unfortunately, `traceroute` usually can't handle multicast addresses. This makes it sometimes difficult to understand what exactly happens. Additionally, the hop limit that the ping program uses with multicast destinations may be inappropriate for our purposes:

Debian Sarge *By default, `ping6` sets the hop limit on packets to multicast addresses to 1. We need to use the `-t` option to set it to a larger value.*

FreeBSD 6.1, Solaris 10 *The ping programs here use the standard hop limit even on packets to multicast addresses.* 113

Another tool that is indispensable for network debugging is our favourite packet sniffer. When we use it on any node that we expect to handle multicast packets, rather than sniffing the network itself, then we must ensure that the sniffer does *not* switch the network interface into promiscuous mode; otherwise we may change the behaviour of that node quite significantly when we start the sniffer.

ethereal *The capture dialog box has a check box that disables promiscuous mode support in `ethereal`.*

snoop (Solaris 10) *The `-P` option disables the use of promiscuous mode.*

tcpdump *The option `-p` prevents `tcpdump` from switching an interface into promiscuous mode.* 114

If we run the packet sniffer on a separate machine attached to a subnet we want to monitor, then we may need promiscuous mode to see all the packets there.

18.1.3 Inside IPv6: Multicast Listener Discovery (MLD)

What happens when we start `mcjoin`? Using a packet sniffer we can observe part of the *multicast listener discovery* (*MLD*) protocol that tells routers and switches about the multicast groups that nodes on a subnet want to receive.

There are two versions of multicast listener discovery: version 1 (*MLDv1*) as of RFC 2710 [23] and version 2 (*MLDv2*) as of RFC 3810 [36]. Both differ in the ICMPv6 type numbers they use, so depending on the Unix we use we can observe either MLDv1 or MLDv2 packets with our packet sniffer.

Debian Sarge *The standard kernel uses MLDv2 packets.*

FreeBSD 6.1, Solaris 10 *These use MLDv1 packets.* 115

For our tests we use the site-local transient multicast address `ff15::ffff`; even if we were connected to an upstream provider, using site-local multicasts shouldn't cause any trouble, and using a transient address will avoid any interference with established multicast services.

When we start `mcjoin ff15::ffff` on a FreeBSD machine, the packet sniffer shows a summary buffer like this:

```
Time  Source                  Destination Proto. Info
0.00  fe80::20c:29ff:fed8:4d3f ff15::ffff  ICMPv6 Multicast listener
                                                 report
7.20  fe80::20c:29ff:fed8:4d3f ff15::ffff  ICMPv6 Multicast listener
                                                 report
```

So as soon as we join a multicast group the kernel sends these *multicast listener report* packets out. They tell all routers and multicast-aware ("intelligent") switches that the node wants to receive multicast packets sent to `ff15::ffff`. Now we take a look at the first packet's details. The packet starts with an Ethernet header which is irrelevant except for the destination address:

```
Ethernet II, Src: 00:0c:29:d8:4d:3f, Dst: 33:33:00:00:ff:ff
[...]
```

The destination address starts with `33:33`, which is the address prefix that Ethernet uses for link-layer multicasts. The remaining four bytes are the same as the last four bytes in the IPv6 multicast address. Generally, IPv6 multicast addresses are mapped to Ethernet multicast addresses in this way. Next follows the IPv6 header. The interesting fields are these:

```
Internet Protocol Version 6
    [...]
    Next header: IPv6 hop-by-hop option (0x00)
    Hop limit: 1
    Source address: fe80::20c:29ff:fed8:4d3f
    Destination address: ff15::ffff
```

So following the base header there is a *hop-by-hop option header* which we'll take a look at below. The hop limit is set to 1, so the packet won't be routed beyond the subnet. The source address of all multicast listener reports is a link-local unicast address. Finally, the packet is sent to the multicast address that we want to join.

The hop-by-hop option header is vital for multicast listener reports: Routers as well as intelligent switches watch for this option header. When they receive a packet with this header, they know they need to inspect it even if the destination address isn't theirs.

```
Hop-by-hop Option Header
    [...]
    Router alert: MLD (4 bytes)
```

As a hop-by-hop header it signals all switches and routers that it contains information relevant to them. The subtype, *router alert*, contains the actual information—the packet is a multicast listener discovery packet. Finally follows the ICMPv6 packet that contains the relevant information.

```
Internet Control Message Protocol v6
    Type: 131 (Multicast listener report)
    Code: 0
    [...]
    Maximum response delay: 0
    Multicast Address: ff15::ffff
```

The ICMPv6 type and code identify the packet as a multicast listener report, version 1. The *maximum response delay* is irrelevant for a listener; we'll see later on how multicast routers use this field. Finally follows the multicast address that we subscribe to.

A few seconds after the node sent its first listener report, it sends a second report in case the first packet got lost. There is no universal standard how often and with what delay these additional packets are sent; RFC 2710 [23, section 7] explains the details and the rationale if you want to tune these parameters.

When we terminate the `mcjoin` command, then the kernel sends out another packet. This time it is a *multicast listener done* packet that notifies the switches and routers that we are no longer interested in the multicast group.

```
Time Source                   Destination Proto. Info
11.1 fe80::20c:29ff:fed8:4d3f ff02::2     ICMPv6 Multicast listener
                                                 done
```

Again the packet details are worth a closer look. The Ethernet header doesn't show anything interesting, so we start with the IPv6 header:

```
Internet Protocol Version 6
    [...]
    Next header: IPv6 hop-by-hop option (0x00)
    Hop limit: 1
    Source address: fe80::20c:29ff:fed8:4d3f
    Destination address: ff02::2
```

Again we see that there is a hop-by-hop option header following. Again the hop limit is set to 1 and the source address is a link-local unicast address. But the destination address is different: The multicast listener done packet isn't sent to the multicast address but to the *all routers link-local multicast group* `ff02::2`. The hop-by-hop option header again tells switches and routers that this is a multicast listener discovery packet.

```
Hop-by-hop Option Header
    [...]
    Router alert: MLD (4 bytes)
```

Finally the ICMPv6 packet tells all routers that we are leaving a multicast group.

```
Internet Control Message Protocol v6
    Type: 132 (Multicast listener done)
    Code: 0
    [...]
    Maximum response delay: 0
    Multicast Address: ff15::ffff
```

The ICMPv6 type differs from that of a multicast listener report. But again there is a maximum response delay that is irrelevant for a multicast listener done packet and the multicast address that we are now leaving.

If we do the same with a Debian node, then we see the equivalent MLDv2 behaviour. When we start the `mcjoin` command, then we first notice that there is only a single packet sent.

```
Time Source                    Destination Proto. Info
0.00 fe80::20c:29ff:fe59:60a9 ff02::16    ICMPv6 Multicast Listener
                                                 Report Message v2
```

The destination address is not the multicast address to listen to, but the *all MLDv2-capable routers multicast group* `ff02::16`, which all MLDv2 multicast routers listen to.

```
Ethernet II, Src: 00:0c:29:59:60:a9, Dst: 33:33:00:00:00:16
  Destination: 33:33:00:00:00:16
  [...]
```

Accordingly, the Ethernet multicast address is now `33:33:00:00:00:16`; using this address instead of the joined multicast address ensures that the information reaches the receivers that need to know: the switches and routers in the subnet. The IPv6 base header doesn't show any relevant changes compared to MLDv1:

```
Internet Protocol Version 6
    Next header: IPv6 hop-by-hop option (0x00)
    Hop limit: 1
    Source address: fe80::20c:29ff:fe59:60a9
    Destination address: ff02::16
```

The only difference is the new destination address explained above. Similarly, the hop-by-hop option header again contains a router alert.

```
Hop-by-hop Option Header
    [...]
    Router alert: MLD (4 bytes)
```

The final ICMPv6 part of the packet looks quite different than its MLDv1 cousin:

```
Internet Control Message Protocol v6
    Type: 143 (Multicast Listener Report Message v2)
    Code: 0 (Should always be zero)
    [...]
    Changed to exclude: ff15::ffff
        Mode: Changed to exclude
        Aux data len: 0
        Multicast Address: ff15::ffff
```

The first thing we notice is the fact that the ICMPv6 type has changed to 143. Beyond that, the only resemblance to an MLDv1 multicast listener report is the field with the multicast address that we want to join. The "aux data len" indicates that there is no "auxiliary data" following; while MLDv2 doesn't use any "auxiliary data", later versions might do, so MLDv2 already provides for them.

The most important difference between MLDv1 and MLDv2 is the added *source filtering* feature: A listener may explicitly tell not only which multicast address it is interested in, but from which source addresses it is willing to receive packets to that multicast address. This listener report states that we are willing to receive packets to `ff15::ffff` except from source addresses that are in the attached list—which happens to be empty.

When we kill the `mcjoin` process, then the kernel sends another multicast listener report packet.

```
Internet Control Message Protocol v6
    Type: 143 (Multicast Listener Report Message v2)
    Code: 0 (Should always be zero)
    Checksum: 0xe5e9 (correct)
    Changed to include: ff15::ffff
        Mode: Changed to include
        Aux data len: 0
        Multicast Address: ff15::ffff
```

It is the equivalent of the MLDv1 multicast listener done packet, but MLDv2 doesn't use different packet types for listener report and listener done messages anymore. Instead, the only difference to the initial packet is that it tells the switches and routers that the node is only willing to receive packets from source addresses included in the attached list—which happens to be empty again.

Why did we use `mcjoin` to join a multicast group and didn't use those multicast groups we already know, like the solicited-node multicast groups or the all-nodes link-local multicast group? Because these are special cases: Since all nodes must join the all-nodes link-local multicast group, it doesn't make sense to do multicast listener discovery here. And when a node brings up a new address on an interface, it needs to subscribe to the solicited-node

multicast address for its duplicate address detection; but since it isn't allowed to use the intended address yet, it has to use `::`, the unspecified address; RFC 3590 [50] has the details.

At this point we need a multicast router to see how multicast listener discovery between routers and hosts works. So we defer further investigations to section 18.2.3 and install a multicast router next.

18.2 Protocol Independent Multicast—Dense Mode (PIM-DM)

There is a major difference between unicast and multicast routing: With multicasts there are no static routes. All multicast routing is done dynamically, using multicast listener discovery to find all nodes that actually want to receive multicast packets and some inter-router protocol to ensure that multicast routers are aware of each other.

In this section we start with the simple scenario that we have a well-defined network cloud and want to route multicast traffic within this cloud only. The routing protocol we use here is called *protocol independent multicast—dense mode* (*PIM-DM*). It uses an optimized flooding algorithm which works automatically without any substantial configuration.

18.2.1 Installation

Figure 18.1 shows the simple test environment we use. We have three subnets, each with a host attached to it, and two multicast routers between the subnets. The routers are set up as advertising routers and we use RIPng for dynamic routing. Next we need to install the routing daemon software.

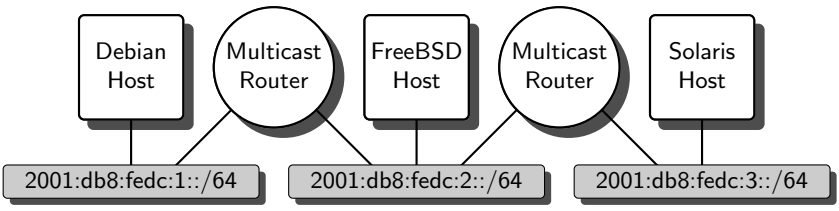

Fig. 18.1. A multicast routing test setup

Debian Sarge *There is a recent port of the BSD/KAME implementation of PIM-DM available at* `http://sourceforge.net/projects/mcast-tools/`. *Unfortunately, it doesn't compile on Debian Sarge, apparently due to some header file problems.*

For now this leaves us with three options: Skip to section 18.3 for a different multicast routing protocol, wait until the `mcast-tools` *have been made available as a binary package and then treat them like the FreeBSD package, or use* `ecmh`*, an "MLD proxy" available at* `http://unfix.org/projects/ecmh/`*. While* `ecmh` *will cause a network meltdown in redundant network topologies, it may be useful as a workaround in some cases. The Debian package available at the web site, currently version 2005.02.09, installs out of the box.*

While experimenting with `ecmh` *and PIM-DM it became apparent that interoperation can still be a problem, so using both together is not advisable at this time.*

FreeBSD 6.1 *First we build and install a kernel with the additional kernel options* `MROUTING` *and* `PIM`*. Then we install the* `mcast-tools` *package from the ports collection. Finally we add a line*

/etc/rc.conf

```
mroute6d_enable=YES
```

to `/etc/rc.conf` *and either reboot or start the* `pim6dd` *daemon using the* `mroute6d` *boot script.*

Solaris 10 *There is no support for multicast routing available.* 116

VMware Workstation (at least version 5.5.1) has a problem with the LANCE/PCnet network card emulation: The Ethernet multicast filtering doesn't work and the multicast packets never arrive. If you want to run multicast routing in a VMware environment, then it is easiest to use a 64 bit CPU and run the virtual machines in 64 bit mode, which emulates an Intel Gigabit Ethernet adapter that doesn't exhibit these problems. Putting the interfaces on the multicast routers (not the hosts) into promiscuous mode also works, but in some situations causes undesirable side effects.

We are now ready to run a first quick test to see if everything works as expected.

- ☐ All nodes must show properly configured routable addresses.
- ☐ The routers must have a proper route to the subnet that they don't connect to directly.
- ☐ The hosts and routers must be able to ping each other.
- ☐ If any host starts `mcjoin` with a routable multicast address, then all other hosts must be able to ping this multicast address.

18.2.2 Essential Configurations: Filters

PIM-DM uses a flooding algorithm to distribute multicast packets, so it is vital to keep PIM-DM routers from distributing multicast packets beyond their

scope. On border routers that connect to the edge of any given scope we need to restrict the propagation of multicasts with some additional configuration.

The `pim6dd` configuration only allows for filters that explicitly prevent the routing of certain multicast addresses on certain interfaces. This is unfortunate; it would be nicer to configure explicitly which multicast addresses to route to which interfaces.

As a reasonable starting point for border router configurations I suggest a configuration like

pim6dd.conf

```
filter ff20::-ffff:ffff:ffff:ffff:ffff:ffff:ffff:ffff lnc1 lnc0
filter ff00::-ff05:ffff:ffff:ffff:ffff:ffff:ffff:ffff lnc1
filter ff10::-ff15:ffff:ffff:ffff:ffff:ffff:ffff:ffff lnc1
```

if we have an inner interface `lnc0` and a border interface `lnc1`. This configuration blocks all packets with a flag nibble that is neither 0 nor 1 on all interfaces and all multicast packets with a scope smaller than site-local going out of the border interface `lnc1`.

The filter configuration only filters outbound packets. That may be enough in some cases, but in general it seems preferable to use a packet filter for better control of inbound and outbound multicast packets.

18.2.3 Inside IPv6: More on Multicast Listener Discovery

With a multicast router at hand we can now continue to explore the multicast listener discovery protocol.

First we turn off all nodes and start the packet sniffer. Then we bring up a multicast router. Between the duplicate address detection and multicast listener reports for some essential multicast groups that a router listens to we should observe some *multicast listener queries* like these:

```
Time       Source                    Dest.   Proto. Info
4.260603   fe80::20c:29ff:fe57:5309  ff02::1 ICMPv6 Multicast
                                                    listener query
5.534291   fe80::20c:29ff:fe57:5309  ff02::1 ICMPv6 Multicast
                                                    listener query
129.469979 fe80::20c:29ff:fe57:5309  ff02::1 ICMPv6 Multicast
                                                    listener query
254.649394 fe80::20c:29ff:fe57:5309  ff02::1 ICMPv6 Multicast
                                                    listener query
```

With these the router attempts to find all multicast listeners in the subnet. It first sends two queries in short succession to deal with the situation that the first packet is lost, and afterwards periodically sends new queries to see if listeners have started or stopped.

The query packet has a link-local source address, the all-nodes link-local multicast address `ff02::1` as destination address and a hop limit of 1. It

contains a hop-by-hop option header with a router alert. Finally the packet contains an ICMPv6 payload:

```
Internet Control Message Protocol v6
    Type: 130 (Multicast listener query)
    Code: 0
    Maximum response delay: 10000
    Multicast Address: ::
    [...]
```

Here the router asks all nodes if they are listening to any multicast address; if the "Multicast Address" field contained a multicast address rather than the unspecified address, then the router would only ask the other nodes about that specific multicast address, but at this time the router wants to get an overview about all multicast addresses that have listeners, so in this case the field contains `::`. To avoid a network overload, the router specifies a *maximum response delay* of 10 000 ms to make all hosts wait a random interval between zero and ten seconds before they reply.

The multicast listener query looks the same with both MLDv1 and MLDv2, but there is a difference that tells all the listeners that in this case the router speaks MLDv1: The MLDv2 version of the query contains additional data and is therefore longer.

So what happens if there are any listeners around? They will pick a random delay interval of up to the maximum response delay and then they will send their listener reports. MLDv1-only listeners will listen for other replies and suppress their own if another node signals that it is listening to the same multicast group that they listen to; MLDv2 listeners will send their announcement in any case. Additionally, if an MLDv2 listener detects that the router sent an MLDv1 query, then it will reply with an MLDv1 rather than an MLDv2 report.

Next we start `mcjoin` on a new multicast address. As we have seen before, the node sends its multicast listener report; nothing else happens. When we stop `mcjoin` again, then something new happens: The router will follow up on the listener done packet and do a listener query, but this time it will specifically ask if any other node still listens to the multicast group that `mcjoin` just stopped listening to.

Multicast routers aren't required to keep track of which nodes, or even how many nodes, listen to any given multicast group. So as soon as a listener leaves they check if any other listeners remain. This saves resources on the router and makes the behaviour more robust in the case that a listener is forcefully disconnected before it can send a listener done message.

Another surprising behaviour occurs when we start up a second router: There is only one router sending listener queries; more specifically, the router with the lower interface ID becomes the *querier* and is solely responsible for sending listener queries to the subnet.

As usual we have ignored a host of interesting details of the multicast listener discovery protocol. But as long as we don't try to implement MLD ourselves, this outline suffices to understand how MLD works and what the traffic we observe with a packet sniffer means. Beyond that, RFC 2710 [23] and RFC 3810 [36] are the definitive specifications of MLDv1 and MLDv2, respectively.

18.2.4 Inside IPv6: The PIM-DM Protocol

Now that we have two multicast routers running PIM-DM in our environment, we can also observe PIM-DM at work.

In our test setup in figure 18.1 on page 271 we connect a packet sniffer to the middle subnet. As long as we don't actually route any multicast traffic, all we see are *PIM hello messages*; the routers send these approximately every thirty seconds to the *all PIM routers multicast group* `ff02::d`. Thanks to these, both routers know of each other.

The PIM packets use their own protocol type 103 (0x67) rather than UDP. The hello packets only contain a packet type that identifies them as hello messages and a *hold time* that tells the peer routers for how long they should assume the router online if they don't receive any further hello messages.

Next we run `mcjoin ff15::ffff` on the Solaris host on the right subnet. The right router will receive the multicast listener report, so it knows that it must now forward multicast packets to this address. At this point we won't notice anything in the packet sniffer.

Now we ping the multicast address from the Debian host on the left subnet. Since Debian uses a hop limit of 1 for pings to multicast addresses, we must use the option `-t 3` for this to work. At this point something unexpected happens: Without any further PIM traffic between the routers the ping packets are routed from the Debian host to the Solaris host.

What happens here is that PIM-DM initially *floods* all multicast packets through the entire network cloud. The left router receives the packet on its left interface and sends it out through all other interfaces, in this case its right interface to the middle subnet. There the right router picks it up on its left interface and forwards it again, this time to the subnet with the listening Solaris host. The Solaris host then sends its unicast reply back.

Does PIM-DM flood all packets through the entire network cloud? No, and if we stop the `mcjoin` process running on the Solaris host, then we notice some additional PIM traffic in the middle subnet: Both routers send *PIM join/prune messages*. With these messages routers tell each other that they don't have any listeners for a specific multicast group on their other interfaces. As soon as a router realizes that its peer doesn't want to receive the specific multicast address it will stop to forward them.

When the `mcjoin` process dies, the Solaris host sends an MLD listener done message to the right router. When the Debian host sends another ping, the left

router receives it and forwards it to the middle subnet. There the right router receives it, realizes that it doesn't have any listener for the multicast address on its right interface and sends a join/prune message to tell the left router that it doesn't want to receive packets to this multicast address anymore. The join/prune message looks like this:

```
Internet Protocol Version 6
    [...]
    Hop limit: 1
    Source address: fe80::20c:29ff:fe98:832a
    Destination address: ff02::d
Protocol Independent Multicast
    Version: 2
    Type: Join/Prune (3)
    [...]
    PIM parameters
        Upstream-neighbor: fe80::20c:29ff:fe57:5313
        Groups: 1
        Holdtime: 210
        Group 0: ff15::ffff/128
            Join: 0
            Prune: 1
                IP address: 2001:db8:fedc:2:20c:29ff:fe59:60a9/128
```

The right router sends the message to the PIM multicast address from its link-local address, using a hop limit of 1. The PIM payload has the type "join/prune" and tells the upstream neighbor, the left router in our example, to stop forwarding multicasts to the multicast address `ff15::ffff` if they have the source address shown in the last line. The *hold time* is set to 210 seconds, so this prune request expires in three and a half minutes.

The left router then sends a similar message, effectively confirming that it isn't interested in these multicasts, either. While this doesn't make sense in a two router setup like ours, it is necessary as soon as a third router is attached to the network.

The left router then keeps forwarding the multicast packets for a little longer. It waits for another router to send a "join/prune" message that signals it wants to keep receiving these multicasts. Since no router sends such a join message, three seconds later the left router stops forwarding the ping packets for 210 seconds.

After those 210 seconds the left router starts to forward the ping packets again and the same process repeats.

What happens if we restart the `mcjoin` program on the Solaris host? Then the Solaris host will send another MLD report and the right router will send a *PIM graft message* to the middle subnet. The left router acknowledges this graft message with a *PIM graft acknowledgment message* and starts forwarding the ping packets again.

So far we have assumed that a multicast router will forward an incoming multicast packet on all interfaces except the one that it received the packet on and those that it has received a prune message on. That is true in our test setup, but in a redundant network topology this behaviour would cause the packet to loop through the network until its hop limit expires, possibly causing a network meltdown. To avoid this, PIM-DM uses yet another optimization: When it receives a multicast packet it first does a *reverse path forwarding* (*RPF*) *check*: It verifies that the interface on which the packet arrived and the interface to which unicast packets to the source address are routed back are actually the same. If they are not, then it assumes a routing loop and uses a *PIM assert message* exchange to decide from which interface to accept packets with this source. The router will then only accept packets from the given source address if they arrive on the chosen interface, thereby eliminating routing loops.

As usual, this overview of the PIM-DM protocol doesn't cover all the details. If you need or want to learn more about the protocol, then RFC 3973 [1] is the definitive reference.

18.2.5 Advantages and Limitations

The PIM-DM protocol and its associated algorithms have a few properties that limit it to a certain kind of environment only. The entire design of PIM-DM is based on the assumption that most multicast packets must be routed to most connected subnets and that it is more efficient to track exceptions from this assumption than to keep "listener lists" that contain all listeners to a given multicast group.

This assumption has some positive effects: The PIM-DM protocol is reasonably simple, it doesn't need any configuration beyond filters on border routers and it is quite efficient at distributing packets to a network cloud with densely populated listeners.

But if the assumption is wrong, then PIM-DM won't work. If the listeners are sparsely distributed, then the majority of multicast routers need to keep track of excessively large pruning tables. The initial flooding of multicast packets will become a major burden to the entire network.

In short, PIM-DM doesn't scale too well. It works fine in a reasonably sized local network cloud, but it can't possibly work in the Internet at large. This leads to another multicast routing protocol that assumes a sparsely distributed listener community: PIM-SM, which we investigate next.

18.3 Protocol Independent Multicast—Sparse Mode (PIM-SM)

While the PIM-DM protocol assumes that multicast listeners are densely distributed, the *protocol independent multicast—sparse mode* (*PIM-SM*) protocol caters for the opposite situation and assumes that there are few listeners to any specific multicast group.

PIM-SM scales much better than PIM-DM, but the better scalability comes at a price: the PIM-SM protocol is far more complex than PIM-DM and it takes some essential configuration to make it work.

18.3.1 Installation and Basic Configuration

As with PIM-DM, we first need to build a kernel that can do multicast routing and install some additional multicast routing software.

Debian Sarge with mrd6 *The* `mcast-tools` *package, which isn't available for Debian yet, also contains the same* `pim6sd` *daemon as the FreeBSD variant. The* `mrd6` *package, available from* `http://hng.av.it.pt/mrd6/` *as source code and from* `http://www.backports.org/` *as a Debian binary package, supports PIM-SM and as of version 0.9.5 it runs out of the box; this makes it the first choice for multicast routing on Debian at least at this time.*

FreeBSD 6.1 *We need a custom kernel as we have built it in the previous section for PIM-DM. The* `mcast-tools` *package that we have installed for PIM-DM also contains a PIM-SM routing daemon called* `pim6sd`*. To enable the daemon we need the lines*

`/etc/rc.conf`

```
mroute6d_enable=YES
mroute6d_program=/usr/local/sbin/pim6sd
```

in `/etc/rc.conf` *and a configuration file* `/usr/local/etc/pim6sd.conf` *which doesn't need to contain anything on most routers.*

Solaris 10 *Again, Solaris doesn't offer any multicast routing functionality.* 117

Next we need to set up a special functionality on at least one of the routers in our multicast routing cloud: We need a *candidate rendezvous point* (*Cand-RP*). PIM-SM doesn't flood the entire routing cloud with initial packets, so there has to be another way to join a multicast group; the *rendezvous points* (*RPs*) serve this purpose. When a multicast router receives a join report from a host, then it contacts the rendezvous point managing the multicast group to set up the routing. If the router receives a packet to a multicast group, then it forwards it to the rendezvous point using unicast and a special encapsulation mechanism; the rendezvous point then distributes the decapsulated packet to

the entire multicast group unless an additional optimization that we'll consider later on kicks in.

Configuring a rendezvous point isn't particularly difficult—we just need to tell the router that it is a candidate rendezvous point for a given multicast prefix or set of prefixes:

Debian Sarge with mrd6 *A minimal configuration looks like this:*

/etc/mrd6.conf

```
pim {
        disable bsr-candidate;
        enable  rp-candidate;
}
groups {
        ff00::/8 { pim rp_adv = true; }
}
```

Strictly speaking the `disable bsr-candidate` *statement isn't necessary, but we only look at this in the following section.*

FreeBSD 6.1 *Different than* `mrd6`*, the* `pim6sd` *daemon doesn't expect a configuration that tells it for which multicast prefixes to serve as a rendezvous point. So all we need are the first two lines of*

/usr/local/etc/pim6sd.conf

```
cand_rp;
cand_bootstrap_router;
group_prefix ff00::/8;
```

in `/usr/local/etc/pim6sd.conf`*; the second line is necessary only on at least one router; we'll investigate this in the next section. The third line restricts rendezvous services to the given prefix.*

When we use `pim6sd` *as the rendezvous point, then it is essential to use its highest address in the configuration of all other routers below as the rendezvous point address;* `pim6sd` *will discard all RP-related traffic that arrives at a different address.* 118

On all other routers we need to configure where the rendezvous point for a given multicast prefix is. Assuming that our rendezvous point has an address `2001:db8:fedc:ffff::ffff`, the configuration on these routers looks like this:

Debian Sarge with mrd6 *The configuration looks similar to that above, but we need to change two things: We disable the* `rp-candidate` *option and for the multicast prefix* `ff00::/8` *we don't tell the daemon to advertise itself as a rendezvous point but where to find the rendezvous point instead.*

/etc/mrd6.conf

```
pim {
        disable bsr-candidate;
        disable rp-candidate;
}
groups {
        ff00::/8 { pim { rp 2001:db8:fedc:ffff::ffff; } }
}
```

FreeBSD 6.1 *If we want to use a single rendezvous point for all multicast addresses, then the line*

/usr/local/etc/pim6sd.conf

```
static_rp ff00::/8 2001:db8:fedc:ffff::ffff;
```

in `/usr/local/etc/pim6sd.conf` *is all we need.* 119

18.3.2 Bootstrap Routers

Configuring the rendezvous point manually is not the originally intended way to distribute the rendezvous points within a routing cloud, but due to a problem with `mrd6` we don't have another choice if we are using `mrd6` based routers. If we use `pim6sd`, or the bug in `mrd6` gets fixed, then we can use *bootstrap routers* (*BSRs*) to disseminate the rendezvous point configuration throughout the routing cloud.

The basic concept works like this: In a routing cloud we have a number of special routers called *candidate bootstrap routers* (*CandBSRs*). Using flooding between the multicast routers these candidate bootstrap routers find each other and use a simple election algorithm to choose one of them as the bootstrap router. The address of the bootstrap router is then flooded to all multicast routers in the cloud. When a candidate rendezvous router comes up, it first asks its peers about the bootstrap router. Then it contacts the bootstrap router, offering its service as a rendezvous router. The bootstrap router periodically floods a list of active rendezvous routers to all multicast routers. They can then choose the correct rendezvous router when they start to provide routing functionality for a new multicast group.

Unfortunately `mrd6` as of version 0.9.5 doesn't accept these lists of active rendezvous routers from the bootstrap router, so we had to configure them statically. If we still want to use bootstrap routers at least as far as possible, then we need to adapt our configuration accordingly.

Setting up a candidate bootstrap router is simple, even with `mrd6`:

Debian Sarge with mrd6 *We just change the* `bsr-candidate` *statement in the* `pim` *section to* `enable bsr-candidate`.

FreeBSD 6.1 *The* `pim6sd` *daemon won't start as a rendezvous point unless it can register with a bootstrap router; for that reason we needed the* `cand_bootstrap_router` *statement in the configuration above. So there is no reason to change any configuration.* (120)

On all other routers we can now remove the statically configured rendezvous points:

Debian Sarge with mrd6 *As explained before, this doesn't work (yet).*

FreeBSD 6.1 *We simply remove the* `static_rp` *statements from the configuration. We can't delete the entire configuration file because then* `pim6sd` *won't start, but an empty file is perfectly fine for many routers.* (121)

18.3.3 Running PIM-SM

It should be reasonably simple to set up a test environment this way. If we use the same environment as in section 18.2.1, then we can use the same basic test sequence as there.

PIM-SM differs from PIM-DM in a major way: It behaves fairly sluggishly if we add another router. Once the router has figured out where the rendezvous routers are, then a non-RP router runs quite nicely. But if a rendezvous point or the bootstrap router is restarted, then it takes some time for the routing cloud to recover. A number of timeouts default to 150 seconds, so it may take several minutes for the PIM-SM cloud to converge to a stable state if anything happens, especially if both the rendezvous point and bootstrap router run on the same node or are rebooted simultaneously.

This has some consequences: Restarting a PIM-SM daemon does have some impact on the multicast routing, especially if we restart a rendezvous point or the bootstrap router. As with RIPng, we should leave PIM-SM time to converge to normal operations. During that time we should watch the routers rather than change the configuration too quickly.

There are tools to watch the routing daemons at work:

Debian Sarge with mrd6 *The command* `mrd6sh` *lets us manipulate a running* `mrd6` *daemon. Since this won't write updates to the configuration, using it for these purposes is likely to cause trouble, though. But running* `mrd6sh show` *shows a fairly detailed status of the running daemon.*

FreeBSD 6.1 *Running the command* `pim6stat` *shows similar status information of the running daemon.* (122)

Beyond that a packet sniffer is sometimes useful to watch PIM-SM at work.

18.3.4 Inside IPv6: The PIM-SM Protocol

To understand the PIM-SM traffic we need to take a closer look at the protocol again. Since PIM-SM is far more complex than PIM-DM, we can't look at all the packets in a sniffer, though.

When we use a packet sniffer to observe the PIM-SM traffic, we notice first that PIM-SM uses the same protocol type 103 (0x67) and multicast group `ff02::d` as PIM-DM. Even the hello messages between the routers look strikingly similar. And as we'll see later on, other message types are also shared between the protocols.

The hello messages within a single subnet serve an additional purpose: Every subnet has a *designated router* (*DR*), which is the one to receive multicast packets and forward them to the appropriate rendezvous point. All routers in a subnet watch for the hello packets. The one with the largest IP address becomes the designated router.

The *bootstrap messages* that PIM-SM routers flood through the cloud contain the IP address of the bootstrap router and a list of rendezvous points with the multicast address prefixes they are willing to serve.

When a new candidate bootstrap router comes up and realizes that there is either no bootstrap router yet or the current bootstrap router has a lower IP address than itself, then it will start to send out bootstrap packets. The candidate bootstrap router with the highest address assumes the role of bootstrap router and sends bootstrap packets in a 60 second interval to the routing cloud. The other candidates just watch for periodic bootstrap packets and if they don't arrive in time, they will start to send them out themselves again.

The flooding mechanism is similar to that of PIM-DM but is only used for PIM-SM internal purposes, not to distribute routed multicast packets.

When a candidate rendezvous point comes up, then it first obtains the address of the bootstrap router through the bootstrap messages from its peers. Then it sends a *candidate RP advertisement* to the bootstrap router; this is a unicast packet that contains a list of prefixes that the candidate rendezvous point is willing to handle as a rendezvous point. The bootstrap router will then include the new rendezvous point in its periodic bootstrap messages.

The rendezvous point keeps sending its advertisement to the bootstrap router at 60 second intervals. If these advertisements cease to arrive at the bootstrap router, then the bootstrap router will drop the rendezvous router from its bootstrap messages.

When a host starts to send multicast packets to the subnet it connects to, then the designated router picks them up and forwards them to the rendezvous point in a *register* message. These packets are sent unicast to the rendezvous point; besides a few flags they contain the entire original packet.

If the rendezvous point doesn't know of any listeners to the multicast group, then it sends a *register stop message* back to the designated router.

This packet tells the designated router to stop sending register packets for this multicast address.

When a listener subscribes to a multicast group, it sends a *join message* to its upstream peer—its next peer towards the rendezvous point. The upstream peer sends another such join message and so on, all the way to the rendezvous point, which then starts to send packets for that multicast group back. With multiple listeners, this creates a routing tree with the rendezvous point as the root.

When a listener shuts down, it sends a *prune message* to signal its upstream router that it doesn't need that multicast group anymore. If the upstream router doesn't have any more listeners for the multicast group, it will send another prune message to its upstream and so on.

The join and prune messages use the same packet type as the join/prune messages with PIM-DM.

The routing tree with the rendezvous point at the root is called a (*shared*) *rendezvous point tree* (*RP-tree*), because it is used for all packets to the multicast group independently of the source. If there are only few but high-volume senders to a multicast group, then this strategy is inefficient: Packets travel in a register packet from the sender to the rendezvous point and then to the recipients, which may be a detour adding delay and creating unnecessary traffic if sender and receiver are close together but far apart from the rendezvous point.

So in this situation the rendezvous point or the designated router of the receiver may use source-specific join/prune messages to establish a separate *shortest path tree* (*SP-tree*) or *source-based forwarding tree* for the source. This shortest path tree has the designated router of the sender as the root, so the routing is more efficient.

These SP-trees put an additional burden on the multicast routers involved, however: Instead of keeping a single multicast routing entry $(*, G)$ for the multicast group G, they now need to maintain a routing entry (S, G) for every sender S to that multicast group. For this reason an SP-tree is only established if the sender exceeds a configurable traffic threshold.

As usual these explanations ignore a number of fine points, but they should suffice to understand how PIM-SM works even for debugging purposes. Beyond that RFC 2362 [39] is the definitive reference.

18.3.5 Source-specific Multicasts (SSM)

While the so-called *any source multicasts* (*ASMs*) we have used so far let a listener subscribe to a multicast group, *source-specific multicasts* (*SSMs*) let a listener subscribe to a multicast address from a specific source address. This improves the efficiency of multicast routing in some cases.

Addresses with an `ff3`x`:0:0:0:0:0::/96` prefix (with x being the scope flag we already know) are treated specially with PIM-SM: When a listener

subscribes to such an address it must also supply the address of the sender it wants to listen to. The combination (S, G) of the sender address and multicast group is called a *channel* in SSM terminology. The listener uses MLDv2 to tell the next designated router that it wants to receive such a channel—MLDv1 doesn't support source address selection. Then the PIM-SM router sets up a source-specific shortest path tree to the sender.

This approach has three major advantages: SSM addresses can be allocated locally on the sender, SSMs avoid the switch from an RP-tree to an SP-tree by using an SP-tree from the beginning, and SSMs don't need a rendezvous point, so source-specific multicasts scale well enough to be used throughout the entire Internet.

The major disadvantage of source-specific multicasts is its need for MLDv2: The IPv6 stacks of most Unixen simply don't support it yet or the header files are missing the necessary macro definitions. None of the Unixen presented here support source-specific multicasts on end points.

Both `mrd6` and `pim6sd` do however support source-specific multicasts. Since they implement the router part of MLD, they should even work on IPv6 implementations that don't support SSM as end points.

If we had SSM-capable senders and receivers, then the `mcjoin` tool would let us test the SSM routing: If we pass it a unicast address as a second command-line parameter, then it will use this as the source address for source-specific multicasts.

The official specifications for source-specific multicasts are spread over various RFCs. To ease understanding of SSM, RFC 3569 [9] provides an overview and refers to the other RFCs where necessary.

18.3.6 Embedded Rendezvous Point Addresses

Source-specific multicasts make rendezvous points unnecessary in the special case of a single sender. If there are multiple senders, then we still need a rendezvous point. RFC 3956 [98] uses a special range of multicast addresses that embed the address of the rendezvous point within the multicast address.

The rendezvous points need a special type of address for this: Their interface ID must have the form `0000:0000:0000:000`i, where i is anything from 1 to `f`. A multicast address with embedded rendezvous point starts with `ff7`s, where s identifies the scope as usual. Then follows a zero nibble, then a nibble containing the last nibble i of the interface ID of the rendezvous point. The next two nibbles contain the length of the network prefix, which then follows. The remaining 32 bits contain the multicast group ID. For example, the multicast address `ff75:0a40:2001:db8:fedc:1:42:678` contains the embedded rendezvous point address `2001:db8:fedc:1::a`. Figure 18.2 shows how the rendezvous point address is extracted from such a multicast address.

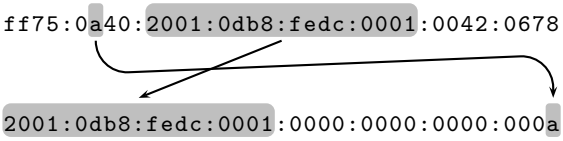

Fig. 18.2. A rendezvous point address embedded in a multicast address

Embedded rendezvous points addresses don't need explicit support on the sender or receiver side, only the routing daemons must support them; both `mrd6` and `pim6sd` do.

While squeezing a rendezvous point address into a multicast address like this doesn't look particularly aesthetic, doing so makes us independent of bootstrap routers. Since the bootstrap protocol doesn't scale but embedded RP addresses do, using them makes multicast routing scale even if we want to use it across the entire Internet.

18.4 Multicast Address Allocation

How do we choose and possibly announce the multicast addresses that we want to use for our programs and services? So far we have simply used an address with site-local scope and assumed that nobody else uses it. But there are scenarios that simply forbid this approach.

If source-specific multicasts were widely available, then we could use them to deal with the "Internet radio" scenario: A single sender distributing packets via multicasts to a large group of sparsely distributed listeners could use an SSM address which we could simply publish through the DNS.

But if we have few listeners and a large number of senders, possibly outside our administrative domain, then things do get slightly more difficult. We can use a multicast address with an embedded rendezvous point address; if we also run the rendezvous point, then an address clash is effectively impossible as long as we properly allocate all addresses used by our rendezvous point. The same approach also supports the "Internet radio" scenario as long as SSM functionality is still unavailable.

Even if we don't want to embed a rendezvous point address in a multicast address we use, for example if we use PIM-DM instead of PIM-SM, then we have a range of multicast addresses at our own disposal: RFC 3306 [51] allocates all addresses of the pattern `ff3`*s*`:00`*nn*`:/24` in a way very similar to the way we have embedded the rendezvous point in a multicast address: Here *s* identifies the scope of the multicast address again and *nn* holds the number of bits that contain the network prefix that "owns" the address. Next follows the network prefix and finally the remaining bits contain a shorter than usual multicast group ID.

Consider an example again: If our assigned global routing prefix is still `2001:db8:fedc::/48` and we need a global scope multicast address, then we can use any address with the prefix `ff3e:30:2001:db8:fedc::/80` for our purposes. The difference between the embedded RP addresses and these *unicast-prefix-based multicast addresses* is the second bit in the third nibble: If it is set, then the address contains an embedded RP address, otherwise it is just an address allocated to a site together with their unicast prefix.

This still leaves us with an unsolved problem: If we want to allocate addresses on a long-term basis, then we just assign them manually and possibly store them in the DNS. But if a program needs a large number of dynamically allocated multicast addresses, then we need to find a way to do that. RFC 2730 [55] specifies the *multicast address dynamic client allocation protocol* (*MADCAP*). It works much like traditional DHCP, but allocates multicast addresses. Unfortunately, implementations aren't available yet.

18.5 Operational Issues

The most prominent reason why large-scale multicasting never became overly popular in the IPv4 world is its complexity; since multicasts have been retrofitted into an existing IPv4 base they were complex, tedious to handle, didn't scale properly and were often unfit for production-grade purposes. With IPv6, multicasts have become significantly easier to use. Still, a number of operations-related issues do exist.

Multicasts, and especially multicast routing at a large scale, are a complex technology. It still requires experience and intimate knowledge of their inner workings for reliable operations—and as usual, we have omitted a lot of the finer details. And even then, a rendezvous point failover—one of the finer details we have omitted—takes far longer than an OSPF route update.

Multicast routing is inherently resource-intensive on all multicast routers. For every rendezvous point or SSM (S, G) pair the routers that forward this traffic to a listener need to maintain state. There is no such thing as route aggregation, so the multicast routing tables can become large enough to cause serious problems. Implementing them in hardware seems to be much more difficult than implementing unicast routing, though only time will tell if the dedicated router hardware vendors will come up with hardware-based solutions or not.

But IPv6 makes multicasting much easier than IPv4: We can embed RP addresses in multicast addresses, which effectively solves the most critical scalability issues. As soon as source-specific multicasts become generally available, multicast routing will improve even more. Every allocated unicast prefix automatically comes with a matching multicast range, so there is no need to obtain multicast addresses from a RIR, Internet service provider or from IANA. Anycasts let us set up rendezvous points in a redundant manner, making multicast routing reliable enough for many production grade purposes.

Beyond these rather fundamental considerations, two questions remain: How do we arrange between PIM-SM and PIM-DM, and what is the best approach to inter-domain multicast routing?

Using both PIM-SM and PIM-DM requires some sort of "border router" that translates between the two. To my knowledge no such translator exists, so we are effectively forced to choose between the two. In a BSD-only environment that uses multicasts only within a reasonably small cloud, PIM-DM may be a good move; certain high performance clusters might be a typical example for such an environment. Beyond that, and as soon as Linux comes in, PIM-SM is probably the best move since it avoids a later switch if we need inter-domain multicast routing.

If we need inter-domain multicast routing, using embedded RP addresses is the first thing to do; there are protocols to route between multicast routing domains, but they aren't widely available and don't scale as well as embedded RP addresses. If we want to avoid running a multicast routing daemon on all routers, then we can use 6in6 or configured 6in4 tunnels to connect disjoint multicast routing clouds with each other.

If performance becomes a problem, then it can be helpful to "short-circuit" communications between some multicast routers by using tunnels between them. That way we can reduce the size of the multicast routing tables of all other routers between them.

If we run a multicast router as both bootstrap router and rendezvous point, then recovery takes a long time if this router fails: First a new bootstrap router must be chosen, then all rendezvous points must advertise their service to the new bootstrap router, then the bootstrap router must flood its bootstrap messages to the routing cloud and then all other multicast routers must reconfigure their rendezvous points. To speed up recovery we should always put candidate bootstrap routers and candidate rendezvous points on different routers.

18.6 Packet Filter Considerations

There are three types of packets that relate to multicasts: ICMPv6 packets for multicast listener discovery, PIM packets for PIM-SM and PIM-DM, and the multicast packets themselves. We may need to deal with all of them in a packet filter configuration.

Multicast listener discovery queries have an ICMPv6 type 130, a link-local unicast address as the source address, and either a valid unicast or a multicast address as the destination. MLDv1 listener reports have an ICMPv6 type 131, either a link-local unicast address or the unspecified address as source and the address of the multicast group to listen to as the destination address. MLDv2 listener done messages have the ICMPv6 type 132, a link-local unicast address as source and `ff02::2` as the destination. MLDv2 listener reports have the

ICMPv6 type 143, either a link-local unicast address or the unspecified address as the source and `ff02::16` as the destination address. All MLD packets have a hop limit of 1.

PIM uses its own transport layer protocol number, 103 or 0x67, and its own link-local multicast group `ff02::d`. Packets are either sent to this multicast address or to the unicast address of the intended recipient; since the bootstrap routers and rendezvous points may be located on a different subnet, routable unicast addresses may be used for both the source and destination addresses here.

Finally, of course we may want to filter the multicast packets themselves; the configuration depends entirely on the particular requirements in the given environment.

18.7 Advanced Topics and Further Reading

Multicast routing is an extremely complex subject well worth an entire book of its own. So again we have skipped a number of advanced topics. Probably most important are *anycast rendezvous points*, as described in RFC 3446 [82]. The *multicast source discovery protocol* (*MSDP*) defined in RFC 3618 [41] may be relevant for inter-domain multicast routing if we can't use embedded rendezvous point addresses. As soon as MADCAP implementations as defined in RFC 2730 [55] become available, they are worth a closer look.

For the definitive specifications on any multicast topic, check these RFCs: MLD is defined in RFC 2710 [23], RFC 3590 [50] and RFC 3810 [36]. PIM-SM and PIM-DM are specified in RFC 2362 [39] and RFC 3973 [1], respectively. RFC 3569 [9] gives an overview of SSM and RFC 3956 [98] standardizes embedded RP addresses. RFC 3306 [51] and RFC 3307 [49] cover allocation guidelines for multicast addresses and unicast-prefix-based multicast addresses in detail. If you want to use multicasts in your own programs, RFC 3678 [107] documents the related socket API extensions.

Finally, multicasts and multicast routing are still areas of active research, so if multicast routing is relevant to you, then watch out for new developments.

19

The Dynamic Host Configuration Protocol (DHCPv6)

In the IPv4 world the most common means to assign dynamically allocated IP addresses is the *dynamic host configuration protocol* (*DHCP*). A DHCP server usually provides a client with a temporarily allocated address plus some additional information like the addresses of the closest DNS servers.

Despite the fact that stateless autoconfiguration is widely considered the canonical way to assign IPv6 addresses, there is also an IPv6 variant of DHCP called *DHCPv6* defined in RFC 3315 [35]. Like DHCP, DHCPv6 also serves two purposes: It provides an alternate means to assign IPv6 addresses to clients and it disseminates general information about the network and its local facilities.

19.1 Installation

As of today, not all Unixen bring DHCPv6 support with them. Since autoconfiguration is generally superior to DHCP with respect to address management, the need for DHCPv6 isn't that pressing. But then, distributing general network information with DHCPv6 does have its merits, so in a number of cases it is worth the trouble to install even if this involves building it from sources ourselves.

Debian Sarge *There is no DHCPv6 support included in the Debian distribution, but there are at least two DHCPv6 implementations available for Linux. According to the documentation,* Dibbler *seems to be the most usable. Its sources are available at* `http://klub.com.pl/dhcpv6`*. The latest release there is 0.4.1; it compiles without problems. (There are also Debian packages available, but they don't install on Sarge.) Running* `make install` *installs the three daemons* `dibbler-client`, `dibbler-server` *and* `dibbler-relay`.

Running `make install` also sets up some non-empty configuration files for the three daemons; these files are named `/etc/dibbler/{client,relay,server}.conf` and contain some randomly chosen addresses. Rename these files right away to avoid trouble starting daemons with broken configuration data later on.

Next we need to set up our own boot script. A very rudimentary version looks like this:

/etc/init.d/dibbler

```
#! /bin/bash
for dmn in client server relay
do
    [ -e /etc/dibbler/$dmn.conf ] && /usr/local/sbin/dibbler-$dmn $1
done
sleep 3    # Wait for daemon to get data or shut down
```

This script will start and stop all daemons that have a matching configuration file in `/etc/dibbler`: Invoking the daemons with either `start` or `stop` as the command line parameter will start or stop them, respectively. We should then set the appropriate link in `/etc/rc2.d/` to the script; if we run the server or relay, then it should start late in the boot process, otherwise we should run it near the network initialization.

FreeBSD 6.1 *The `dhcp6` package contains the client `dhcp6c`, server `dhcp6s` and relay `dhcp6relay` daemons, so we need to install the same package on all DHCPv6 machines.*

The variables `dhcp6c_enable`, `dhcp6relay_enable` and `dhcp6s_enable` in `/etc/rc.conf` control which daemon to start. We must set the appropriate variable to `YES` to start the daemon we want.

Additionally the `dhcp6c` and `dhcp6s` programs need to know on which interface to run. We pass the interface name in the `dhcp6c_flags` and `dhcp6s_flags` environment variables. So the boot configuration for a client looks like

/etc/rc.conf

```
dhcp6c_enable=YES
dhcp6c_flags="lnc0"
```

and, quite similarly, the server boot configuration is

/etc/rc.conf

```
dhcp6s_enable=YES
dhcp6s_flags="lnc0"
```

Finally, due to the way that the client and server daemons work, we need to set up a shared secret in a file `/usr/local/etc/dhcp6[cs]ctlkey`. It must be a random base64 encoded string. We can generate the file using `openssl` like

```
# openssl rand -base64 64 >/usr/local/etc/dhcp6cctlkey     || Client
# openssl rand -base64 64 >/usr/local/etc/dhcp6sctlkey     || Server
```

for the client and accordingly for the server, too.

Solaris 10 *So far there is no support for DHCPv6 available.* 123

At this point our DHCP daemons are ready to run. But we still need to tell the server which information to distribute and the client how to use it. That turns out to be more work than expected.

19.2 Stateless DHCPv6

We start with the most simple, and in many cases most important, aspect of DHCPv6: the stateless distribution of various site-wide information, especially the DNS resolver configuration.

The subset of DHCPv6 that doesn't deal with address allocation is often referred to as *stateless DHCPv6*, because the DHCP server doesn't need to keep track of the addresses assigned to clients. It is perfectly feasible to set up multiple independent DHCP servers within a network to achieve a virtually unlimited degree of reliability. Except for the need for relay routers, which we'll take a look at below, stateless DHCPv6 avoids all of the problems we have considered in section 4.3.1 when we introduced autoconfiguration as a successor of DHCPv4 for address allocation.

19.2.1 The First Step: Resolver Configuration

First we tell the DHCP server the IPv6 addresses of the name servers and the local domain search list. To show how to add multiple name servers and domain names we assume that we have two name servers at `2001:db8:fedc::1` and `2001:db8:fedc::2` and use the domains `example.com` and `example.net` in the domain search list.

Debian Sarge with Dibbler *In `/etc/dibbler/server.conf` we set up our configuration according to this pattern:*

`/etc/dibbler/server.conf`

```
stateless
iface eth0 {
    option dns-server 2001:db8:fedc::1, 2001:db8:fedc::2
    option domain example.com, example.net
}
```

With this we first tell the server to offer only stateless configuration data. Then we configure the data it sends out; we must do so for every interface individually. Finally we must restart the daemon.

FreeBSD 6.1 *We create a configuration file* `/usr/local/etc/dhcp6s.conf` *for the DHCP daemon with the lines*

`/usr/local/etc/dhcp6s.conf`

```
option domain-name-servers 2001:db8:fedc::1;
option domain-name-servers 2001:db8:fedc::2;
option domain-name "example.com";
option domain-name "example.net";
```

and then restart the `dhcp6s` *server either by hand or through its boot script.* 124

Next we check that our client can actually reach the server and obtain its data.

Debian Sarge with Dibbler *There is no way to do a "dry run" that shows the data we receive.*

FreeBSD 6.1 *The* `dhcp6c` *program has an option* `-i` *to find a server and query for its stateless data. Assuming that our client uses interface* `lnc0` *for its network connection, we can check with*

```
# dhcp6c -i lnc0
nameserver[0] 2001:db8:fedc::1
nameserver[1] 2001:db8:fedc::2
Domain search list[0] example.com.
Domain search list[1] example.net.
```

that the client actually receives data from the server. Before we run this command we must first stop all running `dhcp6c` *daemons, otherwise the two processes will attempt to use the same port number on the client.* 125

Finally we need to tell our client what to do with the data it receives. The implementations differ significantly in the way they handle the client-side configuration.

Debian Sarge with Dibbler *We just tell the Dibbler client what data we are actually interested in. The configuration*

`/etc/dibbler/client.conf`

```
stateless
iface eth0 {
  option dns-server
  option domain
}
```

in `/etc/dibbler/client.conf` *tells the client that we are only using stateless configuration data in general and only want to obtain the DNS servers and domain name search list from interface* `eth0`*.*

The Dibbler client knows what to do with the received information and automatically updates the resolver configuration. When the client shuts down it even restores `/etc/resolv.conf` *to its old contents.*

FreeBSD 6.1 *Different than Dibbler, the* `dhcp6c` *follows an approach more similar to that of the ISC DHCP implementation for IPv4: It just passes the data it finds to a shell script and lets the script deal with everything else. Different than ISC DHCP, the* `dhcp6` *package doesn't ship with a pre-built script; it is left to us to set up a script according to our needs.*

First we tell the client what information we are interested in. We need an entry like

/usr/local/etc/dhcp6c.conf

```
interface lnc0 {
  information-only;
  request domain-name-servers, domain-name;
  script "/usr/local/etc/dhcp6c-script";
};
```

in `/usr/local/etc/dhcp6c.conf` *to tell the client that we only want stateless information and that we need the DNS server addresses and the domain name search list from the server. The* `script` *statement tells the client where to find the shell script that actually configures the system; it must refer to an executable file that is owned by the same user as the one that the* `dhcp6c` *is running as.*

Next we need the shell script. A look at the ISC DHCP script immediately shows that a proper script isn't trivial to write. We limit ourselves to a minimum and just rewrite the `/etc/resolv.conf`*:*

/usr/local/etc/dhcp6c-script

```
#! /bin/sh
mv /etc/resolv.conf /etc/resolv.conf.old
( echo "# Configuration created by $0"
  echo "# on `date`"
  echo "$new_domain_name_servers" | tr ' ' '\n' \
  | sed '/^ */d;s/^/nameserver /'
  echo "search $new_domain_name"
) >/etc/resolv.conf
```

126

Configuring the resolver is probably the most pressing reason to use DHCPv6, and fortunately it works without any major problems. Other configuration data may be more troublesome, as we see next.

19.2.2 Adding More Stateless Data

The second important configuration data that we may want to distribute with DHCPv6 is the address of the local time server. Since short round-trip times and low latency improve the precision of NTP quite significantly, using a close time server is sometimes important.

Unfortunately, support for NTP was only specified in RFC 4075 [77] in May 2005 and hasn't found its way into all implementations yet.

Debian Sarge with Dibbler *NTP support according to RFC 4075 is available.*

FreeBSD 6.1 *The NTP support in the* `dhcp6` *package is based on a draft of the standard and by default not compiled in.* (127)

Surprisingly enough, the first service that DHCPv6 actually supported was not DNS but SIP, the session initiation protocol used for voice over IP signalling. RFC 3319 [100] specifies how to distribute information about the local SIP and DNS servers with DHCPv6. All implementations support it.

Additionally, RFC 3898 [76] defines the distribution of some information about the local *network information system* (*NIS* and *NIS+*) infrastructure.

Debian Sarge with Dibbler *The NIS/NIS+ support is available out of the box.*

FreeBSD 6.1 *NIS/NIS+ is not supported.* (128)

In short, some work still needs to be done before distributing more than the resolver configuration with DHCPv6 is generally feasible.

19.3 Address Management with DHCPv6

Despite the fact that stateless address autoconfiguration provides such a superior way to assign IPv6 addresses to hosts, the DHCPv6 standard still supports the dynamic allocation of addresses through a server.

The *managed flag* in router advertisements explicitly notifies a client of the existence of an address managing DHCPv6 server in the subnet. Unix clients don't use this flag, however.

Not all implementations support this feature, at least not in the way that RFC 3315 expects:

Debian Sarge with Dibbler *It is possible to set up address pools for dynamic allocation.*

FreeBSD 6.1 *There is no dynamic allocation available, but it is possible to assign fixed addresses to fixed clients.* (129)

In general, address management based on DHCPv6 is rather unexciting in virtually all practical cases.

19.4 DHCPv6 Across Subnet Borders

DHCP was originally designed as a service for clients that don't have a routable address yet. As a consequence, DHCP servers need to be connected to the same subnet as their clients. To minimize the need for DHCP servers, it is possible to connect *DHCP relays* to all subnets with DHCP clients; these relays forward the DHCP requests to the actual servers using routable addresses.

Unfortunately, even if a client has already obtained its routable address and default route through autoconfiguration, there is still no way for it to contact a DHCP server in another subnet without the need for a relay.

So we now take a look at how to set up a DHCP relay. There are two ways a relay can find a server: We can tell it the routable IPv6 address of a server or we can use a site-local scoped multicast address.

With IPv4, using DHCP relays was extremely useful because it limited the number of nodes that needed to keep track of the assigned address leases; this was essential to enable dedicated router hardware without hard disks to provide DHCP services in the subnets that they connect to. With IPv6 and stateless DHCPv6 the only advantage of DHCP relays is the fact that the configuration data doesn't need to be replicated to every single subnet.

Still, DHCPv6 does support relays. So we might as well take a look at how to set them up.

19.4.1 Setting Up a DHCP Relay

First we configure the relay explicitly with the unicast address of the server. While this implies an additional configuration effort on all relays, it doesn't depend on a multicast routing infrastructure.

Debian Sarge with Dibbler *We can set up a relay using a configuration file* `/etc/dibbler/relay.conf` *like*

/etc/dibbler/relay.conf

```
iface eth0 {
  server multicast no
  server unicast 2001:db8:fedc::2
  client multicast yes
  interface-id 1234
}
```

to run a relay on interface `eth0`*. The relay won't try to reach the server on a multicast address but only on its unicast address* `2001:db8:fedc::2`*. It accepts client requests via multicasts. The* `interface-id` *defines a numeric value that serves to disambiguate multiple servers. This value must be configured the same on the server and relay.*

Additionally we must update the server configuration to support relayed requests. We add three lines to the configuration for the interface we listen to in `/etc/dibbler/server.conf` *like*

/etc/dibbler/server.conf

```
iface eth0 {
  relay eth0
  interface-id 1234
  unicast 2001:db8:fedc::2
  [...]
}
```

The `relay` *option tells the daemon that we are willing to handle relayed requests on this interface. The* `interface-id` *option must match the configuration of the relay. Finally we must also specify the unicast address that we accept relayed requests on.*

FreeBSD 6.1 *The* `dhcp6relay` *daemon is only configured through command line arguments. We can pass it an arbitrary number of interface names to listen to. Additionally, we tell it where to find its DHCPv6 server using the option* `-s`*. For a permanent configuration we can simply set the variable* `dhcp6relay_flags` *in* `/etc/rc.conf` *accordingly:*

/etc/rc.conf

```
dhcp6relay_enable=YES
dhcp6relay_flags="-s 2001:db8:fedc::2 lnc0"
```

Servers will automatically serve requests from relays. 130

19.4.2 Multicasts from Relay to Server

Instead of configuring the relays with the address of the server we can use routed multicasts from the relays to the servers using the site-local multicast address `ff05::1:3`. This approach either requires the relays to have an interface in the same subnet as the servers or a multicast routing setup, but it doesn't need the relays to know the address of the server and lets us set up redundant servers.

Configuring a relay that uses routed multicasts to the server is actually simpler than configuring it with the unicast address of a server.

Debian Sarge with Dibbler *At this time the Dibbler relay has a major limitation: It forwards requests as multicasts with a hop limit of 1, thereby preventing them from being routed to the server. If we equip the relay with multiple interfaces, one connected to the subnet with the clients and the other to the subnet with the servers, then we can still use the relays. If* `eth0` *connects to the client subnet and* `eth1` *to the server subnet, then we can configure the relay as*

/etc/dibbler/relay.conf

```
iface eth0 {
  client multicast yes
  interface-id 1234
}
iface eth1 {
  server multicast yes
}
```

The server needs to be configured as with an explicit relay except that the `unicast` *statement may be missing.*

FreeBSD 6.1 *If the relay node has multiple interfaces, then we need to start the relay daemon with the additional option* `-r` *to specify the interface through which we want to send the multicasts to the server. If we only run the daemon on a single interface anyway, then we can omit this additional option.*

On the server we don't need to touch the configuration at all. 131

The need for a relay is a historic legacy; with IPv4, where DHCP is primarily used to assign addresses, a client doesn't yet have the necessary routable address to reach a DHCP server in another subnet. With IPv6, multicast routing and stateless DHCPv6 it is possible to make a client use the relay-server protocol to query the server directly and do away with the relays. Implementations don't support this, however.

19.5 Interoperation Problems

There exist at least two interoperation problems between Dibbler and the BSD `dhcp6` package that are worth knowing about.

Dibbler considers the "interface ID" DHCP option mandatory for communication between relays and servers while the BSD relay doesn't use it at all. In consequence, a Dibbler server and BSD relay won't interoperate.

The Dibbler server only provides the data that the client requests while the BSD server sends out all the data it has. The BSD client won't explicitly ask the server for the data it needs but expects the server to send all the data it has. In consequence, a BSD client won't work with a Dibbler server.

To avoid these problems it is easiest either to use only a single implementation across all networks or use the BSD implementation on all relays and servers and whatever is available on the clients.

19.6 Conceptual Security Aspects

By its very concept, DHCP is an inherently insecure protocol: It provides a trusting client with some extremely security-critical information.

According to the standard, DHCPv6 supports some authentication feature based on shared secrets between the client and server. Since these need to be distributed to the client, their use is questionable—if we need to touch the client, then we might as well install the configuration data statically. It is obviously both safer and more flexible to update files like `/etc/resolv.conf` or `/etc/ntp.conf` using tools like SSH and Rsync or Rdist. Only in the case of a node that frequently changes its network connectivity does DHCP prove useful. But in that case handling the shared secret becomes so much of a burden that it is effectively useless for practical purposes.

Debian Sarge with Dibbler *There is no support for authentication.* 132

In consequence, DHCP can't be used in security-critical environments. Providing "guest" subnets with a DHCP server may be quite useful, but "guest" nodes connecting to such a subnet must then be aware that the information obtained by DHCP is inherently untrustworthy.

19.7 Packet Filter Considerations

Clients listen on UDP port 546 while relays and servers listen on port 547. Clients use their link-local unicast address to contact the server or relay on the link-local *all DHCP relay agents and servers* multicast address `ff02::1:2`. Relays contact the server on the site-local *all DHCP servers* multicast address `ff05::1:3`.

20

Bridging the DNS Gap

Stateless autoconfiguration offers a reliable and highly automatic way to assign IPv6 hosts their addresses, avoiding all of the problems that arise from traditional DHCP-based address management. But how do these addresses find their way into the DNS?

20.1 From Autoconfiguration to the DNS

When I first got seriously involved with IPv6, this question stumped me. As a Unixer I expected to register all nodes with the DNS, if only to make them accessible to remote administration. But if autoconfiguration changes addresses on the fly, then how can the DNS be kept in sync with the addresses allocated by autoconfiguration?

First I assumed that I missed some obvious but radically un-IPv4-ish solution, but I couldn't find one anywhere. I hacked up some primitive scripts to deal with my immediate needs and deferred the problem.

When I finally did some more research on the issue I found out that no, apparently nobody had taken care of it properly. People got interested in the scripts, so I spent some more work on them. At least one of them is still "lab grade" software, but still they are quite useful to maintain the DNS entries in an IPv6 environment.

20.2 Solution Strategies

I asked around for a "production grade" solution but couldn't find any. Some suggestions that came up were simply impractical, others focused on protocol standards rather than existing implementations.

20.2.1 "But Only Servers Need DNS Entries"

One of the first reactions was: "Why do you want to register all machines with the DNS?" While this question is perfectly reasonable if the environment consists of large numbers of Microsoft Windows desktop machines and a few servers, in a Unixoid data center with lots of servers, or a Unix environment with a large number of workstations that are remotely operated using SSH, the answer to this question is obvious: Not putting all nodes into the DNS makes them remotely inaccessible. From a security standpoint this may be desirable in some cases, but as a general strategy this approach doesn't work.

Worst case scenarios that require large numbers of nodes to be tracked in the DNS are web hosters offering root servers to their customers and high performance cluster environments using large numbers of cheap machines. But even the average medium-sized data center has enough servers that need DNS entries to make this approach infeasible.

20.2.2 Manual DNS Entries

Another reply I received was "How many machines do you have that you can't maintain the DNS entries manually?"

In my case this question led back to the scenario with the large data center. Even in a small to medium size data center with a few hundred servers, maintaining all addresses manually involves a lot of tedious, error-prone work.

With a bit of scripting know-how, or some experience with a decent file editor, a lot of scenarios are easy to handle. But for the initial entries we need to collect all interface IDs, which is infeasible if machines change a lot. Environments that offer virtual root servers based on virtualization software like VMware are an extreme example of this.

20.2.3 The DHCP Non-solution

Another suggestion was to use DHCP for address assignment and let the DHCP server update the name server. Besides the fact that DHCPv6 implementations so far don't generally support DNS updates, this approach has a number of flaws, depending on the way that the DHCP server matches IP addresses with DNS names.

DHCP servers that do DNS updates operate in one of three modes: They use the name that the client provides, they assign names to addresses no matter which client is currently using an address or they look the name up in a manually maintained table of (link-layer address, IP address, DNS name) triples for every client.

The first approach is unacceptable unless the client authenticates itself; even IPv4 DHCP implementations don't generally support this. Otherwise any client claiming to be `www.example.com` might mimic as our web server.

Tying the DNS name to the address doesn't help either: If the IP address `192.168.42.42` and the DNS name `desktop42-42.example.com` are statically associated but the clients using the address change, then we simply don't know how to reach Mr Turing's desktop computer when we need to.

Maintaining link-layer addresses manually does what we need, but then we might as well maintain them in the DNS, since autoconfiguration maps the link-layer address to the interface ID. This is tedious, error-prone and exactly what we want to avoid.

20.2.4 Dynamic DNS (DDNS) Updates

The more helpful replies I received showed that my scripts were heading the right way: All the pieces were there, but nobody had put them all together yet.

With autoconfiguration only the host itself knows its current addresses, so the host needs to make sure that its DNS entries are correct. If they are incorrect, then the host must update them using *dynamic DNS* (*DDNS*) update messages. To deal with renumberings, the host must monitor its addresses so it can update its DNS entry after a renumbering. To make DNS updates secure we need to use the *DNS security* (*DNSSEC*) extensions to authenticate the updates. Since authentication keys are always tied to the DNS record name, this only works for forward zone entries. Reverse zones can't be authenticated by the host, so we need a different strategy there.

The solution works like this: When we install a new host, then we create a so-called *transaction signature* (*TSIG*) on the name server; this "signature" is actually just a shared secret between the name server and client. On the name server side we tie it to the host's DNS name. We copy the "signature" to the host and run a program there that checks if its DNS entry is correct and if necessary uses the "signature" to send a DNS update to the name server. The name server periodically updates the reverse zones from the data it finds in the forward zones to keep both in sync.

The approach in itself and the implementation so far both have a number of inherent problems that we'll investigate in section 20.4. But until now there is no better solution available.

20.3 A Preliminary Implementation

The implementation consists of three separate programs called `maketsigkey`, `nsautoupdate` and `syncrevzone`. All three are available from my home page at `http://www.benedikt-stockebrand.net/`.

20.3.1 Configuring BIND for Dynamic Updates

First we have to make our name server accept authenticated dynamic updates: We must set up our zone files and the directory containing them so that the files can be updated dynamically and configure the `named` daemon to read the TSIG keys we generate.

The zone files themselves as well as the directory with the zone files must be writable to the user that the `named` daemon runs as. The configuration in appendix A uses `/var/named/zonedata` for the zone files so that the `/etc/` directory can be mounted read-only or managed using configuration management tools.

Next we create an empty file `/var/named/tsig.keys.conf`; later on the `maketsigkey` script will store the shared secrets in it. The script stores the key files for the clients in `/var/named/keys/`, so we must also create this directory.

To make the `named` daemon use the keys, we use an `include` statement in its configuration to read the `tsig.keys.conf` file. If we run `named` without a `chroot` environment, then we need a line

named.conf

```
include "/var/named/tsig.keys.conf";
```

in our `named.conf`; if we use the `chroot` feature and run `named` from within `/var/named`, then the line should read

named.conf

```
include "/tsig.keys.conf";
```

instead. If we want to run `named` from a different `chroot` directory, then we must adapt the `maketsigkey` script to write the file to the `chroot` environment.

Additionally we need to allow dynamic updates for our forward zones. For our purposes the most restrictive configuration we may want to use looks like this:

named.conf

```
zone "example.com" {
    type master;
    file "example.com.fwd";
    update-policy { grant *.example.com. self example.com. AAAA; };
};
```

The `update-policy` statement lets every client with a TSIG key issued to a DNS name in "`.example.com`" update its own AAAA records but nothing else.

At this point it is normally a good idea to restart the name server and ensure that the changes to `named.conf` work without any obvious problems.

20.3.2 Creating and Installing TSIG Keys

Next we create TSIG keys for our clients. The `dnssec-keygen` program must be installed on our name server; it usually ships with either the full BIND distribution or a separate "BIND utilities" package, so it should already be there; if not, we must install it. Then we install the `maketsigkey` program from my web site on the name server.

Now we run `maketsigkey`. As arguments it needs the DNS names that we want to create TSIG keys for. For example,

```
# maketsigkey  www.example.com.  ftp.example.com  ntp.example.com
Kwww.example.com.+157+09202
Kftp.example.com.+157+50018
Kntp.example.com.+157+04719
```

creates three TSIG keys, one for `www.example.com`, one for `ftp.example.com` and one for `ntp.example.com`; the trailing period is optional here. The output shows the base names of the key files created; they always start with an uppercase "K" followed by the fully qualified domain name, a plus sign, a numeric code for the authentication algorithm, another plus sign and a final checksum for the randomly chosen secret key. Accordingly, we can find the key files in `/var/named/keys`:

```
# ls /var/named/keys
Kftp.example.com.+157+50018.key
Kftp.example.com.+157+50018.private
Kntp.example.com.+157+04719.key
Kntp.example.com.+157+04719.private
Kwww.example.com.+157+09202.key
Kwww.example.com.+157+09202.private
```

Additionally, `maketsigkey` extracts the keys from the key files and writes them to the `/var/named/tsig.keys.conf` file:

/var/named/tsig.keys.conf

```
[...]
key ftp.example.com.
        { algorithm hmac-md5; secret "KB9GfwCxiKOA7h5hXNbJrA==";};
key ntp.example.com.
        { algorithm hmac-md5; secret "3sRrRinSdbri1AqZ1AthwA==";};
key www.example.com.
        { algorithm hmac-md5; secret "Q8Axaax94ehEikCwR65N3A==";};
```

Finally, `maketsigkey` signals the `named` daemon to re-read its configuration, including the updated `tsig.keys.conf` file.

Now the name server is ready to receive authenticated DNS updates for the three example domain names.

20.3.3 Updating the DNS Forward Zone Records

On the clients we need to install the `dig` and `nsupdate` binaries from the BIND 9 distribution; `nsautoupdate` uses them to communicate with the name server.

Debian Sarge *The* `dnsutils` *package contains everything we need.*

FreeBSD 6.1 *The programs are part of the core system.*

Solaris 10 *Unfortunately these programs are only distributed as part of the BIND 9 server package, so we need to install the entire* `SUNWbind` *package from the distribution media.* 133

Next we install the `nsautoupdate` package from my web site. It is a fairly simple shell script and should install without any problems.

To avoid replay attacks, authenticated DNS updates rely on proper time synchronization. If we don't use NTP yet, then we must ensure that the system clocks on the DNS server and client run in sync approximately within a minute.

Next we need to configure the `nsautoupdate` script. To do so we choose an interface and a scope, either `global` or `site` for site-local. The script will check this interface and the given scope to pick the address that it sends to the DNS server. If we use interface `eth0` and global scope, then we create a directory `/etc/tsig/eth0.global/`. Into that directory we copy both the `*.key` and `*.private` file from the TSIG key we have just created on the DNS server.

Finally we run `nsautoupdate`. If everything works as expected, then it will terminate without any output. To ensure that the update was successful we should finally `dig` for the DNS name we have just added.

Note that `nsautoupdate` adds only *one* address to the DNS. This is probably not the best strategy, but it keeps the script simple.

20.3.4 Maintaining DNS Reverse Zones

Unfortunately, `nsautoupdate` can't possibly update the reverse DNS entries; to do so it would need a TSIG key that authenticated the IP address in its pseudo DNS name format. So we need a different strategy for the reverse zones. In an administrative domain that controls both the forward and reverse zones, we can retrieve the forward zone data and use it to update the reverse zones.

The `syncrevzone` program uses this approach. It can be installed on an arbitrary node, but usually it runs on the name server. To use `syncrevzone`, we need to create a special TSIG key that we grant permission to update all our reverse zones. By convention we call this key `master`, so we create it with the command

```
# maketsigkey master.
```

on our name server. Next we grant this key permission to update our reverse zones. In our `named.conf` we augment the zone declarations of our reverse zones with an `update-policy` statement like

named.conf

```
zone "c.d.e.f.0.b.d.0.1.0.0.2.ip6.arpa." {
   type master;
   file "2001.db8.fedc.rev";
   update-policy {
      grant master.
        wildcard *.c.d.e.f.8.b.d.0.1.0.0.2.ip6.arpa. PTR;
   };
};
```

Analogously we need to update the matching `ip6.int` zone.

Now we run the `syncrevzone` command. We need to pass it the TSIG key and the names of the forward and reverse zones to synchronize. We must specify the reverse zones as a regular address prefix; `syncrevzone` will then update both the `ip6.arpa` and `ip6.int` pseudo domains. Additionally, `syncrevzone` searches `/var/named/keys` for the given key name if we don't give it a file name. So if we run

```
# syncrevzone -k master example.com 2001:db8:fedc::/48
```

then we should afterwards find our reverse zones updated as expected.

- ☐ Make sure that everything worked by querying the name server for the data in the reverse zones. The command to do this without any unnecessary output from dig is

  ```
  # dig +noall +answer c.d.e.f.8.b.d.0.1.0.0.2.ip6.arpa. axfr
  ```

  (and accordingly for `ip6.int`). The output must match the data in the forward zone.

20.3.5 Security Considerations

Considering the importance of the DNS for network operations it is obvious that it is generally very security sensitive. So how do dynamic updates and the tools we've seen affect security?

Generally speaking, enabling dynamic DNS updates doesn't improve the security of the DNS. Authentication needs some CPU time, so allowing DNS updates introduces an increased risk of denial-of-service attacks. But then, the DNS protocol in itself isn't particularly secure; without it, clients may be unable to find the servers they want to connect to, but all application-layer protocols that rely on the DNS to "authenticate" a peer are inherently

insecure. So the potential security impact of dynamic DNS updates depends on the particular environment.

TSIG authentication uses system clocks to prevent replay attacks. This approach has a major drawback: It makes the NTP service extremely security-critical. The time window for replay attacks, even if the NTP service wasn't manipulated, is several minutes, which may be sufficient for some automated attacks.

Finally, the TSIG "certificates" and the `tsig.keys.conf` file must be protected properly. If regular users have accounts to the name server, then the `/var/named/keys/` directory and the `tsig.keys.conf` files must be handled with particular care.

20.4 Operational Issues

Security considerations aside, are these tools fit for use in a production environment? Not in an unsupervised way. But if we just consider them tools to make the DNS administration less tedious, then they can be quite useful even in their current state.

The `syncrevzone` program is reasonably mature; its messages aren't the most elaborate and it still relies on `dig` and `nsupdate` as external programs, but otherwise it does its job. The `maketsigkey` script is fairly uncritical because it doesn't run unsupervised. The `nsautoupdate` script is immature in several ways: It uses the output of `ifconfig` to pick an address. It only adds a single address and depending on the output of `ifconfig` the choice may be suboptimal. It might be used unsupervised in a test environment, but definitely not in a production environment.

Running `syncrevzone` through `cron` is possible. Even if something goes seriously wrong, it only affects the reverse zones; `syncrevzone` won't touch the forward zones, so even in the worst case we can still rely on them to clean up the mess remotely.

Using `nsautoupdate` via cron is dangerous: It may store useless addresses in the DNS so that we can't reach the node involved remotely and during a network renumbering event with a large number of nodes the many concurrent `nsautoupdate` invocations may bring the name server down. In a production environment `nsautoupdate` should only be run manually. This simplifies the DNS management quite significantly, especially if networks change or machines move often, while it still minimizes the adverse effects of a malfunction.

Using dynamic updates introduces another problem: We can't manually update the DNS data by editing the zone files and then restarting the name server; if we did, we would lose all dynamic changes that came in meanwhile. So if we use dynamic updates, then we need to use them for all zone data administration, using a command like `nsupdate` or similar.

For this purpose I habitually generate another TSIG key called `admin` with `maketsigkey` and extend the `update-policy` statement for all zones with an additional line like

named.conf

```
zone "example.com" {
    [...]
    update-policy {
        [...]
        grant admin.  wildcard  *.example.com.  ANY;    || Allow updates
    };
};
```

Afterwards I use the command

```
# nsupdate -k /var/named/keys/Kadmin.+157+12345.key
```

to update all the DNS data in my zones; it takes a bit of practice to get used to this, but it works quite well.

20.5 Future Work

It should be obvious by now that the tools we have introduced in this chapter are a long way from a production grade solution to the DNS gap problem. Some additional work is necessary here.

The limitations of `nsautoupdate` are an obvious first candidate for improvements. The script should take into consideration all the interfaces and addresses in a more intelligent manner, maybe implementing an address selection algorithm similar to those we have seen in section 16.4 for source and destination addresses in applications. Additionally it should delay updates in a way that prevents load peaks at the name server during a network renumbering event and possibly allow for "batch" updates at the name server.

A more rigorous approach might replace the TSIG based upgrades with something different, like an SSL/TLS based update protocol. That would solve the replay attack issues and the dependency on time synchronization between the DNS clients and servers. Additionally it might offer updates beyond the DNS, like properly authenticated changes to various address based access control or firewall configurations—somewhat like universal plug'n'play (UPnP) done properly.

Even more fundamental problems are inherent to the DNS protocol. The DNS by design assumes that the association between nodes and their addresses is mostly static. It provides an efficient and scalable way to publish these associations, making heavy use of caching. With autoconfiguration, nodes may change their addresses quite frequently, for example if they move between different WLAN subnets, so the association between nodes and addresses becomes more or less volatile. At this time we can only configure the DNS

records with a short time to live if we want to ensure that outdated data in the DNS caches expires quickly, but that impacts the performance and scalability of the entire DNS.

In short, the DNS has some fundamental problems with the way it handles IPv6 addresses, so its entire architecture may experience significant changes in the future.

Part V

New Functionalities

21

IP Security (IPsec)

Of the new functionalities that IPv6 introduced to the TCP/IP stack, IPsec is best known, "backported" to IPv4 without loss of functionality and most widely implemented.

21.1 Basic Concepts

The ideas behind IPsec are quite straightforward: Additional option headers provide for the authentication and encryption of IP packets. In practice, IPsec itself is dauntingly complex. Implementations follow suit, ranging from "mostly undocumented" to "configuration nightmare" to "missing essential parts".

So we just take a look at the concepts and its operational implications but don't even attempt to set up a working installation.

21.1.1 Authentication and Encryption

IPsec supports two independent features: The authentication of a packet's source and the encryption of its contents. Both features can be used independently.

An authenticated packet contains an *authentication header* (*AH*) which certifies that the packet originated from the source address shown in the base header. Authentication uses a shared secret between the communication peers. The sender computes a checksum, or hash, over the shared secret, the relevant parts of the IP headers and the IP payload and stores it in the authentication header. The receiver recomputes the checksum and compares its result with the checksum found in the authentication header. If they match, then the receiver knows that the sender also holds a copy of the shared secret.

Encryption also uses an option header, called *encapsulating security payload* (*ESP*). It also uses a pre-installed shared secret between the peers. The

sender inserts an ESP header into an IP packet and encrypts all data following that header. The receiver finds the ESP header and then decrypts all the following data before it processes the packet.

21.1.2 Transport and Tunnel Mode

So far we have assumed that we want end-to-end encryption and authentication. But in section 21.2.1 we'll see that end-to-end encryption is sometimes a problem rather than a solution. To work around this and to use IPsec for *virtual private networks* (*VPN*s), IPsec also works together with IP-in-IP encapsulation.

End-to-end encryption is called *transport mode* while the combination of IPsec and encapsulation is called *tunnel mode*. Figure 21.1 shows the differ-

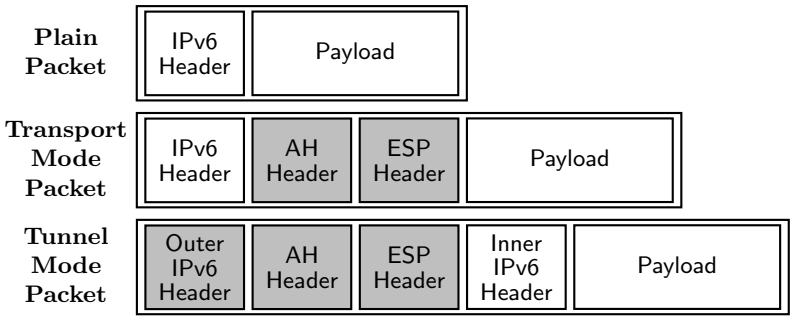

Fig. 21.1. Plain IPv6 packet and IPsec packets

ences between the packets. Things get slightly more complex if more option headers get involved; for our purposes it suffices to know that those option headers that are read by routers are placed before the IPsec headers and those that are only used by the final recipient are placed after. Everything that follows the ESP header is encrypted. Additionally, either the AH or ESP header may be missing.

21.1.3 Policy and Key Management Within the Kernel

When setting up IPsec, one of the first things we need to do is to decide which nodes will use which IPsec features between each other. The kernel maintains a *security policy database* (*SPD*) to decide which IPsec features to apply to outbound packets and which to require from inbound ones.

The SPD is quite similar to a packet filter configuration except that it doesn't filter packets that are forwarded. It is usually quite static and manually configured.

The kernel also stores all the shared secrets in a *security association database* (*SAD* or sometimes *SADB*). The SAD contains a set of *security associations* (*SAs*). Every SA consists of a source and destination IP address that it relates to, a key, an encryption or authentication algorithm, and a *security parameter index* (*SPI*). The SPI is a 32 bit integer that ESP and AH headers use to refer to an SA.

Consider an example: Node A wants to send a packet to node B. It first checks its SPD to see if it should apply IPsec, and if so, with what features and algorithms. Assume that the SPD on A says to use ESP with AES encryption and AH with an SHA1 hash. Next, A takes a look at the SAD to find the necessary SAs—one for ESP and another for AH. It takes the key and algorithm from the ESP SA, applies the algorithm to the data using the key and adds an ESP header behind the base header. The ESP header contains the SPI so the receiver can figure out which SA to use to decrypt the packet. Next, node A does almost the same for the AH header: It takes the key from the authentication SA, uses it to create a hash value from the key, the payload and those parts of the IP header that don't change in transit, and stores the hash value and the SPI in an AH header it puts behind the base header. Then it sends the packet to B.

When B receives the packet, it first takes a look at its own SPD to decide which IPsec features it requires from the packet. If any are missing or use an insufficiently secure algorithm, then B will silently discard the packet. Otherwise it will search its SAD for the matching SAs; if multiple SAs for the given source and destination address exist, then B will use the SPIs from the headers to find the correct ones in its SAD. It can then check the authenticity of the packet by verifying the hash value from the AH header and decrypt the payload. Then it processes the packet as usual.

While this example ignores a number of fine points, it explains how IPsec works within the kernel.

21.1.4 The Internet Key Exchange Protocol (IKE)

At this point, a major issue remains: How do we distribute those shared keys between nodes?

We can configure the SAs manually. This approach is simple and avoids a number of notorious interoperability problems. It is also tedious and error-prone. And what's worse, in a way it is less secure than dynamically exchanged keys: The longer a key is used, the more ciphertext an attacker can gather for cryptographic analysis for that key. Once the attacker has broken the key, he can decrypt a huge amount of traffic. What's worse, even if the attacker couldn't break the key, he could still use captured traffic for a replay attack; with captured NTP packets he could for example turn back a node's system clock to prepare for an attack on Kerberos, which relies heavily on synchronized clocks.

With dynamically distributed keys IPsec can use sequence numbers, much like TCP, to prevent these replay attacks. For this purpose the kernel asks a userland program to update the SAD if necessary. This userland program then uses a public-key cryptography protocol to set up the needed SAs and sequence numbers on both sides.

The protocol is called *Internet key exchange* (*IKE*). It uses parts of the *Internet security association and key management protocol* (*ISAKMP*), which is why some implementations refer to it as ISAKMP.

IKE operates in two phases. The first phase establishes a secure channel between the IKE daemons, possibly using public-key cryptography to authenticate each other. This secure channel is then used during the second phase to exchange shared secrets for the SAs.

In the most simple setup, IKE uses pre-shared keys between nodes to authenticate and encrypt the exchange of the IPsec keys; this is safer than using manually configured IPsec keys but still requires a separate key for every pair of nodes that communicate and a secure means to distribute the keys.

In a more advanced setup IKE uses public key cryptography. With unauthenticated, or self-signed, X.509 certificates we only need to distribute an X.509 certificate for every node to all other nodes; the distribution doesn't need to be secret as with shared keys anymore.

Finally, IKE can use X.509 certificates that are signed by a *certificate authority* (*CA*). With these, we only need to distribute the public key of the CA to all machines, like we do with HTTPS web servers. In fact, since both HTTPS and its underlying SSL/TLS protocol use X.509 certificates, we can use the same CA for both SSL/TLS and IKE. If we want to communicate securely with peers outside our administrative control, then we can use the services of well-known CAs; their certificates are distributed with virtually all HTTPS-capable web browsers.

Unfortunately, this scenario isn't widely supported by implementations. Taking a look at the documentation available today, IPsec is most commonly used either between individual hosts or in tunnel mode between VPN end points. Some implementations don't actually bother to compare the source address of an IKE packet with the address stored in the certificate.

The original IKE specification as of RFC 2409 [57] showed two problems: The key exchange was unnecessarily complicated and a cryptographic weakness in the IKE exchange was discovered. In December 2005, RFC 4306 [31] specified a successor protocol *Internet Key Exchange Version 2* (*IKEv2*) to remedy these problems. Unfortunately, IKEv2 implementations aren't generally available yet.

21.1.5 References

IPsec is standardized in an entire cluster of RFCs; these are the most important ones:

RFC 4301 [81] defines the fundamental IPsec architecture, RFC 4302 [78] the authentication header and RFC 4303 [79] the encapsulating security payload. RFC 3280 [68] specifies a X.509 public key infrastructure for the Internet in general, RFC 2409 [57] IKEv1 and RFC 4306 [31] IKEv2.

A number of additional RFCs specify the cryptographic algorithms that IPsec supports; they are referenced in the main RFCs.

21.2 Open Problems

Before we consider setting up IPsec in an environment we should take a closer look at its security implications. Some of these are direct effects of the fact that IPsec operates within the network layer; others are consequences of the IPsec standards and their implementations; yet others are related to authentication and encryption in general.

In short, IPsec solves a limited range of security concerns at a very high price. It is a long way from the catch-all security solution that some people claim it to be.

21.2.1 Inherent Limitations

The specifications for IPsec are excessively long. The most relevant and current RFCs exceed 450 pages (see section 21.1.5), not including the now outdated IKEv1. The consequences are notorious interoperability problems and a high chance of faulty implementations that might be exploitable.

IPsec operates within the network layer, so it only provides authentication and encryption between nodes. It is *not* a substitute for SSL/TLS, or SSH, or other application-layer mechanisms that provide cryptographic protection for individual users or processes. Neither is it a full substitute for missing encryption and authentication at the application level—telnet over IPsec is still inferior to SSH because it doesn't authenticate a user by a public key. But then, SSL/TLS and such don't provide proper protection between nodes: Running neighbor discovery through SSL/TLS won't work, because first SSL/TLS needs neighbor discovery to have finished before it can set up its TCP connection.

SSL/TLS provides authentication based on DNS names while IPsec authenticates IP addresses only. Since few people bother to remember the IP address of `online-banking.example.com` and even fewer will bother to track the occasional renumberings, authentication should really be based on DNS names. IPsec can't offer this: while it *may* be possible to do a DNS lookup on a DNS name stored in an X.509 certificate, doing so relies on the security of the DNS system; even though attempts have been made to retrofit DNS with up-to-date security features, these can't compete with the way that SSL/TLS authenticates the DNS name directly.

Originally, IPsec only supported unicast packets. Multicast groups were considered "publicly accessible" and therefore couldn't be encrypted. Authentication support was also missing; in theory it is possible to sign every packet using public key cryptography, but with today's public-key algorithms doing so is prohibitively CPU-expensive. Since IPv6 uses multicasts for a number of internal purposes, from neighbor solicitations and router discovery to various dynamic routing protocols, all of these are still unprotected even when IPsec is in use. RFC 3740 [56] and the latest IPsec RFC suite add multicast support to the IPsec framework but still don't specify any cryptographic algorithms that could actually be used for multicast authentication or encryption.

According to RFC 2406 [80], IPsec implementations were only required to provide the "NULL" encryption, which doesn't actually encrypt the payload at all, and *DES* (*data encryption standard*) in *CBC* (*cipher block chaining*) mode, which has been broken using brute force as far back as 1998. Cryptographically strong algorithms were optional. Microsoft's Windows XP doesn't even implement DES, so its IPsec implementation not only violates RFC 2406 but provides no encryption at all. RFC 4305 [29] recently changed these requirements but hasn't been widely implemented yet.

Surprisingly enough, encryption can actually reduce security. If we allow end-to-end encryption between machines inside our network and others on the outside, people can send data in and out without our control: Business secrets, malware or whatever. With IPsec, a packet filter can't even determine the transport layer protocol and port number used; it just notices some kind of traffic from workstation X to `dns0.example.net` but can't tell if it is genuine DNS traffic or anything else.

Finally, encryption and authentication tend to provide people with an exaggerated feeling of security. The huge increase of phishing attacks using HTTPS with invalid certificates demonstrates that a large fraction of people believe online banking is safe "because it is encrypted and everything".

21.2.2 Implementation Issues

Beyond the fundamental concerns we've addressed in the previous section, real-world implementations suffer from a number of additional problems.

Most implementations provide some strong encryption algorithms, but in a heterogeneous environment it may happen that no universally supported algorithm exists, even without Microsoft Windows XP boxes that only support "NULL" encryption.

Debian Sarge exhibits some erratic behaviour concerning IPv6 entries in the security policy database: Requiring encryption for UDP packets blocks neighbor discovery.

Setting up a certificate authority requires OpenSSL 0.9.8 or later; older versions don't support IPv6 addresses in the `altSubjectName` field.

IKEv2 support is mostly missing; so far only FreeBSD 6.1 offers an implementation `racoon2`, which comes without proper documentation. Using

signed X.509 certificates is apparently possible, but not documented. Debian Sarge and Solaris 10 don't support IKEv2 at all.

Virtually all implementations focus on setting up a VPN using tunnel mode. Using IPsec throughout a local network based on signed X.509 certificates is tedious at best.

Solaris 10 ignores the IP address stored in an X.509 certificate; if a node holds a certificate signed by a trusted CA, then it can use this certificate with arbitrary addresses.

21.3 Packet Filter Considerations

Setting up a packet filter for IPsec is fairly simple: We need to let IKE traffic at port 500/UDP through. In some cases we can filter by AH and ESP headers:

Debian Sarge *The* `ip6tables` *packet filter supports a matching extension* `esp` *that lets us filter by SPIs.*

FreeBSD 6.1 *There is no documented feature in* `pf` *to filter by IPsec headers, but apparently the* `proto` *keyword can be used.* 134

If we want to ensure that packets are authenticated, we may need to do so on the destination node using an appropriate policy.

If we use encryption across the packet filter, then we can't filter by port numbers or even protocols anymore. Neither can we filter by packet contents using "deep inspection" or application level gateways.

22

Mobile IPv6 (MIPv6)

The TCP/IP stack was originally developed when computers were several tons too heavy to be carried around. Since they were immobile it was only reasonable to assume that their network connectivity would be static. With todays smart phones, PDAs and similar devices, network connectivity changes very often. *Mobile IPv6* (*MIPv6*) offers a mechanism that allows a mobile node to change its network connectivity while it still keeps its IP address.

Implementations are currently experimental to non-existent and not part of standard distributions. A wide range of security problems are still unresolved. For these reasons we only take a look at the basic concepts and the problems that they raise.

22.1 Concepts

Mobile IPv6 is fairly complex and introduces an extensive set of new terms. So we start with a look at the core idea and the features it offers, then learn about some common optimizations and finally introduce three major functional extensions.

22.1.1 Basic Mobile IPv6

With mobile IPv6, a special *mobile node* (*MN*) may temporarily leave its *home network* (*HN*) or *home link* and connect to the Internet at a different point, possibly using a different link-layer technology than in its home network. While abroad, the mobile node still keeps its IP address from the home network, called the *home address* (*HoA*), so even TCP connections will "survive" a change in network connectivity.

Consider an example: Before I leave my office to do a training class, my notebook is connecting to my LAN, which is its home network. When I leave, I disconnect it and it automatically switches to its GSM/GPRS cell phone

interface. At the training center it switches to the WLAN available there. On return it switches back to GSM/GPRS and then back to my office LAN. During the entire time it keeps its regular address from the home network and its IMAP connection to the mail server at my ISP stays up without interruption.

So how does this work? Figure 22.1 shows that the key trick is another node in the home network called the *home agent* (*HA*). It behaves much like a secretary in an office who is always around and always knows where people are and how to reach them. When a mobile node changes its connectivity, then it immediately notifies the home agent of the address through which it is reachable. If this address is not from the home network, then it is called a *care-of address* (*CoA*) and the home agent starts to intercept all traffic to the home address and forwards it through a tunnel to the care-of address. If the mobile node returns to its home network and notifies the home agent that it is reachable at its home address, then the home agent stops intercepting and forwarding the traffic for the mobile node.

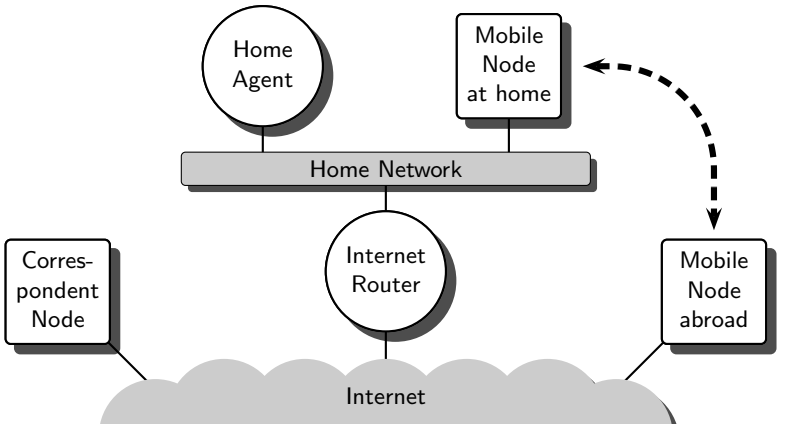

Fig. 22.1. A mobile IPv6 environment

So even when a mobile node is abroad it can still use its home address. A *correspondent node* (*CN*), like the IMAP server at my ISP, can still send packets to my home address; the home agent will forward them to my notebook. The notebook will also send packets to the correspondent node using its home address as the source address. Mobile IPv6 tunnels them to the home agent which then sends them untunneled to the correspondent node. If my network connectivity changes, the correspondent node won't notice because the home agent takes care of tracking my care-of address and forwarding packets there.

22.1.2 Telling the Home Agent: Binding Updates

Some care has been taken to secure the communication between the mobile node and its home agent.

When a mobile node moves to another care-of address, then it sends the home agent a *binding update*; a *binding* is an association of (home address, care-of address) that a home agent maintains for all its mobile nodes. These binding updates are obviously very sensitive; they allow an attacker to divert all traffic to the mobile node if they can be faked.

To protect them, the standards require that binding updates are encrypted using IPsec ESP. We'll see in section 22.2.2 that the need to keep the binding updates not only authenticated but also encrypted is most unfortunate, but at this point we may assume that simple-minded attacks based on forged binding updates are infeasible.

22.1.3 Bidirectional Tunneling and Route Optimization

Up to this point mobile IPv6 doesn't offer anything that a reasonably clever OpenVPN setup or a tunnel broker couldn't provide either. But mobile IPv6 offers a feature that may significantly reduce latency if the correspondent node supports it, too.

So far we have assumed that all traffic between the correspondent node and mobile node passes through the home agent. This is called *bidirectional tunneling*. The tunnel between the home agent and mobile node doesn't use 6in6 encapsulation, but a *type 2 routing header*. This is a special routing header, the IPv6 equivalent of loose source routing, that only supports a single intermediate address. In IPv4 terms, mobile IPv6 uses loose source routing between the home agent and the mobile node.

If the correspondent node is willing to support mobile IPv6, then routing can be improved by sending the traffic as figure 22.2 shows. The solid line denotes the untunneled traffic between the correspondent node and the home agent and the hollow line the bidirectional tunnel between the home agent and mobile node. If correspondent node and mobile node are reasonably close together but the home agent is far away latency-wise, then we can reduce latency drastically by sending the traffic directly between the correspondent and mobile node. This concept, called *route optimization* (*RO*), is shown in the diagram as a dashed line.

Route optimization requires the mobile node to send binding updates not only to the home agent but also to the correspondent node. The correspondent node has to implement part of the home agent functionality to keep track of the bindings with mobile nodes.

Since correspondent and mobile node don't necessarily "know" each other in the sense that a trustworthy IPsec security association exists between them,

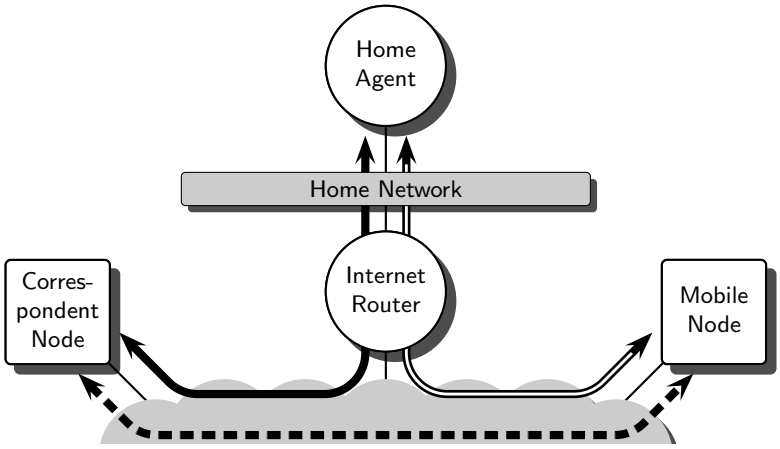

Fig. 22.2. Bidirectional tunneling vs. route optimization

sending binding updates to correspondent nodes introduces a number of security risks. To detect bogus binding updates at least in simple cases, a protocol called *return routability test* lets a correspondent node check if the binding update it receives is valid.

This optimization is a feature that we can't possibly implement with OpenVPN or a tunnel broker. Near-realtime applications like IP telephony are sufficiently sensitive to latency to warrant the use of route optimization in certain situations.

22.1.4 Network Mobility (NEMO)

So far we talked about individual mobile nodes, which should really be called "mobile hosts". The *network mobility* (*NEMO*) extension to mobile IPv6 features mobile networks which behave similarly to a single mobile node.

The most often heard scenarios for network mobility involve expensive cars with "advanced on-board telemetric, information and entertainment functionality" and so-called "intelligent soldiers" equipped with network-based communications equipment and weapons. Both scenarios demand a high level of security, so it remains to be seen if mobile IPv6 is applicable for these purposes.

It may seem reasonably straightforward to extend mobile IPv6 to support entire mobile networks, but it requires some radical changes on the home agent, which must now proxy for an entire network prefix or set of network prefixes. The mobile node becomes a *mobile router* through which the entire mobile network connects to its home agent. Communication between the mobile router and home agent uses a 6in6 tunnel rather than the type 2

routing header. Finally, mobile networks can't use route optimization; they must always use a bidirectional tunnel.

22.1.5 Fast Handovers

When a mobile node moves and changes its network connectivity, then it isn't reachable for some time: It must establish its new care-of address, send binding updates to the home agent and its correspondent nodes and do the return routability test with the correspondent nodes. This delay disrupts near-realtime services like voice over IP.

When a mobile node obtains a new care-of address while its old care-of address is still functional, then the *fast handover* extension ensures a seamless migration to the new care-of address by using the old address for the binding updates.

In typical scenarios a mobile node will switch between different link-layer technologies, for example between Ethernet, WLAN and GSM/GPRS. If the switch uses the same link-layer interface, for example when it uses a single WLAN interface to switch between different WLAN access points, then fast handovers are infeasible.

22.1.6 Hierarchical Mobile IPv6

If fast handovers don't apply, for example because we just want to use mobile IPv6 between adjacent WLAN networks, then another extension to mobile IPv6 becomes interesting: *Hierarchical mobile IPv6* (*HMIPv6*) avoids sending binding updates to far away home agents and correspondent nodes.

To do so a *mobile anchor point* (*MAP*), which is located in close proximity to the WLAN networks in our example, tracks the mobile node as long as it moves between the subnets that the mobile anchor point handles. The mobile node sends binding updates to the mobile anchor point only, which then ensures that all traffic is routed to the mobile node in whichever subnet it currently connects to.

This extension is probably very useful especially when large WLAN environments need to support roaming. WLAN offers some roaming functionality at the link layer, but this doesn't scale arbitrarily. Mobile IPv6 scales much better, but if fast handovers can't be used, then every move between subnets causes a short loss of connectivity. Hierarchical mobile IPv6 mitigates this problem at least in certain scenarios.

22.2 Open Problems

At this time, mobile IPv6 is unfit for practical use for two major reasons: Implementations are scarce and incomplete and a variety of fundamental security concerns are entirely unresolved.

22.2.1 Available Implementations

At the time of this writing, mobile IPv6 support is barely available.

Debian Sarge *An experimental implementation for Linux is available from the "mobile IPv6 for Linux" (MIPL) project, but is based on drafts of the standards and doesn't use IPsec to protect the communication between the home agent and mobile node. It isn't available with Debian but must be added manually and requires a patched kernel. For details, check the MIPL web site at* `http://www.mobile-ipv6.org/`.

FreeBSD 6.1 *The KAME project has implemented mobile IPv6, again based on drafts of the standards. The results were not incorporated into mainstream FreeBSD, but the sources are available from the KAME project;* `http://www.kame.net/newsletter/20031007/` *has the details.*

Solaris 10 *There is no support for mobile IPv6 available.* 135

In other words, mobile IPv6 support is still a long way from a production grade feature at this time.

22.2.2 Unanswered Security Questions

It should be obvious by now that mobile IPv6 raises a number of security concerns that scare any security-conscious network administrator. It is reasonable to take a look at these concerns even though mobile IPv6 isn't really available yet.

We won't even attempt to do an exhaustive security evaluation of the mobile IPv6 design. To demonstrate the risks that it introduces we focus on a single scenario: A company network with some notebooks that need access to the company network as well as the Internet while abroad. Today these scenarios are usually implemented with some sort of more ore less standardized VPN software, in Unix environments often with OpenVPN.

Like all VPN solutions, mobile IPv6 exposes mobile nodes to potentially hostile environments. Host security on the mobile node must be as strong as the company firewall or the mobile node may be attacked while abroad to obtain access to the company network. Commonly used VPN packages often offer a feature to disable all direct network access while the VPN is established. It remains to be seen if mobile IPv6 implementations will eventually offer such a feature.

Another problem with mobile IPv6 concerns privacy: If the mobile node supports route optimization, then an attacker only needs to ping the node at its home address to track down its network connectivity. It doesn't take much to ping the notebook of a competitor's CEO and watch the binding updates we receive; there is nothing illegal about it and it offers some exciting business intelligence. If the CEO wants to use voice over IP, then route optimization

becomes almost inevitable. At the same time we can't configure a stateful packet filter within the company firewall to allow only outbound traffic for the home addresses of mobile nodes, because that will make voice over IP difficult to impossible.

Yet another major problem are packet filters and firewalls in general. Today's packet filters don't provide the functionality to deal with mobile IPv6 to protect the home network. An entire range of problems exist: Packet filters don't filter by routing header and by the addresses that the routing headers contain, so they can't filter by home and care-of addresses. Since the binding updates between home agent and mobile node require IPsec encryption we have to let all ESP packets through to the home agent; in consequence we must let these ESP packets through without filtering. While there is a theoretical possibility to store all IPsec keys on a packet filter, doing so doesn't really help because it makes the packet filter vulnerable to denial of service attacks that use CPU-intensive cryptography with a minimum of network traffic.

The routing headers are a problem with respect to at least one more aspect: Loose source routing in the IPv4 world has been blocked on virtually all packet filters because it offers a huge variety of potential attack vectors. Route optimization needs it, and therefore opens the systems involved to these attacks.

Correspondent nodes can also be victims of an attack based on route optimization: Many access controls are based on IP addresses and network prefixes. If a correspondent node supports route optimization and offers a service only accessible to an address A, then an attacker can send a binding update to the correspondent node with its actual address as the care-of address and A as the home address. Afterwards it can access the service even though it doesn't really have the authorization to do so. What's worse, log files on the correspondent node will show that the service has been accessed from A. If A is actually a valid address, then somebody or some node will be blamed for accessing the service even though they never did.

These considerations are by no means complete. But they should show that mobile IPv6 is an extremely risky technology that shouldn't be deployed without pressing need and extreme care. As of today it is probably advisable to use OpenVPN or another VPN product that doesn't even attempt to offer route optimization.

22.3 Further Reading

Mobile IPv6 is defined in a fairly large number of RFCs. These are the most important ones.

RFC 3753 [33] contains a glossary of mobility-related terminology and concepts for both IPv4 and IPv6. RFC 3775 [75] contains the core mobile

IPv6 specification and RFC 3776 [3] covers in detail the IPsec communication between mobile node and home agent.

RFC 3963 [25] defines network mobility, RFC 4068 [34] fast handovers and RFC 4140 [102] hierarchical mobile IPv6.

RFC 4225 [93] explains the security aspects of the route optimization design. Finally, RFC 4487 [83] analyzes in detail which problems arise with firewalls when they must support mobile IPv6.

23

Quality of Service (QoS)

For a long time the telephony world considered the TCP/IP stack inferior to the ISO/OSI stack because its best effort strategy on packet forwarding was considered unfit for near-realtime services like telephony. The IETF attempted to add quality of service features to IPv6 to make IPv6 "near-realtime capable".

Since then, H.323, the session initiation protocol (SIP) and proprietary protocols like Skype have proven that even IPv4 without quality of service features is "good enough" for telephony services. Still the quality of service (QoS) features are at least specified, so we take a look at them even though implementations are generally not available.

23.1 Concepts

Traditional telephony services allocate fixed bandwidth slots between any two routers on the communications path between two end devices. This guarantees that the bandwidth the communications need are always available—and introduces a number of problems that make telephony networks so complex and expensive. One approach, called integrated services, tries to introduce a similar functionality into IPv6.

An alternative approach, called differentiated services, offers a different strategy. Many routers today offer some sort of *traffic shaping* that employs multiple queues to replace the "first in, first out" principle of traditional packet forwarding with a reordering feature that ensures that "critical" packets are forwarded with a higher priority than "less critical" ones. In a way, differentiated services tries to standardize the various traffic shaping frameworks used in many IP implementations.

Neither approach extends routing in a way that "low latency" packets are routed differently than "high bandwidth" packets. Such an approach would require both a new kind of routing table concept and an entire set of new dynamic routing protocols.

23.1.1 Integrated Services (IntServ)

The base IPv6 header contains a 20 bit field called the *flow label*. Together with the source and destination addresses in the header it identifies a *flow*, a unidirectional connection from a single source address to a single destination address. By default the flow label is set to zero to indicate that a packet doesn't belong to a specific flow. But applications may set the flow label to a different value and *integrated services* (*IntServ*) lets them request a fixed, guaranteed amount of bandwidth from all the routers along the path from the source to the destination using the *resource reservation protocol* (*RSVP*).

Every router supporting integrated services must keep track of the flows allocated to ensure that it doesn't guarantee more bandwidth than it has actually available. It must also check for every flow that it doesn't exceed the allocated amount of bandwidth.

This puts an enormous workload on large routers. On a single 2.4 Gbit/s link on a router we could allocate almost 40 000 flows at 64 kbit/s. If we used an average link-layer frame size of 100 bytes, then we could send more than 3 million packets per second through that interface. That is one approximately every 300 nanoseconds, or every 1300 CPU cycles on a 4 GHz CPU.

To reduce the number of flows, RFC 3175 [4] added a feature to integrated services that offers *aggregated flows*. This may reduce the number of flows that a link has to manage, but still the sheer number of packets passing through makes resource reservation extremely resource intensive.

Integrated services only work if all routers along the path from source to destination support it. Then it offers guaranteed bandwidth and predictable latency. If only one router doesn't support IntServ, then neither bandwidth nor latency are guaranteed anymore.

To my knowledge no usable implementations of integrated services are available yet.

23.1.2 Differentiated Services (DiffServ)

A far less ambitious approach that fits much better into the TCP/IP stack design is called *differentiated services* (*DiffServ*). It assumes that some routers may use traffic shaping features to re-order packets queued to an outbound interface. Individual packets may be tagged using the *traffic class* (*TC*) field in the IPv6 base header to notify routers about their queueing requirements.

Routers along the way may rewrite the traffic class field to adapt the packet to local queueing policies. This makes differentiated services exciting: Even different implementations from different vendors with different traffic shaping capabilities can be used in a consistent way if differentiated services form the base for the traffic shaping.

Routers that don't actively support differentiated services inherently implement a single "first in, first out" queueing policy. High bandwidth routers like those mentioned in the previous section are rarely dropping packets due

to overload and the *jitter*, the variance of delay, is usually very small, so there is simply no need to implement elaborate traffic shaping features on them.

Many of the possible values for the traffic class field are available for local use, so it is perfectly reasonable to define local policies and implement them using DiffServ. Border routers to the outside may rewrite the traffic class field to values that the bordering networks understand.

Unfortunately, until now no basic set of standardized traffic classes has been specified. So at this point differentiated services aren't really applicable at an Internet-wide scope. There are no usable implementations available either.

23.2 Is It Necessary?

IP telephony over plain IPv4 without quality of service features works reasonably well. So do these quality of service features make any sense at all?

23.2.1 Technical Considerations

From a technical perspective there is a fairly simple answer to that question: Integrated services is extremely complex, doesn't scale, needs excessive CPU time on large routers and requires support on all routers along the connection path; it will be useful only in very special cases.

Differentiated services is simple, scales reasonably well and doesn't need support on all routers along the connection path. It simplifies traffic shaping within an administrative domain by offering a vendor-independent standardized mechanism. Throughout the Internet it can be useful even between different administrative domains as soon as standard traffic classes are specified.

But do we actually need it? Figure 23.1 shows a common scenario with two company LANs connected to the Internet with low-bandwidth WAN lines and a public server in the Internet. Where do we need traffic shaping? In the LANs we may need it in some cases, for example if we run a network-based backup and IP telephony across the same subnets. With respect to the Internet however, the links with the lowest bandwidth are the most critical, because they tend to be either idle or saturated. In the Internet and towards public servers the lines are statistically hardly ever saturated thanks to the law of big numbers; the most critical links in the example are the ADSL links, and since they usually have asymmetric bandwidths, the direction upstream from the LAN to the Internet most desperately needs traffic shaping. We can achieve this on the WAN routers without excessive problems either using differentiated services or a non-standard traffic shaping implementation.

In summary, differentiated services may improve and simplify traffic shaping quite significantly as soon as it becomes available.

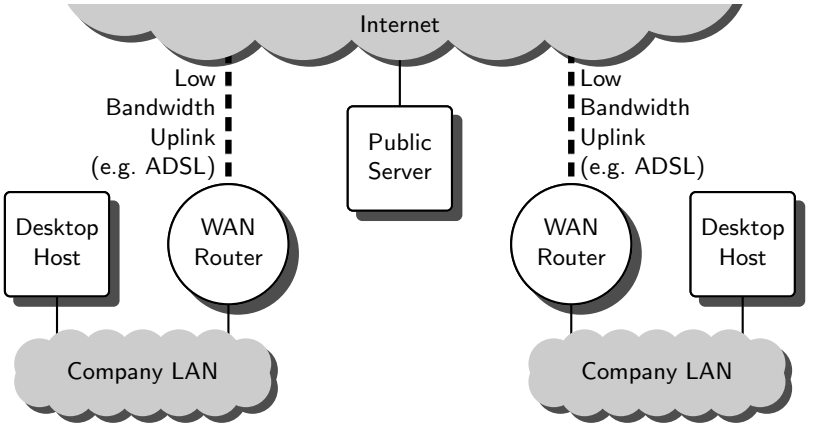

Fig. 23.1. A typical QoS scenario

23.2.2 Political and Economic Aspects

Recent discussions about the large Internet carriers trying to charge content providers for the traffic they generate show that quality of service is very important to the carriers, which are generally telcos as well. There is a strong non-technical element to quality of service that we should understand before we even try to use it.

The telcos lose more and more of their telephony business to IP telephony, so their associated Internet carriers would like it best to sell expensive high-bandwidth Internet connectivity that still doesn't support telephony. Alternatively they want to charge "premium rates" for "critical traffic".

But the same performance problems that prevent the implementation of integrated services on core routers will make it quite challenging to implement "premium services". As a personal opinion I suspect that the carriers will try to degrade performance artificially for traffic that they consider detrimental to their telephony business rather than improving quality for better-paying customers; this can be done on any single router, rather than all routers along the path. Fortunately, switching ISPs isn't particularly difficult with IPv6, as we'll see in section 24.4.

23.2.3 Common Misunderstandings

Possibly due to politically motivated misinformation there are at least two common misunderstandings with respect to quality of service that deserve to be resolved.

With respect to integrated services there are frequent rumors that the resource reservation protocol can trivially be used for a denial of service attack by simply allocating all available bandwidth on a router.

Of course, if the router is properly configured, then it will simply deny the request. Additionally, RSVP allocates the bandwidth on all routers along the path. So the attacker can only reserve as much bandwidth as he has on his connection. If you consider figure 23.1 again and assume that the attacker is using one of the desktop hosts, then he can only allocate the bandwidth that the ADSL link offers—and in doing so he will block his own line rather than anything else. In a company network nodes that do this can be easily traced.

This doesn't preclude the possibility that a more elaborate attack might be successful, but at least the trivial scenario often described won't work.

Differentiated services are often suspected to be useless "because everybody will tag everything as 'premium' and then we have the same effect as without quality of service".

As we have seen above, differentiated services are primarily relevant within a local network and on uplinks towards an ISP. So again the culprit is easily traced. Additionally, differentiated services allow routers to change the traffic class of packets passing through. Strategically placed routers that sanitize the traffic class fields will deal with this problem up to the point that users that abuse the traffic class field only affect their own network performance.

23.3 Further Reading

The RFCs most relevant to integrated services and the resource reservation protocol are RFC 2205 [30] and RFC 2210 [112]; additionally, RFC 3175 [4] addresses the aggregation of multiple flows.

RFC 2474 [92], RFC 2475 [11] and RFC 3260 [48] cover differentiated services.

Part VI

Architectural and Operational Topics

24

Renumbering Procedures

Different than IPv4, IPv6 lets us assign multiple addresses to a single interface at the same time. Together with stateless autoconfiguration this makes it feasible to change IPv6 addresses in a network on the fly without any downtime.

In theory this sounds quite simple and straightforward. In practice however, a renumbering event requires a little planning and effort. This chapter points out a strategy and the basic procedures necessary to make the transition from one global routing prefix to another run smoothly and without disruption of service. We assume that we want to replace a global routing prefix with another, keeping the subnet and interface IDs unchanged.

24.1 Preparations

Before we start with the actual renumbering procedure we need to prepare the network and ourselves for the renumbering. These preparations can be done long before the actual renumbering, but double-checking them immediately before an imminent renumbering event is an excellent idea.

We should consider setting up a reasonably sophisticated network monitoring system that automatically checks if everything works as expected.

Then we should make sure that we have a fallback access path to all network components. So we should install site-local addresses, usually from one of the unique-local address pools, to our network if we haven't done so from the beginning. Section 24.2.1 explains in general how to deploy a new address, so we don't go into the details here. We should make sure that we can reach all nodes in our network using these addresses; this includes appropriate DNS entries, possibly in a dedicated subdomain like `fallback.example.com`.

Next we deal with the addresses and address prefixes occasionally found hard-coded in various configuration files. This includes statically configured addresses in some nodes, like routers and DNS servers, in resolver, packet filter and access control list (ACL) configurations. We must take care not to miss

any of these configurations during the actual renumbering, so it pays to `grep` all configuration files on all nodes for hard-coded addresses and prefixes. In a perfect world all these would be replaced by DNS entries, but the DNS offers neither a commonly used way to store prefix lengths nor a degree of security fit for use in packet filter or ACL configurations. In an almost perfect world we would maintain a correct and complete list of all hard-coded addresses and prefixes in our network. In practice such a list is often handy but we can't possibly rely on it alone, so `grep` helps to find some items missing on the list.

We need to decide how we want to do the renumbering. Normally we do a *soft renumbering* with a grace period during which we may use both old and new addresses in parallel. In some emergency situations this may not be feasible, so a *hard renumbering* without any grace period may be necessary. A hard renumbering will cause temporary loss of service, so we should avoid it. But if we have reason to expect such an event, then we can prepare for it to keep the downtime as short as possible.

In any case we should have a reasonably detailed plan for the renumbering event. The following sections give a generic outline, but they are no substitute for a plan that takes local peculiarities into consideration. As usual, the plan consists of many small steps that can be checked for success and if necessary allow for a quick rollback.

24.2 Soft Renumberings with a Grace Period

The preferred network renumbering procedure uses both old and new prefixes during a limited transition period. As the first major step we establish the new prefix. After that we expire and remove the old prefix.

RFC 4192 [5] suggests a procedure similar to the one presented here. We consider a few additional details that were beyond the scope of the RFC and ignore others, like DHCP-based address management, that we consider too far off mainstream IPv6 usage.

24.2.1 Deploying a New Prefix

Obviously we need a new prefix to install. Together with the new prefix we need a DNS delegation for the reverse zone. We can set up the reverse zone in advance, since it doesn't affect regular operations. We also need to set up our outside connectivity to support the new prefix, so we must ensure that the new prefix is routed to our network. Additionally we need to check with our ISPs that outbound traffic doesn't get caught in ingress filters as of RFC 2827 [42] and RFC 3704 [6], especially if we run a multi-homed site as in chapter 25.1.

- ☐ From our border router(s) network connectivity with the new prefix must work without problems.

Now we should notify the users about the impending changes and ensure that all relevant system and network administrators are within reach.

Next we install the new prefix on all our routers, configuring their interfaces, routing tables and possibly their packet filter rules. The new prefix must *not* yet be distributed using autoconfiguration, though. In parallel we can update all statically configured addresses, access control lists and other configurations on all nodes to contain both the old and new prefix. In some cases it may be necessary to restart various daemons so they re-read their configuration, which may cause the occasional service interruption if the daemons are badly written. If possible, do this off business hours and ask all users to check if everything works when they return to their offices.

- ☐ Ensure that all services are still available. If you have a proper monitoring setup installed, double check it to ensure that everything works fine.

Now we can distribute the new prefix to all hosts using autoconfiguration. At this point we don't really want to use the new addresses yet, so we set the preferred lifetime to 0. To allow for a fast rollback we also set the valid lifetime to a low value, like three times the maximum router advertisement interval. This is $3 \times 600 = 1800$ seconds or 30 minutes by default, so it is likely a good idea to lower the maximum advertisement interval first, maybe down to 60 seconds—then we can invalidate the addresses again within three minutes.

- ☐ All hosts must show an address with the new prefix.

Next we lower the TTL in our DNS zones to a few minutes. This isn't strictly necessary, but it helps if we need to roll back the impending changes to the DNS. Then we add the new addresses to the DNS. A little bit of shell scripting power is quite useful with the forward zones; the reverse zones can be copied if we use the `$ORIGIN` statement judiciously.

- ☐ Querying the name server using `dig` must show that all forward and reverse zones show correct data.
- ☐ Doing the same with tools like `ping` and `traceroute` reveals problems related to the resolver rather than the DNS itself.

At this point even services that use the DNS for "authentication" will be able to use the new addresses and we are ready to use them. So we configure the router advertisement daemons with a small but non-zero preferred lifetime for the new prefix. In reasonably large environments we do this subnet by subnet so we can monitor the progress and fix problems quickly. If we use a tool like `nsautoupdate` via `cron` jobs, then we must take particular care to bring up hosts in small batches to avoid overloading the DNS server.

When all subnets use the new prefix, we have successfully deployed the new prefix. Only testing and a few cleanup tasks are left.

- ☐ Check that all services work as expected.
- ☐ If feasible, consider scheduled reboots for all servers to ensure that they come up with a working configuration again.

If we obtained the new prefix through a temporary tunnel, then we switch to the final uplink. Section 24.4 takes a closer look at switching providers and the associated procedures.

As final cleanup tasks we configure all advertising routers to advertise standard valid and preferred lifetime settings and all name servers to use the standard TTL value.

24.2.2 Revoking an Old Prefix

With the new prefix up and running we can now revoke the old prefix from our network. In many aspects we undo the steps in the previous section, but again it pays off to consider the procedure in detail before executing it.

As the very first step we set the preferred and valid lifetimes of the old prefix on advertising routers so that hosts shouldn't use them anymore: We set the preferred lifetime to 0 to deprecate the old addresses and the valid lifetime to 7200 seconds to make it possible to invalidate the old address within two hours; remember that hosts will ignore smaller values for the valid lifetime to avoid denial of service attacks using bogus router advertisements.

- ☐ Check that all services work correctly.

Next we drop the old addresses from the forward DNS zones. This reduces the traffic using the old addresses quite significantly.

- ☐ Again, check that all services work correctly.

Now we remove the old prefix from all configuration except routers and packet filters, restarting services where necessary. This may again cause some minor disruptions with badly written daemons and should be done off business hours if possible.

- ☐ Yet again, check that all services work correctly.

Use a packet sniffer to watch for traffic still using the old addresses. There is bound to be some traffic using existing connections for a long time and it may be necessary to restart them manually so they switch to the new prefix.

When no more traffic uses the old addresses, we remove them from the router advertisements. Two hours later the addresses have expired on all nodes and we can remove the prefix from the router interfaces, routing tables and packet filter configurations. We can also drop the DNS reverse zone associated with the prefix.

- ☐ For a last time, check that all services work correctly.
- ☐ Additionally, check all error logs for hints of problems.

Now we are done except for some minor cleaning up. If our ISPs do ingress filtering, then we notify them that we don't use the prefix anymore; this is particularly important with multi-homed sites. Finally we notify our users that the migration is done.

24.3 Emergency Renumberings

If we have to renumber our network without a grace period, then we must follow a different approach. A *hard renumbering* is a renumbering event where we lose the old address before we obtain a new one. It implies a temporary loss of service and is generally both undesirable as well as avoidable. But depending on the network peculiarities it may be necessary to prepare for a hard renumbering as a contingency plan. In other words: Don't do this if you can possibly avoid it. A hard renumbering should only be necessary if we unexpectedly lose our global routing prefix without due warning.

When a hard renumbering event occurs, we need to do several things in parallel. First we must sever our Internet connectivity so we can still use the prefix internally. Then we must remove the old prefix, possibly after deploying a unique-local prefix if we haven't done so yet; we can follow the same procedures as in section 24.2. At the same time we must obtain a new prefix somewhere, possibly together with a new uplink. Then we install the new prefix. As soon as we have the old prefix removed and the new installed we can re-connect to the Internet.

Since this is an emergency that should never really happen, we are under more or less severe time pressure. We have to cut some corners, do only the most rudimentary intermediate checks and just plan for a lengthy clean-up period when we deal with the mistakes and omissions we made. Section 25.1 explains how to set up redundant network connectivity to make this most unfortunate situation as unlikely as possible.

24.4 Changing the Internet Service Provider

A special case of renumbering that is worth some consideration is the situation that we need to switch from ISP A to ISP B. Again we want to do this with a minimal and predictable loss of service.

If we can convince management to pay for two uplinks during the transition period, then we can proceed as explained before with one exception: We need to route all outbound traffic to the new ISP and tell them about the prefix from the old provider so they can set up their ingress filtering accordingly.

The situation is more difficult if we only have a single uplink and switch that from one provider to another. This happens for example with ADSL lines

that are provided by a phone company but connected to the Internet through an ISP that only rents the ADSL frequency range from the phone company.

Assuming that our old ISP isn't particularly cooperative but the new one is, we can use a configured or 6in6 tunnel to the new ISP while the old ISP still operates our uplink. We need to route all outbound traffic through this tunnel in the case that the old ISP uses ingress filtering and drops our outbound packets if their source address is from the new prefix. Incoming traffic to the old prefix will arrive untunneled and traffic to the new prefix will be tunneled to us. This provides us with the necessary transition period to set up the new prefix. When we only use the new prefix and the tunnel as our uplink, then we can switch our uplink to the new ISP, deconfigure the tunnel and resume normal operations. As long as the old ISP doesn't delay the actual switch to the new uplink—and cases have been reported—we can switch to the new service provider with a scheduled, very short loss of external network connectivity.

If even this downtime is undesirable, then redundant uplinks through multiple ISPs are necessary at least for the transition period. The next chapter explains how set them up.

25

Multi-homing

So far we have assumed that end sites connect to the Internet through a single provider and hosts connect to a single subnet only. While this is a reasonable assumption in some cases, in others multiple connections are necessary.

In this chapter we take a look at the problems that we face when we set up a multi-homed network or host.

25.1 Multi-homed Networks

Any site that needs a reliable Internet access should have redundant uplinks. With IPv4 this is reasonably complex, requiring *provider-independent addresses* (*PI addresses*), an *autonomous system number* (*ASN*) and local BGP routers. With IPv6 we need to follow a radically different approach because provider-independent addresses are simply not available.

25.1.1 Life Without Provider-independent Addresses

The IPv4 approach has at least two major drawbacks. Every autonomous system and its address prefix must be tracked with BGP throughout the entire default free zone worldwide. According to BGP statistics available at `http://bgp.potaroo.net/`, in June 2006 the average core router had to maintain about 185 000 individual routes. For every packet that a core router forwards it has to search this routing table. The IPv6 routing tables hold less than 700 routes and since there are no PI addresses for IPv6, it is expected to stay reasonably small even when IPv6 becomes the mainstream network layer protocol.

The other problem is not entirely technical: Even *leaf autonomous systems* that don't route foreign traffic through need to run BGP, which is a bit of a challenge to the average local network administrator. Mistakes are both common and critical enough so that providers set up BGP filters to stop broken BGP data entering their BGP setup.

If provider-independent addresses are such a problem, then why do people use them? If you tell the average IPv6-unaware network administrator that there are no PI addresses available for IPv6, then you will quite likely hear statements like "Then we can't use it for production, because we need redundant network connectivity." But this misses the point: Technically it is perfectly feasible to get addresses from one ISP and then announce them with BGP through multiple ISPs; we might have to get our own autonomous system, but we don't need PI addresses for this.

The reason why IPv4 network administrators want their PI addresses is unrelated to multi-homing. If we ever need to switch a moderately large IPv4 network from one ISP to another, then we don't want to renumber the entire network.

IPv4 administrators tend to mix up these two problems because in the IPv4 world both are very closely related. Most BGP routers beyond the border routers on leaf autonomous systems tend to ignore BGP announcements if the prefix is too long; this filters out bad BGP announcements usually caused by faulty configurations of leaf site BGP routers and keeps the routing tables in the default free zone from growing even faster. So the only way to set up redundant Internet4 connectivity takes a reasonably short prefix, an autonomous system number and a BGP leaf configuration.

With IPv6 the situation looks entirely different. A renumbering event involves a bit of work, but the procedure is manageable and doesn't cause any significant downtime as we have seen in chapter 24. Additionally we can use the fact that IPv6 lets us assign multiple addresses to a single interface. This allows for a radically new approach to multi-homing.

The following sections explain this new approach. Since it is a bit more complex we take a look at two partial solutions that are reasonably easy to understand and then merge them into the final solution.

25.1.2 Redundant Links to a Single Provider

The easiest way to make our Internet connection redundant is quite simple and commonly used in small environments: We use multiple links to our provider, as figure 25.1 shows. The most simple case uses a fallback dial-up connection or similar.

These setups don't take much: A second line, routers and dynamic routing on either side of the WAN links. If we use a router appliance that can handle both the main and fallback line, then we can spare us both the additional router and the dynamic routing at the price of introducing a single point of failure. In larger environments we might use multiple active links and distribute the traffic across them, but right now we focus on a single active link and a fallback link because this gets us closer to the final solution.

The problem with such a setup is the single provider: If both WAN links belong to the same provider, then there is a non-negligible risk that this

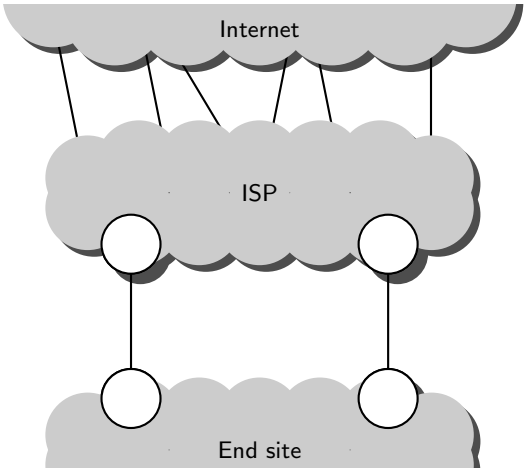

Fig. 25.1. Redundant links to a single ISP

provider will connect both lines to the same router, run the links at least partly through the same conduits, be disconnected by their upstream providers, misconfigure their routing due to serious staff incompetence or work overload or might be hit by sabotage like a remote denial-of-service attack. So we may not want to rely on a single provider.

25.1.3 Non-redundant Links to Multiple Providers

To work around these risks we have to use multiple providers. We start with a configuration that uses only a single WAN link per provider as figure 25.2 shows.

Despite the fact that the network diagram doesn't look too different from figure 25.1, the configuration here is noticeably more complex and deserves some attention.

Both ISPs will assign us a global routing prefix and we must use both throughout the site. If they implement ingress filtering as of RFC 2827 [42] and RFC 3704 [6] to block spoofed packets from our site, then we need to notify each ISP of the prefix that the other has assigned us. Otherwise, outbound traffic doesn't change: We just run dynamic routing within the end site to ensure that all outbound traffic is forwarded to a router with a working WAN link.

Inbound traffic to the prefix assigned by ISP A will always be routed from the Internet to ISP A and then across our link from ISP A; similarly, all traffic to the prefix we received from ISP B will always arrive through their link.

If we run a server that is accessible from the outside, like a web or SMTP server, then we need to add both its addresses to the DNS. A correctly written program will try whichever address it finds first and, if it fails, it will switch to

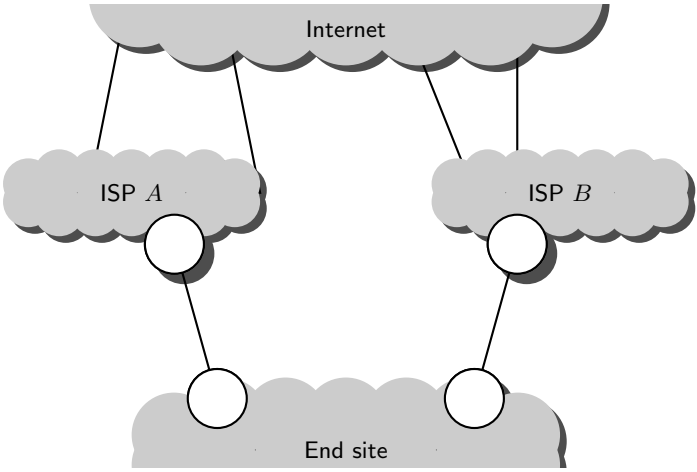

Fig. 25.2. Non-redundant links to a redundant ISPs

the second. This makes the server accessible even during a link failure—but at a price: The client may wait for a lengthy timeout before it tries the second address.

To make this setup more or less redundant we need to implement part of the failover ourselves. We need to monitor the availability of our links, and possibly even the ISP's connection to the Internet. When we detect a failure, for example with our link to ISP A, then we need to deprecate the prefix from ISP A in all our subnets by setting the preferred lifetime for that prefix to 0 on all advertising routers. Additionally we must temporarily remove all addresses with that prefix from the DNS.

Alternatively we might use BGP to announce a temporary route to our prefix from ISP A through our link to ISP B. If we only announce these routes while a link is down, then we don't clutter the routing tables in the default free zone more than necessary. We will however need our own autonomous system with BGP border routers. Additionally we must ensure that both ISPs will accept the temporary routes when we announce them.

No matter which strategy we use, they are reasonably complex to set up, a failover will take some time and existing connections may fail.

25.1.4 Redundant Internet Connectivity

If we combine redundant providers with redundant uplinks, then we can make our network connectivity redundant with a very quick failover if a link fails.

There is an obvious way to combine both strategies: Set up redundant links to multiple ISPs. Unfortunately, four uplinks tend to be slightly expensive, so we use a trick to keep the number of physical links down at two: Instead

of a physical fallback link to ISP A we set up a tunnel from ISP A through the physical link of ISP B to our side and another tunnel from ISP B through the physical link of ISP A. Figure 25.3 shows the setup with the two tunnels as the dotted and dashed lines, respectively.

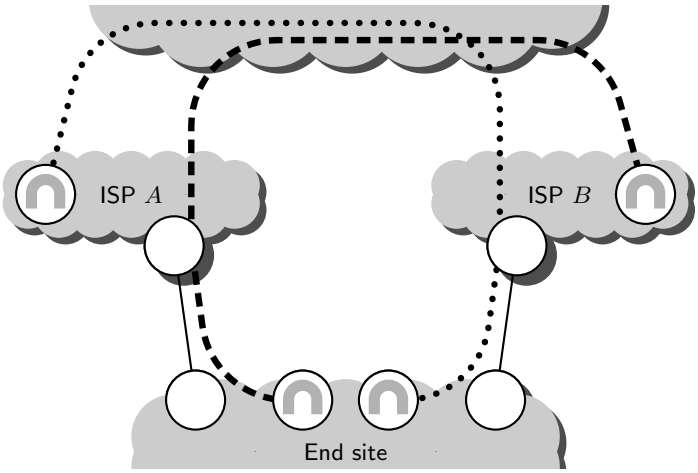

Fig. 25.3. A redundant connectivity architecture

The configuration here is rather complex, but after we have seen the preliminary steps it shouldn't be impossible to understand. We just need to set things up step by step.

First we set up those links to both ISPs and make our advertising routers within the site distribute the global routing prefixes we have received from them. Within our site we also need to run dynamic routing, so all traffic to the Internet is routed to one of these links.

At this point we run into a problem: Quagga doesn't support multiple OSPFv3 instances on a single router yet. Within our site we probably want to run OSPFv3 rather than RIPng to speed up convergence after a link failure; so do the ISPs. If we use Quagga-based routers on both ends of the WAN links, then we need to work around this limitation. I'd personally run RIPng across the WAN links, set the route announcement interval to a very short value (consciously violating the standards while doing so) and make sure that the ISP-side router announces only a default route and my own router announces a single aggregated route to the prefix assigned by the ISP.

At this point we should have Internet connectivity as long as both WAN links are up and running.

Next we configure the tunnels. The actual tunnels are just plain 6in6 tunnels, or possibly configured 6in4 tunnels. The tunnel routers on our side must have their default route configured to the WAN router that the tunnel

runs through. Alternatively we can run a dynamic routing protocol on the LAN interface of the tunnel routers, but then they must run it in passive mode and not announce any routes themselves. To make sure that the tunnel router at the ISP routes traffic back through the Internet and the other ISP, the remote address of their tunnel configuration must be the one with the prefix from the other ISP. The ISP's tunnel router must run a dynamic routing protocol on its LAN interface and announce a route through the tunnel to the prefix they assigned us with a metric worse than the direct WAN link.

Now we have a redundant Internet connectivity that will work seamlessly even if one of the WAN links fails.

There are two optimizations we may apply to save hardware: We can use a single tunnel endpoint at the end site; then we must use passive dynamic routing so the tunnel endpoint is capable of sending out traffic through whichever WAN link is still working. The ISPs will probably not use a dedicated tunnel router for every customer but rather configure many tunnels on a single router.

At this point we still run into problems if an entire ISP fails, possibly because they lose all Internet connectivity; that will bring down the tunnel, too. If we want to cope with such a situation, then we must still resort to the solutions from section 25.1.3 for this special situation.

This entire approach is complex, hard to understand especially when debugging a broken configuration, and difficult to set up correctly; I have personally heard a number of comments along the line "You don't seriously expect anyone to do this, do you?" But if I consider the alternative, running a BGP leaf site, then I personally prefer this setup.

25.2 Multi-homed Hosts

Another multi-homing topic relates to individual hosts that have multiple network interfaces. This turns out to be troublesome because multi-homed hosts don't fit the router–host scheme.

Consider a firewall configuration with two separate packet filters and an application gateway between them. While these setups are slightly more expensive than using a single packet filter, some highly security-sensitive environments really use a setup as in figure 25.4. The packet filters serve several major purposes: They filter bad packets at the network and transport layer, prevent link-layer based attacks and control access to the application level gateway according to the addresses in the IP packets and transport layer port numbers. The application level gateway inspects the contents of the packets at the application layer; typical examples are spam and virus filters for e-mail or web proxies that enforce a security policy like "no ActiveX" to all nodes in the protected network. It doesn't do any forwarding, so all traffic has to pass

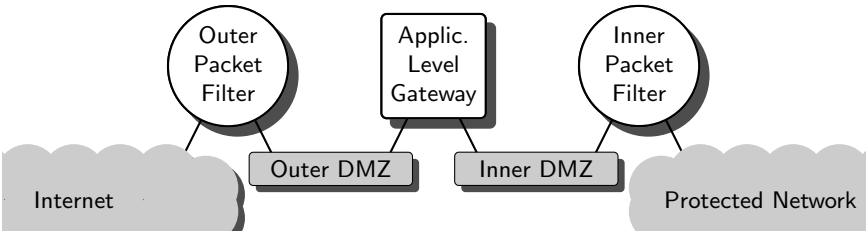

Fig. 25.4. A multi-level firewall with a multi-homed host

through the proper secure proxy services on the gateway. The subnets connecting the gateway to the packet filters are called *demilitarized zones* (*DMZ*) since only authorized nodes are allowed to connect to them.

Setting up the application level gateway as a multi-homed host isn't too difficult in this case. We just don't do any autoconfiguration in the DMZs and configure both the interface addresses and the routes statically.

But things can get complex as soon as either autoconfiguration or dynamic routing get involved. In section 4.4 we have already seen that some implementations make it difficult to prevent a host from listening to router advertisements. In this case the host may configure an undesired default route.

Dynamic routing is another problem: All dynamic routing implementations are expected to check that packet forwarding is enabled before they even start. But on a multi-homed host we *don't* want to forward any packets. So if we wanted to make our firewall setup highly available and used two packet filters and dynamic routing on either side of the application level gateway, then we were in serious trouble. Yes, we could leave it to the packet filters to block all traffic that would need to be routed through, but in a really paranoid (or professional) firewall an application level gateway is meant to be inherently unable to do any packet forwarding at all.

In summary, multi-homed hosts are feasible at least in certain situations, but as soon as they are connected to a network with an advertising router or need to run a dynamic routing protocol, then they are a serious problem.

A

Crash Course: DNS & BIND

If you are familiar with the DNS and operating BIND, then this appendix is probably irrelevant to you. But if you have never set up a name server before, doing so may be a daunting task. To get started, this section provides the absolutely necessary basics, so you might get away without reading some more advanced documentation, like Albitz' and Liu's "DNS and BIND" [2] or the "BIND 9 Administrator Reference Manual" [73] for now. To keep things simple, we ignore IPv6 and just focus on IPv4 addresses.

A.1 Domain Name System (DNS) Basics

The *domain name system* (*DNS*) takes a *domain name* and uses it to look up *resource records* (*RRs*) associated with that name. Each RR consists of a *domain name*, a *record type*, a *record class* which is always `IN` for "Internet" for our purposes, a *time to live* (*TTL*) that specifies how long the entry may be cached, and finally the associated data.

If we want to surf "`www.example.com`", our web server uses the *resolver library* to send a *forward query* to a name server, asking for the "type `A`" records associated with the domain name "`www.example.com.`" to learn about the IP address of the web server with that name. The web server, when accepting the connection from the web browser, may use the IP address used by the browser to do a *reverse lookup*, making the resolver turn the IP address into a funny-looking domain name that can be queried for the associated "type `PTR`" record. This record should contain the host name of the machine we run the web browser on.

To make the DNS scalable, names are structured hierarchically. The *root domain*, "`.`" contains all names. Most importantly, it contains the *top-level domains*, like "`net.`" or "`org.`". These domains contain the *second-level domains* like "`benedikt-stockebrand.net.`" or the one we use here for our demonstration purposes, "`example.com.`". Within these domains we may have additional subdomains as well as RRs that actually contain data.

Reverse lookups convert IP addresses into domain names in the *pseudo domain* "`in-addr.arpa.`" by reverting the order of address bytes, thus turning `192.0.2.80` into "`80.2.9.192.in-addr-arpa.`".

If you look closely, you see that a domain name always ends in a trailing dot. This shows that it is a *fully qualified domain name* (*FQDN*) and is rooted in the root zone. A name without a trailing dot is an *unqualified domain name* and a default domain name needs to be appended to form the FQDN. If we configure our system to use "`example.com.`" as the default domain, then we can use "`www`" as an abbreviation for "`www.example.com.`".

To make the DNS scale, a domain may *delegate* a *zone* to another name server or set of servers. A zone is like a subdomain, except that it doesn't contain the zones it delegates itself. The root servers delegate the "`com.`" zone to some other servers, so "`example.com.`" is still within the root domain, but not within the root zone. Every RR belongs to a single zone but to an entire series of domains. (Later on we'll learn about the two exceptions from this rule necessary to delegate a zone.)

To make the DNS scale to Internet proportions it is necessary to maintain forward and reverse zones independently of each other. This is a major nuisance in many cases but can't be helped.

For every zone there is a *primary name server* (sometimes called the *master* for historical reasons) and a set of *secondary name servers* (or *slaves*). The zone data is actively maintained on the primary and from there distributed to the secondaries. These servers are called *authoritative name servers* because they are assumed to hold correct data at all times.

In contrast to that, name servers are called *non-authoritative* if they only provide cached data for the zone in question. They have either queried an authoritative name server themselves or asked another "upstream" caching name server called a *forwarder*. Normally, programs always use a caching name server, which builds up a rich cache that takes a lot of load from the authoritative servers while providing for fast DNS lookups. All name servers, even authoritative ones, usually also operate as a caching server.

A.2 The BIND Name Server

The standard name server software used with Unix is called *BIND*, short for *Berkeley Internet name domain*.

It consists of the actual name server daemon called `named` and a set of additional tools.

A.2.1 Installation

To set up a name server, we may first need to install some additional packages:

Debian Sarge *The name server proper is in package* `bind9` *and some essential tools like* `dig` *in* `dnsutils`*.*

FreeBSD 6.1 *Here the name server is part of the core system, so no additional packages are necessary. But we need to build some of the default configuration by doing*

```
# cd /etc/namedb
# sh make-localhost example.com
```

to make our name server usable.

Solaris 10 *The name server service manifest and the name server proper are in the packages* `SUNWbindr` *and* `SUNWbind`*. Depending on the subrelease and patchlevel, we may also need to install patch 119783 (SPARC) or 119784 (x86) and reboot the machine to fix a bug*[1]*.* 136

If you want to use an existing BIND installation, make sure the version of BIND is at least 9.2.3 or up; otherwise you will run into problems later on.

A.2.2 Base Configuration

The `named` daemon maintains its configuration in a file called `named.conf`. Authoritative name servers store the data of their zones in separate *zone files*.

Unfortunately, different Unixen have not only complex but also widely differing default configurations. The `named.conf` file may reside in `/etc` (Solaris 10), `/etc/bind` (Debian Sarge), `/etc/namedb` (FreeBSD 6.1) or wherever else an installation chooses to put it. Additionally, the name server daemon `named` is sometimes run in a chroot environment, adding more problems: The configuration may actually reside in the chroot environment instead of `/etc` while a symlink in `/etc` points to the actual location, usually somewhere within `/var`—this can prove disastrous if our backup doesn't include these files. In other cases, the boot scripts copy the configuration files from `/etc` to the chroot environment whenever we restart `named`—as a result, they may happily overwrite any changes we applied there directly, and restarting the `named` directly will keep it running with the old configuration.

This leaves you a difficult choice: Either abandon all the standard configuration and start from scratch or adapt the default configuration accordingly. If you want to start from scratch, a complete albeit minimal configuration may look like this:

named.conf

```
options {
    directory    "/var/named/zonedata";
    pid-file     "/var/run/named.pid";
    allow-query  { "any"; };
};
```

[1] On an unpatched system a reboot will disable the DNS service again.

It tells the name server that all the zone data files are kept in the directory `/var/named/zonedata`, to keep the PID file in a standard place and to allow queries from any address. At this point the `named` works as a caching-only server that is not authoritative for any zones.

Debian Sarge *The configuration options for* `named` *are kept in a separate file* `/etc/bind/named.conf.options` *while the zone configurations go to* `/etc/bind/named.conf.local`*. You need to adapt the following examples accordingly.*

FreeBSD 6.1 *The* `named.conf` *file is located in* `/etc/namedb/`*.*

Solaris 10 *The packages don't include a default configuration, so we create a file* `/etc/named.conf` *with the minimal configuration shown above and a directory* `/var/named/zonedata` *to hold the zone files.* 137

A.2.3 Forwarder Configuration and Fake Root Zones

Next we need to tell the name server how to deal with requests for data that it isn't authoritative for. It is easiest to ask our provider for the IP address of their DNS server and add another line to the `options` section of `named.conf` that reads something like

named.conf

```
options {
    [...]
    forwarders { 192.0.2.1; };
    [...]
}
```

given that the IP address of our provider's DNS server is `192.0.2.1`. The alternative approach, using a hints file that contains the addresses of root name servers, is beyond the scope of this crash course—see the BIND literature, for example Albitz and Liu [2, pages 67–69] if you want to do this.

In an isolated test network without Internet connectivity this doesn't work, because we can't reach a forwarder or root name server. Section 5.2.3 has a solution to deal with this problem: Setting up a fake server for the root zone will ensure that all requests will be served without waiting for a timeout, but possibly with a negative result.

A.2.4 Starting the Name Server

Now we enable the `named` daemon.

Debian Sarge *The* `bind9` *package automatically installs the appropriate boot scripts; we just run* `/etc/init.d/bind9 start` *or reboot our machine.*

FreeBSD 6.1 *We add a line*

/etc/rc.conf

```
named_enable=YES
```

to `/etc/rc.conf` *and either reboot or run* `/etc/rc.d/named start` *by hand.*

Solaris 10 *We run the command* `svcadm enable dns/server`. 138

At this point we should have a running name server. Checking that it works is reasonably straightforward:

☐ Check with `ps` that the `named` process is running.

A.2.5 Adding Forward Zones

To make the name server the primary name server for a zone "`example.com.`", we next add the line

named.conf

```
zone "example.com." { type master; file "example.com.fwd"; };
```

to our `named.conf`. The top of a forward zone file looks like this:

example.com.fwd

```
$TTL 3600
@  SOA  dns1.example.com. root.dns1.example.com. (1 15m 5m 30d 1h)
   NS   dns1.example.com.
   NS   dns2.example.com.
```

The first line tells that the default time to live (TTL) is 3600 seconds, or one hour.

The second line contains the first resource record. The "`@`" refers to the zone name, "`example.com.`". The *SOA* record type, for *start of authority*, tells a few important things about this zone. The data following consists of the name of the primary name server (`dns1`), the mail address of the administrator with the "`@`" replaced by a dot, and a list of five numeric parameters used by the secondary name servers. The *serial number* is used by the secondaries to decide if they need to fetch updated zone data from the primary. It must be incremented whenever the zone data on the primary is changed. The *refresh* and *retry* values specify the interval at which a secondary name server is supposed to contact the primary to see if the zone has changed and at what interval to retry doing so if the primary server is unreachable. The *expire* value defines for how long a secondary should continue to retry reaching the primary until it considers its data entirely outdated and stop serving the zone. The last one is called the *minTTL* for historical reasons but actually defines how long a cache stores negative results, i.e. the fact that a given name doesn't have a DNS entry. Appending an `m`, `h` or `d` to the last four numbers specifies them in minutes, hours or days, respectively.

The third line starts with white space. This indicates that it refers to the same domain name as the previous entry. The type *NS*, for *name server*, indicates that it contains a name server for the domain. All authoritative servers should be listed like this, so the last line specifies a secondary name server for the domain.

Then follow the IP addresses and other data resource records. It is a good habit to add `localhost` here so the loopback address can be reached through that name independent of the default domain configuration. Additionally we should add the address of our name server here.

example.com.fwd

```
localhost.example.com.  A  127.0.0.1
dns1.example.com.       A  192.0.2.1
```

Where these files are kept again depends on the particular Unix and their standard name server configuration.

Debian Sarge *The default configuration expects the zone files to be kept in* `/var/cache/bind/`.

FreeBSD 6.1 *The default configuration expects zone files in* `/etc/namedb`. *But it also assumes that primary zone files are kept in a subdirectory* `master/` *and secondary zones files in* `slave/` *with the* `file` *statements in the* `zone` *clauses to be adapted accordingly.*

Solaris 10 *With the sample configuration shown above, all zone files go to* `/var/named/zonedata/`. 139

A.2.6 Adding Reverse Zones

Reverse zones are quite similar. If we have the address range `192.0.2.0/24` assigned to us, we set up a reverse zone for "`2.0.192.in-addr.arpa.`". The entry in `named.conf` contains the line

named.conf

```
zone "2.0.192.in-addr.arpa." { type master; file "192.0.2.rev"; };
```

if we keep the zone file in `192.0.2.rev`. The zone file itself looks like this:

192.0.2.rev

```
$TTL 3600
@ SOA    dns1.example.com. root.dns1.example.com. (1 15m 5m 30d 1h)
  NS     dns1.example.com.
  NS     dns2.example.com.
1 PTR    dns1.example.com.
```

Except for the last line it looks exactly like a forward zone. The last line

however contains a `PTR` record that associates an IP address in pseudo-domain notation with a "normal" domain name.

The last line makes use of the fact that in a zone file all names without a trailing period, like "`1`", have the zone name appended to form a FQDN, like "`1.2.0.192.in-addr.arpa.`" in this case.

A.2.7 Secondary Servers

Finally, if we want to set up our name server as a secondary for yet another zone, like "`example.net.`" with its primary name server on `192.0.2.2`, we need to add the lines

named.conf

```
zone "example.net." {
    type slave;
    masters { 192.0.2.2; };
    file "example.net.fwd";
};
```

and make sure that the directory with the zone files is writable by the user as which `named` is running.

On the primary server, we ensure that an `NS` record with the name of the secondary name server exists in the zone. Otherwise the primary won't notify the secondary when the zone data changes, which will delay the update of the zone data stored on the secondary.

Whenever a primary server updates a zone, it sends an update notification to all secondary name servers which have a `NS` record in the zone file. Additionally, the secondary servers periodically ask the primary for the `SOA` record of the zone and check if the serial number has changed. When a secondary server detects that its zone data is out of date, then it initiates a *zone transfer*, synchronizes its own copy of the zone with the primary server and saves the updated data to its own local zone file.

A.2.8 Restarting the Server

As the final step, we need to tell the name server to re-read its configuration. Especially in the case of chroot'ed name servers it pays to watch out how the particular Unix wants us to do so:

Debian Sarge *We run `/etc/init.d/bind9 reload` here.*

FreeBSD 6.1 *Similarly, we run `/etc/rc.d/named reload`.*

Solaris 10 *We just run `rndc reload` directly to make the `named` reload its configuration.* 140

A.2.9 Testing and Debugging

At this point our name server should be up and running, serving the zones "`example.com.`" and "`2.0.192.in-addr.arpa.`".

The most important tool to see if the name server works as anticipated is `dig`, which is part of the BIND distribution. It directly queries a name server, without using the resolver library. It does however read `/etc/resolv.conf` to find the name server(s) to query unless an option `@`⟨*name server*⟩ is also given. Following that, a fully qualified domain name must be given—unqualified names won't have a default domain name appended. Alternatively, an option `-x` and an IP address can be given; the address will be expanded into the matching name in the "`in-addr.arpa.`" pseudo-domain. Optionally, a query type may follow. If it doesn't, an `A` record type is requested unless we use the `-x` option, which implies a `PTR` query. A query type `ANY` will query for any record type, the query type `AXFR` transfers an entire zone.

- ☐ The easiest way to check that the name server works uses `dig` to look at the zone: Running

  ```
  dig @::1 example.com. AXFR
  ```

  should return all the data stored in the zone, plus some additional noise.
- ☐ If that doesn't work as expected, take a look at the log files. The `named` daemon writes its log messages to the `syslogd`, so it should be fairly obvious to see what exactly went wrong.
- ☐ Next use a tool like `ping6` to see if the resolver deals with the data correctly. If this has an excessive delay (up to two minutes) either the forwarder configuration is broken or the fake root zone doesn't work.

A.2.10 Zone Delegations

To delegate a zone to another nameserver, we add an `NS` record to the parent zone referring it to the delegated zone. Additionally, if the name server for the delegated zone has a name within the delegated zone, its `A` record is kept in both the parent and the delegated zone; the extra `A` records in the parent zone are called *glue record*.

Both the parent zone and the delegated zone must contain `NS` records for the delegated zone.

A.3 Common Pitfalls

The BIND configuration is notorious for a number of common mistakes novice (and not so novice) DNS administrators make. Here are the most common ones:

Missing a trailing period will cause serious problems because the zone name will be appended. If any record in a forward zone seems to be broken or a reverse lookup comes up with something that looks like "`www.example.com.example.com.`", first check if the domain name or PTR data has the necessary trailing dot.

Unchanged SOA serial numbers will make the primary name server deliver6 the updated data while the secondaries don't know of the change and continue to serve stale data. Best make it a habit to test all DNS updates on the secondaries, not the primaries.

Missing root zone configuration will cause some pain because it will lead to long timeouts, up to two minutes depending on the resolver library. If there are such timeouts, check that the forwarder configuration, fake root zone or root zone hints work.

B

Assigned Numbers and Addresses

This appendix contains the assigned addresses and numbers used throughout this book. The lists are meant as a quick reference for the most relevant addresses and numbers only; for a complete and up-to-date list, check the Internet Assigned Numbers Authority (IANA) at `http://www.iana.org/`.

B.1 Addresses and Address Prefixes

Prefix	Description
`::/96`	IPv4-compatible IPv6 addresses (deprecated)
`::ffff:0:0/96`	IPv4-mapped IPv6 addresses
`2000::/3`	Globally routed unicasts (already allocated)
`fc00::/8`	Centrally managed unique-local unicasts (not yet available)
`fd00::/8`	Randomly chosen unique-local unicasts
`fe80::/10`	Link-local unicasts (only `fe80::/64` used)
`fec0::/10`	Site-local unicasts (deprecated)
`ff00::/8`	Multicasts
other	Globally routed unicasts (not yet allocated)

B.1.1 Unicast Addresses

Prefix	Description
`::`	The unspecified address
`::1`	The loopback address
`2001::/32`	Teredo prefix
`2001:db8::/32`	Documentation prefix
`2002::/16`	6to4 tunnel prefix
`2002:c058:6301::`	6to4 public relay router anycast (`192.88.99.1`)
`3ffe::/16`	Historic 6Bone testbed address prefix
`3ffe:831f::/32`	Historic Teredo prefix

B.1.2 Multicast Addresses

Prefix	Description
ff00::/12	Permanent multicast addresses
ff02::1	All-nodes link-local multicasts
ff02::2	All-routers link-local multicasts
ff02::5	All OSPF routers
ff02::6	OSPF designated and backup designated routers
ff02::9	Routing information protocol (RIPng)
ff02::d	All PIM routers
ff02::16	All MLDv2-capable routers
ff02::1:2	All DHCP relay agents and servers
ff02::1:ff00:0/104	Solicited-node multicasts
ff05::2	All-routers site-local multicasts
ff05::1:3	All DHCP servers
ff10::/12	Transient multicast addresses
ff3*s*::/24	Unicast-prefix based multicast addresses
ff3*s*::/96	Source-specific multicast addresses
ff70::/12	Embedded rendezvous point multicast addresses

B.1.3 Multicast Scopes

Value	Description	Value	Description
0	Reserved	6, 7	(unnamed)
1	Interface-local	8	Organization-local
2	Link-local	9–d	(unnamed)
3	Reserved	e	Global
4	Admin-local	f	Reserved
5	Site-local		

B.1.4 Anycast and Other Special Interface IDs

Interface ID	Description
0:0:0:0	Subnet router anycast
0:fefe:0:0/32	ISATAP
200:fefe:0:0/32	ISATAP
dfff:ffff:ffff:ff80 –dfff:ffff:ffff:ffff	Reserved by IANA

B.2 Transport Layer Port Numbers

B.2.1 TCP

Port	Description
7	Echo
22	Secure shell (SSH)
25	Simple mail transfer protocol (SMTP)
53	Domain name system (DNS)
80	Hypertext transport protocol (HTTP)
123	Network time protocol (NTP)
179	Border gateway protocol (BGP)
443	Secure hypertext transport protocol (HTTPS)
2601	Quagga `zebra` virtual terminal
2603	Quagga `ripngd` virtual terminal
2605	Quagga `bgpd` virtual terminal
2606	Quagga `ospf6d` virtual terminal

B.2.2 UDP

Port	Description
7	Echo
53	Domain name system (DNS)
123	Network time protocol (NTP)
500	Internet key exchange (IKE)
514	Syslog
521	Routing information protocol (RIPng)
546	Dynamic host configuration protocol (DHCPv6) client
547	Dynamic host configuration protocol (DHCPv6) server/relay
1194	OpenVPN default port
3544	Teredo

B.3 ICMPv6 Types

Value	Description
1	Destination unreachable
2	Packet too big
3	TTL exceeded
4	Parameter problem
128	Echo request
129	Echo reply
130	Multicast listener discovery (MLD) query
131	Multicast listener discovery version 1 (MLDv1) listener report
132	Multicast listener discovery version 1 (MLDv1) listener done
133	Router solicitation
134	Router advertisement
135	Neighbor solicitation
136	Neighbor advertisement
137	Redirect
138	Router renumbering
143	Multicast listener discovery version 2 (MLDv2) listener report

B.4 Protocol Numbers in Next Header Field

Protocol	Description
0	Hop-by-hop option
4	4in6 encapsulation
6	Transmission control protocol (TCP)
17	User datagram protocol (UDP)
41	Encapsulated IPv6: 6in4 and 6in6 encapsulation
43	Routing Header
44	Fragment header
46	Resource reservation protocol (RSVP)
47	Generic routing encapsulation (GRE)
50	Encapsulated security payload (ESP) for IPsec
51	Authentication header (AH) for IPsec
58	Internet control message protocol, version 6 (ICMPv6)
59	No next header
60	Destination options
89	Open shortest path first (OSPF)
103	Protocol Independent Multicast (PIM)

B.5 Ethernet

B.5.1 Ethernet Types

Hex	Decimal	Allocated to protocol
800	2048	Internet protocol, version 4 (IPv4)
86dd	34525	Internet protocol, version 6 (IPv6)
880b	34827	Point-to-point protocol (PPP)

B.5.2 Ethernet Addresses

IPv6 unicast addresses are mapped to the Ethernet address of the interface they are assigned to.

IPv6 multicast addresses are mapped to Ethernet addresses by appending the last 32 bits of the IPv6 address to the Ethernet multicast prefix `33:33::/16`:

$$x:x:x:x:x:x:abcd:efgh \Longrightarrow \texttt{33:33}:ab:cd:ef:gh$$

Autoconfiguration creates the interface ID from an Ethernet address according to this pattern:

$$ab:cd:ef:gh:ij:kl \quad \Longrightarrow \quad a(b+2)cd:ef\texttt{FF:FE}gh:ijkl$$

References

The RFCs listed here can be downloaded via FTP from the IETF server at `ftp://ftp.ietf.org/rfc/`.

1. Andrew Adams, Jonathan Nicholas, and William Siadak. Protocol Independent Multicast - Dense Mode (PIM-DM): Protocol Specification (Revised). RFC 3973, 2005.
2. Paul Albitz and Cricket Liu. *DNS and BIND*. O'Reilly & Associates, 2001.
3. Jari Arkko, Vijay Devarapalli, and Francis Dupont. Using IPsec to Protect Mobile IPv6 Signaling Between Mobile Nodes and Home Agents. RFC 3776, 2004.
4. Fred Baker, Carol Iturralde, Francois Le Faucheur, and Bruce Davie. Aggregation of RSVP for IPv4 and IPv6 Reservations. RFC 3175, 2001.
5. Fred Baker, Eliot Lear, and Ralph Droms. Procedures for Renumbering an IPv6 Network without a Flag Day. RFC 4192, 2005.
6. Fred Baker and Pekka Savola. Ingress Filtering for Multihomed Networks. RFC 3704, 2004.
7. Tony Bates, Yakov Rekhter, Ravi Chandra, and Dave Katz. Multiprotocol Extensions for BGP-4. RFC 2858, 2000.
8. Tim Berners-Lee, Roy T. Fielding, and Larry Masinter. Uniform Resource Identifier (URI): Generic Syntax. RFC 3986, 2005.
9. Supratik Bhattacharyya. An Overview of Source-Specific Multicast (SSM). RFC 3569, 2003.
10. Peter Bieringer. Linux IPv6 Howto. `http://www.bieringer.de/linux/IPv6/`.
11. Steven Blake, David L. Black, Mark A. Carlson, and Elwyn Davies. An Architecture for Differentiated Services. RFC 2475, 1998.
12. Scott Bradner. Key words for use in RFCs to Indicate Requirement Levels. RFC 2119, 1997.
13. Randy Bush, Alain Durand, Bob Fink, Olafur Gudmundsson, and Tony Hain (eds.). Representing Internet Protocol version 6 (IPv6) Addresses in the Domain Name System (DNS). RFC 3363, 2002.
14. Brian E. Carpenter and Cyndi Jung. Transmission of IPv6 over IPv4 Domains without Explicit Tunnels. RFC 2529, 1999.
15. Brian E. Carpenter and Keith Moore. Connection of IPv6 Domains via IPv4 Clouds. RFC 3056, 2001.

16. Rob Coltun, Dennis Ferguson, and John Moy. OSPF for IPv6. RFC 2740, 1999.
17. Alex Conta and Stephen Deering. Generic Packet Tunneling in IPv6—Specification. RFC 2473, 1998.
18. Matt Crawford. Binary Labels in the Domain Name System. RFC 2673, 1999.
19. Matt Crawford. Non-Terminal DNS Name Redirection. RFC 2672, 1999.
20. Matt Crawford. Router Renumbering for IPv6. RFC 2894, 2000.
21. Matt Crawford and Christian Huitema. DNS Extensions to Support IPv6 Address Aggregation and Renumbering. RFC 2874, 2000.
22. Michael Daniele, Brian Haberman, Shawn A. Routhier, and Juergen Schoenwaelder. Textual Conventions for Internet Network Addresses. RFC 4001, 2005.
23. Stephen E. Deering, William C. Fenner, and Brian Haberman. Multicast Listener Discovery (MLD) for IPv6. RFC 2710, 1999.
24. Stephen E. Deering and Robert M. Hinden. Internet Protocol, Version 6 (IPv6)—Specification. RFC 2460, 1998.
25. Vijay Devarapalli, Ryuji Wakikawa, Alexandru Petrescu, and Pascal Thubert. Network Mobility (NEMO) Basic Support Protocol. RFC 3963, 2005.
26. Richard Draves. Default Address Selection for Internet Protocol version 6 (IPv6). RFC 3484, 2003.
27. Richard Draves and Dave Thaler. Default Router Preferences and More-Specific Routes. RFC 4191, 2005.
28. Alain Durand, Paolo Fasano, Ivano Guardini, and Domenico Lento. IPv6 Tunnel Broker. RFC 3053, 2001.
29. Donald E. Eastlake 3rd. Cryptographic Algorithm Implementation Requirements for Encapsulating Security Payload (ESP) and Authentication Header (AH). RFC 4305, 2005.
30. Bob Braden (ed.), Lixia Zhang, Steve Berson, Shai Herzog, and Sugih Jamin. Resource ReSerVation Protocol (RSVP)—Version 1 Functional Specification. RFC 2205, 1997.
31. Charlie Kaufman (ed.). Internet Key Exchange (IKEv2) Protocol. RFC 4306, 2005.
32. Jon Postel (ed.). INTERNET PROTOCOL. RFC 791, 1981.
33. Jukka Manner (ed.) and Markku Kojo (ed.). Mobility Related Terminology. RFC 3753, 2004.
34. Rajeev Koodli (ed.). Fast Handovers for Mobile IPv6. RFC 4068, 2005.
35. Ralph Droms (ed.), Jim Bound, Bernie Volz, Ted Lemon, Charles E. Perkins, and Mike Carney. Dynamic Host Configuration Protocol for IPv6 (DHCPv6). RFC 3315, 2003.
36. Rolland Vida (ed.) and Luis Henrique Maciel Kosmalski Costa (ed.). Multicast Listener Discovery Version 2 (MLDv2) for IPv6. RFC 3810, 2004.
37. Stephen E. Deering (ed.). ICMP Router Discovery Messages. RFC 1256, 1991.
38. Robert Elz. A Compact Representation of IPv6 Addresses. RFC 1924, 1996.
39. Deborah Estrin, Dino Farinacci, Ahmed Helmy, David Thaler, Stephen Deering, Mark Handley, Van Jacobson, Ching gung Liu, Puneet Sharma, and Liming Wei. Protocol Independent Multicast-Sparse Mode (PIM-SM): Protocol Specification. RFC 2362, 1998.
40. Dino Farinacci, Tony Li, Stan Hanks, David Meyer, and Paul Traina. Generic Routing Encapsulation (GRE). RFC 2784, 2000.

41. Bill Fenner and David Meyer. Multicast Source Discovery Protocol (MSDP). RFC 3618, 2003.
42. Paul Ferguson and Daniel Senie. Network Ingress Filtering: Defeating Denial of Service Attacks which employ IP Source Address Spoofing. RFC 2827, 2000.
43. Vince Fuller, Tony Li, Jessica (Jie Yun) Yu, and Kannan Varadhan. Supernetting: an Address Assignment and Aggregation Strategy. RFC 1338, 1992.
44. Vince Fuller, Tony Li, Jessica (Jie Yun) Yu, and Kannan Varadhan. Classless Inter-Domain Routing (CIDR): an Address Assignment and Aggregation Strategy. RFC 1519, 1993.
45. Bob Gilligan, Susan Thomson, Jim Bound, and Jack McCann. Basic Socket Interface Extensions for IPv6. RFC 3493, 2003.
46. Robert E. Gilligan and Erik Nordmark. Transition Mechanisms for IPv6 Hosts and Routers. RFC 1933, 1996.
47. Robert E. Gilligan and Erik Nordmark. Transition Mechanisms for IPv6 Hosts and Routers. RFC 2893, 2000.
48. Dan Grossman. New Terminology and Clarifications for Diffserv. RFC 3260, 2002.
49. Brian Haberman. Allocation Guidelines for IPv6 Multicast Addresses. RFC 3307, 2002.
50. Brian Haberman. Source Address Selection for the Multicast Listener Discovery (MLD) Protocol. RFC 3590, 2003.
51. Brian Haberman and Dave Thaler. Unicast-Prefix-based IPv6 Multicast Addresses. RFC 3306, 2002.
52. Silvia Hagen. *IPv6 Essentials*. O'Reilly & Associates, 1st edition, 2002.
53. Silvia Hagen. *IPv6: Grundlagen—Funktionalität—Integration*. Sunny Edition, 2004.
54. Silvia Hagen. *IPv6 Essentials*. O'Reilly & Associates, 2nd edition, 2006.
55. Stephen R. Hanna, Baiju V. Patel, and Munil Shah. Multicast Address Dynamic Client Allocation Protocol (MADCAP). RFC 2730, 1999.
56. Thomas Hardjono and Brian Weis. The Multicast Group Security Architecture. RFC 3740, 2004.
57. Dan Harkins and Dave Carrel. The Internet Key Exchange (IKE). RFC 2409, 1998.
58. Dimitry Haskin and Ed Allen. IP Version 6 over PPP. RFC 2023, 1996.
59. Dimitry Haskin and Ed Allen. IP Version 6 over PPP. RFC 2472, 1998.
60. C. Hedrick. Routing Information Protocol. RFC 1058, 1988.
61. Robert M. Hinden and Stephen E. Deering. IP Version 6 Addressing Architecture. RFC 1884, 1995.
62. Robert M. Hinden and Stephen E. Deering. IP Version 6 Addressing Architecture. RFC 2373, 1998.
63. Robert M. Hinden and Stephen E. Deering. Internet Protocol Version 6 (IPv6) Addressing Architecture. RFC 3513, 2003.
64. Robert M. Hinden and Stephen E. Deering. IP Version 6 Addressing Architecture. RFC 4291, 2006.
65. Robert M. Hinden, Stephen E. Deering, and Erik Nordmark. IPv6 Global Unicast Address Format. RFC 3587, 2003.
66. Robert M. Hinden and Brian Haberman. Unique Local IPv6 Unicast Addresses. RFC 4193, 2005.
67. Robert M. Hinden, Mike O'Dell, and Stephen E. Deering. An IPv6 Aggregatable Global Unicast Address Format. RFC 2374, 1998.

68. Russell Housley, Tim Polk, Warwick Ford, and David Solo. Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. RFC 3280, 2002.
69. Christian Huitema. An Anycast Prefix for 6to4 Relay Routers. RFC 3068, 2001.
70. Christian Huitema. Teredo: Tunneling IPv6 over UDP through Network Address Translations (NATs). RFC 4380, 2006.
71. Christian Huitema and Brian Carpenter. Deprecating Site Local Addresses. RFC 3879, 2004.
72. Internet Architecture Board (IAB) and Internet Engineering Steering Group (IESG). IAB/IESG Recommendations on IPv6 Address Allocations to Sites. RFC 3177, 2001.
73. Internet Systems Consortium ("ISC"). BIND 9 Administrator Reference Manual. `http://www.isc.org/sw/bind/arm93/Bv9ARM.pdf`, 2005.
74. Jun-ichiro itojun Hagino and Kazu Yamamoto. An IPv6-to-IPv4 Transport Relay Translator. RFC 3142, 2001.
75. David B. Johnson, Charles E. Perkins, and Jari Arkko. Mobility Support in IPv6. RFC 3775, 2004.
76. Vijayabhaskar A Kalusivalingam. Network Information Service (NIS) Configuration Options for Dynamic Host Configuration Protocol for IPv6 (DHCPv6). RFC 3898, 2004.
77. Vijayabhaskar A. Kalusivalingam. Simple Network Time Protocol (SNTP) Configuration Option for DHCPv6. RFC 4075, 2005.
78. Stephen Kent. IP Authentication Header. RFC 4302, 2005.
79. Stephen Kent. IP Encapsulating Security Payload (ESP). RFC 4303, 2005.
80. Stephen Kent and Randall Atkinson. IP Encapsulating Security Payload (ESP). RFC 2406, 1998.
81. Stephen Kent and Karen Seo. Security Architecture for the Internet Protocol. RFC 4301, 2005.
82. Dorian Kim, David Meyer, Hank Kilmer, and Dino Farinacci. Anycast Rendevous Point (RP) mechanism using Protocol Independent Multicast (PIM) and Multicast Source Discovery Protocol (MSDP). RFC 3446, 2003.
83. Franck Le, Stefano Faccin, Basavaraj Patil, and Hannes Tschofenig. Mobile IPv6 and Firewalls: Problem Statement. RFC 4487, 2006.
84. Gary Scott Malkin. RIP Version 2—Carrying Additional Information. RFC 1388, 1993.
85. Gary Scott Malkin and Robert E. Minnear. RIPng for IPv6. RFC 2080, 1997.
86. Pedro R. Marques and Francis Dupont. Use of BGP-4 Multiprotocol Extensions for IPv6 Inter-Domain Routing. RFC 2545, 1999.
87. Jack McCann, Stephen E. Deering, and Jeffrey Mogul. Path MTU Discovery for IP version 6. RFC 1981, 1996.
88. Yasuhiro Morishita and Tatuya Jinmei. Common Misbehavior Against DNS Queries for IPv6 Addresses. RFC 4074, 2005.
89. John Moy. OSPF Version 2. RFC 2328, 1998.
90. Thomas Narten and Richard Draves. Privacy Extensions for Stateless Address Autoconfiguration in IPv6. RFC 3041, 2001.
91. Thomas Narten, Erik Nordmark, and William Allen Simpson. Neighbor Discovery for IP Version 6 (IPv6). RFC 2461, 1998.

92. Kathleen Nichols, Steven Blake, Fred Baker, and David L. Black. Definition of the Differentiated Services Field (DS Field) in the IPv4 and IPv6 Headers. RFC 2474, 1998.
93. Pekka Nikander, Jari Arkko, Tuomas Aura, Gabriel Montenegro, and Erik Nordmark. Mobile IP Version 6 Route Optimization Security Design Background. RFC 4225, 2005.
94. Erik Nordmark. Stateless IP/ICMP Translation Algorithm (SIIT). RFC 2765, 2000.
95. Erik Nordmark and Robert E. Gilligan. Basic Transition Mechanisms for IPv6 Hosts and Routers. RFC 4213, 2005.
96. Yakov Rekhter and Tony Li. A Border Gateway Protocol 4 (BGP-4). RFC 1771, 1995.
97. Yakov Rekhter, Robert G Moskowitz, Daniel Karrenberg, Geert Jan de Groot, and Eliot Lear. Address Allocation for Private Internets. RFC 1918, 1996.
98. Pekka Savola and Brian Haberman. Embedding the Rendezvous Point (RP) Address in an IPv6 Multicast Address. RFC 3956, 2004.
99. Pekka Savola and Chirayu Patel. Security Considerations for 6to4. RFC 3964, 2004.
100. Henning Schulzrinne and Bernie Volz. Dynamic Host Configuration Protocol (DHCPv6) Options for Session Initiation Protocol (SIP) Servers. RFC 3319, 2003.
101. William Allen Simpson. IP in IP Tunneling. RFC 1853, 1995.
102. Hesham Soliman, Claude Castelluccia, Karim El Malki, and Ludovic Bellier. Hierarchical Mobile IPv6 Mobility Management (HMIPv6). RFC 4140, 2005.
103. W. Richard Stevens. *TCP/IP Illustrated*, volume 1, The Protocols. Addison-Wesley, 1994.
104. Benedikt Stockebrand. IPv6 in der Praxis—Teil 1: Weichenstellung. *iX*, 2/Februar, 2005.
105. Benedikt Stockebrand. IPv6 in der Praxis—Teil 2: Namensdienste und Zeitsynchronisation. *iX*, 3/März, 2005.
106. Benedikt Stockebrand. IPv6 in der Praxis—Teil 3: Diensttauglich. *iX*, 4/April, 2005.
107. Dave Thaler, Bill Fenner, and Bob Quinn. Socket Interface Extensions for Multicast Source Filters. RFC 3678, 2004.
108. Susan Thomson and Christian Huitema. DNS Extensions to support IP version 6. RFC 1886, 1995.
109. Susan Thomson, Christian Huitema, Vladimir Ksinant, and Mohsen Souissi. DNS Extensions to Support IP Version 6. RFC 3596, 2003.
110. Susan Thomson and Thomas Narten. IPv6 Stateless Address Autoconfiguration. RFC 2462, 1998.
111. George Tsirtsis and Pyda Srisuresh. Network Address Translation - Protocol Translation (NAT-PT). RFC 2766, 2000.
112. John Wroclawski. The Use of RSVP with IETF Integrated Services. RFC 2210, 1997.

Index

Rather than providing a separate acronym list, or even a glossary which duplicates various parts of the book proper, this index doubles as an acronym list. Additionally, an online version of this index is available at my home page, `http://www.benedikt-stockebrand.net/`, which should be easier to search than a printed index.