Linux IPv6 HOWTO (en)

Peter Bieringer

pb at bieringer dot de

Revision History

Revision 0.60.2	2007-10-03	Revised by: PB
Revision 0.60.1	2007-06-16	Revised by: PB
Revision 0.60	2007-05-31	Revised by: PB
Revision 0.51	2006-11-08	Revised by: PB

The goal of the Linux IPv6 HOWTO is to answer both basic and advanced questions about IPv6 on the Linux operating system. This HOWTO will provide the reader with enough information to install, configure, and use IPv6 applications on Linux machines. Intermediate releases of this HOWTO are available at mirrors.bieringer.de or mirrors.deepspace6.net. See also revision history for changes.

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Chapter 1. General

Information about available translations you will find in section Translations.

1.1. Copyright, license and others

1.1.1. Copyright

1.1.2. License

1.1.3. About the author

1.1.3.1. Internet/IPv6 history of the author

- 1993: I got in contact with the Internet using console based e-mail and news client (e.g. look for "e91abier" on groups.google.com, that's me).
- 1996: I got a request for designing a course on IPv6, including a workshop with the Linux operating system.
- 1997: Started writing a guide on how to install, configure and use IPv6 on Linux systems, called IPv6 & Linux - HowTo (see IPv6 & Linux - HowTo/History for more information).
- 2001: Started writing this new Linux IPv6 HOWTO.

1.1.3.2. Contact

The author can be contacted via e-mail at <pb at bieringer dot de> and also via his homepage.

He's currently living in Munich [northern part of Schwabing] / Bavaria / Germany (south) / Europe (middle) / Earth (surface/mainland).

1.2. Category

This HOWTO should be listed in category "*Networking/Protocols*".

1.3. Version, History and To-Do

1.3.1. Version

The current version is shown at the beginning of the document.

CVS information: CVS-ID: $Id: Linux+IPv6-HOWTO.lyx,v 1.119 2006/11/08 22:45:25 pbldp Exp $

For other available versions/translations see also http://www.bieringer.de/linux/IPv6/.

1.3.2. History

1.3.2.1. Major history

2001-11-30: Starting to design new HOWTO.

2002-01-02: A lot of content completed, first public release of chapter 1 (version 0.10).

2002-01-14: More completed, some reviews, public release of the whole document (version 0.14).

2002-08-16: Polish translation is in progress

2002-10-31: Chinese translation is available (see Translations for more)

2002-11-10: German translation is in progress

2003-02-10: German translation is available

2003-04-09: French translation is in progress

2003-05-09: French translation is available

2003-08-15: Spanish translation is in progress

2003-10-16: Italian translation is in progress

2004-03-12: Italian translation is available

2004-06-18: Greek translation is in progress

2004-08-29: Spanish translation is still NOT in progress

2005-07-25: Turkish translation is availble

2007-03-28: Portuguese-Brazil is in progress

1.3.2.2. Full history

See revision history at the end of this document.

1.3.3. To-Do

- Fill in missing content
- Finishing grammar checking

1.4. Translations

Translations always have to contain the URL, version number and copyright of the original document (but yours, too). Pls. don't translate the original changelog, this is not very useful - also do not translate the full section about available translations, can be run out-of-date, add an URL to this section here in the English howto.

Looks like the document's change frequency is mostly less than once per month. Since version 0.27 it looks like that most of the content contributed by me has been written. Translations always have to use the English version as source.

1.4.1. To language

1.4.1.1. Chinese

A Chinese translation by Burma Chen <expns at yahoo dot com> (announced to me at 2002-10-31) can be found on the TLDP: http://www.ibiblio.org/pub/Linux/docs/HOWTO/translations/zh/Linux-IPv6-HOWTO.txt.gz (g'zipped txt). It's a snapshot translation, don't know whether kept up-to-date.

1.4.1.2. Polish

Since 2002-08-16 a Polish translation was started and is still in progress by Lukasz Jokiel <Lukasz dot Jokiel at klonex dot com dot pl>. Taken source: CVS-version 1.29 of LyX file, which was source for howto version 0.27. Status is still work-in-progress (2004-08-30).

1.4.1.3. German

With 2002-11-10 a German translation was started by Georg KÃ¤fer <gkaefer at gmx dot at> and the first public version was published 2003-02-10. It's originally available on Deep Space 6 at http://mirrors.deepspace6.net/Linux+IPv6-HOWTO-de/ (mirrored e.g. on http://mirrors.bieringer.de/Linux+IPv6-HOWTO-de/). This version will stay up-to-date as much as possible.

1.4.1.4. French

With 2003-04-09 a French translation was started by Michel Boucey <mboucey at free dot fr> and the first public version was published 2003-05-09. It's originally available on Deep Space 6 at http://mirrors.deepspace6.net/Linux+IPv6-HOWTO-fr/ (mirrored e.g. on http://mirrors.bieringer.de/Linux+IPv6-HOWTO-fr/).

1.4.1.5. Spanish

The first try of a Spanish translations ran into ressource problem, currently I got two other ones, status will be updated when more information is available.

1.4.1.6. Italian

With 2003-10-16 a Italian translation was started by Michele Ferritto <m dot ferritto at virgilio dot it> for the ILDP (Italian Linux Documentation Project) and the first public version was published 2004-03-12. It's originally available on the ILDP at http://it.tldp.org/HOWTO/Linux+IPv6-HOWTO/.

1.4.1.7. Japanese

On 2003-05-14 Shino Taketani <shino_1305 at hotmail dot com> send me a note that he planned to translate the HowTo into Japanese.

1.4.1.8. Greek

On 2004-06-18 Nikolaos Tsarmpopoulos <ntsarb at uth dot gr> send me a note that he planned to translate the HowTo into Greek.

1.4.1.9. Turkish

On 2005-07-18 Necdet Yucel <nyucel at comu dot edu dot tr> send me a note that a Turkish translation is available. It's a snapshot translation and can be found at http://www.belgeler.org/howto/ipv6-howto.html.

1.4.1.10. Portuguese-Brazil

On 2007-03-28 Claudemir da Luz <claudemir dot daluz at virtuallink dot com dot br> send me a note that he planned to translate the HowTo in Portuguese-Brazil.

1.5. Technical

1.5.1. Original source of this HOWTO

This HOWTO is currently written with LyX version 1.4.4 on a Fedora Core 6 system with template SGML/XML (DocBook book). It's available on TLDP-CVS / users / Peter-Bieringer for contribution.

1.5.1.1. Code line wrapping

Code line wrapping is done using selfmade utility "lyxcodelinewrapper.pl", you can get it from CVS for your own usage: TLDP-CVS / users / Peter-Bieringer

---

1.5.1.2. SGML generation

SGML/XML is generated using export function in LyX.

1.5.2. On-line references to the HTML version of this HOWTO (linking/anchors)

1.5.2.1. Master index page

Generally, a reference to the master index page is recommended.

1.5.2.2. Dedicated pages

Because the HTML pages are generated out of the SGML file, the HTML filenames turn out to be quite random. However, some pages are tagged in LyX, resulting in static names. These tags are useful for references and shouldn't be changed in the future.

If you think that I have forgotten a tag, please let me know, and I will add it.

1.6. Preface

Some things first:

1.6.1. How many versions of a Linux & IPv6 related HOWTO are floating around?

Including this, there are three (3) HOWTO documents available. Apologies, if that is too many ;-)

1.6.1.1. Linux IPv6 FAQ/HOWTO (outdated)

The first IPv6 related document was written by *Eric Osborne* and called Linux IPv6 FAQ/HOWTO (please use it only for historical issues). Latest version was 3.2.1 released July, 14 1997.

Please help: if someone knows the date of birth of this HOWTO, please send me an e-mail (information will be needed in "history").

1.6.1.2. IPv6 & Linux - HowTo (maintained)

There exists a second version called IPv6 & Linux - HowTo written by me (*Peter Bieringer*) in pure HTML. It was born April 1997 and the first English version was published in June 1997. I will continue to maintain it, but it will slowly fade (but not full) in favour of the Linux IPv6 HOWTO you are currently reading.

1.6.1.3. Linux IPv6 HOWTO (this document)

Because the IPv6 & Linux - HowTo is written in pure HTML it's not really compatible with the The Linux Documentation Project (TLDP). I (*Peter Bieringer*) got a request in late November 2001 to rewrite the IPv6 & Linux - HowTo in SGML. However, because of the discontinuation of that HOWTO (Future of IPv6 & Linux - HowTo), and as IPv6 is becoming more and more standard, I decided to write a new document covering basic and advanced issues which will remain important over the next few years. More dynamic and some advanced content will be still found further on in the second HOWTO (IPv6 & Linux - HowTo).

1.7. Used terms, glossary and shortcuts

1.7.1. Network related

Base 10
: Well known decimal number system, represent any value with digit 0-9.

Base 16
: Usually used in lower and higher programming languages, known also as hexadecimal number system, represent any value with digit 0-9 and char A-F (case insensitive).

Base 85
: Representation of a value with 85 different digits/chars, this can lead to shorter strings but never seen in the wild.

Bit
: Smallest storage unit, on/true (1) or off/false (0)

Byte
: Mostly a collection of 8 (but not really a must - see older computer systems) bits

Device
: Here, hardware of network connection, see also NIC

Dual homed host
: A dual homed host is a node with two network (physical or virtual) interfaces on two different links, but does not forward any packets between the interfaces.

Host
: Generally a single homed host on a link. Normally it has only one active network interface, e.g. Ethernet or (not and) PPP.

Interface
: Mostly same as "device", see also NIC

IP Header
: Header of an IP packet (each network packet has a header, kind of is depending on network layer)

Link
: A link is a layer 2 network packet transport medium, examples are Ethernet, Token Ring, PPP, SLIP, ATM, ISDN, Frame Relay,...

Node
: A node is a host or a router.

Octet
: A collection of 8 real bits, today also similar to "byte".

Port
: Information for the TCP/UDP dispatcher (layer 4) to transport information to upper layers

Protocol
: Each network layer contains mostly a protocol field to make life easier on dispatching transported information to upper layer, seen in layer 2 (MAC) and 3 (IP)

Router
: A router is a node with two or more network (physical or virtual) interfaces, capable of forwarding packets between the interfaces.

Socket
: An IP socket is defined by source and destination IP addresses and Ports and (binding)

Stack
: Network related a collection of layers

Subnetmask
: IP networks uses bit masks to separate local networks from remote ones

Tunnel
: A tunnel is typically a point-to-point connection over which packets are exchanged which carry the data of another protocol, e.g. an IPv6-in-IPv4 tunnel.

1.7.1.1. Shortcuts

ACL
: Access Control List

API
: Application Programming Interface

ASIC
: Application Specified Integrated Circuit

BSD
: Berkeley Software Distribution

CAN-Bus
: Controller Area Network Bus (physical bus system)

ISP
: Internet Service Provider

KAME
: Project - a joint effort of six companies in Japan to provide a free IPv6 and IPsec (for both IPv4 and IPv6) stack for BSD variants to the world www.kame.net

LIR
: Local Internet Registry

NIC
: Network Interface Card

RFC
: Request For Comments - set of technical and organizational notes about the Internet

USAGI
: UniverSAl playGround for Ipv6 Project - works to deliver the production quality IPv6 protocol stack for the Linux system.

1.7.2. Document related

1.7.2.1. Long code line wrapping signal char

The special character "Â¬" is used for signaling that this code line is wrapped for better viewing in PDF and PS files.

1.7.2.2. Placeholders

In generic examples you will sometimes find the following:

```
<myipaddress>
```

For real use on your system command line or in scripts this has to be replaced with relevant content (removing the < and > of course), the result would be e.g.

```
1.2.3.4
```

1.7.2.3. Commands in the shell

Commands executable as non-root user begin with $, e.g.

```
$ whoami
```

Commands executable as root user begin with #, e.g.

```
# whoami
```

1.8. Requirements for using this HOWTO

1.8.1. Personal prerequisites

1.8.1.1. Experience with Unix tools

You should be familiar with the major Unix tools e.g. *grep*, *awk*, *find*, ... , and know about their most commonly used command-line options.

1.8.1.2. Experience with networking theory

You should know about layers, protocols, addresses, cables, plugs, etc. If you are new to this field, here is one good starting point for you: http://www.rigacci.org/docs/biblio/online/intro_to_networking/book1.htm

1.8.1.3. Experience with IPv4 configuration

You should definitely have some experience in IPv4 configuration, otherwise it will be hard for you to understand what is really going on.

1.8.1.4. Experience with the Domain Name System (DNS)

Also you should understand what the Domain Name System (DNS) is, what it provides and how to use it.

1.8.1.5. Experience with network debugging strategies

You should at least understand how to use *tcpdump* and what it can show you. Otherwise, network debugging will very difficult for you.

1.8.2. Linux operating system compatible hardware

Surely you wish to experiment with real hardware, and not only read this HOWTO to fall asleep here and there. ;-7)

Chapter 2. Basics

2.1. What is IPv6?

IPv6 is a new layer 3 protocol (see linuxports/howto/intro_to_networking/ISO - OSI Model) which will supersede IPv4 (also known as IP). IPv4 was designed long time ago (RFC 760 / Internet Protocol from January 1980) and since its inception, there have been many requests for more addresses and enhanced capabilities. Latest RFC is RFC 2460 / Internet Protocol Version 6 Specification. Major changes in IPv6 are the redesign of the header, including the increase of address size from 32 bits to 128 bits. Because layer 3 is responsible for end-to-end packet transport using packet routing based on addresses, it must include the new IPv6 addresses (source and destination), like IPv4.

For more information about the IPv6 history take a look at older IPv6 related RFCs listed e.g. at SWITCH IPv6 Pilot / References.

2.2. History of IPv6 in Linux

The years 1992, 1993 and 1994 of the IPv6 History (in general) are covered by following document: IPv6 or IPng (IP next generation).

To-do: better time-line, more content...

2.2.1. Beginning

The first IPv6 related network code was added to the Linux kernel 2.1.8 in November 1996 by Pedro Roque. It was based on the BSD API:

```
diff -u --recursive --new-file v2.1.7/linux/include/linux/in6.h
Â¬ linux/include/linux/in6.h
--- v2.1.7/linux/include/linux/in6.h Thu Jan 1 02:00:00 1970
+++ linux/include/linux/in6.h Sun Nov 3 11:04:42 1996
@@ -0,0 +1,99 @@
+/*
+ * Types and definitions for AF_INET6
+ * Linux INET6 implementation
+ * + * Authors:
+ * Pedro Roque <******>
+ *
+ * Source:
+ * IPv6 Program Interfaces for BSD Systems
+ * <draft-ietf-ipngwg-bsd-api-05.txt>
```

The shown lines were copied from patch-2.1.8 (e-mail address was blanked on copy&paste).

2.2.2. In between

Because of lack of manpower, the IPv6 implementation in the kernel was unable to follow the discussed drafts or newly released RFCs. In October 2000, a project was started in Japan, called USAGI, whose aim was to implement all missing, or outdated IPv6 support in Linux. It tracks the current IPv6 implementation in FreeBSD made by the KAME project. From time to time they create snapshots against current vanilla Linux kernel sources.

Until kernel development series 2.5.x was started, the USAGI patch was so big, that Linux networking maintainers were unable to include it completly in the production source of the Linux kernel 2.4.x series.

During kernel development series 2.5.x, USAGI tried to insert all of their current extensions into this.

Some, but not all of them were backpported to series 2.4.x and therefore missing some (many) extensions and also does not confirm to all current drafts and RFCs (see IP Version 6 Working Group (ipv6) Charter). This can cause some interoperability problems with other operating systems.

2.2.3. Current

Many of the long-term developed IPv6 related patches by USAGI and others are integrated into vanilla kernel series 2.6.x.

2.2.4. Future

USAGI and others are still working on implementation of newer features like mobililty and others. From time to time, new extension patches are released and also integration into vanilla kernel series is made.

2.3. What do IPv6 addresses look like?

As previously mentioned, IPv6 addresses are 128 bits long. This number of bits generates very high decimal numbers with up to 39 digits:

```
2^128-1: 340282366920938463463374607431768211455
```

Such numbers are not really addresses that can be memorized. Also the IPv6 address schema is bitwise orientated (just like IPv4, but that's not often recognized). Therefore a better notation of such big numbers is hexadecimal. In hexadecimal, 4 bits (also known as "nibble") are represented by a digit or character from 0-9 and a-f (10-15). This format reduces the length of the IPv6 address to 32 characters.

```
2^128-1: 0xffffffffffffffffffffffffffffffff
```

This representation is still not very convenient (possible mix-up or loss of single hexadecimal digits), so the designers of IPv6 chose a hexadecimal format with a colon as separator after each block of 16 bits. In addition, the leading "0x" (a signifier for hexadecimal values used in programming languages) is removed:

```
2^128-1: ffff:ffff:ffff:ffff:ffff:ffff:ffff:ffff
```

A usable address (see address types later) is e.g.:

```
2001:0db8:0100:f101:0210:a4ff:fee3:9566
```

For simplifications, leading zeros of each 16 bit block can be omitted:

```
2001:0db8:0100:f101:0210:a4ff:fee3:9566  ->
Â¬ 2001:db8:100:f101:210:a4ff:fee3:9566
```

One sequence of 16 bit blocks containing only zeroes can be replaced with "::". But not more than one at a time, otherwise it is no longer a unique representation.

```
2001:0db8:100:f101:0:0:0:1  ->  2001:db8:100:f101::1
```

The biggest reduction is seen by the IPv6 localhost address:

```
0000:0000:0000:0000:0000:0000:0000:0001  ->  ::1
```

There is also a so-called *compact* (base85 coded) representation defined RFC 1924 / A Compact Representation of IPv6 Addresses (published on 1. April 1996), never seen in the wild, probably an April fool's joke, but here is an example:

```
# ipv6calc --addr_to_base85 2001:0db8:0100:f101:0210:a4ff:fee3:9566
9R}vSQZ1W=9A_Q74Lz&R
```

> Info: *ipv6calc* is an IPv6 address format calculator and converter program and can be found here: ipv6calc homepage (Mirror)

2.4. FAQ (Basics)

2.4.1. Why is the name IPv6 and not IPv5 as successor for IPv4?

On any IP header, the first 4 bits are reserved for protocol version. So theoretically a protocol number between 0 and 15 is possible:

- 4: is already used for IPv4
- 5: is reserved for the Stream Protocol (STP, RFC 1819 / Internet Stream Protocol Version 2) (which never really made it to the public)

The next free number was 6. Hence IPv6 was born!

2.4.2. IPv6 addresses: why such a high number of bits?

During the design of IPv4, people thought that 32 bits were enough for the world. Looking back into the past, 32 bits were enough until now and will perhaps be enough for another few years. However, 32 bits are not enough to provide each network device with a global address in the future. Think about mobile phones, cars (including electronic devices on its CAN-bus), toasters, refrigerators, light switches, and so on...

So designers have chosen 128 bits, 4 times more in length than in IPv4 today.

The usable size is smaller than it may appear however. This is because in the currently defined address schema, 64 bits are used for interface identifiers. The other 64 bits are used for routing. Assuming the current strict levels of aggregation (/48, /32, ...), it is still possible to "run out" of space, but hopefully not in the near future.

See also for more information RFC 1715 / The H Ratio for Address Assignment Efficiency and RFC 3194 / The Host-Density Ratio for Address Assignment Efficiency.

2.4.3. IPv6 addresses: why so small a number of bits on a new design?

While, there are (possibly) some people (only know about Jim Fleming...) on the Internet who are thinking about IPv8 and IPv16, their design is far away from acceptance and implementation. In the meantime 128 bits was the best choice regarding header overhead and data transport. Consider the minimum Maximum Transfer Unit (MTU) in IPv4 (576 octets) and in IPv6 (1280 octets), the header length in IPv4 is 20 octets (minimum, can increase to 60 octets with IPv4 options) and in IPv6 is 48 octets (fixed). This is 3.4 % of MTU in IPv4 and 3.8 % of MTU in IPv6. This means the header overhead is almost equal. More bits for addresses would require bigger headers and therefore more overhead. Also, consider the maximum MTU on normal links (like Ethernet today): it's 1500 octets (in special cases: 9k octets using Jumbo frames). Ultimately, it wouldn't be a

proper design if 10 % or 20 % of transported data in a Layer-3 packet were used for addresses and not for payload.

Chapter 3. Address types

Like IPv4, IPv6 addresses can be split into network and host parts using subnet masks.

IPv4 has shown that sometimes it would be nice, if more than one IP address can be assigned to an interface, each for a different purpose (aliases, multi-cast). To remain extensible in the future, IPv6 is going further and allows more than one IPv6 address to be assigned to an interface. There is currently no limit defined by an RFC, only in the implementation of the IPv6 stack (to prevent DoS attacks).

Using this large number of bits for addresses, IPv6 defines address types based on some leading bits, which are hopefully never going to be broken in the future (unlike IPv4 today and the history of class A, B, and C).

Also the number of bits are separated into a network part (upper 64 bits) and a host part (lower 64 bits), to facilitate auto-configuration.

3.1. Addresses without a special prefix

3.1.1. Localhost address

This is a special address for the loopback interface, similiar to IPv4 with its "127.0.0.1". With IPv6, the localhost address is:

```
0000:0000:0000:0000:0000:0000:0000:0001
```

or compressed:

```
::1
```

Packets with this address as source or destination should never leave the sending host.

3.1.2. Unspecified address

This is a special address like "any" or "0.0.0.0" in IPv4 . For IPv6 it's:

```
0000:0000:0000:0000:0000:0000:0000:0000
```

or:

```
::
```

These addresses are mostly used/seen in socket binding (to any IPv6 address) or routing tables.

Note: the unspecified address cannot be used as destination address.

3.1.3. IPv6 address with embedded IPv4 address

There are two addresses which contain an IPv4 address.

3.1.3.1. IPv4-mapped IPv6 address

IPv4-only IPv6-compatible addresses are sometimes used/shown for sockets created by an IPv6-enabled daemon, but only binding to an IPv4 address.

These addresses are defined with a special prefix of length 96 (a.b.c.d is the IPv4 address):

```
0:0:0:0:0:ffff:a.b.c.d/96
```

or in compressed format

```
::ffff:a.b.c.d/96
```

For example, the IPv4 address 1.2.3.4 looks like this:

```
::ffff:1.2.3.4
```

3.1.3.2. IPv4-compatible IPv6 address

Used for automatic tunneling (RFC 2893 / Transition Mechanisms for IPv6 Hosts and Routers), which is being replaced by 6to4 tunneling.

```
0:0:0:0:0:0:a.b.c.d/96
```

or in compressed format

```
::a.b.c.d/96
```

3.2. Network part, also known as prefix

Designers defined some address types and left a lot of scope for future definitions as currently unknown requirements arise. RFC 4291 / IP Version 6 Addressing Architecture defines the current addressing scheme.

Now lets take a look at the different types of prefixes (and therefore address types):

3.2.1. Link local address type

These are special addresses which will only be valid on a link of an interface. Using this address as destination the packet would never pass through a router. It's used for link communications such as:

- anyone else here on this link?
- anyone here with a special address (e.g. looking for a router)?

They begin with (where *"x"* is any hex character, normally *"0*")

```
fe8x:  <- currently the only one in use.
fe9x:
feax:
febx:
```

An address with this prefix is found on each IPv6-enabled interface after stateless auto-configuration (which is normally always the case).

3.2.2. Site local address type

These are addresses similar to the RFC 1918 / Address Allocation for Private Internets in IPv4 today, with the added advantage that everyone who use this address type has the capability to use the given 16 bits for a maximum number of 65536 subnets. Comparable with the 10.0.0.0/8 in IPv4 today.

Another advantage: because it's possible to assign more than one address to an interface with IPv6, you can also assign such a site local address in addition to a global one.

It begins with:

```
fecx:  <- most commonly used
fedx:
feex:
fefx:
```

(where *"x"* is any hex character, normally *"0"*)

This address type is now deprecated RFC 3879 / Deprecating Site Local Addresses, but for a test in a lab, such addresses are still a good choice in my humble opinion.

3.2.3. Unique Local IPv6 Unicast Addresses

Because the original defined site local addresses are not unique, this can lead to major problems, if two former independend networks would be connected later (overlapping of subnets). This and other issues lead to a new address type named RFC 4193 / Unique Local IPv6 Unicast Addresses.

It begins with:

```
fdxx:
fcxx:
```

A part of the prefix (40 bits) are generated using a pseudo-random algorithm and it's improbable, that two generated ones are equal.

Example for a prefix (generated using a web-based tool: Goebel Consult / createLULA):

```
fd0f:8b72:ac90::/48
```

3.2.4. Global address type "(Aggregatable) global unicast"

Today, there is one global address type defined (the first design, called "provider based," was thrown away some years ago RFC 1884 / IP Version 6 Addressing Architecture [obsolete], you will find some remains in older Linux kernel sources).

It begins with (*x* are hex characters)

```
2xxx:
3xxx:
```

Note: the prefix "aggregatable" is thrown away in current drafts. There are some further subtypes defined, see below:

3.2.4.1. 6bone test addresses

These were the first global addresses which were defined and in use. They all start with

```
3ffe:
```

Example:

```
3ffe:ffff:100:f102::1
```

A special 6bone test address which will never be globally unique begins with

```
3ffe:ffff:
```

and is mostly shown in older examples. The reason for this is, if real addresses are are shown, it's possible for someone to do a copy & paste to their configuration files, thus inadvertently causing duplicates on a globally unique address. This would cause serious problems for the original host (e.g. getting answer packets for request that were never sent). Because IPv6 is now in production, this prefix is no longer be delegated and is removed from routing after 6.6.2006 (see RFC 3701 / 6bone Phaseout for more).

3.2.4.2. 6to4 addresses

These addresses, designed for a special tunneling mechanism [RFC 3056 / Connection of IPv6 Domains via IPv4 Clouds and RFC 2893 / Transition Mechanisms for IPv6 Hosts and Routers], encode a given IPv4 address and a possible subnet and begin with

```
2002:
```

For example, representing 192.168.1.1/5:

```
2002:c0a8:0101:5::1
```

A small shell command line can help you generating such address out of a given IPv4 one:

```
ipv4="1.2.3.4"; sla="5"; printf "2002:%02x%02x:%02x%02x:%04x::1" `echo $ipv4
Â¬ | tr "." " "` $sla
```

See also tunneling using 6to4 and information about 6to4 relay routers.

3.2.4.3. Assigned by provider for hierarchical routing

These addresses are delegated to Internet service providers (ISP) and begin currently with

```
2001:
```

Prefixes to major (backbone owning) ISPs (also known as LIRs) are delegated by local registries and currently have a prefix with length 32 assigned.

Any ISP customer can get a prefix with length 48.

3.2.4.4. Addresses reserved for examples and documentation

Currently, two address ranges are reserved for examples and documentation RFC 3849 / IPv6 Address Prefix Reserved for Documentation:

```
3fff:ffff::/32
2001:0DB8::/32   EXAMPLENET-WF
```

These address ranges should be filtered based on source addresses and should NOT be routed on border routers to the internet, if possible.

3.2.5. Multicast addresses

Multicast addresses are used for related services.

They alway start with (*xx* is the scope value)

```
ffxy:
```

They are split into scopes and types:

3.2.5.1. Multicast scopes

Multicast scope is a parameter to specify the maximum distance a multicast packet can travel from the sending entity.

Currently, the following regions (scopes) are defined:

- ffx1: node-local, packets never leave the node.
- ffx2: link-local, packets are never forwarded by routers, so they never leave the specified link.
- ffx5: site-local, packets never leave the site.
- ffx8: organization-local, packets never leave the organization (not so easy to implement, must be covered by routing protocol).
- ffxe: global scope.
- others are reserved

3.2.5.2. Multicast types

There are many types already defined/reserved (see RFC 4291 / IP Version 6 Addressing Architecture for details). Some examples are:

- All Nodes Address: ID = 1h, addresses all hosts on the local node (ff01:0:0:0:0:0:0:1) or the connected link (ff02:0:0:0:0:0:0:1).
- All Routers Address: ID = 2h, addresses all routers on the local node (ff01:0:0:0:0:0:0:2), on the connected link (ff02:0:0:0:0:0:0:2), or on the local site (ff05:0:0:0:0:0:0:2)

3.2.5.3. Solicited node link-local multicast address

Special multicast address used as destination address in neighborhood discovery, because unlike in IPv4, ARP no longer exists in IPv6.

An example of this address looks like

```
ff02::1:ff00:1234
```

Used prefix shows that this is a link-local multicast address. The suffix is generated from the destination address. In this example, a packet should be sent to address "fe80::1234", but the network stack doesn't know the current layer 2 MAC address. It replaces the upper 104 bits with "ff02:0:0:0:0:1:ff00::/104" and leaves the lower 24 bits untouched. This address is now used `on-link' to find the corresponding node which has to send a reply containing its layer 2 MAC address.

3.2.6. Anycast addresses

Anycast addresses are special addresses and are used to cover things like nearest DNS server, nearest DHCP server, or similar dynamic groups. Addresses are taken out of the unicast address space (aggregatable global or site-local at the moment). The anycast mechanism (client view) will be handled by dynamic routing protocols.

Note: Anycast addresses cannot be used as source addresses, they are only used as destination addresses.

3.2.6.1. Subnet-router anycast address

A simple example for an anycast address is the subnet-router anycast address. Assuming that a node has the following global assigned IPv6 address:

```
2001:db8:100:f101:210:a4ff:fee3:9566/64  <- Node's address
```

The subnet-router anycast address will be created blanking the suffix (least significant 64 bits) completely:

```
2001:db8:100:f101::/64  <- subnet-router anycast address
```

3.3. Address types (host part)

For auto-configuration and mobility issues, it was decided to use the lower 64 bits as the host part of the address in most of the current address types. Therefore each single subnet can hold a large amount of addresses.

This host part can be inspected differently:

3.3.1. Automatically computed (also known as stateless)

With auto-configuration, the host part of the address is computed by converting the MAC address of an interface (if available), with the EUI-64 method, to a unique IPv6 address. If no MAC address is available for this device (happens e.g. on virtual devices), something else (like the IPv4 address or the MAC address of a physical interface) is used instead.

E.g. a NIC has following MAC address (48 bit):

```
00:10:A4:E3:95:66
```

This would be expanded according to theIEEE-Tutorial EUI-64 design for EUI-48 identifiers to the 64 bit interface identifier:

```
0210:a4ff:fee3:9566
```

With a given prefix, the result is the IPv6 address shown in example above:

```
2001:0db8:0100:f101:0210:a4ff:fee3:9566
```

3.3.1.1. Privacy problem with automatically computed addresses and a solution

Because the "automatically computed" host part is globally unique (except when a vendor of a NIC uses the same MAC address on more than one NIC), client tracking is possible on the host when not using a proxy of any kind.

This is a known problem, and a solution was defined: privacy extension, defined in RFC 3041 / Privacy Extensions for Stateless Address Autoconfiguration in IPv6 (there is also already a newer draft available: draft-ietf-ipv6-privacy-addrs-v2-*). Using a random and a static value a new suffix is generated from time to time. Note: this is only reasonable for outgoing client connections and isn't really useful for well-known servers.

3.3.2. Manually set

For servers, it's probably easier to remember simpler addresses, this can also be accommodated. It is possible to assign an additional IPv6 address to an interface, e.g.

```
2001:0db8:100:f101::1
```

For manual suffixes like "::1" shown in the above example, it's required that the 7th most significant bit is set to 0 (the universal/local bit of the automatically generated identifier). Also some other (otherwise unchosen) bit combinations are reserved for anycast addresses, too.

3.4. Prefix lengths for routing

In the early design phase it was planned to use a fully hierarchical routing approach to reduce the size of the routing tables maximally. The reasons behind this approach were the number of current IPv4 routing entries in core routers (> 104 thousand in May 2001), reducing the need of memory in hardware routers (ASIC "Application Specified Integrated Circuit" driven) to hold the routing table and increase speed (fewer entries hopefully result in faster lookups).

Todays view is that routing will be mostly hierarchically designed for networks with only one service provider. With more than one ISP connections, this is not possible, and subject to an issue named multi-homing (infos on multi-homing: drafts-ietf-multi6-*,IPv6 Multihoming Solutions).

3.4.1. Prefix lengths (also known as "netmasks")

Similar to IPv4, the routable network path for routing to take place. Because standard netmask notation for 128 bits doesn't look nice, designers employed the IPv4 Classless Inter Domain Routing (CIDR, RFC 1519 / Classless Inter-Domain Routing) scheme, which specifies the number of bits of the IP address to be used for routing. It is also called the "slash" notation.

An example:

```
2001:0db8:100:1:2:3:4:5/48
```

This notation will be expanded:

- Network:

```
2001:0db8:0100:0000:0000:0000:0000:0000
```

- Netmask:

```
ffff:ffff:ffff:0000:0000:0000:0000:0000
```

3.4.2. Matching a route

Under normal circumstances (no QoS), a lookup in a routing table results in the route with the most significant number of address bits being selected. In other words, the route with the biggest prefix length matches first.

For example if a routing table shows following entries (list is not complete):

```
2001:0db8:100::/48     ::            U  1 0 0 sit1
2000::/3               ::192.88.99.1 UG 1 0 0 tun6to4
```

Shown destination addresses of IPv6 packets will be routed through shown device

```
2001:0db8:100:1:2:3:4:5/48  ->  routed through device sit1
2001:0db8:200:1:2:3:4:5/48  ->  routed through device tun6to4
```

Chapter 4. IPv6-ready system check

Before you can start using IPv6 on a Linux host, you have to test, whether your system is IPv6-ready. You may have to do some work to enable it first.

4.1. IPv6-ready kernel

Modern Linux distributions already contain IPv6-ready kernels, the IPv6 capability is generally compiled as a module, but it's possible that this module is not loaded automatically on startup.

Note: you shouldn't anymore use kernel series 2.2.x, because it's not IPv6-up-to-date anymore. Also the IPv6 support in series 2.4.x is no longer improved according to definitions in latest RFCs. It's recommend to use series 2.6.x now.

4.1.1. Check for IPv6 support in the current running kernel

To check, whether your current running kernel supports IPv6, take a look into your /proc-file-system. Following entry must exists:

```
/proc/net/if_inet6
```

A short automatical test looks like:

```
# test -f /proc/net/if_inet6 && echo "Running kernel is IPv6 ready"
```

If this fails, it is quite likely, that the IPv6 module is not loaded.

4.1.2. Try to load IPv6 module

You can try to load the IPv6 module executing

```
# modprobe ipv6
```

If this is successful, this module should be listed, testable with following auto-magically line:

```
# lsmod |grep -w 'ipv6' && echo "IPv6 module successfully loaded"
```

And the check shown above should now run successfully.

Note: unloading the module is currently not supported and can result, under some circumstances, in a kernel crash.

4.1.2.1. Automatically loading of module

Its possible to automatically load the IPv6 module on demand. You only have to add following line in the configuration file of the kernel module loader (normally /etc/modules.conf or /etc/conf.modules):

```
alias net-pf-10 ipv6  # automatically load IPv6 module on demand
```

It's also possible to disable automatically loading of the IPv6 module using following line

```
alias net-pf-10 off   # disable automatically load of IPv6 module on demand
```

Additional note: in kernels series 2.6.x, the module loader mechanism was changed. The new configuration file has to be named /etc/modprobe.conf instead of /etc/modules.conf.

4.1.3. Compile kernel with IPv6 capabilities

If both above shown results were negative and your kernel has no IP6 support, than you have the following options:

- Update your distribution to a current one which supports IPv6 out-of-the-box (recommended for newbies)
- Compile a new vanilla kernel (easy, if you know which options you needed)
- Recompile kernel sources given by your Linux distribution (sometimes not so easy)
- Compile a kernel with USAGI extensions

If you decide to compile a kernel, you should have previous experience in kernel compiling and read the Linux Kernel HOWTO.

A comparison between vanilla and USAGI extended kernels is available on IPv6+Linux-Status-Kernel.

4.1.3.1. Compiling a vanilla kernel

More detailed hints about compiling an IPv6-enabled kernel can be found e.g. on IPv6-HOWTO-2#kernel.

Note: you should use whenever possible kernel series 2.6.x or above, because the IPv6 support in series 2.4.x only will no longer get backported features from 2.6.x and IPv6 support in series 2.2.x is hopeless outdated.

4.1.3.2. Compiling a kernel with USAGI extensions

Same as for vanilla kernel, only recommend for advanced users, which are already familiar with IPv6 and kernel compilation. See also USAGI project / FAQ and Obtaining the best IPv6 support with Linux (Article) (Mirror).

4.1.4. IPv6-ready network devices

Not all existing network devices have already (or ever) the capability to transport IPv6 packets. A current status can be found at IPv6+Linux-status-kernel.html#transport.

A major issue is that because of the network layer structure of kernel implementation an IPv6 packet isn't really recognized by it's IP header number (6 instead of 4). It's recognized by the protocol number of the Layer 2 transport protocol. Therefore any transport protocol which doesn't use such protocol number cannot dispatch IPv6 packets. Note: the packet is still transported over the link, but on receivers side, the dispatching won't work (you can see this e.g. using tcpdump).

4.1.4.1. Currently known never "IPv6 capable links"

- Serial Line IP (SLIP, RFC 1055 / SLIP), should be better called now to SLIPv4, device named: slX
- Parallel Line IP (PLIP), same like SLIP, device names: plipX
- ISDN with encapsulation *rawip*, device names: isdnX

4.1.4.2. Currently known "not supported IPv6 capable links"

- ISDN with encapsulation *syncppp*, device names: ipppX (design issue of the ipppd, will be merged into more general PPP layer in kernel series 2.5.x)

4.2. IPv6-ready network configuration tools

You wont get very far, if you are running an IPv6-ready kernel, but have no tools to configure IPv6. There are several packages in existence which can configure IPv6.

4.2.1. net-tools package

The net-tool package includes some tools like ifconfig and route, which helps you to configure IPv6 on an interface. Look at the output of ifconfig -? or route -?, if something is shown like IPv6 or inet6, then the tool is IPv6-ready.

Auto-magically check:

```
# /sbin/ifconfig –? 2>& 1|grep –qw 'inet6' && echo "utility 'ifconfig' is
Â¬ IPv6–ready"
```

Same check can be done for route:

```
# /sbin/route –? 2>& 1|grep –qw 'inet6' && echo "utility 'route' is IPv6–ready"
```

4.2.2. iproute package

Alexey N. Kuznetsov (current a maintainer of the Linux networking code) created a tool-set which configures networks through the netlink device. Using this tool-set you have more functionality than net-tools provides, but its not very well documented and isn't for the faint of heart.

```
# /sbin/ip 2>&1 |grep –qw 'inet6' && echo "utility 'ip' is IPv6–ready"
```

If the program /sbin/ip isn't found, then I strongly recommend you install the iproute package.

- You can get it from your Linux distribution (if contained)
- You can download the tar-ball and recompile it: Original FTP source and mirror (missing)
- You're able to look for a proper RPM package at RPMfind/iproute (sometimes rebuilding of a SRPMS package is recommended)

4.3. IPv6-ready test/debug programs

After you have prepared your system for IPv6, you now want to use IPv6 for network communications. First you should learn how to examine IPv6 packets with a sniffer program. This is strongly recommended because for debugging/troubleshooting issues this can aide in providing a diagnosis very quickly.

4.3.1. IPv6 ping

This program is normally included in package *iputils*. It is designed for simple transport tests sending ICMPv6 echo-request packets and wait for ICMPv6 echo-reply packets.

Usage

```
# ping6 <hostwithipv6address>
# ping6 <ipv6address>
# ping6 [-I <device>] <link-local-ipv6address>
```

Example

```
# ping6 -c 1 ::1
PING ::1(::1) from ::1 : 56 data bytes
64 bytes from ::1: icmp_seq=0 hops=64 time=292 usec

--- ::1 ping statistics ---
1 packets transmitted, 1 packets received, 0% packet loss
round-trip min/avg/max/mdev = 0.292/0.292/0.292/0.000 ms
```

Hint: ping6 needs raw access to socket and therefore root permissions. So if non-root users cannot use ping6 then there are two possible problems:

1. ping6 is not in users path (probably, because ping6 is generally stored in /usr/sbin -> add path (not really recommended)
2. ping6 doesn't execute properly, generally because of missing root permissions -> chmod u+s /usr/sbin/ping6

4.3.1.1. Specifying interface for IPv6 ping

Using link-local addresses for an IPv6 ping, the kernel does not know through which (physically or virtual) device it must send the packet - each device has a link-local address. A try will result in following error message:

```
# ping6 fe80::212:34ff:fe12:3456
connect: Invalid argument
```

In this case you have to specify the interface additionally like shown here:

```
# ping6 -I eth0 -c 1 fe80::2e0:18ff:fe90:9205
PING fe80::212:23ff:fe12:3456(fe80::212:23ff:fe12:3456) from
Â¬ fe80::212:34ff:fe12:3478 eth0: 56 data bytes
64 bytes from fe80::212:23ff:fe12:3456: icmp_seq=0 hops=64 time=445 usec

--- fe80::2e0:18ff:fe90:9205 ping statistics ---
1 packets transmitted, 1 packets received, 0% packet loss round-trip
Â¬ min/avg/max/mdev = 0.445/0.445/0.445/0.000 ms
```

4.3.1.2. Ping6 to multicast addresses

An interesting mechanism to detect IPv6-active hosts on a link is to ping6 to the link-local all-node multicast address:

```
# ping6 -I eth0 ff02::1
PING ff02::1(ff02::1) from fe80:::2ab:cdff:feef:0123 eth0: 56 data bytes
64 bytes from ::1: icmp_seq=1 ttl=64 time=0.104 ms
64 bytes from fe80::212:34ff:fe12:3450: icmp_seq=1 ttl=64 time=0.549 ms (DUP!)
```

Unlike in IPv4, where replies to a ping on the broadcast address can be disabled, in IPv6 currently this behavior cannot be disable except by local IPv6 firewalling.

4.3.2. IPv6 traceroute6

This program is normally included in package *iputils*. It's a program similar to IPv4 traceroute. Below you will see an example:

```
# traceroute6 www.6bone.net
traceroute to 6bone.net (3ffe:b00:c18:1::10) from 2001:0db8:0000:f101::2, 30
Â¬ hops max, 16 byte packets
 1 localipv6gateway (2001:0db8:0000:f101::1) 1.354 ms 1.566 ms 0.407 ms
 2 swi6T1-T0.ipv6.switch.ch (3ffe:2000:0:400::1) 90.431 ms 91.956 ms 92.377 ms
 3 3ffe:2000:0:1::132 (3ffe:2000:0:1::132) 118.945 ms 107.982 ms 114.557 ms
 4 3ffe:c00:8023:2b::2 (3ffe:c00:8023:2b::2) 968.468 ms 993.392 ms 973.441 ms
 5 3ffe:2e00:e:c::3 (3ffe:2e00:e:c::3) 507.784 ms 505.549 ms 508.928 ms
 6 www.6bone.net (3ffe:b00:c18:1::10) 1265.85 ms * 1304.74 ms
```

Note: unlike some modern versions of IPv4 traceroute, which can use ICMPv4 echo-request packets as well as UDP packets (default), current IPv6-traceroute is only able to send UDP packets. As you perhaps already know, ICMP echo-request packets are more accepted by firewalls or ACLs on routers inbetween than UDP packets.

4.3.3. IPv6 tracepath6

This program is normally included in package *iputils*. It's a program like traceroute6 and traces the path to a given destination discovering the MTU along this path. Below you will see an example:

```
# tracepath6 www.6bone.net
 1?: [LOCALHOST] pmtu 1480
 1: 3ffe:401::2c0:33ff:fe02:14 150.705ms
 2: 3ffe:b00:c18::5 267.864ms
 3: 3ffe:b00:c18::5 asymm 2 266.145ms pmtu 1280
 3: 3ffe:3900:5::2 asymm 4 346.632ms
 4: 3ffe:28ff:ffff:4::3 asymm 5 365.965ms
 5: 3ffe:1cff:0:ee::2 asymm 4 534.704ms
 6: 3ffe:3800::1:1 asymm 4 578.126ms !N
Resume: pmtu 1280
```

4.3.4. IPv6 tcpdump

On Linux, tcpdump is the major tool for packet capturing. Below you find some examples. IPv6 support is normally built-in in current releases of version 3.6.

tcpdump uses expressions for filtering packets to minimize the noise:

- icmp6: filters native ICMPv6 traffic
- ip6: filters native IPv6 traffic (including ICMPv6)
- proto ipv6: filters tunneled IPv6-in-IPv4 traffic
- not port ssh: to suppress displaying SSH packets for running tcpdump in a remote SSH session

Also some command line options are very useful to catch and print more information in a packet, mostly interesting for digging into ICMPv6 packets:

- "-s 512": increase the snap length during capturing of a packet to 512 bytes
- "-vv": really verbose output
- "-n": don't resolve addresses to names, useful if reverse DNS resolving isn't working proper

4.3.4.1. IPv6 ping to 2001:0db8:100:f101::1 native over a local link

```
# tcpdump -t -n -i eth0 -s 512 -vv ip6 or proto ipv6
tcpdump: listening on eth0
2001:0db8:100:f101:2e0:18ff:fe90:9205 > 2001:0db8:100:f101::1: icmp6: echo
Â¬ request (len 64, hlim 64)
2001:0db8:100:f101::1 > 2001:0db8:100:f101:2e0:18ff:fe90:9205: icmp6: echo
Â¬ reply (len 64, hlim 64)
```

4.3.4.2. IPv6 ping to 2001:0db8:100::1 routed through an IPv6-in-IPv4-tunnel

1.2.3.4 and 5.6.7.8 are tunnel endpoints (all addresses are examples)

```
# tcpdump -t -n -i ppp0 -s 512 -vv ip6 or proto ipv6
tcpdump: listening on ppp0
1.2.3.4 > 5.6.7.8: 2002:ffff:f5f8::1 > 2001:0db8:100::1: icmp6: echo request
Â¬ (len 64, hlim 64) (DF) (ttl 64, id 0, len 124)
5.6.7.8 > 1.2.3.4: 2001:0db8:100::1 > 2002:ffff:f5f8::1: icmp6: echo reply (len
Â¬ 64, hlim 61) (ttl 23, id 29887, len 124)
1.2.3.4 > 5.6.7.8: 2002:ffff:f5f8::1 > 2001:0db8:100::1: icmp6: echo request
Â¬ (len 64, hlim 64) (DF) (ttl 64, id 0, len 124)
5.6.7.8 > 1.2.3.4: 2001:0db8:100::1 > 2002:ffff:f5f8::1: icmp6: echo reply (len
Â¬ 64, hlim 61) (ttl 23, id 29919, len 124)
```

4.4. IPv6-ready programs

Current distributions already contain the most needed IPv6 enabled client and servers. See first on IPv6+Linux-Status-Distribution. If still not included, you can check IPv6 & Linux - Current Status - Applications whether the program is already ported to IPv6 and usable with Linux. For common used programs there are some hints available at IPv6 & Linux - HowTo - Part 3 and IPv6 & Linux - HowTo - Part 4.

4.5. IPv6-ready client programs (selection)

To run the following shown tests, it's required that your system is IPv6 enabled, and some examples show addresses which only can be reached if a connection to the 6bone is available.

4.5.1. Checking DNS for resolving IPv6 addresses

Because of security updates in the last years every Domain Name System (DNS) server should run newer software which already understands the (intermediate) IPv6 address-type AAAA (the newer one named A6 isn't still common at the moment because only supported using BIND9 and newer and also the non-existent support of root domain IP6.ARPA). A simple test whether the used system can resolve IPv6 addresses is

```
# host -t AAAA www.join.uni-muenster.de
```

and should show something like following:

```
www.join.uni-muenster.de. is an alias for tolot.join.uni-muenster.de.
tolot.join.uni-muenster.de. has AAAA address 2001:638:500:101:2e0:81ff:fe24:37c6
```

4.5.2. IPv6-ready telnet clients

IPv6-ready telnet clients are available. A simple test can be done with

```
$ telnet 3ffe:400:100::1 80
Trying 3ffe:400:100::1...
Connected to 3ffe:400:100::1.
Escape character is '^]'.
HEAD / HTTP/1.0

HTTP/1.1 200 OK
Date: Sun, 16 Dec 2001 16:07:21
GMT Server: Apache/2.0.28 (Unix)
Last-Modified: Wed, 01 Aug 2001 21:34:42 GMT
ETag: "3f02-a4d-b1b3e080"
Accept-Ranges: bytes
Content-Length: 2637
Connection: close
Content-Type: text/html; charset=ISO-8859-1

Connection closed by foreign host.
```

If the telnet client don't understand the IPv6 address and says something like "cannot resolve hostname", then it's not IPv6-enabled.

4.5.3. IPv6-ready ssh clients

4.5.3.1. openssh

Current versions of openssh are IPv6-ready. Depending on configuring before compiling it has two behavior.

- --without-ipv4-default: the client tries an IPv6 connect first automatically and fall back to IPv4 if not working
- --with-ipv4-default: default connection is IPv4, IPv6 connection must be force like following example shows

```
$ ssh -6 ::1
user@::1's password: ******
[user@ipv6host user]$
```

If your ssh client doesn't understand the option "-6" then it's not IPv6-enabled, like most ssh version 1 packages.

4.5.3.2. ssh.com

SSH.com's SSH client and server is also IPv6 aware now and is free for all Linux and FreeBSD machine regardless if used for personal or commercial use.

4.5.4. IPv6-ready web browsers

A current status of IPv6 enabled web browsers is available at IPv6+Linux-status-apps.html#HTTP.

Most of them have unresolved problems at the moment

1. If using an IPv4 only proxy in the settings, IPv6 requests will be sent to the proxy, but the proxy will fail to understand the request and the request fails. Solution: update proxy software (see later).

2. Automatic proxy settings (*.pac) cannot be extended to handle IPv6 requests differently (e.g. don't use proxy) because of their nature (written in Java-script and well hard coded in source like to be seen in Maxilla source code).

Also older versions don't understand an URL with IPv6 encoded addresses like http://[3ffe:400:100::1]/ (this given URL only works with an IPv6-enabled browser!).

A short test is to try shown URL with a given browser and using no proxy.

4.5.4.1. URLs for testing

A good starting point for browsing using IPv6 is http://www.kame.net/. If the turtle on this page is animated, the connection is via IPv6, otherwise the turtle is static.

4.6. IPv6-ready server programs

In this part of this HOWTO, more client specific issues are mentioned. Therefore hints for IPv6-ready servers like sshd, httpd, telnetd, etc. are shown below in Hints for IPv6-enabled daemons.

4.7. FAQ (IPv6-ready system check)

4.7.1. Using tools

4.7.1.1. Q: Cannot ping6 to link-local addresses

Error message: "*connect: Invalid argument*"

Kernel doesn't know, which physical or virtual link you want to use to send such ICMPv6 packets. Therefore it displays this error message.

Solution: Specify interface like: "ping6 -I eth0 fe80::2e0:18ff:fe90:9205", see also program ping6 usage.

4.7.1.2. Q: Cannot ping6 or traceroute6 as normal user

Error message: "*icmp socket: Operation not permitted*"

These utilities create special ICMPv6 packets and send them out. This is done by using raw sockets in the kernel. But raw sockets can only be used by the "root" user. Therefore normal users get such error message.

Solution: If it's really needed that all users should be able to use these utilities, you can add the "suid" bit using "chmod u+s /path/to/program", see also program ping6 usage. If not all users should be able to, you can change the group of the program to e.g. "wheel", add these power users to this group and remove the execution bit for other users using "chmod o-rwx /path/to/program". Or configure "sudo" to enable your security policy.

Chapter 5. Configuring interfaces

5.1. Different network devices

On a node, there exist different network devices. They can be collected in classes

- Physically bounded, like eth0, tr0
- Virtually existing, like ppp0, tun0, tap0, sit0, isdn0, ippp0

5.1.1. Physically bounded

Physically bounded interfaces like Ethernet or Token-Ring are normal ones and need no special treatment.

5.1.2. Virtually bounded

Virtually bounded interfaces always need special support

5.1.2.1. IPv6-in-IPv4 tunnel interfaces

These interfaces are normally named sit*x*. The name *sit* is a shortcut for Simple Internet Transition. This device has the capability to encapsulate IPv6 packets into IPv4 ones and tunnel them to a foreign endpoint.

sit0 has a special meaning and cannot be used for dedicated tunnels.

5.1.2.2. PPP interfaces

PPP interfaces get their IPv6 capability from an IPv6 enabled PPP daemon.

5.1.2.3. ISDN HDLC interfaces

IPv6 capability for HDLC with encapsulation ip is already built-in in the kernel

5.1.2.4. ISDN PPP interfaces

ISDN PPP interfaces (ippp) aren't IPv6 enabled by kernel. Also there are also no plans to do that because in kernel 2.5.+ they will be replaced by a more generic ppp interface layer.

5.1.2.5. SLIP + PLIP

Like mentioned earlier, this interfaces don't support IPv6 transport (sending is OK, but dispatching on receiving don't work).

5.1.2.6. Ether-tap device

Ether-tap devices are IPv6-enabled and also stateless configured. For use, the module "ethertap" has to be loaded before.

5.1.2.7. tun devices

Currently not tested by me.

5.1.2.8. ATM

01/2002: Aren't currently supported by vanilla kernel, supported by USAGI extension

5.1.2.9. Others

Did I forget an interface?...

5.2. Bringing interfaces up/down

Two methods can be used to bring interfaces up or down.

5.2.1. Using "ip"

Usage:

```
# ip link set dev <interface> up
# ip link set dev <interface> down
```

Example:

```
# ip link set dev eth0 up
# ip link set dev eth0 down
```

5.2.2. Using "ifconfig"

Usage:

```
# /sbin/ifconfig <interface> up
# /sbin/ifconfig <interface> down
```

Example:

```
# /sbin/ifconfig eth0 up
# /sbin/ifconfig eth0 down
```

Chapter 6. Configuring IPv6 addresses

There are different ways to configure an IPv6 address on an interface. You can use use "ifconfig" or "ip".

6.1. Displaying existing IPv6 addresses

First you should check, whether and which IPv6 addresses are already configured (perhaps auto-magically during stateless auto-configuration).

6.1.1. Using "ip"

Usage:

```
# /sbin/ip -6 addr show dev <interface>
```

Example for a static configured host:

```
# /sbin/ip -6 addr show dev eth0
2: eth0: <BROADCAST,MULTICAST,UP&gt; mtu 1500 qdisc pfifo_ fast qlen 100
inet6 fe80::210:a4ff:fee3:9566/10 scope link
inet6 2001:0db8:0:f101::1/64 scope global
inet6 fec0:0:0:f101::1/64 scope site
```

Example for a host which is auto-configured

Here you see some auto-magically configured IPv6 addresses and their lifetime.

```
# /sbin/ip -6 addr show dev eth0
3: eth0: <BROADCAST,MULTICAST,PROMISC,UP&gt; mtu 1500 qdisc pfifo_fast qlen
Â¬ 100
inet6 2002:d950:f5f8:f101:2e0:18ff:fe90:9205/64 scope global dynamic
valid_lft 16sec preferred_lft 6sec
inet6 3ffe:400:100:f101:2e0:18ff:fe90:9205/64 scope global dynamic
valid_lft 2591997sec preferred_lft 604797sec inet6 fe80::2e0:18ff:fe90:9205/10
Â¬ scope link
```

6.1.2. Using "ifconfig"

Usage:

```
# /sbin/ifconfig <interface>
```

Example (output filtered with grep to display only IPv6 addresses). Here you see different IPv6 addresses with different scopes.

```
# /sbin/ifconfig eth0 |grep "inet6 addr:"
inet6 addr: fe80::210:a4ff:fee3:9566/10 Scope:Link
inet6 addr: 2001:0db8:0:f101::1/64 Scope:Global
inet6 addr: fec0:0:0:f101::1/64 Scope:Site
```

6.2. Add an IPv6 address

Adding an IPv6 address is similar to the mechanism of "IP ALIAS" addresses in Linux IPv4 addressed interfaces.

6.2.1. Using "ip"

Usage:

```
# /sbin/ip -6 addr add <ipv6address>/<prefixlength> dev <interface>
```

Example:

```
# /sbin/ip -6 addr add 2001:0db8:0:f101::1/64 dev eth0
```

6.2.2. Using "ifconfig"

Usage:

```
# /sbin/ifconfig <interface> inet6 add <ipv6address>/<prefixlength>
```

Example:

```
# /sbin/ifconfig eth0 inet6 add 2001:0db8:0:f101::1/64
```

6.3. Removing an IPv6 address

Not so often needed, be carefully with removing non existent IPv6 address, sometimes using older kernels it results in a crash.

6.3.1. Using "ip"

Usage:

```
# /sbin/ip -6 addr del <ipv6address>/<prefixlength> dev <interface>
```

Example:

```
# /sbin/ip -6 addr del 2001:0db8:0:f101::1/64 dev eth0
```

6.3.2. Using "ifconfig"

Usage:

```
# /sbin/ifconfig <interface> inet6 del <ipv6address>/<prefixlength>
```

Example:

```
# /sbin/ifconfig eth0 inet6 del 2001:0db8:0:f101::1/64
```

Chapter 7. Configuring normal IPv6 routes

If you want to leave your link and want to send packets in the world wide IPv6-Internet, you need routing. If there is already an IPv6 enabled router on your link, it's possible enough to add IPv6 routes.

7.1. Displaying existing IPv6 routes

First you should check, whether and which IPv6 addresses are already configured (perhaps auto-magically during auto-configuration).

7.1.1. Using "ip"

Usage:

```
# /sbin/ip -6 route show [dev <device>]
```

Example:

```
# /sbin/ip -6 route show dev eth0
2001:0db8:0:f101::/64 proto kernel metric 256 mtu 1500 advmss 1440
fe80::/10             proto kernel metric 256 mtu 1500 advmss 1440
ff00::/8              proto kernel metric 256 mtu 1500 advmss 1440
default               proto kernel metric 256 mtu 1500 advmss 1440
```

7.1.2. Using "route"

Usage:

```
# /sbin/route -A inet6
```

Example (output is filtered for interface eth0). Here you see different IPv6 routes for different addresses on a single interface.

```
# /sbin/route -A inet6 |grep -w "eth0"
2001:0db8:0:f101 ::/64 :: UA  256 0 0 eth0 <- Interface route for global
Â¬ address
fe80::/10        ::       UA  256 0 0 eth0 <- Interface route for link-local
Â¬ address
ff00::/8         ::       UA  256 0 0 eth0 <- Interface route for all multicast
Â¬ addresses
::/0             ::       UDA 256 0 0 eth0 <- Automatic default route
```

7.2. Add an IPv6 route through a gateway

Mostly needed to reach the outside with IPv6 using an IPv6-enabled router on your link.

7.2.1. Using "ip"

Usage:

```
# /sbin/ip -6 route add <ipv6network>/<prefixlength> via <ipv6address>
Â¬ [dev <device>]
```

Example:

```
# /sbin/ip -6 route add 2000::/3 via 2001:0db8:0:f101::1
```

7.2.2. Using "route"

Usage:

```
# /sbin/route -A inet6 add <ipv6network>/<prefixlength> gw
Â¬ <ipv6address> [dev <device>]
```

A device can be needed, too, if the IPv6 address of the gateway is a link local one.

Following shown example adds a route for all currently global addresses (2000::/3) through gateway 2001:0db8:0:f101::1

```
# /sbin/route -A inet6 add 2000::/3 gw 2001:0db8:0:f101::1
```

7.3. Removing an IPv6 route through a gateway

Not so often needed manually, mostly done by network configure scripts on shutdown (full or per interface)

7.3.1. Using "ip"

Usage:

```
# /sbin/ip -6 route del <ipv6network>/<prefixlength> via <ipv6address>
Â¬ [dev <device>]
```

Example:

```
# /sbin/ip -6 route del 2000::/3 via 2001:0db8:0:f101::1
```

7.3.2. Using "route"

Usage:

```
# /sbin/route -A inet6 del <network>/<prefixlength> gw <ipv6address> [dev <device>]
```

Example for removing upper added route again:

```
# /sbin/route -A inet6 del 2000::/3 gw 2001:0db8:0:f101::1
```

7.4. Add an IPv6 route through an interface

Not often needed, sometimes in cases of dedicated point-to-point links.

7.4.1. Using "ip"

Usage:

```
# /sbin/ip -6 route add <ipv6network>/<prefixlength> dev <device>
Â¬ metric 1
```

Example:

```
# /sbin/ip -6 route add 2000::/3 dev eth0 metric 1
```

Metric "1" is used here to be compatible with the metric used by route, because the default metric on using "ip" is "1024".

7.4.2. Using "route"

Usage:

```
# /sbin/route -A inet6 add <ipv6network>/<prefixlength> dev <device>
```

Example:

```
# /sbin/route -A inet6 add 2000::/3 dev eth0
```

7.5. Removing an IPv6 route through an interface

Not so often needed to use by hand, configuration scripts will use such on shutdown.

7.5.1. Using "ip"

Usage:

```
# /sbin/ip -6 route del <ipv6network>/<prefixlength> dev <device>
```

Example:

```
# /sbin/ip -6 route del 2000::/3 dev eth0
```

7.5.2. Using "route"

Usage:

```
# /sbin/route -A inet6 del <network>/<prefixlength> dev <device>
```

Example:

```
# /sbin/route -A inet6 del 2000::/3 dev eth0
```

7.6. FAQ for IPv6 routes

7.6.1. Support of an IPv6 default route

One idea of IPv6 was a hierachical routing, therefore only less routing entries are needed in routers.

There are some issues in current Linux kernels:

7.6.1.1. Clients (not routing any packet!)

Client can setup a default route like prefix "::/0", they also learn such route on autoconfiguration e.g. using radvd on the link like following example shows:

```
# ip -6 route show | grep ^default
```

```
default via fe80::212:34ff:fe12:3450 dev eth0 proto kernel metric 1024 expires
Â¬ 29sec mtu 1500 advmss 1440
```

7.6.1.2. Routers in case of packet forwarding

Older Linux kernel (at least <= 2.4.17) don't support default routes. You can set them up, but the route lookup fails when a packet should be forwarded (normal intention of a router). If you're still using such older kernel, "default routing" can be setup using the currently used global address prefix "2000::/3".

Note: take care about default routing without address filtering on edge routers. Otherwise unwanted multicast or site-local traffic can leave the edge.

Chapter 8. Neighbor Discovery

Neighbor discovery was the IPv6 successor for the ARP (Address Resolution Protocol) in IPv4. You can retrieve information about the current neighbors, in addition you can set and delete entries. The kernel keeps tracking of successful neighbor detection (like ARP in IPv4). You can dig into the learnt table using "ip".

8.1. Displaying neighbors using "ip"

With following command you can display the learnt or configured IPv6 neighbors

```
# ip -6 neigh show [dev <device>]
```

The following example shows one neighbor, which is a reachable router

```
# ip -6 neigh show
fe80::201:23ff:fe45:6789 dev eth0 lladdr 00:01:23:45:67:89 router nud reachable
```

8.2. Manipulating neighbors table using "ip"

8.2.1. Manually add an entry

With following command you are able to manually add an entry

```
# ip -6 neigh add <IPv6 address> lladdr <link-layer address> dev <device>
```

Example:

```
# ip -6 neigh add fec0::1 lladdr 02:01:02:03:04:05 dev eth0
```

8.2.2. Manually delete an entry

Like adding also an entry can be deleted:

```
# ip -6 neigh del <IPv6 address> lladdr <link-layer address> dev <device>
```

Example:

```
# ip -6 neigh del fec0::1 lladdr 02:01:02:03:04:05 dev eth0
```

8.2.3. More advanced settings

The tool "ip" is less documentated, but very strong. See online "help" for more:

```
# ip -6 neigh help
Usage: ip neigh { add | del | change | replace } { ADDR [ lladdr LLADDR ]
          [ nud { permanent | noarp | stale | reachable } ]
          | proxy ADDR } [ dev DEV ]
       ip neigh {show|flush} [ to PREFIX ] [ dev DEV ] [ nud STATE ]
```

Looks like some options are only for IPv4...if you can contribute information about flags and advanced usage, pls. send.

Chapter 9. Configuring IPv6-in-IPv4 tunnels

If you want to leave your link you have no IPv6 capable network around you, you need IPv6-in-IPv4 tunneling to reach the world wide IPv6-Internet.

There are some kind of tunnel mechanism and also some possibilities to setup tunnels.

9.1. Types of tunnels

There are more than one possibility to tunnel IPv6 packets over IPv4-only links.

9.1.1. Static point-to-point tunneling: 6bone

A point-to-point tunnel is a dedicated tunnel to an endpoint, which knows about your IPv6 network (for backward routing) and the IPv4 address of your tunnel endpoint and defined in RFC 2893 / Transition Mechanisms for IPv6 Hosts and Routers. Requirements:

- IPv4 address of your local tunnel endpoint must be static, global unique and reachable from the foreign tunnel endpoint
- A global IPv6 prefix assigned to you (see 6bone registry)
- A foreign tunnel endpoint which is capable to route your IPv6 prefix to your local tunnel endpoint (mostly remote manual configuration required)

9.1.2. Automatically tunneling

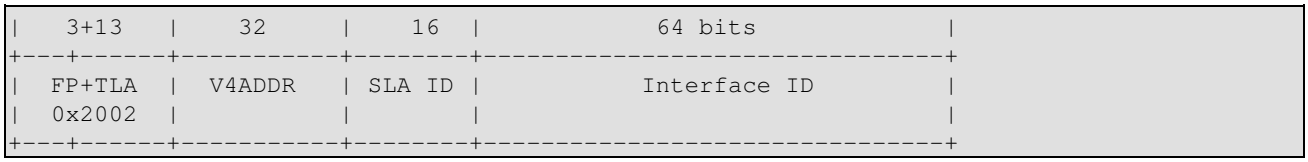
Automatic tunneling occurs, when a node directly connects another node gotten the IPv4 address of the other node before.

9.1.3. 6to4-Tunneling

6to4 tunneling (RFC 3056 / Connection of IPv6 Domains via IPv4 Clouds) uses a simple mechanism to create automatic tunnels. Each node with a global unique IPv4 address is able to be a 6to4 tunnel endpoint (if no IPv4 firewall prohibits traffic). 6to4 tunneling is mostly not a one-to-one tunnel. This case of tunneling can be divided into upstream and downstream tunneling. Also, a special IPv6 address indicates that this node will use 6to4 tunneling for connecting the world-wide IPv6 network

9.1.3.1. Generation of 6to4 prefix

The 6to4 address is defined like following (schema is taken from RFC 3056 / Connection of IPv6 Domains via IPv4 Clouds):



FP and TLA together (16 bits) have the value 0x2002. V4ADDR is the node's global unique IPv4 address (in hexadecimal notation). SLA is the subnet identifier (65536 local subnets possible) and are usable to represent your local network structure.

For gateways, such prefix is generated by normally using SLA "0000" and suffix "::1" (not a must, can be an arbitrary one with local-scope) and assigned to the 6to4 tunnel interface. Note that Microsoft Windows uses V4ADDR also for suffix.

9.1.3.2. 6to4 upstream tunneling

The node has to know to which foreign tunnel endpoint its in IPv4 packed IPv6 packets should be send to. In "early" days of 6to4 tunneling, dedicated upstream accepting routers were defined. See NSayer's 6to4 information for a list of routers.

Nowadays, 6to4 upstream routers can be found auto-magically using the anycast address 192.88.99.1. In the background routing protocols handle this, see RFC 3068 / An Anycast Prefix for 6to4 Relay Routers for details.

9.1.3.3. 6to4 downstream tunneling

The downstream (6bone -> your 6to4 enabled node) is not really fix and can vary from foreign host which originated packets were send to. There exist two possibilities:

- Foreign host uses 6to4 and sends packet direct back to your node (see below)
- Foreign host sends packets back to the world-wide IPv6 network and depending on the dynamic routing a relay router create a automatic tunnel back to your node.

9.1.3.4. Possible 6to4 traffic

- from 6to4 to 6to4: is normally directly tunneled between the both 6to4 enabled hosts
- from 6to4 to non-6to4: is sent via upstream tunneling
- non-6to4 to 6to4: is sent via downstream tunneling

9.2. Displaying existing tunnels

9.2.1. Using "ip"

Usage:

```
# /sbin/ip -6 tunnel show [<device>]
```

Example:

```
# /sbin/ip -6 tunnel show
sit0: ipv6/ip remote any local any ttl 64 nopmtudisc
sit1: ipv6/ip remote 195.226.187.50 local any ttl 64
```

9.2.2. Using "route"

Usage:

```
# /sbin/route -A inet6
```

Example (output is filtered to display only tunnels through virtual interface sit0):

```
# /sbin/route -A inet6 | grep "\Wsit0\W*$"
::/96      ::              U   256  2  0  sit0
```

```
2002::/16  ::               UA  256  0  0  sit0
2000::/3   ::193.113.58.75  UG    1  0  0  sit0
fe80::/10  ::               UA  256  0  0  sit0
ff00::/8   ::               UA  256  0  0  sit0
```

9.3. Setup of point-to-point tunnel

There are 3 possibilities to add or remove point-to-point tunnels.

A good additional information about tunnel setup using "ip" is Configuring tunnels with iproute2 (article) (Mirror).

9.3.1. Add point-to-point tunnels

9.3.1.1. Using "ip"

Common method at the moment for a small amount of tunnels.

Usage for creating a tunnel device (but it's not up afterward, also a TTL must be specified because the default value is 0).

```
# /sbin/ip tunnel add <device> mode sit ttl <ttldefault> remote
Â¬ <ipv4addressofforeigntunnel> local <ipv4addresslocal>
```

Usage (generic example for three tunnels):

```
# /sbin/ip tunnel add sit1 mode sit ttl <ttldefault> remote
Â¬ <ipv4addressofforeigntunnel1> local <ipv4addresslocal>
# /sbin/ip link set dev sit1 up
# /sbin/ip -6 route add <prefixtoroute1> dev sit1 metric 1

# /sbin/ip tunnel add sit2 mode sit ttl <ttldefault>
Â¬ <ipv4addressofforeigntunnel2> local <ipv4addresslocal>
# /sbin/ip link set dev sit2 up
# /sbin/ip -6 route add <prefixtoroute2> dev sit2 metric 1

# /sbin/ip tunnel add sit3 mode sit ttl <ttldefault>
Â¬ <ipv4addressofforeigntunnel3> local <ipv4addresslocal>
# /sbin/ip link set dev sit3 up
# /sbin/ip -6 route add <prefixtoroute3> dev sit3 metric 1
```

9.3.1.2. Using "ifconfig" and "route" (deprecated)

This not very recommended way to add a tunnel because it's a little bit strange. No problem if adding only one, but if you setup more than one, you cannot easy shutdown the first ones and leave the others running.

Usage (generic example for three tunnels):

```
# /sbin/ifconfig sit0 up

# /sbin/ifconfig sit0 tunnel <ipv4addressofforeigntunnel1>
# /sbin/ifconfig sit1 up
# /sbin/route -A inet6 add <prefixtoroute1> dev sit1

# /sbin/ifconfig sit0 tunnel <ipv4addressofforeigntunnel2>
# /sbin/ifconfig sit2 up
```

```
# /sbin/route -A inet6 add <prefixtoroute2> dev sit2

# /sbin/ifconfig sit0 tunnel <ipv4addressofforeigntunnel3>
# /sbin/ifconfig sit3 up
# /sbin/route -A inet6 add <prefixtoroute3> dev sit3
```

Important: DON'T USE THIS, because this setup implicit enable "automatic tunneling" from anywhere in the Internet, this is a risk, and it should not be advocated.

9.3.1.3. Using "route" only

It's also possible to setup tunnels in Non Broadcast Multiple Access (NBMA) style, it's a easy way to add many tunnels at once. But none of the tunnel can be numbered (which is a not required feature).

Usage (generic example for three tunnels):

```
# /sbin/ifconfig sit0 up

# /sbin/route -A inet6 add <prefixtoroute1> gw
Â¬ ::<ipv4addressofforeigntunnel1> dev sit0
# /sbin/route -A inet6 add <prefixtoroute2> gw
Â¬ ::<ipv4addressofforeigntunnel2> dev sit0
# /sbin/route -A inet6 add <prefixtoroute3> gw
Â¬ ::<ipv4addressofforeigntunnel3> dev sit0
```

Important: DON'T USE THIS, because this setup implicit enable "automatic tunneling" from anywhere in the Internet, this is a risk, and it should not be advocated.

9.3.2. Removing point-to-point tunnels

Manually not so often needed, but used by scripts for clean shutdown or restart of IPv6 configuration.

9.3.2.1. Using "ip"

Usage for removing a tunnel device:

```
# /sbin/ip tunnel del <device>
```

Usage (generic example for three tunnels):

```
# /sbin/ip -6 route del <prefixtoroute1> dev sit1
# /sbin/ip link set sit1 down
# /sbin/ip tunnel del sit1

# /sbin/ip -6 route del <prefixtoroute2> dev sit2
# /sbin/ip link set sit2 down
# /sbin/ip tunnel del sit2

# /sbin/ip -6 route del <prefixtoroute3> dev sit3
# /sbin/ip link set sit3 down
# /sbin/ip tunnel del sit3
```

9.3.2.2. Using "ifconfig" and "route" (deprecated because not very funny)

Not only the creation is strange, the shutdown also...you have to remove the tunnels in backorder, means the latest created must be removed first.

Usage (generic example for three tunnels):

```
# /sbin/route -A inet6 del <prefixtoroute3> dev sit3
# /sbin/ifconfig sit3 down

# /sbin/route -A inet6 del <prefixtoroute2> dev sit2
# /sbin/ifconfig sit2 down

# /sbin/route -A inet6 add <prefixtoroute1> dev sit1
# /sbin/ifconfig sit1 down

# /sbin/ifconfig sit0 down
```

9.3.2.3. Using "route"

This is like removing normal IPv6 routes.

Usage (generic example for three tunnels):

```
# /sbin/route -A inet6 del <prefixtoroute1> gw
Â¬ ::<ipv4addressofforeigntunnel1> dev sit0
# /sbin/route -A inet6 del <prefixtoroute2> gw
Â¬ ::<ipv4addressofforeigntunnel2> dev sit0
# /sbin/route -A inet6 del <prefixtoroute3> gw
Â¬ ::<ipv4addressofforeigntunnel3> dev sit0

# /sbin/ifconfig sit0 down
```

9.3.3. Numbered point-to-point tunnels

Sometimes it's needed to configure a point-to-point tunnel with IPv6 addresses like in IPv4 today. This is only possible with the first (ifconfig+route - deprecated) and third (ip+route) tunnel setup. In such cases, you can add the IPv6 address to the tunnel interface like shown on interface configuration.

9.4. Setup of 6to4 tunnels

Pay attention that the support of 6to4 tunnels currently lacks on vanilla kernel series 2.2.x (see systemcheck/kernel for more information). Also note that that the prefix length for a 6to4 address is 16 because of from network point of view, all other 6to4 enabled hosts are on the same layer 2.

9.4.1. Add a 6to4 tunnel

First, you have to calculate your 6to4 prefix using your local assigned global routable IPv4 address (if your host has no global routable IPv4 address, in special cases NAT on border gateways is possible):

Assuming your IPv4 address is

```
1.2.3.4
```

the generated 6to4 prefix will be

```
2002:0102:0304::
```

Local 6to4 gateways should (but it's not a must, you can choose an arbitrary suffix with local-scope, if you feel better) always assigned the suffix "::1", therefore your local 6to4 address will be

```
2002:0102:0304::1
```

Use e.g. following for automatic generation:

```
ipv4="1.2.3.4"; printf "2002:%02x%02x:%02x%02x::1" `echo $ipv4 | tr "." " "`
```

There are two ways possible to setup 6to4 tunneling now.

9.4.1.1. Using "ip" and a dedicated tunnel device

This is now the recommended way (a TTL must be specified because the default value is 0).

Create a new tunnel device

```
# /sbin/ip tunnel add tun6to4 mode sit ttl <ttldefault> remote any local <localipv4address>
```

Bring interface up

```
# /sbin/ip link set dev tun6to4 up
```

Add local 6to4 address to interface (note: prefix length 16 is important!)

```
# /sbin/ip -6 addr add <local6to4address>/16 dev tun6to4
```

Add (default) route to the global IPv6 network using the all-6to4-routers IPv4 anycast address

```
# /sbin/ip -6 route add 2000::/3 via ::192.88.99.1 dev tun6to4 metric 1
```

It was reported that some versions of "ip" (e.g. SuSE Linux 9.0) don't support IPv4-compatible IPv6 addresses for gateways, in this case the related IPv6 address has to be used:

```
# /sbin/ip -6 route add 2000::/3 via 2002:c058:6301::1 dev tun6to4 metric 1
```

9.4.1.2. Using "ifconfig" and "route" and generic tunnel device "sit0" (deprecated)

This is now deprecated because using the generic tunnel device sit0 doesn't let specify filtering per device.

Bring generic tunnel interface sit0 up

```
# /sbin/ifconfig sit0 up
```

Add local 6to4 address to interface

```
# /sbin/ifconfig sit0 add <local6to4address>/16
```

Add (default) route to the global IPv6 network using the all-6to4-relays IPv4 anycast address

```
# /sbin/route -A inet6 add 2000::/3 gw ::192.88.99.1 dev sit0
```

9.4.2. Remove a 6to4 tunnel

9.4.2.1. Using "ip" and a dedicated tunnel device

Remove all routes through this dedicated tunnel device

```
# /sbin/ip -6 route flush dev tun6to4
```

Shut down interface

```
# /sbin/ip link set dev tun6to4 down
```

Remove created tunnel device

```
# /sbin/ip tunnel del tun6to4
```

9.4.2.2. Using "ifconfig" and "route" and generic tunnel device "sit0" (deprecated)

Remove (default) route through the 6to4 tunnel interface

```
# /sbin/route -A inet6 del 2000::/3 gw ::192.88.99.1 dev sit0
```

Remove local 6to4 address to interface

```
# /sbin/ifconfig sit0 del <local6to4address>/16
```

Shut down generic tunnel device (take care about this, perhaps it's still in use...)

```
# /sbin/ifconfig sit0 down
```

Chapter 10. Configuring IPv4-in-IPv6 tunnels

This will be filled in the future. At the moment, such tunnels are more used in test environments but it looks like that support is missing currently for Linux (03/2004).

More information in the meantime: RFC 2473 / Generic Packet Tunneling in IPv6 Specification

Chapter 11. Kernel settings in /proc-filesystem

Note: the source of this section is mostly the file "ip-sysctl.txt" which is included in current kernel sources in directory "Documentation/networking". Credits to Pekka Savola for maintaining the IPv6-related part in this file. Also some text is more or less copied & pasted into this document.

11.1. How to access the /proc-filesystem

11.1.1. Using "cat" and "echo"

Using "cat" and "echo" is the simplest way to access the /proc filesystem, but some requirements are needed for that

- The /proc-filesystem had to be enabled in kernel, means on compiling following switch has to be set

```
CONFIG_PROC_FS=y
```

- The /proc-filesystem was mounted before, which can be tested using

```
# mount | grep "type proc"
none on /proc type proc (rw)
```

- You need read and sometimes also write access (normally root only) to the /proc-filesystem

Normally, only entries in /proc/sys/* are writable, the others are readonly and for information retrieving only.

11.1.1.1. Retrieving a value

The value of an entry can be retrieved using "cat":

```
# cat /proc/sys/net/ipv6/conf/all/forwarding
0
```

11.1.1.2. Setting a value

A new value can be set (if entry is writable) using "echo":

```
# echo "1" >/proc/sys/net/ipv6/conf/all/forwarding
```

11.1.2. Using "sysctl"

Using the "sysctl" program to access the kernel switches is a modern method today. You can use it also, if the /proc-filesystem isn't mounted. But you have only access to /proc/sys/*!

The program "sysctl" is included in package "procps" (on Red Hat Linux systems).

- The sysctl-interface had to be enabled in kernel, means on compiling following switch has to be set

```
CONFIG_SYSCTL=y
```

11.1.2.1. Retrieving a value

The value of an entry can be retrieved now:

```
# sysctl net.ipv6.conf.all.forwarding
net.ipv6.conf.all.forwarding = 0
```

11.1.2.2. Setting a value

A new value can be set (if entry is writable):

```
# sysctl -w net.ipv6.conf.all.forwarding=1
net.ipv6.conf.all.forwarding = 1
```

Note: Don't use spaces around the "=" on setting values. Also on multiple values per line, quote them like e.g.

```
# sysctl -w net.ipv4.ip_local_port_range="32768 61000"
net.ipv4.ip_local_port_range = 32768 61000
```

11.1.2.3. Additionals

Note: There are sysctl versions in the wild which displaying "/" instead of the "."

For more details take a look into sysctl's manpage.

Hint: for digging fast into the settings, use the option "-a" (display all entries) in conjunction with "grep".

11.1.3. Values found in /proc-filesystems

There are several formats seen in /proc-filesystem:

- BOOLEAN: simple a "0" (false) or a "1" (true)
- INTEGER: an integer value, can be unsigned, too
- more sophisticated lines with several values: sometimes a header line is displayed also, if not, have a look into the kernel source to retrieve information about the meaning of each value...

11.2. Entries in /proc/sys/net/ipv6/

11.2.1. conf/default/*

Change the interface-specific default settings.

11.2.2. conf/all/*

Change all the interface-specific settings.

Exception: "conf/all/forwarding" has a different meaning here

11.2.2.1. conf/all/forwarding

- Type: BOOLEAN

This enables global IPv6 forwarding between all interfaces.

In IPv6 you can't control forwarding per device, forwarding control has to be done using IPv6-netfilter (controlled with ip6tables) rulesets and specify input and output devices (see Firewalling/Netfilter6 for more). This is different to IPv4, where you are able to control forwarding per device (decision is made on interface where packet came in).

This also sets all interfaces' Host/Router setting 'forwarding' to the specified value. See below for details. This referred to as global forwarding.

If this value is 0, no IPv6 forwarding is enabled, packets never leave another interface, neither physical nor logical like e.g. tunnels.

11.2.3. conf/interface/*

Change special settings per interface.

The functional behaviour for certain settings is different depending on whether local forwarding is enabled or not.

11.2.3.1. accept_ra

- Type: BOOLEAN
- Functional default: enabled if local forwarding is disabled. disabled if local forwarding is enabled.

Accept Router Advertisements, and autoconfigure this interface with received data.

11.2.3.2. accept_redirects

- Type: BOOLEAN
- Functional default: enabled if local forwarding is disabled. disabled if local forwarding is enabled.

Accept Redirects sent by an IPv6 router.

11.2.3.3. autoconf

- Type: BOOLEAN
- Functional default: enabled if accept_ra_pinfo is enabled. disabled if accept_ra_pinfo is disabled.

Autoconfigure addresses using prefix information from router advertisements.

11.2.3.4. dad_transmits

- Type: INTEGER
- Default: 1

The amount of Duplicate Address Detection probes to send.

11.2.3.5. forwarding

- Type: BOOLEAN
- Default: FALSE if global forwarding is disabled (default), otherwise TRUE

Configure interface-specific Host/Router behaviour.

Note: It is recommended to have the same setting on all interfaces; mixed router/host scenarios are rather uncommon.

- Value FALSE: By default, Host behaviour is assumed. This means:

1. IsRouter flag is not set in Neighbour Advertisements.
2. Router Solicitations are being sent when necessary.
3. If accept_ra is TRUE (default), accept Router Advertisements (and do autoconfiguration).
4. If accept_redirects is TRUE (default), accept Redirects.

- Value TRUE: If local forwarding is enabled, Router behaviour is assumed. This means exactly the reverse from the above:

1. IsRouter flag is set in Neighbour Advertisements.
2. Router Solicitations are not sent.
3. Router Advertisements are ignored.
4. Redirects are ignored.

11.2.3.6. hop_limit

- Type: INTEGER
- Default: 64

Default Hop Limit to set.

11.2.3.7. mtu

- Type: INTEGER
- Default: 1280 (IPv6 required minimum)

Default Maximum Transfer Unit

11.2.3.8. router_solicitation_delay

- Type: INTEGER
- Default: 1

Number of seconds to wait after interface is brought up before sending Router Solicitations.

11.2.3.9. router_solicitation_interval

- Type: INTEGER
- Default: 4

Number of seconds to wait between Router Solicitations.

11.2.3.10. router_solicitations

- Type: INTEGER
- Default: 3

Number of Router Solicitations to send until assuming no routers are present.

11.2.4. neigh/default/*

Change default settings for neighbor detection and some special global interval and threshold values:

11.2.4.1. gc_thresh1

- Type: INTEGER
- Default: 128

More to be filled.

11.2.4.2. gc_thresh2

- Type: INTEGER
- Default: 512

More to be filled.

11.2.4.3. gc_thresh3

- Type: INTEGER
- Default: 1024

Tuning parameter for neighbour table size.

Increase this value if you have a lot of interfaces and problem with routes start to act mysteriously and fail. Or if a running Zebra (routing daemon) reports:

```
ZEBRA: netlink-listen error: No buffer space available, type=RTM_NEWROUTE(24), seq=426, pid=0
```

11.2.4.4. gc_interval

- Type: INTEGER
- Default: 30

More to be filled.

11.2.5. neigh/interface/*

Change special settings per interface for neighbor detection.

11.2.5.1. anycast_delay

- Type: INTEGER
- Default: 100

More to be filled.

11.2.5.2. gc_stale_time

- Type: INTEGER
- Default: 60

More to be filled.

11.2.5.3. proxy_qlen

- Type: INTEGER
- Default: 64

More to be filled.

11.2.5.4. unres_qlen

- Type: INTEGER
- Default: 3

More to be filled.

11.2.5.5. app_solicit

- Type: INTEGER
- Default: 0

More to be filled.

11.2.5.6. locktime

- Type: INTEGER
- Default: 0

More to be filled.

11.2.5.7. retrans_time

- Type: INTEGER
- Default: 100

More to be filled.

11.2.5.8. base_reachable_time

- Type: INTEGER
- Default: 30

More to be filled.

11.2.5.9. mcast_solicit

- Type: INTEGER
- Default: 3

More to be filled.

11.2.5.10. ucast_solicit

- Type: INTEGER
- Default: 3

More to be filled

11.2.5.11. delay_first_probe_time

- Type: INTEGER
- Default: 5

More to be filled.

11.2.5.12. proxy_delay

- Type: INTEGER
- Default: 80

More to be filled.

11.2.6. route/*

Change global settings for routing.

11.2.6.1. flush

Removed in newer kernel releases - more to be filled.

11.2.6.2. gc_interval

- Type: INTEGER
- Default: 30

More to be filled.

11.2.6.3. gc_thresh

- Type: INTEGER
- Default: 1024

More to be filled.

11.2.6.4. mtu_expires

- Type: INTEGER
- Default: 600

More to be filled.

11.2.6.5. gc_elasticity

- Type: INTEGER
- Default: 0

More to be filled.

11.2.6.6. gc_min_interval

- Type: INTEGER
- Default: 5

More to be filled.

11.2.6.7. gc_timeout

- Type: INTEGER
- Default: 60

More to be filled.

11.2.6.8. min_adv_mss

- Type: INTEGER
- Default: 12

More to be filled.

11.2.6.9. max_size

- Type: INTEGER
- Default: 4096

More to be filled.

11.3. IPv6-related entries in /proc/sys/net/ipv4/

At the moment (and this will be until IPv4 is completly converted to an independend kernel module) some switches are also used here for IPv6.

11.3.1. ip_*

11.3.1.1. ip_local_port_range

This control setting is used by IPv6 also.

11.3.2. tcp_*

This control settings are used by IPv6 also.

11.3.3. icmp_*

This control settings are not used by IPv6. To enable ICMPv6 rate limiting (which is very recommended because of the capability of ICMPv6 storms) netfilter-v6 rules must be used.

11.3.4. others

Unknown, but probably not used by IPv6.

11.4. IPv6-related entries in /proc/net/

In /proc/net there are several read-only entries available. You cannot retrieve information using "sysctl" here, so use e.g. "cat".

11.4.1. if_inet6

- Type: One line per addresss containing multiple values

Here all configured IPv6 addresses are shown in a special format. The example displays for loopback interface only. The meaning is shown below (see "net/ipv6/addrconf.c" for more).

```
# cat /proc/net/if_inet6
00000000000000000000000000000001 01 80 10 80 lo
+--------------------------------+ ++ ++ ++ ++ ++
|                                  |  |  |  |  |
1                                  2  3  4  5  6
```

1. IPv6 address displayed in 32 hexadecimal chars without colons as separator
2. Netlink device number (interface index) in hexadecimal (see "ip addr" , too)
3. Prefix length in hexadecimal
4. Scope value (see kernel source " include/net/ipv6.h" and "net/ipv6/addrconf.c" for more)
5. Interface flags (see "include/linux/rtnetlink.h" and "net/ipv6/addrconf.c" for more)
6. Device name

11.4.2. ipv6_route

- Type: One line per route containing multiple values

Here all configured IPv6 routes are shown in a special format. The example displays for loopback interface only. The meaning is shown below (see "net/ipv6/route.c" for more).

```
# cat /proc/net/ipv6_route
00000000000000000000000000000000 00 00000000000000000000000000000000 00
+--------------------------------+ ++ +--------------------------------+ ++
|                                |  |                                  |
1                                2  3                                  4

Â¬ 00000000000000000000000000000000 ffffffff 00000001 00000001 00200200 lo
Â¬ +--------------------------------+ +------+ +------+ +------+ +------+ ++
Â¬ |                                |        |        |        |        |
Â¬ 5                                6        7        8        9        10
```

1. IPv6 destination network displayed in 32 hexadecimal chars without colons as separator
2. IPv6 destination prefix length in hexadecimal
3. IPv6 source network displayed in 32 hexadecimal chars without colons as separator
4. IPv6 source prefix length in hexadecimal
5. IPv6 next hop displayed in 32 hexadecimal chars without colons as separator
6. Metric in hexadecimal
7. Reference counter
8. Use counter
9. Flags
10. Device name

11.4.3. sockstat6

- Type: One line per protocol with description and value

Statistics about used IPv6 sockets. Example:

```
# cat /proc/net/sockstat6
TCP6: inuse 7
UDP6: inuse 2
RAW6: inuse 1
FRAG6: inuse 0 memory 0
```

11.4.4. tcp6

To be filled.

11.4.5. udp6

To be filled.

11.4.6. igmp6

To be filled.

11.4.7. raw6

To be filled.

11.4.8. ip6_flowlabel

To be filled.

11.4.9. rt6_stats

To be filled.

11.4.10. snmp6

- Type: One line per SNMP description and value

SNMP statistics, can be retrieved via SNMP server and related MIB table by network management software.

11.4.11. ip6_tables_names

Available netfilter6 tables

Chapter 12. Netlink-Interface to kernel

To be filled...I have no experience with that...

Chapter 13. Network debugging

13.1. Server socket binding

13.1.1. Using "netstat" for server socket binding check

It's always interesting which server sockets are currently active on a node. Using "netstat" is a short way to get such information:

Used options: -nlptu

Example:

```
# netstat -nlptu
Active Internet connections (only servers)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
Â¬ PID/Program name
tcp        0      0 0.0.0.0:32768           0.0.0.0:*               LISTEN
Â¬ 1258/rpc.statd
tcp        0      0 0.0.0.0:32769           0.0.0.0:*               LISTEN
Â¬ 1502/rpc.mountd
tcp        0      0 0.0.0.0:515             0.0.0.0:*               LISTEN
Â¬ 22433/lpd Waiting
tcp        0      0 1.2.3.1:139             0.0.0.0:*               LISTEN
Â¬ 1746/smbd
tcp        0      0 0.0.0.0:111             0.0.0.0:*               LISTEN
Â¬ 1230/portmap
tcp        0      0 0.0.0.0:6000            0.0.0.0:*               LISTEN
Â¬ 3551/X
tcp        0      0 1.2.3.1:8081            0.0.0.0:*               LISTEN
Â¬ 18735/junkbuster
tcp        0      0 1.2.3.1:3128            0.0.0.0:*               LISTEN
Â¬ 18822/(squid)
tcp        0      0 127.0.0.1:953           0.0.0.0:*               LISTEN
Â¬ 30734/named
tcp        0      0 ::ffff:1.2.3.1:993      :::*                    LISTEN
Â¬ 6742/xinetd-ipv6
tcp        0      0 :::13                   :::*                    LISTEN
Â¬ 6742/xinetd-ipv6
tcp        0      0 ::ffff:1.2.3.1:143      :::*                    LISTEN
Â¬ 6742/xinetd-ipv6
tcp        0      0 :::53                   :::*                    LISTEN
Â¬ 30734/named
tcp        0      0 :::22                   :::*                    LISTEN
Â¬ 1410/sshd
tcp        0      0 :::6010                 :::*                    LISTEN
Â¬ 13237/sshd
udp        0      0 0.0.0.0:32768           0.0.0.0:*
Â¬ 1258/rpc.statd
udp        0      0 0.0.0.0:2049            0.0.0.0:*
Â¬ -
udp        0      0 0.0.0.0:32770           0.0.0.0:*
Â¬ 1502/rpc.mountd
udp        0      0 0.0.0.0:32771           0.0.0.0:*
Â¬ -
udp        0      0 1.2.3.1:137             0.0.0.0:*
Â¬ 1751/nmbd
udp        0      0 0.0.0.0:137             0.0.0.0:*
```

```
Â¬ 1751/nmbd
udp        0      0 1.2.3.1:138             0.0.0.0:*
Â¬ 1751/nmbd
udp        0      0 0.0.0.0:138             0.0.0.0:*
Â¬ 1751/nmbd
udp        0      0 0.0.0.0:33044           0.0.0.0:*
Â¬ 30734/named
udp        0      0 1.2.3.1:53              0.0.0.0:*
Â¬ 30734/named
udp        0      0 127.0.0.1:53            0.0.0.0:*
Â¬ 30734/named
udp        0      0 0.0.0.0:67              0.0.0.0:*
Â¬ 1530/dhcpd
udp        0      0 0.0.0.0:67              0.0.0.0:*
Â¬ 1530/dhcpd
udp        0      0 0.0.0.0:32858           0.0.0.0:*
Â¬ 18822/(squid)
udp        0      0 0.0.0.0:4827            0.0.0.0:*
Â¬ 18822/(squid)
udp        0      0 0.0.0.0:111             0.0.0.0:*
Â¬ 1230/portmap
udp        0      0 :::53                   :::*
Â¬ 30734/named
```

13.2. Examples for tcpdump packet dumps

Here some examples of captured packets are shown, perhaps useful for your own debugging...

...more coming next...

13.2.1. Router discovery

13.2.1.1. Router advertisement

```
15:43:49.484751 fe80::212:34ff:fe12:3450 > ff02::1: icmp6: router
Â¬ advertisement(chlim=64, router_ltime=30, reachable_time=0,
Â¬ retrans_time=0)(prefix info: AR valid_ltime=30, preffered_ltime=20,
Â¬ prefix=2002:0102:0304:1::/64)(prefix info: LAR valid_ltime=2592000,
Â¬ preffered_ltime=604800, prefix=2001:0db8:0:1::/64)(src lladdr:
Â¬ 0:12:34:12:34:50) (len 88, hlim 255)
```

Router with link-local address "fe80::212:34ff:fe12:3450" send an advertisement to the all-node-on-link multicast address "ff02::1" containing two prefixes "2002:0102:0304:1::/64" (lifetime 30 s) and "2001:0db8:0:1::/64" (lifetime 2592000 s) including its own layer 2 MAC address "0:12:34:12:34:50".

13.2.1.2. Router solicitation

```
15:44:21.152646 fe80::212:34ff:fe12:3456 > ff02::2: icmp6: router solicitation
Â¬ (src lladdr: 0:12:34:12:34:56) (len 16, hlim 255)
```

Node with link-local address "fe80::212:34ff:fe12:3456" and layer 2 MAC address "0:12:34:12:34:56" is looking for a router on-link, therefore sending this solicitation to the all-router-on-link multicast address "ff02::2".

13.2.2. Neighbor discovery

13.2.2.1. Neighbor discovery solicitation for duplicate address detection

Following packets are sent by a node with layer 2 MAC address "0:12:34:12:34:56" during autoconfiguration to check whether a potential address is already used by another node on the link sending this to the solicited-node link-local multicast address.

- Node wants to configure its link-local address "fe80::212:34ff:fe12:3456", checks for duplicate now

```
15:44:17.712338 :: > ff02::1:ff12:3456: icmp6: neighbor sol: who has
Â¬ fe80::212:34ff:fe12:3456(src lladdr: 0:12:34:12:34:56) (len 32, hlim 255)
```

- Node wants to configure its global address "2002:0102:0304:1:212:34ff:fe12:3456" (after receiving advertisement shown above), checks for duplicate now

```
15:44:21.905596 :: > ff02::1:ff12:3456: icmp6: neighbor sol: who has
Â¬ 2002:0102:0304:1:212:34ff:fe12:3456(src lladdr: 0:12:34:12:34:56) (len 32,
Â¬ hlim 255)
```

- Node wants to configure its global address "2001:0db8:0:1:212:34ff:fe12:3456" (after receiving advertisement shown above), checks for duplicate now

```
15:44:22.304028 :: > ff02::1:ff12:3456: icmp6: neighbor sol: who has
Â¬ 2001:0db8:0:1:212:34ff:fe12:3456(src lladdr: 0:12:34:12:34:56) (len 32, hlim
Â¬ 255)
```

13.2.2.2. Neighbor discovery solicitation for looking for host or gateway

- Node wants to send packages to "2001:0db8:0:1::10" but has no layer 2 MAC address to send packet, so send solicitation now

```
13:07:47.664538 2002:0102:0304:1:2e0:18ff:fe90:9205 > ff02::1:ff00:10: icmp6:
Â¬ neighbor sol: who has 2001:0db8:0:1::10(src lladdr: 0:e0:18:90:92:5) (len 32,
Â¬ hlim 255)
```

- Node looks for "fe80::10" now

```
13:11:20.870070 fe80::2e0:18ff:fe90:9205 > ff02::1:ff00:10: icmp6: neighbor
Â¬ sol: who has fe80::10(src lladdr: 0:e0:18:90:92:5) (len 32, hlim 255)
```

Chapter 14. Support for persistent IPv6 configuration in Linux distributions

Some Linux distribution contain already support of a persistent IPv6 configuration using existing or new configuration and script files and some hook in the IPv4 script files.

14.1. Red Hat Linux and "clones"

Since starting writing the IPv6 & Linux - HowTo it was my intention to enable a persistent IPv6 configuration which catch most of the wished cases like host-only, router-only, dual-homed-host, router with second stub network, normal tunnels, 6to4 tunnels, and so on. Nowadays there exists a set of configuration and script files which do the job very well (never heard about real problems, but I don't know how many use the set). Because this configuration and script files are extended from time to time, they got their own homepage: initscripts-ipv6 homepage (Mirror). Because I began my IPv6 experience using a Red Hat Linux 5.0 clone, my IPv6 development systems are mostly Red Hat Linux based now, it's kind a logic that the scripts are developed for this kind of distribution (so called *historic issue*). Also it was very easy to extend some configuration files, create new ones and create some simple hook for calling IPv6 setup during IPv4 setup.

Fortunately, in Red Hat Linux since 7.1 a snapshot of my IPv6 scripts is included, this was and is still further on assisted by Pekka Savola.

Mandrake since version 8.0 also includes an IPv6-enabled initscript package, but a minor bug still prevents usage ("ifconfig" misses "inet6" before "add").

14.1.1. Test for IPv6 support of network configuration scripts

You can test, whether your Linux distribution contain support for persistent IPv6 configuration using my set. Following script library should exist:

```
/etc/sysconfig/network-scripts/network-functions-ipv6
```

Auto-magically test:

```
# test -f /etc/sysconfig/network-scripts/network-functions-ipv6 && echo "Main
Â¬ IPv6 script library exists"
```

The version of the library is important if you miss some features. You can get it executing following (or easier look at the top of the file):

```
# source /etc/sysconfig/network-scripts/network-functions-ipv6 &&
Â¬ getversion_ipv6_functions
20011124
```

In shown example, the used version is 20011124. Check this against latest information on initscripts-ipv6 homepage (Mirror) to see what has been changed. You will find there also a change-log.

14.1.2. Short hint for enabling IPv6 on current RHL 7.1, 7.2, 7.3, ...

- Check whether running system has already IPv6 module loaded

```
# modprobe -c | grep net-pf-10
alias net-pf-10 off
```

- If result is "off", then enable IPv6 networking by editing /etc/sysconfig/network, add following new line

```
NETWORKING_IPV6=yes
```

- Reboot or restart networking using

```
# service network restart
```

- Now IPv6 module should be loaded

```
# modprobe -c | grep ipv6
alias net-pf-10 ipv6
```

If your system is on a link which provides router advertisement, autoconfiguration will be done automatically. For more information which settings are supported see /usr/share/doc/initscripts-$version/sysconfig.txt.

14.2. SuSE Linux

In newer 7.x versions there is a really rudimentary support available, see /etc/rc.config for details.

Because of the really different configuration and script file structure it is hard (or impossible) to use the set for Red Hat Linux and clones with this distribution. In versions 8.x they completly change their configuration setup.

14.2.1. SuSE Linux 7.3

- How to setup 6to4 IPv6 with SuSE 7.3

14.2.2. SuSE Linux 8.0

14.2.2.1. IPv6 address configuration

Edit file /etc/sysconfig/network/ifcfg-<Interface-Name> and setup following value

```
IP6ADDR="<ipv6-address>/<prefix>"
```

14.2.2.2. Additional information

See file /usr/share/doc/packages/sysconfig/README

14.2.3. SuSE Linux 8.1

14.2.3.1. IPv6 address configuration

Edit file /etc/sysconfig/network/ifcfg-<Interface-Name> and setup following value

```
IPADDR="<ipv6-address>/<prefix>"
```

14.2.3.2. Additional information

See file /usr/share/doc/packages/sysconfig/Network

14.3. Debian Linux

Following information was contributed by Stephane Bortzmeyer <bortzmeyer at nic dot fr>

1. Be sure that IPv6 is loaded, either because it is compiled into the kernel or because the module is loaded. For the latest, three solutions, adding it to /etc/modules, using the pre-up trick shown later or using kmod (not detailed here).
2. Configure your interface. Here we assume eth0 and address (2001:0db8:1234:5::1:1). Edit /etc/network/interfaces:

```
iface eth0 inet6 static
        pre-up modprobe ipv6
        address 2001:0db8:1234:5::1:1
        # To suppress completely autoconfiguration:
        # up echo 0 > /proc/sys/net/ipv6/conf/all/autoconf
        netmask 64
        # The router is autoconfigured and has no fixed address.
        # It is magically
        # found. (/proc/sys/net/ipv6/conf/all/accept_ra). Otherwise:
        #gateway 2001:0db8:1234:5::1
```

And you reboot or you just

```
# ifup --force eth0
```

and you have your static address.

14.3.1. Further information

- IPv6 on Debian Linux by Craig Small
- Jean-Marc V. Liotier's HOWTO for Freenet6 & Debian Users (announced 24.12.2002 on mailinglist users@ipv6.org)

Chapter 15. Auto-configuration

15.1. Stateless auto-configuration

Is supported and seen on the assigned link-local address after an IPv6-enabled interface is up.

Example:

```
# ip -6 addr show dev eth0 scope link
2: eth0: <BROADCAST,MULTICAST,UP> mtu 1500 qlen1000
    inet6 fe80::211:d8ff:fe6b:f0f5/64 scope link
       valid_lft forever preferred_lft forever
```

15.2. Stateful auto-configuration using Router Advertisement Daemon (radvd)

to be filled. See radvd daemon autoconfiguration below.

15.3. Dynamic Host Configuration Protocol v6 (DHCPv6)

After a long time discussing issues, finally RFC 3315 / Dynamic Host Configuration Protocol for IPv6 (DHCPv6) was finished. At time updating this part (10/2005) currently two implementations are available:

- Dibbler by Tomasz Mrugalski <thomson at klub dot com dot pl>
- DHCPv6 on Sourceforge (Hints for configuration)

Chapter 16. Mobility

16.1. Common information

16.1.1. Node Mobility

Support for IPv6 mobility can be enabled in Linux by installing the MIPL2 implementation found at: http://www.mobile-ipv6.org/

This implementation is compliant with RFC 3775. It is composed of a kernel patch and a mobility daemon called mip6d. Version 2.0.1 applies on Linux kernel 2.6.15.

Installation and setup are described in the Linux Mobile IPv6 HOWTO: http://gnist.org/~lars/doc/Mobile-IPv6-HOWTO/Mobile-IPv6-HOWTO.html (temporary link)

16.1.2. Network Mobility

There also exists an implementation of network mobility for Linux, it is called NEPL and is based on MIPL. It can also be downloaded from: http://www.mobile-ipv6.org/.

The HOWTO document describing setup and configuration is available at: http://www.nautilus6.org/doc/nepl-howto/.

16.1.3. Links

- Mobile IPv6 for Linux (MIPL) project: http://www.mobile-ipv6.org/
- Nautilus6 working group: http://nautilus6.org/
- Fast Handovers for Mobile IPv6 for Linux project: http://www.fmipv6.org/
- RFC 3775 / Mobility Support in IPv6
- RFC 3776 / Using IPsec to Protect Mobile IPv6 Signaling Between Mobile Nodes and Home Agents
- RFC 3963 / Network Mobility (NEMO)
- RFC 4068 / Fast Handovers for Mobile IPv6

Chapter 17. Firewalling

IPv6 firewalling is important, especially if using IPv6 on internal networks with global IPv6 addresses. Because unlike at IPv4 networks where in common internal hosts are protected automatically using private IPv4 addresses like RFC 1918 / Address Allocation for Private Internets or Automatic Private IP Addressing (APIPA)Google search for Microsoft + APIPA, in IPv6 normally global addresses are used and someone with IPv6 connectivity can reach all internal IPv6 enabled nodes.

17.1. Firewalling using netfilter6

Native IPv6 firewalling is only supported in kernel versions 2.4+. In older 2.2- you can only filter IPv6-in-IPv4 by protocol 41.

Attention: no warranty that described rules or examples can really protect your system!

Audit your ruleset after installation, see Section 18.3 for more.

Since kernel version 2.6.20 IPv6 connection tracking is fully working (and does not break IPv4 NAT anymore like versions before)

17.1.1. More information

- Netfilter project
- maillist archive of netfilter users
- maillist archive of netfilter developers
- Unofficial status informations

17.2. Preparation

This step is only needed if distributed kernel and netfilter doesn't fit your requirements and new features are available but still not built-in.

17.2.1. Get sources

Get the latest kernel source: http://www.kernel.org/

Get the latest iptables package:

- Source tarball (for kernel patches): http://www.netfilter.org/

17.2.2. Extract sources

Change to source directory:

```
# cd /path/to/src
```

Unpack and rename kernel sources

```
# tar z|jxf kernel-version.tar.gz|bz2
# mv linux linux-version-iptables-version+IPv6
```

Unpack iptables sources

```
# tar z|jxf iptables-version.tar.gz|bz2
```

17.2.3. Apply latest iptables/IPv6-related patches to kernel source

Change to iptables directory

```
# cd iptables-version
```

Apply pending patches

```
# make pending-patches KERNEL_DIR=/path/to/src/linux-version-iptables-version/
```

Apply additional IPv6 related patches (still not in the vanilla kernel included)

```
# make patch-o-matic KERNEL_DIR=/path/to/src/linux-version-iptables-version/
```

Say yes at following options (iptables-1.2.2)

- ah-esp.patch
- masq-dynaddr.patch (only needed for systems with dynamic IP assigned WAN connections like PPP or PPPoE)
- ipv6-agr.patch.ipv6
- ipv6-ports.patch.ipv6
- LOG.patch.ipv6
- REJECT.patch.ipv6

Check IPv6 extensions

```
# make print-extensions
Extensions found: IPv6:owner IPv6:limit IPv6:mac IPv6:multiport
```

17.2.4. Configure, build and install new kernel

Change to kernel sources

```
# cd /path/to/src/linux-version-iptables-version/
```

Edit Makefile

```
- EXTRAVERSION =
+ EXTRAVERSION = -iptables-version+IPv6-try
```

Run configure, enable IPv6 related

```
            Code maturity level options
                  Prompt for development and/or incomplete code/drivers : yes
            Networking options
                  Network packet filtering: yes
                  The IPv6 protocol: module
                        IPv6: Netfilter Configuration
                              IP6 tables support: module
                              All new options like following:
                                    limit match support: module
                                    MAC address match support: module
                                    Multiple port match support: module
                                    Owner match support: module
```

```
                                    netfilter MARK match support: module
                                    Aggregated address check: module
                                    Packet filtering: module
                                         REJECT target support: module
                                         LOG target support: module
                                    Packet mangling: module
                                    MARK target support: module
```

Configure other related to your system, too

Compilation and installing: see the kernel section here and other HOWTOs

17.2.5. Rebuild and install binaries of iptables

Make sure, that upper kernel source tree is also available at /usr/src/linux/

Rename older directory

```
# mv /usr/src/linux /usr/src/linux.old
```

Create a new softlink

```
# ln -s /path/to/src/linux-version-iptables-version /usr/src/linux
```

Rebuild SRPMS

```
# rpm --rebuild /path/to/SRPMS/iptables-version-release.src.rpm
```

Install new iptables packages (iptables + iptables-ipv6)

- On RH 7.1 systems, normally, already an older version is installed, therefore use "freshen"

```
# rpm -Fhv /path/to/RPMS/cpu/iptables*-version-release.cpu.rpm
```

- If not already installed, use "install"

```
# rpm -ihv /path/to/RPMS/cpu/iptables*-version-release.cpu.rpm
```

- On RH 6.2 systems, normally, no kernel 2.4.x is installed, therefore the requirements don't fit. Use "--nodeps" to install it

```
# rpm -ihv --nodeps /path/to/RPMS/cpu/iptables*-version-release.cpu.rpm
```

Perhaps it's necessary to create a softlink for iptables libraries where iptables looks for them

```
# ln -s /lib/iptables/ /usr/lib/iptables
```

17.3. Usage

17.3.1. Check for support

Load module, if so compiled

```
# modprobe ip6_tables
```

Check for capability

```
# [ ! -f /proc/net/ip6_tables_names ] && echo "Current kernel doesn't support
Â¬ 'ip6tables' firewalling (IPv6)!"
```

17.3.2. Learn how to use ip6tables

17.3.2.1. List all IPv6 netfilter entries

- Short

```
# ip6tables -L
```

- Extended

```
# ip6tables -n -v --line-numbers -L
```

17.3.2.2. List specified filter

```
# ip6tables -n -v --line-numbers -L INPUT
```

17.3.2.3. Insert a log rule at the input filter with options

```
# ip6tables --table filter --append INPUT  -j LOG --log-prefix "INPUT:"
Â¬ --log-level 7
```

17.3.2.4. Insert a drop rule at the input filter

```
# ip6tables --table filter --append INPUT  -j DROP
```

17.3.2.5. Delete a rule by number

```
# ip6tables --table filter --delete INPUT 1
```

17.3.2.6. Enable connection tracking

Since kernel version 2.6.20 IPv6 connection tracking is well supported and should be used instead of using stateless filter rules.

```
# ip6tables -A INPUT -m state --state ESTABLISHED,RELATED -j ACCEPT
```

17.3.2.7. Allow ICMPv6

Using older kernels (unpatched kernel 2.4.5 and iptables-1.2.2) no type can be specified

- Accept incoming ICMPv6 through tunnels

```
# ip6tables -A INPUT -i sit+ -p icmpv6 -j ACCEPT
```

- Allow outgoing ICMPv6 through tunnels

```
# ip6tables -A OUTPUT -o sit+ -p icmpv6 -j ACCEPT
```

Newer kernels allow specifying of ICMPv6 types:

```
# ip6tables -A INPUT -p icmpv6 --icmpv6-type echo-request -j ACCEPT
```

17.3.2.8. Rate-limiting

Because it can happen (author already saw it to times) that an ICMPv6 storm will raise up, you should use available rate limiting for at least ICMPv6 ruleset. In addition logging rules should also get rate limiting to prevent DoS attacks against syslog and storage of log file partition. An example for a rate limited ICMPv6 looks like:

```
# ip6tables -A INPUT --protocol icmpv6 --icmpv6-type echo-request
Â¬ -j ACCEPT --match limit --limit 30/minute
```

17.3.2.9. Allow incoming SSH

Here an example is shown for a ruleset which allows incoming SSH connection from a specified IPv6 address

- Allow incoming SSH from 2001:0db8:100::1/128

```
# ip6tables -A INPUT -i sit+ -p tcp -s 2001:0db8:100::1/128 --sport 512:65535
Â¬ --dport 22 -j ACCEPT
```

- Allow response packets (no longer needed if connection tracking is used!)

```
# ip6tables -A OUTPUT -o sit+ -p tcp -d 2001:0db8:100::1/128 --dport 512:65535
Â¬ --sport 22 ! --syn -j ACCEPT
```

17.3.2.10. Enable tunneled IPv6-in-IPv4

To accept tunneled IPv6-in-IPv4 packets, you have to insert rules in your IPv4 firewall setup relating to such packets, for example

- Accept incoming IPv6-in-IPv4 on interface ppp0

```
# iptables -A INPUT -i ppp0 -p ipv6 -j ACCEPT
```

- Allow outgoing IPv6-in-IPv4 to interface ppp0

```
# iptables -A OUTPUT -o ppp0 -p ipv6 -j ACCEPT
```

If you have only a static tunnel, you can specify the IPv4 addresses, too, like

- Accept incoming IPv6-in-IPv4 on interface ppp0 from tunnel endpoint 192.0.2.2

```
# iptables -A INPUT -i ppp0 -p ipv6 -s 192.0.2.2 -j ACCEPT
```

- Allow outgoing IPv6-in-IPv4 to interface ppp0 to tunnel endpoint 1.2.3.4

```
# iptables -A OUTPUT -o ppp0 -p ipv6 -d 192.0.2.2 -j ACCEPT
```

17.3.2.11. Protection against incoming TCP connection requests

VERY RECOMMENDED! For security issues you should really insert a rule which blocks incoming TCP connection requests. Adapt "-i" option, if other interface names are in use!

- Block incoming TCP connection requests to this host

```
# ip6tables -I INPUT -i sit+ -p tcp --syn -j DROP
```

- Block incoming TCP connection requests to hosts behind this router

```
# ip6tables -I FORWARD -i sit+ -p tcp --syn -j DROP
```

Perhaps the rules have to be placed below others, but that is work you have to think about it. Best way is to create a script and execute rules in a specified way.

17.3.2.12. Protection against incoming UDP connection requests

ALSO RECOMMENDED! Like mentioned on my firewall information it's possible to control the ports on outgoing UDP/TCP sessions. So if all of your local IPv6 systems are using local ports e.g. from 32768 to 60999 you are able to filter UDP connections also (until connection tracking works) like:

- Block incoming UDP packets which cannot be responses of outgoing requests of this host

```
# ip6tables -I INPUT -i sit+ -p udp ! --dport 32768:60999 -j DROP
```

- Block incoming UDP packets which cannot be responses of forwarded requests of hosts behind this router

```
# ip6tables -I FORWARD -i sit+ -p udp ! --dport 32768:60999 -j DROP
```

17.3.3. Examples

17.3.3.1. Simple example for Fedora Core

Following lines show a simple firewall configuration for Fedora Core 6 (since kernel version 2.6.20). It was modfied from the default one (generated by system-config-firewall) for supporting connection tracking and return the proper ICMPv6 code for rejects. Incoming SSH (port 22) connections are allowed.

```
File: /etc/sysconfig/ip6tables

*filter :INPUT ACCEPT [0:0]
:FORWARD ACCEPT [0:0]
:OUTPUT ACCEPT [0:0]
:RH-Firewall-1-INPUT - [0:0]
-A INPUT -j RH-Firewall-1-INPUT
-A FORWARD -j RH-Firewall-1-INPUT
-A RH-Firewall-1-INPUT -i lo -j ACCEPT
-A RH-Firewall-1-INPUT -p icmpv6 -j ACCEPT
-A RH-Firewall-1-INPUT -p 50 -j ACCEPT
-A RH-Firewall-1-INPUT -p 51 -j ACCEPT
-A RH-Firewall-1-INPUT -p udp --dport 5353 -d ff02::fb -j ACCEPT
-A RH-Firewall-1-INPUT -p udp -m udp --dport 631 -j ACCEPT
-A RH-Firewall-1-INPUT -p tcp -m tcp --dport 631 -j ACCEPT
-A RH-Firewall-1-INPUT -m state --state ESTABLISHED,RELATED -j ACCEPT
```

```
-A RH-Firewall-1-INPUT -m state --state NEW -p tcp --dport 22 -j ACCEPT
-A RH-Firewall-1-INPUT -j REJECT --reject-with icmp6-adm-prohibited
COMMIT
```

For completeness also the IPv4 configuration is shown here:

```
File: /etc/sysconfig/iptables

*filter :INPUT ACCEPT [0:0]
:FORWARD ACCEPT [0:0]
:OUTPUT ACCEPT [0:0]
:RH-Firewall-1-INPUT - [0:0]
-A INPUT -j RH-Firewall-1-INPUT
-A FORWARD -j RH-Firewall-1-INPUT
-A RH-Firewall-1-INPUT -i lo -j ACCEPT
-A RH-Firewall-1-INPUT -p icmp --icmp-type any -j ACCEPT
-A RH-Firewall-1-INPUT -p 50 -j ACCEPT
-A RH-Firewall-1-INPUT -p 51 -j ACCEPT
-A RH-Firewall-1-INPUT -p udp --dport 5353 -d 224.0.0.251 -j ACCEPT
-A RH-Firewall-1-INPUT -p udp -m udp --dport 631 -j ACCEPT
-A RH-Firewall-1-INPUT -p tcp -m tcp --dport 631 -j ACCEPT
-A RH-Firewall-1-INPUT -m state --state ESTABLISHED,RELATED -j ACCEPT
-A RH-Firewall-1-INPUT -m state --state NEW -m tcp -p tcp --dport 22 -j ACCEPT
-A RH-Firewall-1-INPUT -j REJECT --reject-with icmp-host-prohibited
COMMIT
```

Usage:

- Create/modify the configuration files
- Activate IPv4 & IPv6 firewalling

```
# service iptables start
# service ip6tables start
```

- Enable automatic start after reboot

```
# chkconfig iptables on
# chkconfig ip6tables on
```

17.3.3.2. Sophisticated example

Following lines show a more sophisticated but still stateless filter setup as an example. Happy netfilter6 ruleset creation....

```
# ip6tables -n -v -L
Chain INPUT (policy DROP 0 packets, 0 bytes)
 pkts bytes target     prot opt in     out     source               destination
    0     0 extIN      all      sit+   *       ::/0                 ::/0
    4   384 intIN      all      eth0   *       ::/0                 ::/0
    0     0 ACCEPT     all      *      *       ::1/128              ::1/128
    0     0 ACCEPT     all      lo     *       ::/0                 ::/0
    0     0 LOG        all      *      *       ::/0                 ::/0
Â¬         LOG flags 0 level 7 prefix `INPUT-default:'
    0     0 DROP       all      *      *       ::/0                 ::/0

Chain FORWARD (policy DROP 0 packets, 0 bytes)
 pkts bytes target     prot opt in     out     source               destination
Â¬
    0     0 int2ext    all      eth0   sit+    ::/0                 ::/0
```

```
    0     0 ext2int    all      sit+   eth0    ::/0                ::/0
    0     0 LOG        all      *      *       ::/0                ::/0
Â¬        LOG flags 0 level 7 prefix `FORWARD-default:'
    0     0 DROP       all      *      *       ::/0                ::/0

Chain OUTPUT (policy DROP 0 packets, 0 bytes)
 pkts bytes target     prot opt in     out     source              destination
Â¬
    0     0 extOUT     all      *      sit+    ::/0                ::/0
    4   384 intOUT     all      *      eth0    ::/0                ::/0
    0     0 ACCEPT     all      *      *       ::1/128             ::1/128
    0     0 ACCEPT     all      *      lo      ::/0                ::/0
    0     0 LOG        all      *      *       ::/0                ::/0
Â¬        LOG flags 0 level 7 prefix `OUTPUT-default:'
    0     0 DROP       all      *      *       ::/0                ::/0

Chain ext2int (1 references)
 pkts bytes target     prot opt in     out     source              destination
Â¬
    0     0 ACCEPT     icmpv6    *      *       ::/0                 ::/0
    0     0 ACCEPT     tcp      *      *       ::/0                ::/0
Â¬        tcp spts:1:65535 dpts:1024:65535 flags:!0x16/0x02
    0     0 LOG        all      *      *       ::/0                ::/0
Â¬        LOG flags 0 level 7 prefix `ext2int-default:'
    0     0 DROP       tcp      *      *       ::/0                ::/0
    0     0 DROP       udp      *      *       ::/0                ::/0
    0     0 DROP       all      *      *       ::/0                ::/0

Chain extIN (1 references)
 pkts bytes target     prot opt in     out     source              destination
Â¬
    0     0 ACCEPT     tcp      *      *       3ffe:400:100::1/128  ::/0
Â¬        tcp spts:512:65535 dpt:22
    0     0 ACCEPT     tcp      *      *       3ffe:400:100::2/128  ::/0
Â¬        tcp spts:512:65535 dpt:22
    0     0 ACCEPT     icmpv6    *      *       ::/0                 ::/0
    0     0 ACCEPT     tcp      *      *       ::/0                ::/0
Â¬        tcp spts:1:65535 dpts:1024:65535 flags:!0x16/0x02
    0     0 ACCEPT     udp      *      *       ::/0                ::/0
Â¬        udp spts:1:65535 dpts:1024:65535
    0     0 LOG        all      *      *       ::/0                ::/0
Â¬        limit: avg 5/min burst 5 LOG flags 0 level 7 prefix `extIN-default:'
    0     0 DROP       all      *      *       ::/0                ::/0

Chain extOUT (1 references)
 pkts bytes target     prot opt in     out     source              destination
Â¬
    0     0 ACCEPT     tcp      *      *       ::/0
Â¬ 2001:0db8:100::1/128tcp spt:22 dpts:512:65535 flags:!0x16/0x02
    0     0 ACCEPT     tcp      *      *       ::/0
Â¬ 2001:0db8:100::2/128tcp spt:22 dpts:512:65535 flags:!0x16/0x02
    0     0 ACCEPT     icmpv6    *      *       ::/0                 ::/0
    0     0 ACCEPT     tcp      *      *       ::/0                ::/0
Â¬        tcp spts:1024:65535 dpts:1:65535
    0     0 ACCEPT     udp      *      *       ::/0                ::/0
Â¬        udp spts:1024:65535 dpts:1:65535
    0     0 LOG        all      *      *       ::/0                ::/0
Â¬        LOG flags 0 level 7 prefix `extOUT-default:'
    0     0 DROP       all      *      *       ::/0                ::/0

Chain int2ext (1 references)
 pkts bytes target     prot opt in     out     source              destination
```

```
Â¬
    0     0 ACCEPT     icmpv6    *      *       ::/0                ::/0
    0     0 ACCEPT     tcp       *      *       ::/0               ::/0
Â¬         tcp spts:1024:65535 dpts:1:65535
    0     0 LOG        all       *      *       ::/0               ::/0
Â¬         LOG flags 0 level 7 prefix `int2ext:'
    0     0 DROP       all       *      *       ::/0               ::/0
    0     0 LOG        all       *      *       ::/0               ::/0
Â¬         LOG flags 0 level 7 prefix `int2ext-default:'
    0     0 DROP       tcp       *      *       ::/0               ::/0
    0     0 DROP       udp       *      *       ::/0               ::/0
    0     0 DROP       all       *      *       ::/0               ::/0

Chain intIN (1 references)
 pkts bytes target     prot opt in     out     source               destination
Â¬
    0     0 ACCEPT     all       *      *       ::/0
Â¬ fe80::/ffc0::
    4   384 ACCEPT     all       *      *       ::/0               ff02::/16

Chain intOUT (1 references)
 pkts bytes target     prot opt in     out     source               destination
Â¬
    0     0 ACCEPT     all       *      *       ::/0
Â¬ fe80::/ffc0::
    4   384 ACCEPT     all       *      *       ::/0               ff02::/16
    0     0 LOG        all       *      *       ::/0               ::/0
Â¬         LOG flags 0 level 7 prefix `intOUT-default:'
    0     0 DROP       all       *      *       ::/0               ::/0
```

Chapter 18. Security

18.1. Node security

It's very recommended to apply all available patches and disable all not necessary services. Also bind services to the needed IPv4/IPv6 addresses only and install local firewalling.

More to be filled...

18.2. Access limitations

Many services uses the tcp_wrapper library for access control. Below is described the use of tcp_wrapper.

More to be filled...

18.3. IPv6 security auditing

Currently there are no comfortable tools out which are able to check a system over network for IPv6 security issues. Neither Nessus nor any commercial security scanner is as far as I know able to scan IPv6 addresses.

18.3.1. Legal issues

ATTENTION: always take care that you only scan your own systems or after receiving a written order, otherwise legal issues are able to come up to you. CHECK destination IPv6 addresses TWICE before starting a scan.

18.3.2. Security auditing using IPv6-enabled netcat

With the IPv6-enabled netcat (see IPv6+Linux-status-apps/security-auditing for more) you can run a portscan by wrapping a script around which run through a port range, grab banners and so on. Usage example:

```
# nc6 ::1 daytime
13 JUL 2002 11:22:22 CEST
```

18.3.3. Security auditing using IPv6-enabled nmap

NMap, one of the best portscaner around the world, supports IPv6 since version 3.10ALPHA1. Usage example:

```
# nmap -6 -sT ::1
Starting nmap V. 3.10ALPHA3 ( www.insecure.org/nmap/ )
Interesting ports on localhost6 (::1):
(The 1600 ports scanned but not shown below are in state: closed)
Port       State       Service
22/tcp     open        ssh
53/tcp     open        domain
515/tcp    open        printer
2401/tcp   open        cvspserver
Nmap run completed -- 1 IP address (1 host up) scanned in 0.525 seconds
```

18.3.4. Security auditing using IPv6-enabled strobe

Strobe is a (compared to NMap) more a low budget portscanner, but there is an IPv6-enabling patch available (see IPv6+Linux-status-apps/security-auditing for more). Usage example:

```
# ./strobe ::1 strobe 1.05 (c) 1995-1999 Julian Assange <proff@iq.org>.
::1 2401 unassigned unknown
::1 22 ssh Secure Shell - RSA encrypted rsh
::1 515 printer spooler (lpd)
::1 6010 unassigned unknown
::1 53 domain Domain Name Server
```

Note: strobe isn't really developed further on, the shown version number isn't the right one.

18.3.5. Audit results

If the result of an audit mismatch your IPv6 security policy, use IPv6 firewalling to close the holes, e.g. using netfilter6 (see Firewalling/Netfilter6 for more).

Info: More detailed information concerning IPv6 Security can be found here:

- IETF drafts - IPv6 Operations (v6ops)
- RFC 3964 / Security Considerations for 6to4

Chapter 19. Encryption and Authentication

Unlike in IPv4, encryption and authentication is a mandatory feature of IPv6. Those features are normally implemented using IPsec (which can be also used by IPv4).

19.1. Modes of using encryption and authentication

Two modes of encryption and authentication of a connection are possible:

19.1.1. Transport mode

Transport mode is a real end-to-end connection mode. Here, only the payload (usually ICMP, TCP or UDP) is encrypted with their particular header, while the IP header is not encrypted (but usually included in authentication).

Using AES-128 for encryption and SHA1 for authentication, this mode decreases the MTU by 42 octets.

19.1.2. Tunnel mode

Tunnel mode can be used either for end-to-end or for gateway-to-gateway connection modes. Here, the complete IP packet is being encrypted and gets a new IP header prepended, all together constituing a new IP packet (this mechanism is also known as "encapsulation")

This mode usually decreases the MTU by 40 octets from the MTU of transport mode. I.e. using AES-128 for encryption and SHA1 for authentication 82 octets less than the normal MTU.

19.2. Support in kernel (ESP and AH)

19.2.1. Support in vanilla Linux kernel 2.4.x

At the time of writing missing in vanilla up to 2.4.28. There was an issue about keeping the Linux kernel source free of export/import-control-laws regarding encryption code. This is also one case why FreeS/WAN project wasn't included in vanilla source. Perhaps a backport from 2.6.x will be done in the future.

19.2.2. Support in vanilla Linux kernel 2.6.x

Current versions (as time of writing 2.6.9 and upper) support native IPsec for IPv4 and IPv6.

Implementation was helped by the USAGI project.

19.3. Automatic key exchange (IKE)

IPsec requires a key exchange of a secret. This is mostly done automatically by so called IKE daemons. They also handle the authentication of the peers, either by a common known secret (so called "pre-shared secret") or by RSA keys (which can also be used from X.509 certificates).

Currently, two different IKE daemons are available for Linux, which totally differ in configuration and usage.

I prefer "pluto" from the *S/WAN implementation because of the easier and one-config-only setup.

19.3.1. IKE daemon "racoon"

The IKE daemon "racoon" is taken from the KAME project and ported to Linux. Modern Linux distributions contain this daemon in the package "ipsec-tools". Two executables are required for a proper IPsec setup. Take a look on Linux Advanced Routing & Traffic Control HOWTO / IPSEC, too.

19.3.1.1. Manipulation of the IPsec SA/SP database with the tool "setkey"

"setkey" is important to define the security policy (SP) for the kernel.

File: /etc/racoon/setkey.sh

- Example for an end-to-end encrypted connection in transport mode

```
#!/sbin/setkey -f
flush;
spdflush;
spdadd 2001:db8:1:1::1 2001:db8:2:2::2 any -P out ipsec esp/transport//require;
spdadd 2001:db8:2:2::2 2001:db8:1:1::1 any -P in  ipsec esp/transport//require;
```

- Example for a end-to-end encrypted connection in tunnel mode

```
#!/sbin/setkey -f
flush;
spdflush;
spdadd 2001:db8:1:1::1 2001:db8:2:2::2 any -P out ipsec
Â¬ esp/tunnel/2001:db8:1:1::1-2001:db8:2:2::2/require;
spdadd 2001:db8:2:2::2 2001:db8:1:1::1 any -P in  ipsec
Â¬ esp/tunnel/2001:db8:2:2::2-2001:db8:1:1::1/require;
```

For the other peer, you have to replace "in" with "out".

19.3.1.2. Configuration of the IKE daemon "racoon"

"racoon" requires a configuration file for proper execution. It includes the related settings to the security policy, which should be set up previously using "setkey".

File: /etc/racoon/racoon.conf

```
# Racoon IKE daemon configuration file.
# See 'man racoon.conf' for a description of the format and entries.
path include "/etc/racoon";
path pre_shared_key "/etc/racoon/psk.txt";

listen
{
        isakmp 2001:db8:1:1::1;
}

remote 2001:db8:2:2::2
{
        exchange_mode main;
        lifetime time 24 hour;
        proposal
```

```
        {
                encryption_algorithm 3des;
                hash_algorithm md5;
                authentication_method pre_shared_key;
                dh_group 2;
        }
}

# gateway-to-gateway
sainfo address 2001:db8:1:1::1 any address 2001:db8:2:2::2 any
{
        lifetime time 1 hour;
        encryption_algorithm 3des;
        authentication_algorithm hmac_md5;
        compression_algorithm deflate;
}

sainfo address 2001:db8:2:2::2 any address 2001:db8:1:1::1 any
{
        lifetime time 1 hour;
        encryption_algorithm 3des;
        authentication_algorithm hmac_md5;
        compression_algorithm deflate;
}
```

Also set up the pre-shared secret:

File: /etc/racoon/psk.txt

```
# file for pre-shared keys used for IKE authentication
# format is: 'identifier' 'key'

2001:db8:2:2::2 verysecret
```

19.3.1.3. Running IPsec with IKE daemon "racoon"

At least the daemon needs to be started. For the first time, use debug and foreground mode. The following example shows a successful IKE phase 1 (ISAKMP-SA) and 2 (IPsec-SA) negotiation:

```
# racoon -F -v -f /etc/racoon/racoon.conf
Foreground mode.
2005-01-01 20:30:15: INFO: @(#)ipsec-tools 0.3.3 (http://ipsec-tools.sourceforge.net)
2005-01-01 20:30:15: INFO: @(#)This product linked
Â¬ OpenSSL 0.9.7a Feb 19 2003 (http://www.openssl.org/)
2005-01-01 20:30:15: INFO: 2001:db8:1:1::1[500] used as isakmp port (fd=7)
2005-01-01 20:31:06: INFO: IPsec-SA request for 2001:db8:2:2::2
Â¬ queued due to no phase1 found.
2005-01-01 20:31:06: INFO: initiate new phase 1 negotiation:
Â¬ 2001:db8:1:1::1[500]<=>2001:db8:2:2::2[500]
2005-01-01 20:31:06: INFO: begin Identity Protection mode.
2005-01-01 20:31:09: INFO: ISAKMP-SA established
Â¬ 2001:db8:1:1::1[500]-2001:db8:2:2::2[500] spi:da3d3693289c9698:ac039a402b2db401
2005-01-01 20:31:09: INFO: initiate new phase 2 negotiation:
Â¬ 2001:6f8:900:94::2[0]<=>2001:db8:2:2::2[0]
2005-01-01 20:31:10: INFO: IPsec-SA established:
Â¬ ESP/Tunnel 2001:db8:2:2::2->2001:db8:1:1::1 spi=253935531(0xf22bfab)
2005-01-01 20:31:10: INFO: IPsec-SA established:
Â¬ ESP/Tunnel 2001:db8:1:1::1->2001:db8:2:2::2 spi=175002564(0xa6e53c4)
```

Each direction got its own IPsec-SA (like defined in the IPsec standard). With "tcpdump" on the related interface, you will see as result of an IPv6 ping:

```
20:35:55.305707 2001:db8:1:1::1 > 2001:db8:2:2::2: ESP(spi=0x0a6e53c4,seq=0x3)
20:35:55.537522 2001:db8:2:2::2 > 2001:db8:1:1::1: ESP(spi=0x0f22bfab,seq=0x3)
```

As expected, the negotiated SPIs are being used here.

And using "setkey", current active parameters are shown:

```
# setkey -D
2001:db8:1:1::1 2001:db8:2:2::2
        esp mode=tunnel spi=175002564(0x0a6e53c4) reqid=0(0x00000000)
        E: 3des-cbc  bd26bc45 aea0d249 ef9c6b89 7056080f 5d9fa49c 924e2edd
        A: hmac-md5  60c2c505 517dd8b7 c9609128 a5efc2db
        seq=0x00000000 replay=4 flags=0x00000000 state=mature
        created: Jan  1 20:31:10 2005   current: Jan  1 20:40:47 2005
        diff: 577(s)    hard: 3600(s)   soft: 2880(s)
        last: Jan  1 20:35:05 2005      hard: 0(s)      soft: 0(s)
        current: 540(bytes)     hard: 0(bytes)  soft: 0(bytes)
        allocated: 3    hard: 0 soft: 0
        sadb_seq=1 pid=22358 refcnt=0
2001:db8:2:2::2 2001:db8:1:1::1
        esp mode=tunnel spi=253935531(0x0f22bfab) reqid=0(0x00000000)
        E: 3des-cbc  c1ddba65 83debd62 3f6683c1 20e747ac 933d203f 4777a7ce
        A: hmac-md5  3f957db9 9adddc8c 44e5739d 3f53ca0e
        seq=0x00000000 replay=4 flags=0x00000000 state=mature
        created: Jan  1 20:31:10 2005   current: Jan  1 20:40:47 2005
        diff: 577(s)    hard: 3600(s)   soft: 2880(s)
        last: Jan  1 20:35:05 2005      hard: 0(s)      soft: 0(s)
        current: 312(bytes)     hard: 0(bytes)  soft: 0(bytes)
        allocated: 3    hard: 0 soft: 0
        sadb_seq=0 pid=22358 refcnt=0
```

19.3.2. IKE daemon "pluto"

The IKE daemon "pluto" is included in distributions of the *S/WAN projects. *S/WAN project starts at the beginning as FreeS/WAN. Unfortunately, the FreeS/WAN project stopped further development in 2004. Because of the slow pace of development in the past, two spin-offs started: strongSwan and Openswan. Today, readily installable packages are available for at least Openswan (included in Fedora Core 3).

A major difference to "racoon", only one configuration file is required. Also, an initscript exists for automatic setup after booting.

19.3.2.1. Configuration of the IKE daemon "pluto"

The configuration is very similar to the IPv4 one, only one important option is necessary.

File: /etc/ipsec.conf

```
# /etc/ipsec.conf - Openswan IPsec configuration file
#
# Manual:     ipsec.conf.5
version 2.0     # conforms to second version of ipsec.conf specification

# basic configuration
config setup
        # Debug-logging controls:  "none" for (almost) none, "all" for lots.
        # klipsdebug=none
        # plutodebug="control parsing"
```

```
#Disable Opportunistic Encryption
include /etc/ipsec.d/examples/no_oe.conf

conn ipv6-p1-p2
        connaddrfamily=ipv6       # Important for IPv6!
        left=2001:db8:1:1::1
        right=2001:db8:2:2::2
        authby=secret
        esp=aes128-sha1
        ike=aes128-sha-modp1024
        type=transport
        #type=tunnel
        compress=no
        #compress=yes
        auto=add
        #auto=start
```

Don't forget to define the pre-shared secret here also.

File: /etc/ipsec.secrets

```
2001:db8:1:1::1 2001:db8:2:2::2 : PSK      "verysecret"
```

19.3.2.2. Running IPsec with IKE daemon "pluto"

If installation of Openswan was successfully, an initscript should exist for starting IPsec, simply run (on each peer):

```
# /etc/rc.d/init.d/ipsec start
```

Afterwards, start this connection on one peer. If you saw the line "IPsec SA established", all worked fine.

```
# ipsec auto --up ipv6-peer1-peer2
104 "ipv6-p1-p2" #1: STATE_MAIN_I1: initiate
106 "ipv6-p1-p2" #1: STATE_MAIN_I2: sent MI2, expecting MR2
108 "ipv6-p1-p2" #1: STATE_MAIN_I3: sent MI3, expecting MR3
004 "ipv6-p1-p2" #1: STATE_MAIN_I4: ISAKMP SA established
112 "ipv6-p1-p2" #2: STATE_QUICK_I1: initiate
004 "ipv6-p1-p2" #2: STATE_QUICK_I2: sent QI2,
Â¬ IPsec SA established {ESP=>0xa98b7710 <0xa51e1f22}
```

Because *S/WAN and setkey/racoon do use the same IPsec implementation in Linux 2.6.x kernel, "setkey" can be used here too to show current active parameters:

```
# setkey -D
2001:db8:1:1::1 2001:db8:2:2::2
        esp mode=transport spi=2844489488(0xa98b7710) reqid=16385(0x00004001)
        E: aes-cbc  082ee274 2744bae5 7451da37 1162b483
        A: hmac-sha1  b7803753 757417da 477b1c1a 64070455 ab79082c
        seq=0x00000000 replay=64 flags=0x00000000 state=mature
        created: Jan  1 21:16:32 2005   current: Jan  1 21:22:20 2005
        diff: 348(s)    hard: 0(s)      soft: 0(s)
        last:                           hard: 0(s)      soft: 0(s)
        current: 0(bytes)       hard: 0(bytes)  soft: 0(bytes)
        allocated: 0    hard: 0 soft: 0
        sadb_seq=1 pid=23825 refcnt=0
2001:db8:2:2::2 2001:db8:1:1::1
        esp mode=transport spi=2770214690(0xa51e1f22) reqid=16385(0x00004001)
        E: aes-cbc  6f59cc30 8d856056 65e07b76 552cac18
        A: hmac-sha1  c7c7d82b abfca8b1 5440021f e0c3b335 975b508b
        seq=0x00000000 replay=64 flags=0x00000000 state=mature
```

```
        created: Jan  1 21:16:31 2005   current: Jan  1 21:22:20 2005
        diff: 349(s)    hard: 0(s)      soft: 0(s)
        last:                           hard: 0(s)      soft: 0(s)
        current: 0(bytes)       hard: 0(bytes)  soft: 0(bytes)
        allocated: 0    hard: 0 soft: 0
        sadb_seq=0 pid=23825 refcnt=0
```

19.4. Additional informations:

On Linux Kernel 2.6.x you can get the policy and status of IPsec also using "ip":

```
# ip xfrm policy
...

# ip xfrm state
...
```

Chapter 20. Quality of Service (QoS)

IPv6 supports QoS with use of Flow Labels and Traffic Classes. This can be controlled using "tc" (contained in package "iproute").

Additional infos:

- RFC 3697 / IPv6 Flow Label Specification

more to be filled...

Chapter 21. Hints for IPv6-enabled daemons

Here some hints are shown for IPv6-enabled daemons.

21.1. Berkeley Internet Name Daemon BIND (named)

IPv6 is supported since version 9. Always use newest available version. At least version 9.1.3 must be used, older versions can contain remote exploitable security holes.

21.1.1. Listening on IPv6 addresses

Note: unlike in IPv4 current versions doesn't allow to bind a server socket to dedicated IPv6 addresses, so only *any* or *none* are valid. Because this can be a security issue, check the Access Control List (ACL) section below, too!

21.1.1.1. Enable BIND named for listening on IPv6 address

To enable IPv6 for listening, following options are requested to change

```
options {
        # sure other options here, too
        listen-on-v6 { any; };
};
```

This should result after restart in e.g.

```
# netstat -lnptu |grep "named\W*$"
tcp 0 0 :::53         :::*      LISTEN 1234/named
Â¬ # incoming TCP requests
udp 0 0 1.2.3.4:53    0.0.0.0:*        1234/named
Â¬ # incoming UDP requests to IPv4 1.2.3.4
udp 0 0 127.0.0.1:53  0.0.0.0:*        1234/named
Â¬ # incoming UDP requests to IPv4 localhost
udp 0 0 0.0.0.0:32868 0.0.0.0:*        1234/named
Â¬ # dynamic chosen port for outgoing queries
udp 0 0 :::53         :::*             1234/named
Â¬ # incoming UDP request to any IPv6
```

And a simple test looks like

```
# dig localhost @::1
```

and should show you a result.

21.1.1.2. Disable BIND named for listening on IPv6 address

To disable IPv6 for listening, following options are requested to change

```
options {
        # sure other options here, too
        listen-on-v6 { none; };
};
```

21.1.2. IPv6 enabled Access Control Lists (ACL)

IPv6 enabled ACLs are possible and should be used whenever it's possible. An example looks like following:

```
acl internal-net {
        127.0.0.1;
        1.2.3.0/24;
        2001:0db8:100::/56;
        ::1/128;
        ::ffff:1.2.3.4/128;
};
acl ns-internal-net {
        1.2.3.4;
        1.2.3.5;
        2001:0db8:100::4/128;
        2001:0db8:100::5/128;
};
```

This ACLs can be used e.g. for queries of clients and transfer zones to secondary name-servers. This prevents also your caching name-server to be used from outside using IPv6.

```
options {
        # sure other options here, too
        listen-on-v6 { none; };
        allow-query { internal-net; };
        allow-transfer { ns-internal-net; };
};
```

It's also possible to set the *allow-query* and *allow-transfer* option for most of single zone definitions, too.

21.1.3. Sending queries with dedicated IPv6 address

This option is not required, but perhaps needed:

```
query-source-v6 address <ipv6address|*> port <port|*>;
```

21.1.4. Per zone defined dedicated IPv6 addresses

It's also possible to define per zone some IPv6 addresses.

21.1.4.1. Transfer source address

Transfer source address is used for outgoing zone transfers:

```
transfer-source-v6 <ipv6addr|*> [port port];
```

21.1.4.2. Notify source address

Notify source address is used for outgoing notify messages:

```
notify-source-v6 <ipv6addr|*> [port port];
```

21.1.5. IPv6 DNS zone files examples

Some information can be also found at IPv6 DNS Setup Information (article). Perhaps also helpful is the IPv6 Reverse DNS zone builder for BIND 8/9 (webtool).

21.1.6. Serving IPv6 related DNS data

For IPv6 new types and root zones for reverse lookups are defined:

- AAAA and reverse IP6.INT: specified in RFC 1886 / DNS Extensions to support IP version 6, usable since BIND version 4.9.6
- A6, DNAME (DEPRECATED NOW!) and reverse IP6.ARPA: specified in RFC 2874 / DNS Extensions to Support IPv6 Address Aggregation and Renumbering, usable since BIND 9, but see also an information about the current state at Domain Name System Extension (dnsext)

Perhaps filled later more content, for the meantime take a look at given RFCs and

- AAAA and reverse IP6.INT: IPv6 DNS Setup Information
- A6, DNAME (DEPRECATED NOW!) and reverse IP6.ARPA: take a look into chapter 4 and 6 of the BIND 9 Administrator Reference Manual (ARM) distributed with the bind-package or get this here: BIND manual version 9.3

Because IP6.INT is deprecated (but still in use), a DNS server which will support IPv6 information has to serve both reverse zones.

21.1.6.1. Current best practice

Because there are some troubles around using the new formats, current best practice is:

Forward lookup support:

- AAAA

Reverse lookup support:

- Reverse nibble format for zone ip6.int (FOR BACKWARD COMPATIBILITY)
- Reverse nibble format for zone ip6.arpa (RECOMMENDED)

21.1.7. Checking IPv6-enabled connect

To check, whether BIND is listening on an IPv6 socket and serving data see following examples.

21.1.7.1. IPv6 connect, but denied by ACL

Specifying a dedicated server for the query, an IPv6 connect can be forced:

```
$ host -t aaaa www.6bone.net 2001:0db8:200:f101::1
Using domain server:
Name: 2001:0db8:200:f101::1
Address: 2001:0db8:200:f101::1#53
Aliases:
```

```
Host www.6bone.net. not found: 5(REFUSED)
```

Related log entry looks like following:

```
Jan 3 12:43:32 gate named[12347]: client
Â¬ 2001:0db8:200:f101:212:34ff:fe12:3456#32770:
 query denied
```

If you see such entries in the log, check whether requests from this client should be allowed and perhaps review your ACL configuration.

21.1.7.2. Successful IPv6 connect

A successful IPv6 connect looks like following:

```
$ host -t aaaa www.6bone.net 2001:0db8:200:f101::1
Using domain server:
Name: 2001:0db8:200:f101::1
Address: 2001:0db8:200:f101::1#53
Aliases:

www.6bone.net. is an alias for 6bone.net.
6bone.net. has AAAA address 3ffe:b00:c18:1::10
```

21.2. Internet super daemon (xinetd)

IPv6 is supported since xinetd version around 1.8.9. Always use newest available version. At least version 2.3.3 must be used, older versions can contain remote exploitable security holes.

Some Linux distribution contain an extra package for the IPv6 enabled xinetd, some others start the IPv6-enabled xinetd if following variable is set: NETWORKING_IPV6="yes", mostly done by /etc/sysconfig/network (only valid for Red Hat like distributions). In newer releases, one binary supports IPv4 and IPv6.

If you enable a built-in service like e.g. daytime by modifying the configuration file in /etc/xinetd.d/daytime like

```
# diff -u /etc/xinetd.d/daytime.orig /etc/xinetd.d/daytime
--- /etc/xinetd.d/daytime.orig Sun Dec 16 19:00:14 2001
+++ /etc/xinetd.d/daytime Sun Dec 16 19:00:22 2001
@@ -10,5 +10,5 @@
        protocol = tcp
        user = root
        wait = no
-       disable = yes
+       disable = no
 }
```

After restarting the xinetd you should get a positive result like:

```
# netstat -lnptu -A inet6 |grep "xinetd*"
tcp 0 0 ::ffff:192.168.1.1:993  :::*  LISTEN  12345/xinetd-ipv6
tcp 0 0 :::13                   :::*  LISTEN  12345/xinetd-ipv6 <- service
Â¬ daytime/tcp
tcp 0 0 ::ffff:192.168.1.1:143  :::*  LISTEN  12345/xinetd-ipv6
```

Shown example also displays an IMAP and IMAP-SSL IPv4-only listening xinetd.

Note: earlier versions had a problem that an IPv4-only xinetd won't start on an IPv6-enabled node and also the IPv6-enabled xinetd won't start on an IPv4-only node. This is known to be fixed in later versions, at least version 2.3.11.

21.3. Webserver Apache2 (httpd2)

Apache web server supports IPv6 native by maintainers since 2.0.14. Available patches for the older 1.3.x series are not current and shouldn't be used in public environment, but available at KAME / Misc.

21.3.1. Listening on IPv6 addresses

Note: virtual hosts on IPv6 addresses are broken in versions until 2.0.28 (a patch is available for 2.0.28). But always try latest available version first because earlier versions had some security issues.

21.3.1.1. Virtual host listen on an IPv6 address only

```
Listen [2001:0db8:100::1]:80
<VirtualHost [2001:0db8:100::1]:80>
        ServerName ipv6only.yourdomain.yourtopleveldomain
        # ...sure more config lines
</VirtualHost>
```

21.3.1.2. Virtual host listen on an IPv6 and on an IPv4 address

```
Listen [2001:0db8:100::2]:80
Listen 1.2.3.4:80
<VirtualHost [2001:0db8:100::2]:80 1.2.3.4:80>
        ServerName ipv6andipv4.yourdomain.yourtopleveldomain
        # ...sure more config lines
</VirtualHost>
```

This should result after restart in e.g.

```
# netstat -lnptu |grep "httpd2\W*$"
tcp 0 0 1.2.3.4:80          0.0.0.0:* LISTEN 12345/httpd2
tcp 0 0 2001:0db8:100::1:80 :::*      LISTEN 12345/httpd2
tcp 0 0 2001:0db8:100::2:80 :::*      LISTEN 12345/httpd2
```

For simple tests use the telnet example already shown.

21.3.1.3. Additional notes

- Apache2 supports a method called "sendfile" to speedup serving data. Some NIC drivers also support offline checksumming. In some cases, this can lead to connection problems and invalid TCP checksums. In this cases, disable "sendfile" either by recompiling using configure option "--without-sendfile" or by using the "EnableSendfile off" directive in configuration file.

21.4. Router Advertisement Daemon (radvd)

The router advertisement daemon is very useful on a LAN, if clients should be auto-configured. The daemon itself should run on the Linux default IPv6 gateway router (it's not required that this is also the default IPv4 gateway, so pay attention who on your LAN is sending router advertisements).

You can specify some information and flags which should be contained in the advertisement. Common used are

- Prefix (needed)
- Lifetime of the prefix
- Frequency of sending advertisements (optional)

After a proper configuration, the daemon sends advertisements through specified interfaces and clients are hopefully receive them and auto-magically configure addresses with received prefix and the default route.

21.4.1. Configuring radvd

21.4.1.1. Simple configuration

Radvd's config file is normally /etc/radvd.conf. An simple example looks like following:

```
interface eth0 {
        AdvSendAdvert on;
        MinRtrAdvInterval 3;
        MaxRtrAdvInterval 10;
        prefix 2001:0db8:0100:f101::/64 {
                AdvOnLink on;
                AdvAutonomous on;
                AdvRouterAddr on;
        };
};
```

This results on client side in

```
# ip -6 addr show eth0
3: eth0: <BROADCAST,MULTICAST,UP> mtu 1500 qdisc pfifo_fast qlen 100
    inet6 2001:0db8:100:f101:2e0:12ff:fe34:1234/64 scope global dynamic
       valid_lft 2591992sec preferred_lft 604792sec
    inet6 fe80::2e0:12ff:fe34:1234/10 scope link
```

Because no lifetime was defined, a very high value was used.

21.4.1.2. Special 6to4 configuration

Version since 0.6.2pl3 support the automatic (re)-generation of the prefix depending on an IPv4 address of a specified interface. This can be used to distribute advertisements in a LAN after the 6to4 tunneling has changed. Mostly used behind a dynamic dial-on-demand Linux router. Because of the sure shorter lifetime of such prefix (after each dial-up, another prefix is valid), the lifetime configured to minimal values:

```
interface eth0 {
        AdvSendAdvert on;
        MinRtrAdvInterval 3;
        MaxRtrAdvInterval 10;
        prefix 0:0:0:f101::/64 {
                AdvOnLink off;
                AdvAutonomous on;
                AdvRouterAddr on;
                Base6to4Interface ppp0;
                AdvPreferredLifetime 20;
                AdvValidLifetime 30;
        };
};
```

This results on client side in (assuming, ppp0 has currently 1.2.3.4 as local IPv4 address):

```
# /sbin/ip -6 addr show eth0
3: eth0: <BROADCAST,MULTICAST,UP> mtu 1500 qdisc pfifo_fast qlen 100
   inet6 2002:0102:0304:f101:2e0:12ff:fe34:1234/64 scope global dynamic
      valid_lft 22sec preferred_lft 12sec
   inet6 fe80::2e0:12ff:fe34:1234/10 scope link
```

Because a small lifetime was defined, such prefix will be thrown away quickly, if no related advertisement was received.

Additional note: if you do not used special 6to4 support in initscripts, you have to setup a special route on the internal interface on the router, otherwise you will get some backrouting problems. for the example showh here:

```
# /sbin/ip -6 route add 2002:0102:0304:f101::/64 dev eth0 metric 1
```

This route needs to be replaced every time the prefix changes, which is the case after a new IPv4 address was assigned to the dial-up interface.

21.4.2. Debugging

A program called "radvdump" can help you looking into sent or received advertisements. Simple to use:

```
# radvdump
Router advertisement from fe80::280:c8ff:feb9:cef9 (hoplimit 255)
        AdvCurHopLimit: 64
        AdvManagedFlag: off
        AdvOtherConfigFlag: off
        AdvHomeAgentFlag: off
        AdvReachableTime: 0
        AdvRetransTimer: 0
        Prefix 2002:0102:0304:f101::/64
                AdvValidLifetime: 30
                AdvPreferredLifetime: 20
                AdvOnLink: off
                AdvAutonomous: on
                AdvRouterAddr: on
        Prefix 2001:0db8:100:f101::/64
                AdvValidLifetime: 2592000
                AdvPreferredLifetime: 604800
                AdvOnLink: on
                AdvAutonomous: on
                AdvRouterAddr: on
        AdvSourceLLAddress: 00 80 12 34 56 78
```

Output shows you each advertisement package in readable format. You should see your configured values here again, if not, perhaps it's not your radvd which sends the advertisement...look for another router on the link (and take the LLAddress, which is the MAC address for tracing).

21.5. Dynamic Host Configuration v6 Server (dhcp6s)

DHCPv6 can be used for stateful configurations. The daemon itself need not necessary run on the Linux default IPv6 gateway router.

You can specify more information than by using radvd. The are most similar to IPv4 DHCP server.

After a proper configuration, the daemon reacts on received ICMPv6 multicast packets sent by a client to address ff02::16

21.5.1. Configuration of the DHCPv6 server (dhcp6s)

21.5.1.1. Simple configuration

dhcp6s's config file is normally /etc/dhcp6s.conf. An simple example looks like following:

```
interface eth0 {
        server-preference 255;
        renew-time 60;
        rebind-time 90;
        prefer-life-time 130;
        valid-life-time 200;
        allow rapid-commit;
        option dns_servers 2001:db8:0:f101::1 sub.domain.example;
        link AAA {
                range 2001:db8:0:f101::1000 to 2001:db8:0:f101::ffff/64;
                prefix 2001:db8:0:f101::/64;
        };
};
```

21.5.2. Configuration of the DHCPv6 client (dhcp6c)

21.5.2.1. Simple configuration

dhcp6c's config file is normally /etc/dhcp6c.conf. An simple example looks like following:

```
interface eth0 {
        send rapid-commit;
        request domain-name-servers;
};
```

21.5.3. Usage

21.5.3.1. dhcpv6_server

Start server, e.g.

```
# service dhcp6s start
```

21.5.3.2. dhcpv6_client

Start client in foreground, e.g.

```
# # dhcp6c -f eth0
```

21.5.4. Debugging

21.5.4.1. dhcpv6_server

The server has one foreground and two debug toggles (both should be used for debugging), here is an example:

```
# dhcp6s -d -D -f eth0
```

21.5.4.2. dhcpv6_client

The client has one foreground and two debug toggles, here is an example:

```
# dhcp6c -d -f eth0
Oct/03/2005 17:18:16 dhcpv6 doesn't support hardware type 776
Oct/03/2005 17:18:16 doesn't support sit0 address family 0
Oct/03/2005 17:18:16 netlink_recv_rtgenmsg error
Oct/03/2005 17:18:16 netlink_recv_rtgenmsg error
Oct/03/2005 17:18:17 status code for this address is: success
Oct/03/2005 17:18:17 status code: success
Oct/03/2005 17:18:17 netlink_recv_rtgenmsg error
Oct/03/2005 17:18:17 netlink_recv_rtgenmsg error
Oct/03/2005 17:18:17 assigned address 2001:db8:0:f101::1002 prefix len is not in any RAs prefix l
Oct/03/2005 17:18:17 renew time 60, rebind time 9
```

Note that the netlink error messages have no impact.

21.6. tcp_wrapper

tcp_wrapper is a library which can help you to protect service against misuse.

21.6.1. Filtering capabilities

You can use tcp_wrapper for

- Filtering against source addresses (IPv4 or IPv6)
- Filtering against users (requires a running ident daemon on the client)

21.6.2. Which program uses tcp_wrapper

Following are known:

- Each service which is called by xinetd (if xinetd is compiled using tcp_wrapper library)
- sshd (if compiled using tcp_wrapper)

21.6.3. Usage

tcp_wrapper is controlled by two files name /etc/hosts.allow and /etc/hosts.deny. For more information see

```
$ man hosts.allow
```

21.6.3.1. Example for /etc/hosts.allow

In this file, each service which should be positive filtered (means connects are accepted) need a line.

```
sshd:           1.2.3. [2001:0db8:100:200::]/64
daytime-stream: 1.2.3. [2001:0db8:100:200::]/64
```

Note: there are broken implementations around, which uses following broken IPv6 network description: [2001:0db8:100:200::/64]. Hopefully, such versions will be fixed soon.

21.6.3.2. Example for /etc/hosts.deny

This file contains all negative filter entries and should normally deny the rest using

```
ALL: ALL
```

If this node is a more sensible one you can replace the standard line above with this one, but this can cause a DoS attack (load of mailer and spool directory), if too many connects were made in short time. Perhaps a logwatch is better for such issues.

```
ALL: ALL: spawn (echo "Attempt from %h %a to %d at `date`"
 | tee -a /var/log/tcp.deny.log | mail root@localhost)
```

21.6.4. Logging

Depending on the entry in the syslog daemon configuration file /etc/syslog.conf the tcp_wrapper logs normally into /var/log/secure.

21.6.4.1. Refused connection

A refused connection via IPv4 to an xinetd covered daytime service produces a line like following example

```
Jan 2 20:40:44 gate xinetd-ipv6[12346]: FAIL: daytime-stream libwrap
Â¬ from=::ffff:1.2.3.4
Jan 2 20:32:06 gate xinetd-ipv6[12346]: FAIL: daytime-stream libwrap
 from=2001:0db8:100:200::212:34ff:fe12:3456
```

A refused connection via IPv4 to an dual-listen sshd produces a line like following example

```
Jan 2 20:24:17 gate sshd[12345]: refused connect from ::ffff:1.2.3.4
Â¬ (::ffff:1.2.3.4)
Jan 2 20:39:33 gate sshd[12345]: refused connect
 from 2001:0db8:100:200::212:34ff:fe12:3456
Â¬ (2001:0db8:100:200::212:34ff:fe12:3456)
```

21.6.4.2. Permitted connection

A permitted connection via IPv4 to an xinetd covered daytime service produces a line like following example

```
Jan 2 20:37:50 gate xinetd-ipv6[12346]: START: daytime-stream pid=0
Â¬ from=::ffff:1.2.3.4
Jan 2 20:37:56 gate xinetd-ipv6[12346]: START: daytime-stream pid=0
 from=2001:0db8:100:200::212:34ff:fe12:3456
```

A permitted connection via IPv4 to an dual-listen sshd produces a line like following example

```
Jan 2 20:43:10 gate sshd[21975]: Accepted password for user from ::ffff:1.2.3.4
```

```
Â¬ port 33381 ssh2
Jan 2 20:42:19 gate sshd[12345]: Accepted password for user
 from 2001:0db8:100:200::212:34ff:fe12:3456 port 33380 ssh2
```

21.7. vsftpd

21.7.1. Listening on IPv6 addresses

Edit the configuration file, ususally /etc/vsftpd/vsftpd.conf, and adjust the listen option like

```
listen_ipv6=yes
```

That's all.

21.8. proftpd

21.8.1. Listening on IPv6 addresses

Edit the configuration file, ususally /etc/proftpd.conf, but take care, not 100% logical in virtual host setup

```
<VirtualHost 192.0.2.1>
        ...
        Bind 2001:0DB8::1
        ...
</VirtualHost>
```

That's all.

21.9. Other daemons

Nowadays it's mostly simple, look for either a command line option or a configuration value to enable IPv6 listening. See manual page of the daemon or check related FAQs. It can happen that you can bind a daemon only to the IPv6-"any"-address (::) and not to bind to a dedicated IPv6 address, because the lack of support (depends on that what the programmer has implemented so far...).

Chapter 22. Programming

22.1. Programming using C-API

Related RFCs:

- RFC 3493 / Basic Socket Interface Extensions for IPv6
- RFC 3542 / Advanced Sockets Application Program Interface (API) for IPv6

Following contents of this section is contributed by John Wenker, Sr. Software Engineer Performance Technologies San Diego, CA USA http://www.pt.com/.

This section describes how to write IPv6 client-server applications under the Linux operating system. First thing's first, and credit must be given where it is due. The information contained in this section is derived from Chapters 2 through 4 of IPv6 Network Programming by Jun-ichiro itojun Hagino (ISBN 1-55558-318-0). The reader is encouraged to consult that book for more detailed information. It describes how to convert IPv4 applications to be IPv6 compatible in a protocol-independent way, and describes some of the common problems encountered during the conversion along with suggested solutions. At the time of this writing, this is the only book of which the author is aware that specifically addresses how to program IPv6 applications [since writing this section, the author has also become aware of the Porting applications to IPv6 HowTo by Eva M. Castro at Since writing this HowTo, the author has also become aware of the Porting applications to IPv6 HowTo by Eva M. Castro at http://jungla.dit.upm.es/~ecastro/IPv6-web/ipv6.html]. Unfortunately, of the almost 360 pages in the book, maybe 60 are actually useful (the chapters mentioned). Nevertheless, without the guidance of that book, the author would have been unable to perform his job duties or compose this HowTo. While most (but certainly not all) of the information in the Hagino book is available via the Linux 'man' pages, application programmers will save a significant amount of time and frustration by reading the indicated chapters of the book rather than searching through the 'man' pages and online documentation.

Other than the Hagino book, any other information presented in this HowTo was obtained through trial and error. Some items or explanations may not be entirely "correct" in the grand IPv6 scheme, but seem to work in practical application.

The discussion that follows assumes the reader is already experienced with the traditional TCP/IP socket API. For more information on traditional socket programming, the Internetworking with TCP/IP series of textbooks by Comer & Stevens is hard to beat, specifically Volume III: Client-Server Programming and Applications, Linux/POSIX Sockets Version (ISBN 0-13-032071-4). This HowTo also assumes that the reader has had at least a bare basic introduction to IPv6 and in particular the addressing scheme for network addresses (see Section 2.3).

22.1.1. Address Structures

This section provides a brief overview of the structures provided in the socket API to represent network addresses (or more specifically transport endpoints) when using the Internet protocols in a client-server application.

22.1.1.1. IPv4 sockaddr_in

In IPv4, network addresses are 32 bits long and define a network node. Addresses are written in dotted decimal notation, such as 192.0.2.1, where each number represents eight bits of the address. Such an IPv4 address is represented by the struct sockaddr_in data type, which is defined in <netinet/in.h>.

```
struct sockaddr_in
{
   sa_family_t    sin_family;
   in_port_t      sin_port;
   struct in_addr sin_addr;
   /* Plus some padding for alignment */
};
```

The sin_family component indicates the address family. For IPv4 addresses, this is always set to AF_INET. The sin_addr field contains the 32-bit network address (in network byte order). Finally, the sin_port component represents the transport layer port number (in network byte order). Readers should already be familiar with this structure, as this is the standard IPv4 address structure.

22.1.1.2. IPv6 sockaddr_in6

The biggest feature of IPv6 is its increased address space. Instead of 32-bit network addresses, IPv6 allots 128 bits to an address. Addresses are written in colon-hex notation of the form fe80::2c0:8cff:fe01:2345, where each hex number separated by colons represents 16 bits of the address. Two consecutive colons indicate a string of consecutive zeros for brevity, and at most only one double-colon may appear in the address. IPv6 addresses are represented by the struct sockaddr_in6 data type, also defined in <netinet/in.h>.

```
struct sockaddr_in6
{
   sa_family_t     sin6_family;
   in_port_t       sin6_port;
   uint32_t        sin6_flowinfo;
   struct in6_addr sin6_addr;
   uint32_t        sin6_scope_id;
};
```

The sin6_family, sin6_port, and sin6_addr components of the structure have the same meaning as the corresponding fields in the sockaddr_in structure. However, the sin6_family member is set to AF_INET6 for IPv6 addresses, and the sin6_addr field holds a 128-bit address instead of only 32 bits.

The sin6_flowinfo field is used for flow control, but is not yet standardized and can be ignored.

The sin6_scope_id field has an odd use, and it seems (at least to this naÃ¯ve author) that the IPv6 designers took a huge step backwards when devising this. Apparently, 128-bit IPv6 network addresses are not unique. For example, it is possible to have two hosts, on separate networks, with the same link-local address (see Figure 1). In order to pass information to a specific host, more than just the network address is required; the scope identifier must also be specified. In Linux, the network interface name is used for the scope identifier (e.g. "eth0") [be warned that the scope identifier is implementation dependent!]. Use the ifconfig(1M) command to display a list of active network interfaces.

A colon-hex network address can be augmented with the scope identifier to produce a "scoped address". The percent sign ('%') is used to delimit the network address from the scope identifier. For example, fe80::1%eth0 is a scoped IPv6 address where fe80::1 represents the 128-bit network address and eth0 is the network interface (i.e. the scope identifier). Thus, if a host resides on two networks, such as Host B in example below, the user now has to know which path to take in order to get to a particular host. In Figure 1, Host B addresses

Host A using the scoped address fe80::1%eth0, while Host C is addressed with fe80::1%eth1.

```
Host A (fe80::1) ---- eth0 ---- Host B ---- eth1 ---- Host C (fe80::1)
```

Getting back to the sockaddr_in6 structure, its sin6_scope_id field contains the index of the network interface on which a host may be found. Server applications will have this field set automatically by the socket API when they accept a connection or receive a datagram. For client applications, if a scoped address is passed as the node parameter to getaddrinfo(3) (described later in this HowTo), then the sin6_scope_id field will be filled in correctly by the system upon return from the function; if a scoped address is not supplied, then the sin6_scope_id field must be explicitly set by the client software prior to attempting to communicate with the remote server. The if_nametoindex(3) function is used to translate a network interface name into its corresponding index. It is declared in <net/if.h>.

22.1.1.3. Generic Addresses

As any programmer familiar with the traditional TCP/IP socket API knows, several socket functions deal with "generic" pointers. For example, a pointer to a generic struct sockaddr data type is passed as a parameter to some socket functions (such as connect(2) or bind(2)) rather than a pointer to a specific address type. Be careful... the sockaddr_in6 structure is larger than the generic sockaddr structure! Thus, if your program receives a generic address whose actual type is unknown (e.g. it could be an IPv4 address structure or an IPv6 address structure), you must supply sufficient storage to hold the entire address. The struct sockaddr_storage data type is defined in <bits/socket.h> for this purpose [do not #include this file directly within an application; use <sys/socket.h> as usual, and <bits/socket.h> will be implicitly included].

For example, consider the recvfrom(2) system call, which is used to receive a message from a remote peer. Its function prototype is:

```
ssize_t recvfrom( int              s,
                  void            *buf,
                  size_t           len,
                  int              flags,
                  struct sockaddr *from,
                  socklen_t       *fromlen );
```

The from parameter points to a generic sockaddr structure. If data can be received from an IPv6 peer on the socket referenced by s, then from should point to a data type of struct sockaddr_storage, as in the following dummy example:

```
/*
** Read a message from a remote peer, and return a buffer pointer to
** the caller.
**
** 's' is the file descriptor for the socket.
*/
char *rcvMsg( int s )
{
   static char             bfr[ 1025 ];  /* Where the msg is stored. */
   ssize_t                 count;
   struct sockaddr_storage ss;           /* Where the peer adr goes. */
   socklen_t               sslen;
   sslen = sizeof( ss );
   count = recvfrom( s,
                     bfr,
                     sizeof( bfr ) - 1,
                     0,
                     (struct sockaddr*) &ss,
                     &sslen );
```

```
   bfr[ count ] = '\0';   /* Null-terminates the message. */
   return bfr;
}  /* End rcvMsg() */
```

As seen in the above example, ss (a struct sockaddr_storage data object) is used to receive the peer address information, but it's address is typecast to a generic struct sockaddr* pointer in the call to recvfrom(2).

22.1.2. Lookup Functions

Traditionally, hostname and service name resolution were performed by functions such as gethostbyname(3) and getservbyname(3). These traditional lookup functions are still available, but they are not forward compatible to IPv6. Instead, the IPv6 socket API provides new lookup functions that consolidate the functionality of several traditional functions. These new lookup functions are also backward compatible with IPv4, so a programmer can use the same translation algorithm in an application for both the IPv4 and IPv6 protocols. This is an important feature, because obviously a global IPv6 infrastructure isn't going to be put in place overnight. Thus, during the transition period from IPv4 to IPv6, client-server applications should be designed with the flexibility to handle both protocols simultaneously. The example programs at the end of this chapter do just that.

The primary lookup function in the new socket API is getaddrinfo(3). Its prototype is as follows.

```
int getaddrinfo( const char             *node,
                 const char             *service,
                 const struct addrinfo  *hints,
                 struct addrinfo       **res );
```

The node parameter is a pointer to the hostname or IP address being translated. The referenced string can be a hostname, IPv4 dotted decimal address, or IPv6 colon-hex address (possibly scoped). The service parameter is a pointer to the transport layer's service name or port number. It can be specified as a name found in /etc/services or a decimal number. getaddrinfo(3) resolves the host/service combination and returns a list of address records; a pointer to the list is placed in the location pointed at by res. For example, suppose a host can be identified by both an IPv4 and IPv6 address, and that the indicated service has both a TCP entry and UDP entry in /etc/services. In such a scenario, it is not inconceivable that four address records are returned; one for TCP/IPv6, one for UDP/IPv6, one for TCP/IPv4, and one for UDP/IPv4.

The definition for struct addrinfo is found in <netdb.h> (as is the declaration for getaddrinfo(3) and the other functions described in this section). The structure has the following format:

```
struct addrinfo
{
   int              ai_flags;
   int              ai_family;
   int              ai_socktype;
   int              ai_protocol;
   socklen_t        ai_addrlen;
   struct sockaddr *ai_addr;
   char            *ai_canonname;
   struct addrinfo *ai_next;
};
```

Consult the 'man' page for getaddrinfo(3) for detailed information about the various fields; this HowTo only describes a subset of them, and only to the extent necessary for normal IPv6 programming.

The ai_family, ai_socktype, and ai_protocol fields have the exact same meaning as the parameters to the socket(2) system call. The ai_family field indicates the protocol family (not the address family) associated with the record, and will be PF_INET6 for IPv6 or PF_INET for IPv4. The ai_socktype parameter indicates

the type of socket to which the record corresponds; SOCK_STREAM for a reliable connection-oriented byte-stream or SOCK_DGRAM for connectionless communication. The ai_protocol field specifies the underlying transport protocol for the record.

The ai_addr field points to a generic struct sockaddr object. Depending on the value in the ai_family field, it will point to either a struct sockaddr_in (PF_INET) or a struct sockaddr_in6 (PF_INET6). The ai_addrlen field contains the size of the object pointed at by the ai_addr field.

As mentioned, getaddrinfo(3) returns a list of address records. The ai_next field points to the next record in the list.

The hints parameter to getaddrinfo(3) is also of type struct addrinfo and acts as a filter for the address records returned in res. If hints is NULL, all matching records are returned; but if hints is non-NULL, the referenced structure gives "hints" to getaddrinfo(3) about which records to return. Only the ai_flags, ai_family, ai_socktype, and ai_protocol fields are significant in the hints structure, and all other fields should be set to zero.

Programs can use hints->ai_family to specify the protocol family. For example, if it is set to PF_INET6, then only IPv6 address records are returned. Likewise, setting hints->ai_family to PF_INET results in only IPv4 address records being returned. If an application wants both IPv4 and IPv6 records, the field should be set to PF_UNSPEC.

The hints->socktype field can be set to SOCK_STREAM to return only records that correspond to connection-oriented byte streams, SOCK_DGRAM to return only records corresponding to connectionless communication, or 0 to return both.

For the Internet protocols, there is only one protocol associated with connection-oriented sockets (TCP) and one protocol associated with connectionless sockets (UDP), so setting hints->ai_socktype to SOCK_STREAM or SOCK_DGRAM is the same as saying, "Give me only TCP records," or "Give me only UDP records," respectively. With that in mind, the hints->ai_protocol field isn't really that important with the Internet protocols, and pretty much mirrors the hints->ai_socktype field. Nevertheless, hints->ai_protocol can be set to IPPROTO_TCP to return only TCP records, IPPROTO_UDP to return only UDP records, or 0 for both.

The node or service parameter to gethostbyname(3) can be NULL, but not both. If node is NULL, then the ai_flags field of the hints parameter specifies how the network address in a returned record is set (i.e. the sin_addr or sin6_addr field of the object pointed at by the ai_addr component in a returned record). If the AI_PASSIVE flag is set in hints, then the returned network addresses are left unresolved (all zeros). This is how server applications would use getaddrinfo(3). If the flag is not set, then the address is set to the local loopback address (::1 for IPv6 or 127.0.0.1 for IPv4). This is one way a client application can specify that the target server is running on the same machine as the client. If the service parameter is NULL, the port number in the returned address records remains unresolved.

The getaddrinfo(3) function returns zero on success, or an error code. In the case of an error, the gai_strerror(3) function is used to obtain a character pointer to an error message corresponding to the error code, just like strerror(3) does in the standard 'C' library.

Once the address list is no longer needed, it must be freed by the application. This is done with the freeaddrinfo(3) function.

The last function that will be mentioned in this section is getnameinfo(3). This function is the inverse of getaddrinfo(3); it is used to create a string representation of the hostname and service from a generic struct sockaddr data object. It has the following prototype.

```
int getnameinfo( const struct sockaddr *sa,
                 socklen_t              salen,
                 char                  *host,
                 size_t                 hostlen,
                 char                  *serv,
                 size_t                 servlen,
                 int                    flags );
```

The sa parameter points to the address structure in question, and salen contains its size. The host parameter points to a buffer where the null-terminated hostname string is placed, and the hostlen parameter is the size of that buffer. If there is no hostname that corresponds to the address, then the network address (dotted decimal or colon-hex) is placed in host. Likewise, the serv parameter points to a buffer where the null-terminated service name string (or port number) is placed, and the servlen parameter is the size of that buffer. The flags parameter modifies the function's behavior; in particular, the NI_NUMERICHOST flag indicates that the converted hostname should always be formatted in numeric form (i.e. dotted decimal or colon-hex), and the NI_NUMERICSERV flag indicates that the converted service should always be in numeric form (i.e. the port number).

The symbols NI_MAXHOST and NI_MAXSERV are available to applications and represent the maximum size of any converted hostname or service name, respectively. Use these when declaring output buffers for getnameinfo(3).

22.1.3. Quirks Encountered

Before jumping into the programming examples, there are several quirks in IPv6 of which the reader should be aware. The more significant ones (in addition to the non-uniqueness of IPv6 network addresses already discussed) are described in the paragraphs below.

22.1.3.1. IPv4 Mapped Addresses

For security reasons that this author won't pretend to understand, "IPv4 mapped addresses" should not be allowed in IPv6-capable server applications. To put it in terms that everyone can understand, this simply means that a server should not accept IPv4 traffic on an IPv6 socket (an otherwise legal operation). An IPv4 mapped address is a mixed-format address of the form:

```
::ffff:192.0.2.1
```

where the first portion is in IPv6 colon-hex format and the last portion is in IPv4 dotted decimal notation. The dotted decimal IPv4 address is the actual network address, but it is being mapped into an IPv6 compatible format.

To prevent IPv4 mapped addresses from being accepted on an IPv6 socket, server applications must explicitly set the IPV6_V6ONLY socket option on all IPv6 sockets created [the Hagino book implies that this is only a concern with server applications. However, it has been observed during testing that if a client application uses an IPv4 mapped address to specify the target server, and the target server has IPv4 mapped addresses disabled, the connection still completes regardless. On the server side, the connection endpoint is an IPv4 socket as desired; but on the client side, the connection endpoint is an IPv6 socket. Setting the IPV6_V6ONLY socket option on the client side as well as the server side prevents any connection from being established at all.]. There's only one problem. Apparently, IPV6_V6ONLY isn't defined on all systems [or at least it wasn't in 2005 when the Hagino book was written]. The server example at the end of this chapter

provides a method for handling this problem.

If IPv4 traffic cannot be handled on IPv6 sockets, then that implies that server applications must open both an IPv4 and IPv6 socket for a particular network service if it wants to handle requests from either protocol. This goes back to the flexibility issue mentioned earlier. If getaddrinfo(3) returns multiple address records, then server applications should traverse the list and open a passive socket for each address provided.

22.1.3.2. Cannot Specify the Scope Identifier in /etc/hosts

It is possible to assign a hostname to an IPv6 network address in /etc/hosts. For example, the following is an excerpt from the /etc/hosts file on the author's development system.

```
::1                     localhost
127.0.0.1               localhost
fe80::2c0:8cff:fe01:2345   pt141
192.0.2.1               pt141
```

The "localhost" and "pt141" hostnames can be translated to either an IPv4 or IPv6 network address. So, for example, if "pt141" is passed as the node parameter to getaddrinfo(3), the function returns both an IPv4 and IPv6 address record for the host (assuming the behavior hasn't been modified by the hints parameter). Unfortunately, a scoped address cannot be used in /etc/hosts. Doing so results in getaddrinfo(3) returning only the IPv4 record.

22.1.3.3. Client & Server Residing on the Same Machine

Suppose a machine has the IPv4 address 192.0.2.1. A client application running on that machine can connect to a server application on the same machine by using either the local loopback address (127.0.0.1) or the network address (192.0.2.1) as the target server. Much to this author's surprise (and dismay), it turns out that an IPv6 client application cannot connect to a server application on the same machine if it uses the network address of that machine as the target; it must use the local loopback address (::1).

22.1.4. Putting It All Together (A Client-Server Programming Example)

Now it's time to put everything discussed thus far together into a sample client-server application. The remainder of this section is devoted to a remote time-of-day application (the 'daytime' Internet service) [I noticed that Ms. Castro used a 'daytime' example in her *Porting applications to IPv6 HowTo*. For the record, the source code presented here is original, developed from scratch, and any similarity between it and any other publicly available 'daytime' example is purely coincidental.]. The source code presented in this section was developed and tested on a RedHat Linux release using the 2.6 kernel (2.6.9 to be specific). Readers may use the source code freely, so long as proper credit is attributed; but of course the standard disclaimer must be given first:

> Although the sample source code is believed to be free of errors, the author makes no guarantees as to its reliability, especially considering that some error paths were intentionally omitted for brevity. Use it at your own risk!

When you get right down to it, there really aren't that many differences between IPv4 and IPv6 applications. The trick is to code IPv6 applications in a protocol-independent manner, such that they can handle both IPv4 and IPv6 simultaneously and transparently. This sample application does just that. The only protocol-dependent code in the example occurs when printing network addresses in verbose mode; but only after the ai_family field in the addrinfo structure has been checked, so the programs know exactly what type

of address they're handling at the time.

22.1.4.1. 'Daytime' Server Code

The server code is found in file tod6d.c (time-of-day IPv6 daemon). Once built, the server may be started using the following command syntax (assuming tod6d is the executable file):

```
tod6d [-v] [service]
```

ARGUMENTS:

service
: The service (or well-known port) on which to listen. Default is "daytime".

OPTIONS:

-v
: Turn on verbose mode.

The server handles both TCP and UDP requests on the network. The server source code contained in tod6d.c follows:

```
/*************************************************************************
* File: tod6d.c
* Description: Contains source code for an IPv6-capable 'daytime' server.
* Author: John Wenker, Sr. Software Engineer,
*         Performance Technologies, San Diego, USA
*************************************************************************/
/*
** System header files.
*/
#include <errno.h>        /* errno declaration & error codes.            */
#include <netdb.h>        /* getaddrinfo(3) et al.                       */
#include <netinet/in.h>   /* sockaddr_in & sockaddr_in6 definition.      */
#include <stdio.h>        /* printf(3) et al.                            */
#include <stdlib.h>       /* exit(2).                                    */
#include <string.h>       /* String manipulation & memory functions.     */
#include <sys/poll.h>     /* poll(2) and related definitions.            */
#include <sys/socket.h>   /* Socket functions (socket(2), bind(2), etc). */
#include <time.h>         /* time(2) & ctime(3).                         */
#include <unistd.h>       /* getopt(3), read(2), etc.                    */
/*
** Constants.
*/
#define DFLT_SERVICE "daytime"   /* Default service name.                  */
#define INVALID_DESC -1          /* Invalid file descriptor.               */
#define MAXCONNQLEN  3           /* Max nbr of connection requests to queue. */
#define MAXTCPSCKTS  2           /* One TCP socket for IPv4 & one for IPv6.  */
#define MAXUDPSCKTS  2           /* One UDP socket for IPv4 & one for IPv6.  */
#define VALIDOPTS    "v"         /* Valid command options.                 */
/*
** Simple boolean type definition.
*/
typedef enum { false = 0, true } boolean;
/*
** Prototypes for internal helper functions.
*/
static int  openSckt( const char *service,
```

```
                     const char *protocol,
                     int         desc[ ],
                     size_t     *descSize );
static void tod( int    tSckt[ ],
                 size_t tScktSize,
                 int    uSckt[ ],
                 size_t uScktSize );
/*
** Global (within this file only) data objects.
*/
static char        hostBfr[ NI_MAXHOST ];   /* For use w/getnameinfo(3).    */
static const char *pgmName;                 /* Program name w/o dir prefix. */
static char        servBfr[ NI_MAXSERV ];   /* For use w/getnameinfo(3).    */
static boolean     verbose = false;         /* Verbose mode indication.     */
/*
** Usage macro for command syntax violations.
*/
#define USAGE                                       \
        {                                           \
           fprintf( stderr,                         \
                    "Usage: %s [-v] [service]\n",   \
                    pgmName );                      \
           exit( 127 );                             \
        }  /* End USAGE macro. */
/*
** Macro to terminate the program if a system call error occurs.  The system
** call must be one of the usual type that returns -1 on error.  This macro is
** a modified version of a macro authored by Dr. V. Vinge, SDSU Dept. of
** Computer Science (retired)... best professor I ever had.  I hear he writes
** great science fiction in addition to robust code, too.
*/
#define CHK(expr)                                                   \
        do                                                          \
        {                                                           \
           if ( (expr) == -1 )                                      \
           {                                                        \
              fprintf( stderr,                                      \
                       "%s (line %d): System call ERROR - %s.\n",   \
                       pgmName,                                     \
                       __LINE__,                                    \
                       strerror( errno ) );                         \
              exit( 1 );                                            \
           }   /* End IF system call failed. */                     \
        } while ( false )
/*****************************************************************************
* Function: main
*
* Description:
*    Set up a time-of-day server and handle network requests.  This server
*    handles both TCP and UDP requests.
*
* Parameters:
*    The usual argc and argv parameters to a main() function.
*
* Return Value:
*    This is a daemon program and never returns.  However, in the degenerate
*    case where no sockets are created, the function returns zero.
*****************************************************************************/
int main( int   argc,
          char *argv[ ] )
{
   int         opt;
```

```
const char *service   = DFLT_SERVICE;
int         tSckt[ MAXTCPSCKTS ];     /* Array of TCP socket descriptors. */
size_t      tScktSize = MAXTCPSCKTS;  /* Size of uSckt (# of elements).   */
int         uSckt[ MAXUDPSCKTS ];     /* Array of UDP socket descriptors. */
size_t      uScktSize = MAXUDPSCKTS;  /* Size of uSckt (# of elements).   */
/*
** Set the program name (w/o directory prefix).
*/
pgmName = strrchr( argv[ 0 ], '/' );
pgmName = pgmName == NULL  ?  argv[ 0 ]  :  pgmName + 1;
/*
** Process command options.
*/
opterr = 0;   /* Turns off "invalid option" error messages. */
while ( ( opt = getopt( argc, argv, VALIDOPTS ) ) >= 0 )
{
   switch ( opt )
   {
      case 'v':   /* Verbose mode. */
      {
         verbose = true;
         break;
      }
      default:
      {
         USAGE;
      }
   }  /* End SWITCH on command option. */
}  /* End WHILE processing options. */
/*
** Process command line arguments.
*/
switch ( argc - optind )
{
   case 0:  break;
   case 1:  service = argv[ optind ]; break;
   default: USAGE;
}  /* End SWITCH on number of command line arguments. */
/*
** Open both a TCP and UDP socket, for both IPv4 & IPv6, on which to receive
** service requests.
*/
if ( ( openSckt( service, "tcp", tSckt, &tScktSize ) < 0 ) ||
     ( openSckt( service, "udp", uSckt, &uScktSize ) < 0 ) )
{
   exit( 1 );
}
/*
** Run the time-of-day server.
*/
if ( ( tScktSize > 0 ) || ( uScktSize > 0 ) )
{
   tod( tSckt,         /* tod() never returns. */
        tScktSize,
        uSckt,
        uScktSize );
}
/*
** Since tod() never returns, execution only gets here if no sockets were
** created.
*/
if ( verbose )
```

```
   {
      fprintf( stderr,
               "%s: No sockets opened... terminating.\n",
               pgmName );
   }
   return 0;
}  /* End main() */
/*****************************************************************************
* Function: openSckt
*
* Description:
*    Open passive (server) sockets for the indicated inet service & protocol.
*    Notice in the last sentence that "sockets" is plural.  During the interim
*    transition period while everyone is switching over to IPv6, the server
*    application has to open two sockets on which to listen for connections...
*    one for IPv4 traffic and one for IPv6 traffic.
*
* Parameters:
*    service  - Pointer to a character string representing the well-known port
*               on which to listen (can be a service name or a decimal number).
*    protocol - Pointer to a character string representing the transport layer
*               protocol (only "tcp" or "udp" are valid).
*    desc     - Pointer to an array into which the socket descriptors are
*               placed when opened.
*    descSize - This is a value-result parameter.  On input, it contains the
*               max number of descriptors that can be put into 'desc' (i.e. the
*               number of elements in the array).  Upon return, it will contain
*               the number of descriptors actually opened.  Any unused slots in
*               'desc' are set to INVALID_DESC.
*
* Return Value:
*    0 on success, -1 on error.
*****************************************************************************/
static int openSckt( const char *service,
                     const char *protocol,
                     int         desc[ ],
                     size_t     *descSize )
{
   struct addrinfo *ai;
   int              aiErr;
   struct addrinfo *aiHead;
   struct addrinfo  hints    = { .ai_flags  = AI_PASSIVE,    /* Server mode.  */
                                 .ai_family = PF_UNSPEC };   /* IPv4 or IPv6. */
   size_t           maxDescs = *descSize;
   /*
   ** Initialize output parameters.  When the loop completes, *descSize is 0.
   */
   while ( *descSize > 0 )
   {
      desc[ --( *descSize ) ] = INVALID_DESC;
   }
   /*
   ** Check which protocol is selected (only TCP and UDP are valid).
   */
   if ( strcmp( protocol, "tcp" ) == 0 )        /* TCP protocol.     */
   {
      hints.ai_socktype = SOCK_STREAM;
      hints.ai_protocol = IPPROTO_TCP;
   }
   else if ( strcmp( protocol, "udp" ) == 0 )   /* UDP protocol.     */
   {
      hints.ai_socktype = SOCK_DGRAM;
```

```
   hints.ai_protocol = IPPROTO_UDP;
}
else                                        /* Invalid protocol. */
{
   fprintf( stderr,
            "%s (line %d): ERROR - Unknown transport "
            "layer protocol \"%s\".\n",
            pgmName,
            __LINE__,
            protocol );
   return -1;
}
/*
** Look up the service's well-known port number.  Notice that NULL is being
** passed for the 'node' parameter, and that the AI_PASSIVE flag is set in
** 'hints'.  Thus, the program is requesting passive address information.
** The network address is initialized to :: (all zeros) for IPv6 records, or
** 0.0.0.0 for IPv4 records.
*/
if ( ( aiErr = getaddrinfo( NULL,
                            service,
                            &hints,
                            &aiHead ) ) != 0 )
{
   fprintf( stderr,
            "%s (line %d): ERROR - %s.\n",
            pgmName,
            __LINE__,
            gai_strerror( aiErr ) );
   return -1;
}
/*
** For each of the address records returned, attempt to set up a passive
** socket.
*/
for ( ai = aiHead;
      ( ai != NULL ) && ( *descSize < maxDescs );
      ai = ai->ai_next )
{
   if ( verbose )
   {
      /*
      ** Display the current address info.   Start with the protocol-
      ** independent fields first.
      */
      fprintf( stderr,
               "Setting up a passive socket based on the "
               "following address info:\n"
               "   ai_flags     = 0x%02X\n"
               "   ai_family    = %d (PF_INET = %d, PF_INET6 = %d)\n"
               "   ai_socktype  = %d (SOCK_STREAM = %d, SOCK_DGRAM = %d)\n"
               "   ai_protocol  = %d (IPPROTO_TCP = %d, IPPROTO_UDP = %d)\n"
               "   ai_addrlen   = %d (sockaddr_in = %d, "
               "sockaddr_in6 = %d)\n",
               ai->ai_flags,
               ai->ai_family,
               PF_INET,
               PF_INET6,
               ai->ai_socktype,
               SOCK_STREAM,
               SOCK_DGRAM,
               ai->ai_protocol,
```

```
            IPPROTO_TCP,
            IPPROTO_UDP,
            ai->ai_addrlen,
            sizeof( struct sockaddr_in ),
            sizeof( struct sockaddr_in6 ) );
      /*
      ** Now display the protocol-specific formatted socket address.  Note
      ** that the program is requesting that getnameinfo(3) convert the
      ** host & service into numeric strings.
      */
      getnameinfo( ai->ai_addr,
                   ai->ai_addrlen,
                   hostBfr,
                   sizeof( hostBfr ),
                   servBfr,
                   sizeof( servBfr ),
                   NI_NUMERICHOST | NI_NUMERICSERV );
      switch ( ai->ai_family )
      {
         case PF_INET:   /* IPv4 address record. */
         {
            struct sockaddr_in *p = (struct sockaddr_in*) ai->ai_addr;
            fprintf( stderr,
                     "   ai_addr      = sin_family:   %d (AF_INET = %d, "
                     "AF_INET6 = %d)\n"
                     "                  sin_addr:     %s\n"
                     "                  sin_port:     %s\n",
                     p->sin_family,
                     AF_INET,
                     AF_INET6,
                     hostBfr,
                     servBfr );
            break;
         }  /* End CASE of IPv4. */
         case PF_INET6:   /* IPv6 address record. */
         {
            struct sockaddr_in6 *p = (struct sockaddr_in6*) ai->ai_addr;
            fprintf( stderr,
                     "   ai_addr      = sin6_family:   %d (AF_INET = %d, "
                     "AF_INET6 = %d)\n"
                     "                  sin6_addr:     %s\n"
                     "                  sin6_port:     %s\n"
                     "                  sin6_flowinfo: %d\n"
                     "                  sin6_scope_id: %d\n",
                     p->sin6_family,
                     AF_INET,
                     AF_INET6,
                     hostBfr,
                     servBfr,
                     p->sin6_flowinfo,
                     p->sin6_scope_id );
            break;
         }  /* End CASE of IPv6. */
         default:   /* Can never get here, but just for completeness. */
         {
            fprintf( stderr,
                     "%s (line %d): ERROR - Unknown protocol family (%d).\n",
                     pgmName,
                     __LINE__,
                     ai->ai_family );
            freeaddrinfo( aiHead );
            return -1;
```

```
            }  /* End DEFAULT case (unknown protocol family). */
         }  /* End SWITCH on protocol family. */
      }  /* End IF verbose mode. */
      /*
      ** Create a socket using the info in the addrinfo structure.
      */
      CHK( desc[ *descSize ] = socket( ai->ai_family,
                                       ai->ai_socktype,
                                       ai->ai_protocol ) );
      /*
      ** Here is the code that prevents "IPv4 mapped addresses", as discussed
      ** in Section 22.1.3.1.  If an IPv6 socket was just created, then set the
      ** IPV6_V6ONLY socket option.
      */
      if ( ai->ai_family == PF_INET6 )
      {
#if defined( IPV6_V6ONLY )
         /*
         ** Disable IPv4 mapped addresses.
         */
         int v6Only = 1;
         CHK( setsockopt( desc[ *descSize ],
                          IPPROTO_IPV6,
                          IPV6_V6ONLY,
                          &v6Only,
                          sizeof( v6Only ) ) );
#else
         /*
         ** IPV6_V6ONLY is not defined, so the socket option can't be set and
         ** thus IPv4 mapped addresses can't be disabled.  Print a warning
         ** message and close the socket.  Design note: If the
         ** #if...#else...#endif construct were removed, then this program
         ** would not compile (because IPV6_V6ONLY isn't defined).  That's an
         ** acceptable approach; IPv4 mapped addresses are certainly disabled
         ** if the program can't build!  However, since this program is also
         ** designed to work for IPv4 sockets as well as IPv6, I decided to
         ** allow the program to compile when IPV6_V6ONLY is not defined, and
         ** turn it into a run-time warning rather than a compile-time error.
         ** IPv4 mapped addresses are still disabled because _all_ IPv6 traffic
         ** is disabled (all IPv6 sockets are closed here), but at least this
         ** way the server can still service IPv4 network traffic.
         */
         fprintf( stderr,
                  "%s (line %d): WARNING - Cannot set IPV6_V6ONLY socket "
                  "option.  Closing IPv6 %s socket.\n",
                  pgmName,
                  __LINE__,
                  ai->ai_protocol == IPPROTO_TCP  ?  "TCP"  :  "UDP" );
         CHK( close( desc[ *descSize ] ) );
         continue;   /* Go to top of FOR loop w/o updating *descSize! */
#endif /* IPV6_V6ONLY */
      }  /* End IF this is an IPv6 socket. */
      /*
      ** Bind the socket.  Again, the info from the addrinfo structure is used.
      */
      CHK( bind( desc[ *descSize ],
                 ai->ai_addr,
                 ai->ai_addrlen ) );
      /*
      ** If this is a TCP socket, put the socket into passive listening mode
      ** (listen is only valid on connection-oriented sockets).
      */
```

```
      if ( ai->ai_socktype == SOCK_STREAM )
      {
         CHK( listen( desc[ *descSize ],
                      MAXCONNQLEN ) );
      }
      /*
      ** Socket set up okay.  Bump index to next descriptor array element.
      */
      *descSize += 1;
   }  /* End FOR each address info structure returned. */
   /*
   ** Dummy check for unused address records.
   */
   if ( verbose && ( ai != NULL ) )
   {
      fprintf( stderr,
               "%s (line %d): WARNING - Some address records were "
               "not processed due to insufficient array space.\n",
               pgmName,
               __LINE__ );
   }  /* End IF verbose and some address records remain unprocessed. */
   /*
   ** Clean up.
   */
   freeaddrinfo( aiHead );
   return 0;
}  /* End openSckt() */
/*****************************************************************************
* Function: tod
*
* Description:
*    Listen on a set of sockets and send the current time-of-day to any
*    clients.  This function never returns.
*
* Parameters:
*    tSckt     - Array of TCP socket descriptors on which to listen.
*    tScktSize - Size of the tSckt array (nbr of elements).
*    uSckt     - Array of UDP socket descriptors on which to listen.
*    uScktSize - Size of the uSckt array (nbr of elements).
*
* Return Value: None.
*****************************************************************************/
static void tod( int    tSckt[ ],
                 size_t tScktSize,
                 int    uSckt[ ],
                 size_t uScktSize )
{
   char                     bfr[ 256 ];
   ssize_t                  count;
   struct pollfd           *desc;
   size_t                   descSize = tScktSize + uScktSize;
   int                      idx;
   int                      newSckt;
   struct sockaddr         *sadr;
   socklen_t                sadrLen;
   struct sockaddr_storage  sockStor;
   int                      status;
   size_t                   timeLen;
   char                    *timeStr;
   time_t                   timeVal;
   ssize_t                  wBytes;
   /*
```

```
** Allocate memory for the poll(2) array.
*/
desc = malloc( descSize * sizeof( struct pollfd ) );
if ( desc == NULL )
{
   fprintf( stderr,
            "%s (line %d): ERROR - %s.\n",
            pgmName,
            __LINE__,
            strerror( ENOMEM ) );
   exit( 1 );
}
/*
** Initialize the poll(2) array.
*/
for ( idx = 0;     idx < descSize;     idx++ )
{
   desc[ idx ].fd      = idx < tScktSize  ?  tSckt[ idx ]
                                          :  uSckt[ idx - tScktSize ];
   desc[ idx ].events  = POLLIN;
   desc[ idx ].revents = 0;
}
/*
** Main time-of-day server loop.  Handles both TCP & UDP requests.  This is
** an interative server, and all requests are handled directly within the
** main loop.
*/
while ( true )   /* Do forever. */
{
   /*
   ** Wait for activity on one of the sockets.  The DO..WHILE construct is
   ** used to restart the system call in the event the process is
   ** interrupted by a signal.
   */
   do
   {
      status = poll( desc,
                     descSize,
                     -1 /* Wait indefinitely for input. */ );
   } while ( ( status < 0 ) && ( errno == EINTR ) );
   CHK( status );   /* Check for a bona fide system call error. */
   /*
   ** Get the current time.
   */
   timeVal = time( NULL );
   timeStr = ctime( &timeVal );
   timeLen = strlen( timeStr );
   /*
   ** Indicate that there is new network activity.
   */
   if ( verbose )
   {
      char *s = malloc( timeLen+1 );
      strcpy( s, timeStr );
      s[ timeLen-1 ] = '\0';   /* Overwrite '\n' in date string. */
      fprintf( stderr,
               "%s: New network activity on %s.\n",
               pgmName,
               s );
      free( s );
   }  /* End IF verbose. */
   /*
```

```
      ** Process sockets with input available.
      */
      for ( idx = 0;     idx < descSize;     idx++ )
      {
         switch ( desc[ idx ].revents )
         {
            case 0:        /* No activity on this socket; try the next. */
               continue;
            case POLLIN:   /* Network activity.  Go process it.         */
               break;
            default:       /* Invalid poll events.                      */
            {
               fprintf( stderr,
                        "%s (line %d): ERROR - Invalid poll event (0x%02X).\n",
                        pgmName,
                        __LINE__,
                        desc[ idx ].revents );
               exit( 1 );
            }
         }  /* End SWITCH on returned poll events. */
         /*
         ** Determine if this is a TCP request or UDP request.
         */
         if ( idx < tScktSize )
         {
            /*
            ** TCP connection requested.  Accept it.  Notice the use of
            ** the sockaddr_storage data type.
            */
            sadrLen = sizeof( sockStor );
            sadr    = (struct sockaddr*) &sockStor;
            CHK( newSckt = accept( desc[ idx ].fd,
                                   sadr,
                                   &sadrLen ) );
            CHK( shutdown( newSckt,       /* Server never recv's anything. */
                           SHUT_RD ) );
            if ( verbose )
            {
               /*
               ** Display the socket address of the remote client.  Begin with
               ** the address-independent fields.
               */
               fprintf( stderr,
                        "Sockaddr info for new TCP client:\n"
                        "   sa_family = %d (AF_INET = %d, AF_INET6 = %d)\n"
                        "   addr len  = %d (sockaddr_in = %d, "
                        "sockaddr_in6 = %d)\n",
                        sadr->sa_family,
                        AF_INET,
                        AF_INET6,
                        sadrLen,
                        sizeof( struct sockaddr_in ),
                        sizeof( struct sockaddr_in6 ) );
               /*
               ** Display the address-specific fields.
               */
               getnameinfo( sadr,
                            sadrLen,
                            hostBfr,
                            sizeof( hostBfr ),
                            servBfr,
                            sizeof( servBfr ),
```

```
                    NI_NUMERICHOST | NI_NUMERICSERV );
      /*
      ** Notice that we're switching on an address family now, not a
      ** protocol family.
      */
      switch ( sadr->sa_family )
      {
         case AF_INET:   /* IPv4 address. */
         {
            struct sockaddr_in *p = (struct sockaddr_in*) sadr;
            fprintf( stderr,
                     "   sin_addr  = sin_family: %d\n"
                     "               sin_addr:   %s\n"
                     "               sin_port:   %s\n",
                     p->sin_family,
                     hostBfr,
                     servBfr );
            break;
         }  /* End CASE of IPv4. */
         case AF_INET6:   /* IPv6 address. */
         {
            struct sockaddr_in6 *p = (struct sockaddr_in6*) sadr;
            fprintf( stderr,
                     "   sin6_addr = sin6_family:   %d\n"
                     "               sin6_addr:     %s\n"
                     "               sin6_port:     %s\n"
                     "               sin6_flowinfo: %d\n"
                     "               sin6_scope_id: %d\n",
                     p->sin6_family,
                     hostBfr,
                     servBfr,
                     p->sin6_flowinfo,
                     p->sin6_scope_id );
            break;
         }  /* End CASE of IPv6. */
         default:   /* Can never get here, but for completeness. */
         {
            fprintf( stderr,
                     "%s (line %d): ERROR - Unknown address "
                     "family (%d).\n",
                     pgmName,
                     __LINE__,
                     sadr->sa_family );
            break;
         }  /* End DEFAULT case (unknown address family). */
      }  /* End SWITCH on address family. */
   }  /* End IF verbose mode. */
   /*
   ** Send the TOD to the client.
   */
   wBytes = timeLen;
   while ( wBytes > 0 )
   {
      do
      {
         count = write( newSckt,
                        timeStr,
                        wBytes );
      } while ( ( count < 0 ) && ( errno == EINTR ) );
      CHK( count );   /* Check for a bona fide error. */
      wBytes -= count;
   }  /* End WHILE there is data to send. */
```

```
            CHK( close( newSckt ) );
         }  /* End IF this was a TCP connection request. */
         else
         {
            /*
            ** This is a UDP socket, and a datagram is available.  The funny
            ** thing about UDP requests is that this server doesn't require any
            ** client input; but it can't send the TOD unless it knows a client
            ** wants the data, and the only way that can occur with UDP is if
            ** the server receives a datagram from the client.  Thus, the
            ** server must receive _something_, but the content of the datagram
            ** is irrelevant.  Read in the datagram.  Again note the use of
            ** sockaddr_storage to receive the address.
            */
            sadrLen = sizeof( sockStor );
            sadr    = (struct sockaddr*) &sockStor;
            CHK( count = recvfrom( desc[ idx ].fd,
                                   bfr,
                                   sizeof( bfr ),
                                   0,
                                   sadr,
                                   &sadrLen ) );
            /*
            ** Display whatever was received on stdout.
            */
            if ( verbose )
            {
               ssize_t rBytes = count;
               fprintf( stderr,
                        "%s: UDP datagram received (%d bytes).\n",
                        pgmName,
                        count );
               while ( count > 0 )
               {
                  fputc( bfr[ rBytes - count-- ],
                         stdout );
               }
               if ( bfr[ rBytes-1 ] != '\n' )
                  fputc( '\n', stdout );   /* Newline also flushes stdout. */
               /*
               ** Display the socket address of the remote client.  Address-
               ** independent fields first.
               */
               fprintf( stderr,
                        "Remote client's sockaddr info:\n"
                        "   sa_family = %d (AF_INET = %d, AF_INET6 = %d)\n"
                        "   addr len  = %d (sockaddr_in = %d, "
                        "sockaddr_in6 = %d)\n",
                        sadr->sa_family,
                        AF_INET,
                        AF_INET6,
                        sadrLen,
                        sizeof( struct sockaddr_in ),
                        sizeof( struct sockaddr_in6 ) );
               /*
               ** Display the address-specific information.
               */
               getnameinfo( sadr,
                            sadrLen,
                            hostBfr,
                            sizeof( hostBfr ),
                            servBfr,
```

```
                    sizeof( servBfr ),
                    NI_NUMERICHOST | NI_NUMERICSERV );
      switch ( sadr->sa_family )
      {
         case AF_INET:   /* IPv4 address. */
         {
            struct sockaddr_in *p = (struct sockaddr_in*) sadr;
            fprintf( stderr,
                     "   sin_addr  = sin_family: %d\n"
                     "               sin_addr:   %s\n"
                     "               sin_port:   %s\n",
                     p->sin_family,
                     hostBfr,
                     servBfr );
            break;
         }  /* End CASE of IPv4 address. */
         case AF_INET6:   /* IPv6 address. */
         {
            struct sockaddr_in6 *p = (struct sockaddr_in6*) sadr;
            fprintf( stderr,
                     "   sin6_addr = sin6_family:   %d\n"
                     "               sin6_addr:     %s\n"
                     "               sin6_port:     %s\n"
                     "               sin6_flowinfo: %d\n"
                     "               sin6_scope_id: %d\n",
                     p->sin6_family,
                     hostBfr,
                     servBfr,
                     p->sin6_flowinfo,
                     p->sin6_scope_id );
            break;
         }  /* End CASE of IPv6 address. */
         default:   /* Can never get here, but for completeness. */
         {
            fprintf( stderr,
                     "%s (line %d): ERROR - Unknown address "
                     "family (%d).\n",
                     pgmName,
                     __LINE__,
                     sadr->sa_family );
            break;
         }  /* End DEFAULT case (unknown address family). */
      }  /* End SWITCH on address family. */
   }  /* End IF verbose mode. */
   /*
   ** Send the time-of-day to the client.
   */
   wBytes = timeLen;
   while ( wBytes > 0 )
   {
      do
      {
         count = sendto( desc[ idx ].fd,
                         timeStr,
                         wBytes,
                         0,
                         sadr,        /* Address & address length   */
                         sadrLen );   /*    received in recvfrom(). */
      } while ( ( count < 0 ) && ( errno == EINTR ) );
      CHK( count );   /* Check for a bona fide error. */
      wBytes -= count;
   }  /* End WHILE there is data to send. */
```

```
        }  /* End ELSE a UDP datagram is available. */
        desc[ idx ].revents = 0;   /* Clear the returned poll events. */
      }  /* End FOR each socket descriptor. */
   }  /* End WHILE forever. */
}  /* End tod() */
```

22.1.4.2. 'Daytime' TCP Client Code

The TCP client code is found in file tod6tc.c (time-of-day IPv6 TCP client). Once built, the TCP client may be started using the following command syntax (assuming tod6tc is the executable file):

```
tod6tc [-v] [-s scope_id] [host [service]]
```

ARGUMENTS:

host
: The hostname or IP address (dotted decimal or colon-hex) of the remote host providing the service. Default is "localhost".

service
: The TCP service (or well-known port number) to which a connection attempt is made. Default is "daytime".

OPTIONS:

-s
: This option is only meaningful for IPv6 addresses, and is used to set the scope identifier (i.e. the network interface on which to establish the connection). Default is "eth0". If host is a scoped address, this option is ignored.

-v
: Turn on verbose mode.

The TCP client source code contained in tod6tc.c follows:

```
/*************************************************************************
* File: tod6tc.c
* Description: Contains source code for an IPv6-capable 'daytime' TCP client.
* Author: John Wenker, Sr. Software Engineer
*         Performance Technologies, San Diego, USA
*************************************************************************/
/*
** System header files.
*/
#include <errno.h>        /* errno declaration and error codes.             */
#include <net/if.h>       /* if_nametoindex(3).                             */
#include <netdb.h>        /* getaddrinfo(3) and associated definitions.     */
#include <netinet/in.h>   /* sockaddr_in and sockaddr_in6 definitions.      */
#include <stdio.h>        /* printf(3) et al.                               */
#include <stdlib.h>       /* exit(2).                                       */
#include <string.h>       /* String manipulation and memory functions.      */
#include <sys/socket.h>   /* Socket functions (socket(2), connect(2), etc). */
#include <unistd.h>       /* getopt(3), read(2), etc.                       */
/*
** Constants & macros.
*/
#define DFLT_HOST      "localhost"   /* Default server name.              */
#define DFLT_SCOPE_ID  "eth0"        /* Default scope identifier.         */
#define DFLT_SERVICE   "daytime"     /* Default service name.             */
```

```
#define INVALID_DESC   -1           /* Invalid file (socket) descriptor. */
#define MAXBFRSIZE     256          /* Max bfr sz to read remote TOD.    */
#define VALIDOPTS      "s:v"        /* Valid command options.            */
/*
** Type definitions (for convenience).
*/
typedef enum { false = 0, true } boolean;
typedef struct sockaddr_in       sockaddr_in_t;
typedef struct sockaddr_in6      sockaddr_in6_t;
/*
** Prototypes for internal helper functions.
*/
static int  openSckt( const char   *host,
                      const char   *service,
                      unsigned int  scopeId );
static void tod( int sckt );
/*
** Global (within this file only) data objects.
*/
static const char *pgmName;           /* Program name (w/o directory). */
static boolean     verbose = false;   /* Verbose mode.                 */
/*
** Usage macro.
*/
#define USAGE                                                           \
        {                                                               \
           fprintf( stderr,                                             \
                    "Usage: %s [-v] [-s scope_id] [host [service]]\n",  \
                    pgmName );                                          \
           exit( 127 );                                                 \
        }  /* End USAGE macro. */
/*
** This "macro" (even though it's really a function) is loosely based on the
** CHK() macro by Dr. V. Vinge (see server code).  The status parameter is
** a boolean expression indicating the return code from one of the usual system
** calls that returns -1 on error.  If a system call error occurred, an alert
** is written to stderr.  It returns a boolean value indicating success/failure
** of the system call.
**
** Example: if ( !SYSCALL( "write",
**                         count = write( fd, bfr, size ) ) )
**          {
**             // Error processing... but SYSCALL() will have already taken
**             // care of dumping an error alert to stderr.
**          }
*/
static __inline boolean SYSCALL( const char *syscallName,
                                 int         lineNbr,
                                 int         status )
{
   if ( ( status == -1 ) && verbose )
   {
      fprintf( stderr,
               "%s (line %d): System call failed ('%s') - %s.\n",
               pgmName,
               lineNbr,
               syscallName,
               strerror( errno ) );
   }
   return status != -1;   /* True if the system call was successful. */
}  /* End SYSCALL() */
/*****************************************************************************
```

```
* Function: main
*
* Description:
*    Connect to a remote time-of-day service and write the remote host's TOD to
*    stdout.
*
* Parameters:
*    The usual argc & argv parameters to a main() program.
*
* Return Value:
*    This function always returns zero.
*******************************************************************************/
int main( int   argc,
          char *argv[ ] )
{
   const char   *host     = DFLT_HOST;
   int           opt;
   int           sckt;
   unsigned int  scopeId  = if_nametoindex( DFLT_SCOPE_ID );
   const char   *service  = DFLT_SERVICE;
   /*
   ** Determine the program name (w/o directory prefix).
   */
   pgmName = (const char*) strrchr( argv[ 0 ], '/' );
   pgmName = pgmName == NULL  ?  argv[ 0 ]  :  pgmName+1;
   /*
   ** Process command line options.
   */
   opterr = 0;   /* Turns off "invalid option" error messages. */
   while ( ( opt = getopt( argc, argv, VALIDOPTS ) ) != -1 )
   {
      switch ( opt )
      {
         case 's':   /* Scope identifier (IPv6 kluge). */
         {
            scopeId = if_nametoindex( optarg );
            if ( scopeId == 0 )
            {
               fprintf( stderr,
                        "%s: Unknown network interface (%s).\n",
                        pgmName,
                        optarg );
               USAGE;
            }
            break;
         }
         case 'v':   /* Verbose mode. */
         {
            verbose = true;
            break;
         }
         default:
         {
            USAGE;
         }
      }  /* End SWITCH on command option. */
   } /* End WHILE processing command options. */
   /*
   ** Process command arguments.  At the end of the above loop, optind is the
   ** index of the first NON-option argv element.
   */
   switch ( argc - optind )
```

```
   {
      case 2:   /* Both host & service are specified on the command line. */
      {
          service = argv[ optind + 1 ];
          /***** Fall through *****/
      }
      case 1:   /* Host is specified on the command line. */
      {
          host = argv[ optind ];
          /***** Fall through *****/
      }
      case 0:   /* Use default host & service. */
      {
          break;
      }
      default:
      {
         USAGE;
      }
   }  /* End SWITCH on number of command arguments. */
   /*
   ** Open a connection to the indicated host/service.
   **
   ** Note that if all three of the following conditions are met, then the
   ** scope identifier remains unresolved at this point.
   **    1) The default network interface is unknown for some reason.
   **    2) The -s option was not used on the command line.
   **    3) An IPv6 "scoped address" was not specified for the hostname on the
   **       command line.
   ** If the above three conditions are met, then only an IPv4 socket can be
   ** opened (connect(2) fails without the scope ID properly set for IPv6
   ** sockets).
   */
   if ( ( sckt = openSckt( host,
                           service,
                           scopeId ) ) == INVALID_DESC )
   {
      fprintf( stderr,
               "%s: Sorry... a connection could not be established.\n",
               pgmName );
      exit( 1 );
   }
   /*
   ** Get the remote time-of-day.
   */
   tod( sckt );
   /*
   ** Close the connection and terminate.
   */
   (void) SYSCALL( "close",
                   __LINE__,
                   close( sckt ) );
   return 0;
}  /* End main() */
/*****************************************************************************
* Function: openSckt
*
* Description:
*    Sets up a TCP connection to a remote server.  Getaddrinfo(3) is used to
*    perform lookup functions and can return multiple address records (i.e. a
*    list of 'struct addrinfo' records).  This function traverses the list and
*    tries to establish a connection to the remote server.  The function ends
```

```
*    when either a connection has been established or all records in the list
*    have been processed.
*
* Parameters:
*    host    - A pointer to a character string representing the hostname or IP
*              address (IPv4 or IPv6) of the remote server.
*    service - A pointer to a character string representing the service name or
*              well-known port number.
*    scopeId - For IPv6 sockets only.  This is the index corresponding to the
*              network interface on which to set up the connection.  This
*              parameter is ignored for IPv4 sockets or when an IPv6 "scoped
*              address" is specified in 'host' (i.e. where the colon-hex
*              network address is augmented with the scope ID).
*
* Return Value:
*    Returns the socket descriptor for the connection, or INVALID_DESC if all
*    address records have been processed and a connection could not be
*    established.
*****************************************************************************/
static int openSckt( const char   *host,
                     const char   *service,
                     unsigned int  scopeId )
{
   struct addrinfo *ai;
   int              aiErr;
   struct addrinfo *aiHead;
   struct addrinfo  hints;
   sockaddr_in6_t  *pSadrIn6;
   int              sckt;
   /*
   ** Initialize the 'hints' structure for getaddrinfo(3).
   **
   ** Notice that the 'ai_family' field is set to PF_UNSPEC, indicating to
   ** return both IPv4 and IPv6 address records for the host/service.  Most of
   ** the time, the user isn't going to care whether an IPv4 connection or an
   ** IPv6 connection is established; the user simply wants to exchange data
   ** with the remote host and doesn't care how it's done.  Sometimes, however,
   ** the user might want to explicitly specify the type of underlying socket.
   ** It is left as an exercise for the motivated reader to add a command line
   ** option allowing the user to specify the IP protocol, and then process the
   ** list of addresses accordingly (it's not that difficult).
   */
   memset( &hints, 0, sizeof( hints ) );
   hints.ai_family   = PF_UNSPEC;     /* IPv4 or IPv6 records (don't care). */
   hints.ai_socktype = SOCK_STREAM;   /* Connection-oriented byte stream.   */
   hints.ai_protocol = IPPROTO_TCP;   /* TCP transport layer protocol only. */
   /*
   ** Look up the host/service information.
   */
   if ( ( aiErr = getaddrinfo( host,
                               service,
                               &hints,
                               &aiHead ) ) != 0 )
   {
      fprintf( stderr,
               "%s (line %d): ERROR - %s.\n",
               pgmName,
               __LINE__,
               gai_strerror( aiErr ) );
      return INVALID_DESC;
   }
   /*
```

```
** Go through the list and try to open a connection.  Continue until either
** a connection is established or the entire list is exhausted.
*/
for ( ai = aiHead,   sckt = INVALID_DESC;
      ( ai != NULL ) && ( sckt == INVALID_DESC );
      ai = ai->ai_next )
{
   /*
   ** IPv6 kluge.  Make sure the scope ID is set.
   */
   if ( ai->ai_family == PF_INET6 )
   {
      pSadrIn6 = (sockaddr_in6_t*) ai->ai_addr;
      if ( pSadrIn6->sin6_scope_id == 0 )
      {
         pSadrIn6->sin6_scope_id = scopeId;
      }  /* End IF the scope ID wasn't set. */
   }  /* End IPv6 kluge. */
   /*
   ** Display the address info for the remote host.
   */
   if ( verbose )
   {
      /*
      ** Temporary character string buffers for host & service.
      */
      char hostBfr[ NI_MAXHOST ];
      char servBfr[ NI_MAXSERV ];
      /*
      ** Display the address information just fetched.  Start with the
      ** common (protocol-independent) stuff first.
      */
      fprintf( stderr,
               "Address info:\n"
               "   ai_flags     = 0x%02X\n"
               "   ai_family    = %d (PF_INET = %d, PF_INET6 = %d)\n"
               "   ai_socktype  = %d (SOCK_STREAM = %d, SOCK_DGRAM = %d)\n"
               "   ai_protocol  = %d (IPPROTO_TCP = %d, IPPROTO_UDP = %d)\n"
               "   ai_addrlen   = %d (sockaddr_in = %d, "
               "sockaddr_in6 = %d)\n",
               ai->ai_flags,
               ai->ai_family,
               PF_INET,
               PF_INET6,
               ai->ai_socktype,
               SOCK_STREAM,
               SOCK_DGRAM,
               ai->ai_protocol,
               IPPROTO_TCP,
               IPPROTO_UDP,
               ai->ai_addrlen,
               sizeof( struct sockaddr_in ),
               sizeof( struct sockaddr_in6 ) );
      /*
      ** Display the protocol-specific formatted address.
      */
      getnameinfo( ai->ai_addr,
                   ai->ai_addrlen,
                   hostBfr,
                   sizeof( hostBfr ),
                   servBfr,
                   sizeof( servBfr ),
```

```
                NI_NUMERICHOST | NI_NUMERICSERV );
      switch ( ai->ai_family )
      {
         case PF_INET:   /* IPv4 address record. */
         {
            sockaddr_in_t *pSadrIn = (sockaddr_in_t*) ai->ai_addr;
            fprintf( stderr,
                     "   ai_addr      = sin_family: %d (AF_INET = %d, "
                     "AF_INET6 = %d)\n"
                     "                  sin_addr:   %s\n"
                     "                  sin_port:   %s\n",
                     pSadrIn->sin_family,
                     AF_INET,
                     AF_INET6,
                     hostBfr,
                     servBfr );
            break;
         }  /* End CASE of IPv4 record. */
         case PF_INET6:   /* IPv6 address record. */
         {
            pSadrIn6 = (sockaddr_in6_t*) ai->ai_addr;
            fprintf( stderr,
                     "   ai_addr      = sin6_family:   %d (AF_INET = %d, "
                     "AF_INET6 = %d)\n"
                     "                  sin6_addr:     %s\n"
                     "                  sin6_port:     %s\n"
                     "                  sin6_flowinfo: %d\n"
                     "                  sin6_scope_id: %d\n",
                     pSadrIn6->sin6_family,
                     AF_INET,
                     AF_INET6,
                     hostBfr,
                     servBfr,
                     pSadrIn6->sin6_flowinfo,
                     pSadrIn6->sin6_scope_id );
            break;
         }  /* End CASE of IPv6 record. */
         default:   /* Can never get here, but just for completeness. */
         {
            fprintf( stderr,
                     "%s (line %d): ERROR - Unknown protocol family (%d).\n",
                     pgmName,
                     __LINE__,
                     ai->ai_family );
            break;
         }  /* End DEFAULT case (unknown protocol family). */
      }  /* End SWITCH on protocol family. */
   }  /* End IF verbose mode. */
   /*
   ** Create a socket.
   */
   if ( !SYSCALL( "socket",
                  __LINE__,
                  sckt = socket( ai->ai_family,
                                 ai->ai_socktype,
                                 ai->ai_protocol ) ) )
   {
      sckt = INVALID_DESC;
      continue;   /* Try the next address record in the list. */
   }
   /*
   ** Connect to the remote host.
```

```
      */
      if ( !SYSCALL( "connect",
                     __LINE__,
                     connect( sckt,
                              ai->ai_addr,
                              ai->ai_addrlen ) ) )
      {
         (void) close( sckt );   /* Could use SYSCALL() again here, but why? */
         sckt = INVALID_DESC;
         continue;   /* Try the next address record in the list. */
      }
   }  /* End FOR each address record returned by getaddrinfo(3). */
   /*
   ** Clean up & return.
   */
   freeaddrinfo( aiHead );
   return sckt;
}  /* End openSckt() */
/*****************************************************************************
* Function: tod
*
* Description:
*    Receive the time-of-day from the remote server and write it to stdout.
*
* Parameters:
*    sckt - The socket descriptor for the connection.
*
* Return Value: None.
*****************************************************************************/
static void tod( int sckt )
{
   char bfr[ MAXBFRSIZE+1 ];
   int  inBytes;
   /*
   ** The client never sends anything, so shut down the write side of the
   ** connection.
   */
   if ( !SYSCALL( "shutdown",
                  __LINE__,
                  shutdown( sckt, SHUT_WR ) ) )
   {
      return;
   }
   /*
   ** Read the time-of-day from the remote host.
   */
   do
   {
      if ( !SYSCALL( "read",
                     __LINE__,
                     inBytes = read( sckt,
                                     bfr,
                                     MAXBFRSIZE ) ) )
      {
         return;
      }
      bfr[ inBytes ] = '\0';   /* Null-terminate the received string. */
      fputs( bfr, stdout );    /* Null string if EOF (inBytes == 0).  */
   } while ( inBytes > 0 );
   fflush( stdout );
}  /* End tod() */
```

22.1.4.3. 'Daytime' UDP Client Code

The UDP client code is found in file tod6uc.c (time-of-day IPv6 UDP client). It is almost an exact duplicate of the TCP client (and in fact was derived from it), but is included in this HowTo for completeness. Once built, the UDP client may be started using the following command syntax (assuming tod6uc is the executable file):

```
tod6uc [-v] [-s scope_id] [host [service]]
```

ARGUMENTS:

host
: The hostname or IP address (dotted decimal or colon-hex) of the remote host providing the service. Default is "localhost".

service
: The UDP service (or well-known port number) to which datagrams are sent. Default is "daytime".

OPTIONS:

-s
: This option is only meaningful for IPv6 addresses, and is used to set the scope identifier (i.e. the network interface on which to exchange datagrams). Default is "eth0". If host is a scoped address, this option is ignored.

-v
: Turn on verbose mode.

The UDP client source code contained in tod6uc.c follows:

```
/*************************************************************************
* File: tod6uc.c
* Description: Contains source code for an IPv6-capable 'daytime' UDP client.
* Author: John Wenker, Sr. Software Engineer
*         Performance Technologies, San Diego, USA
*************************************************************************/
/*
** System header files.
*/
#include <errno.h>        /* errno declaration and error codes.            */
#include <net/if.h>       /* if_nametoindex(3).                            */
#include <netdb.h>        /* getaddrinfo(3) and associated definitions.    */
#include <netinet/in.h>   /* sockaddr_in and sockaddr_in6 definitions.     */
#include <stdio.h>        /* printf(3) et al.                              */
#include <stdlib.h>       /* exit(2).                                      */
#include <string.h>       /* String manipulation and memory functions.     */
#include <sys/socket.h>   /* Socket functions (socket(2), connect(2), etc). */
#include <unistd.h>       /* getopt(3), recvfrom(2), sendto(2), etc.       */
/*
** Constants & macros.
*/
#define DFLT_HOST      "localhost"   /* Default server name.              */
#define DFLT_SCOPE_ID  "eth0"        /* Default scope identifier.         */
#define DFLT_SERVICE   "daytime"     /* Default service name.             */
#define INVALID_DESC   -1            /* Invalid file (socket) descriptor. */
#define MAXBFRSIZE     256           /* Max bfr sz to read remote TOD.    */
#define VALIDOPTS      "s:v"         /* Valid command options.            */
/*
** Type definitions (for convenience).
```

```
*/
typedef enum { false = 0, true } boolean;
typedef struct sockaddr_in       sockaddr_in_t;
typedef struct sockaddr_in6      sockaddr_in6_t;
/*
** Prototypes for internal helper functions.
*/
static int  openSckt( const char   *host,
                      const char   *service,
                      unsigned int  scopeId );
static void tod( int sckt );
/*
** Global (within this file only) data objects.
*/
static const char *pgmName;           /* Program name (w/o directory). */
static boolean     verbose = false;   /* Verbose mode.                 */
/*
** Usage macro.
*/
#define USAGE                                                           \
        {                                                               \
           fprintf( stderr,                                             \
                    "Usage: %s [-v] [-s scope_id] [host [service]]\n",  \
                    pgmName );                                          \
           exit( 127 );                                                 \
        }  /* End USAGE macro. */
/*
** This "macro" (even though it's really a function) is loosely based on the
** CHK() macro by Dr. V. Vinge (see server code).  The status parameter is
** a boolean expression indicating the return code from one of the usual system
** calls that returns -1 on error.  If a system call error occurred, an alert
** is written to stderr.  It returns a boolean value indicating success/failure
** of the system call.
**
** Example: if ( !SYSCALL( "write",
**                         count = write( fd, bfr, size ) ) )
**          {
**             // Error processing... but SYSCALL() will have already taken
**             // care of dumping an error alert to stderr.
**          }
*/
static __inline boolean SYSCALL( const char *syscallName,
                                 int         lineNbr,
                                 int         status )
{
   if ( ( status == -1 ) && verbose )
   {
      fprintf( stderr,
               "%s (line %d): System call failed ('%s') - %s.\n",
               pgmName,
               lineNbr,
               syscallName,
               strerror( errno ) );
   }
   return status != -1;   /* True if the system call was successful. */
}  /* End SYSCALL() */
/*****************************************************************************
* Function: main
*
* Description:
*    Connect to a remote time-of-day service and write the remote host's TOD to
*    stdout.
```

```
*
* Parameters:
*    The usual argc & argv parameters to a main() program.
*
* Return Value:
*    This function always returns zero.
*****************************************************************************/
int main( int   argc,
          char *argv[ ] )
{
   const char   *host     = DFLT_HOST;
   int           opt;
   int           sckt;
   unsigned int  scopeId  = if_nametoindex( DFLT_SCOPE_ID );
   const char   *service  = DFLT_SERVICE;
   /*
   ** Determine the program name (w/o directory prefix).
   */
   pgmName = (const char*) strrchr( argv[ 0 ], '/' );
   pgmName = pgmName == NULL  ?  argv[ 0 ]  :  pgmName+1;
   /*
   ** Process command line options.
   */
   opterr = 0;   /* Turns off "invalid option" error messages. */
   while ( ( opt = getopt( argc, argv, VALIDOPTS ) ) != -1 )
   {
      switch ( opt )
      {
         case 's':   /* Scope identifier (IPv6 kluge). */
         {
            scopeId = if_nametoindex( optarg );
            if ( scopeId == 0 )
            {
               fprintf( stderr,
                        "%s: Unknown network interface (%s).\n",
                        pgmName,
                        optarg );
               USAGE;
            }
            break;
         }
         case 'v':   /* Verbose mode. */
         {
            verbose = true;
            break;
         }
         default:
         {
            USAGE;
         }
      }  /* End SWITCH on command option. */
   } /* End WHILE processing command options. */
   /*
   ** Process command arguments.  At the end of the above loop, optind is the
   ** index of the first NON-option argv element.
   */
   switch ( argc - optind )
   {
      case 2:   /* Both host & service are specified on the command line. */
      {
          service = argv[ optind + 1 ];
          /***** Fall through *****/
```

```
      }
      case 1:   /* Host is specified on the command line. */
      {
          host = argv[ optind ];
          /***** Fall through *****/
      }
      case 0:   /* Use default host & service. */
      {
          break;
      }
      default:
      {
         USAGE;
      }
   }  /* End SWITCH on number of command arguments. */
   /*
   ** Open a connection to the indicated host/service.
   **
   ** Note that if all three of the following conditions are met, then the
   ** scope identifier remains unresolved at this point.
   **    1) The default network interface is unknown for some reason.
   **    2) The -s option was not used on the command line.
   **    3) An IPv6 "scoped address" was not specified for the hostname on the
   **       command line.
   ** If the above three conditions are met, then only an IPv4 socket can be
   ** opened (connect(2) fails without the scope ID properly set for IPv6
   ** sockets).
   */
   if ( ( sckt = openSckt( host,
                           service,
                           scopeId ) ) == INVALID_DESC )
   {
      fprintf( stderr,
               "%s: Sorry... a connectionless socket could "
               "not be set up.\n",
               pgmName );
      exit( 1 );
   }
   /*
   ** Get the remote time-of-day.
   */
   tod( sckt );
   /*
   ** Close the connection and terminate.
   */
   (void) SYSCALL( "close",
                   __LINE__,
                   close( sckt ) );
   return 0;
}  /* End main() */
/*****************************************************************************
* Function: openSckt
*
* Description:
*    Sets up a UDP socket to a remote server.  Getaddrinfo(3) is used to
*    perform lookup functions and can return multiple address records (i.e. a
*    list of 'struct addrinfo' records).  This function traverses the list and
*    tries to establish a connection to the remote server.  The function ends
*    when either a connection has been established or all records in the list
*    have been processed.
*
* Parameters:
```

```
*    host    - A pointer to a character string representing the hostname or IP
*              address (IPv4 or IPv6) of the remote server.
*    service - A pointer to a character string representing the service name or
*              well-known port number.
*    scopeId - For IPv6 sockets only.  This is the index corresponding to the
*              network interface on which to exchange datagrams.  This
*              parameter is ignored for IPv4 sockets or when an IPv6 "scoped
*              address" is specified in 'host' (i.e. where the colon-hex
*              network address is augmented with the scope ID).
*
* Return Value:
*    Returns the socket descriptor for the connection, or INVALID_DESC if all
*    address records have been processed and a socket could not be initialized.
*****************************************************************************/
static int openSckt( const char   *host,
                     const char   *service,
                     unsigned int  scopeId )
{
   struct addrinfo *ai;
   int              aiErr;
   struct addrinfo *aiHead;
   struct addrinfo  hints;
   sockaddr_in6_t  *pSadrIn6;
   int              sckt;
   /*
   ** Initialize the 'hints' structure for getaddrinfo(3).
   **
   ** Notice that the 'ai_family' field is set to PF_UNSPEC, indicating to
   ** return both IPv4 and IPv6 address records for the host/service.  Most of
   ** the time, the user isn't going to care whether an IPv4 connection or an
   ** IPv6 connection is established; the user simply wants to exchange data
   ** with the remote host and doesn't care how it's done.  Sometimes, however,
   ** the user might want to explicitly specify the type of underlying socket.
   ** It is left as an exercise for the motivated reader to add a command line
   ** option allowing the user to specify the IP protocol, and then process the
   ** list of addresses accordingly (it's not that difficult).
   */
   memset( &hints, 0, sizeof( hints ) );
   hints.ai_family   = PF_UNSPEC;     /* IPv4 or IPv6 records (don't care). */
   hints.ai_socktype = SOCK_DGRAM;    /* Connectionless communication.      */
   hints.ai_protocol = IPPROTO_UDP;   /* UDP transport layer protocol only. */
   /*
   ** Look up the host/service information.
   */
   if ( ( aiErr = getaddrinfo( host,
                               service,
                               &hints,
                               &aiHead ) ) != 0 )
   {
      fprintf( stderr,
               "%s (line %d): ERROR - %s.\n",
               pgmName,
               __LINE__,
               gai_strerror( aiErr ) );
      return INVALID_DESC;
   }
   /*
   ** Go through the list and try to open a connection.  Continue until either
   ** a connection is established or the entire list is exhausted.
   */
   for ( ai = aiHead,   sckt = INVALID_DESC;
         ( ai != NULL ) && ( sckt == INVALID_DESC );
```

```
       ai = ai->ai_next )
  {
     /*
     ** IPv6 kluge.  Make sure the scope ID is set.
     */
     if ( ai->ai_family == PF_INET6 )
     {
        pSadrIn6 = (sockaddr_in6_t*) ai->ai_addr;
        if ( pSadrIn6->sin6_scope_id == 0 )
        {
           pSadrIn6->sin6_scope_id = scopeId;
        }  /* End IF the scope ID wasn't set. */
     }  /* End IPv6 kluge. */
     /*
     ** Display the address info for the remote host.
     */
     if ( verbose )
     {
        /*
        ** Temporary character string buffers for host & service.
        */
        char hostBfr[ NI_MAXHOST ];
        char servBfr[ NI_MAXSERV ];
        /*
        ** Display the address information just fetched.  Start with the
        ** common (protocol-independent) stuff first.
        */
        fprintf( stderr,
                 "Address info:\n"
                 "   ai_flags     = 0x%02X\n"
                 "   ai_family    = %d (PF_INET = %d, PF_INET6 = %d)\n"
                 "   ai_socktype  = %d (SOCK_STREAM = %d, SOCK_DGRAM = %d)\n"
                 "   ai_protocol  = %d (IPPROTO_TCP = %d, IPPROTO_UDP = %d)\n"
                 "   ai_addrlen   = %d (sockaddr_in = %d, "
                 "sockaddr_in6 = %d)\n",
                 ai->ai_flags,
                 ai->ai_family,
                 PF_INET,
                 PF_INET6,
                 ai->ai_socktype,
                 SOCK_STREAM,
                 SOCK_DGRAM,
                 ai->ai_protocol,
                 IPPROTO_TCP,
                 IPPROTO_UDP,
                 ai->ai_addrlen,
                 sizeof( struct sockaddr_in ),
                 sizeof( struct sockaddr_in6 ) );
        /*
        ** Display the protocol-specific formatted address.
        */
        getnameinfo( ai->ai_addr,
                     ai->ai_addrlen,
                     hostBfr,
                     sizeof( hostBfr ),
                     servBfr,
                     sizeof( servBfr ),
                     NI_NUMERICHOST | NI_NUMERICSERV );
        switch ( ai->ai_family )
        {
           case PF_INET:   /* IPv4 address record. */
           {
```

```
            sockaddr_in_t *pSadrIn = (sockaddr_in_t*) ai->ai_addr;
            fprintf( stderr,
                     "   ai_addr      = sin_family: %d (AF_INET = %d, "
                     "AF_INET6 = %d)\n"
                     "                  sin_addr:   %s\n"
                     "                  sin_port:   %s\n",
                     pSadrIn->sin_family,
                     AF_INET,
                     AF_INET6,
                     hostBfr,
                     servBfr );
            break;
         }  /* End CASE of IPv4 record. */
         case PF_INET6:   /* IPv6 address record. */
         {
            pSadrIn6 = (sockaddr_in6_t*) ai->ai_addr;
            fprintf( stderr,
                     "   ai_addr      = sin6_family:   %d (AF_INET = %d, "
                     "AF_INET6 = %d)\n"
                     "                  sin6_addr:     %s\n"
                     "                  sin6_port:     %s\n"
                     "                  sin6_flowinfo: %d\n"
                     "                  sin6_scope_id: %d\n",
                     pSadrIn6->sin6_family,
                     AF_INET,
                     AF_INET6,
                     hostBfr,
                     servBfr,
                     pSadrIn6->sin6_flowinfo,
                     pSadrIn6->sin6_scope_id );
            break;
         }  /* End CASE of IPv6 record. */
         default:   /* Can never get here, but just for completeness. */
         {
            fprintf( stderr,
                     "%s (line %d): ERROR - Unknown protocol family (%d).\n",
                     pgmName,
                     __LINE__,
                     ai->ai_family );
            break;
         }  /* End DEFAULT case (unknown protocol family). */
      }  /* End SWITCH on protocol family. */
   }  /* End IF verbose mode. */
   /*
   ** Create a socket.
   */
   if ( !SYSCALL( "socket",
                  __LINE__,
                  sckt = socket( ai->ai_family,
                                 ai->ai_socktype,
                                 ai->ai_protocol ) ) )
   {
      sckt = INVALID_DESC;
      continue;   /* Try the next address record in the list. */
   }
   /*
   ** Set the target destination for the remote host on this socket.  That
   ** is, this socket only communicates with the specified host.
   */
   if ( !SYSCALL( "connect",
                  __LINE__,
                  connect( sckt,
```

```
                              ai->ai_addr,
                              ai->ai_addrlen ) ) )
      {
         (void) close( sckt );   /* Could use SYSCALL() again here, but why? */
         sckt = INVALID_DESC;
         continue;   /* Try the next address record in the list. */
      }
   }  /* End FOR each address record returned by getaddrinfo(3). */
   /*
   ** Clean up & return.
   */
   freeaddrinfo( aiHead );
   return sckt;
}  /* End openSckt() */
/*****************************************************************************
* Function: tod
*
* Description:
*    Receive the time-of-day from the remote server and write it to stdout.
*
* Parameters:
*    sckt - The socket descriptor for the connection.
*
* Return Value: None.
*****************************************************************************/
static void tod( int sckt )
{
   char bfr[ MAXBFRSIZE+1 ];
   int  inBytes;
   /*
   ** Send a datagram to the server to wake it up.  The content isn't
   ** important, but something must be sent to let it know we want the TOD.
   */
   if ( !SYSCALL( "write",
                  __LINE__,
                  write( sckt, "Are you there?", 14 ) ) )
   {
      return;
   }
   /*
   ** Read the time-of-day from the remote host.
   */
   if ( !SYSCALL( "read",
                  __LINE__,
                  inBytes = read( sckt,
                                  bfr,
                                  MAXBFRSIZE ) ) )
   {
      return;
   }
   bfr[ inBytes ] = '\0';   /* Null-terminate the received string. */
   fputs( bfr, stdout );    /* Null string if EOF (inBytes == 0).  */
   fflush( stdout );
}  /* End tod() */
```

22.2. Other programming languages

22.2.1. JAVA

Sun Java versions since 1.4 are IPv6 enabled, see e.g. Inet6Address (1.5/5.0) class. Hints are available in the *Networking IPv6 User Guide for JDK/JRE* 1.4 and 1.5 (5.0).

22.2.2. Perl

As of May 2007 it's not known that the Perl core itself already supports IPv6. It can be added by using following modules:

- Socket6

Anyway, some other modules exist for/with IPv6 support (e.g. Net::IP), search for "IPv6" on http://search.cpan.org/.

Chapter 23. Interoperability

The TAHI Project checks the interoperability of different operating systems regarding the implementation of IPv6 features. Linux kernel already got the IPv6 Ready Logo Phase 1.

Chapter 24. Further information and URLs

24.1. Paper printed books, articles, online reviews (mixed)

24.1.1. Printed Books (English)

24.1.1.1. Cisco

- Cisco Self-Study: Implementing IPv6 Networks (IPV6) by Regis Desmeules. Cisco Press; ISBN 1587050862; 500 pages; 1st edition (April 11, 2003). Note: This item will be published on April 11, 2003.
- Configuring IPv6 with Cisco IOS by Sam Brown, Sam Browne, Neal Chen, Robbie Harrell, Edgar, Jr. Parenti (Editor), Eric Knipp (Editor), Paul Fong (Editor)362 pages; Syngress Media Inc; ISBN 1928994849; (July 12, 2002).

24.1.1.2. General

- IPv6 in Practice: A Unixer's Guide to the Next Generation Internet von Benedikt Stockebrand, November 2006; ISBN 3-540-24524-3
- IPv6 Essentials by Silvia Hagen, 2nd Edition, May 2006; ISBN 0-5961-0058-2 ToC, Index, Sample Chapter etc.; O'Reilly Pressrelease
- IPv6: The New Internet Protocol. By Christian Huitema; Published by Prentice-Hall; ISBN 0138505055. Description: This book, written by Christian Huitema - a member of the InternetArchitecture Board, gives an excellent description of IPv6, how it differs from IPv4, and the hows and whys of it's development. Source: http://www.cs.uu.nl/wais/html/na-dir/internet/tcp-ip/resource-list.html
- IPv6 Networks by Niles, Kitty; (ISBN 0070248079); 550 pages; Date Published 05/01/1998.
- Implementing IPV6. Supporting the Next Generation Internet Protocols by P. E. Miller, Mark A. Miller; Publisher: John Wiley & Sons; ISBN 0764545892; 2nd edition (March 15, 2000); 402 pages.
- Big Book of Ipv6 Addressing Rfcs by Peter H. Salus (Compiler), Morgan Kaufmann Publishers, April 2000, 450 pages ISBN 0126167702.
- Understanding IPV6 by Davies, Joseph; ISBN 0735612455; Date Published 05/01/2001; Number of Pages: 350. Understanding IPV6 by Davies, Joseph; ISBN 0735612455; Date Published 13/11/2002; Number of Pages 544.
- Migrating to IPv6 - IPv6 in Practice by Marc Blanchet Publisher: John Wiley & Sons; ISBN 0471498920; 1st edition (November 2002); 368 pages.
- Ipv6 Network Programming by Jun-ichiro Hagino; ISBN 1555583180
- Wireless boosting IPv6 by Carolyn Duffy Marsan, 10/23/2000.
- O'reilly Network search for keyword IPv6 results in 29 hits (28. January 2002)

24.1.2. Articles, eBooks, Online Reviews (mixed)

- Getting Connected with 6to4 by Huber Feyrer, 06/01/2001
- Transient Addressing for Related Processes: Improved Firewalling by Using IPv6 and Multiple Addresses per Host; written by Peter M. Gleiz, Steven M. Bellovin (PC-PDF-Version; Palm-PDF-Version; PDB-Version)
- Internetworking IPv6 with Cisco Routers by Silvano Gai, McGrawHill Italia, 1997. The 13 chapters and appendix A-D are downloadable as PDF-documents.
- Aufbruch in die neue Welt - IPv6 in IPv4 Netzen by Dipl.Ing. Ralf DÃ¶ring, TU Illmenau, 1999

- Migration and Co-existence of IPv4 and IPv6 in Residential Networks by Pekka Savola, CSC/FUNET, 2002

24.1.3. Science Publications (abstracts, bibliographies, online resources)

See also: liinwww.ira.uka.de/ipv6 or Google / Scholar / IPv6

- GEANT IPv6 Workplan
- IPv6 Trials on UK Academic Networks: Bermuda Project Aug.2002: Participants - Getting connected - Project deliverables - Network topology - Address assignments - Wireless IPv6 access - IPv6 migration - Project presentations - Internet 2 - Other IPv6 projects - IPv6 fora and standards Bermuda 2...
- http://www.ipv6.ac.uk/
- IPv6 at the University of Southampton
- Microsoft Research IPv6 Implementation (MSRIPv6): MSRIPv6 Configuring 6to4 - Connectivity with MSR IPv6 - Our 6Bone Node...

24.1.4. Others

See following URL for more: SWITCH IPv6 Pilot / References

24.2. Conferences, Meetings, Summits

24.2.1. 2002

- IPv6 Deployment Summit at INET 2002

Something missing? Suggestions are welcome!

24.2.2. 2003

Suggestions are welcome!

24.2.3. 2004

- 1st Global IPv6 Summit in Sao Paul, Brazil

24.3. Online information

24.3.1. Join the IPv6 backbone

More to be filled later...suggestions are welcome!

24.3.1.1. Global registries

See regional registries.

24.3.1.2. Major regional registries

- America: ARIN, ARIN / registration page, ARIN / IPv6 guidelines
- EMEA: Ripe NCC, Ripe NCC / registration page, Ripe NCC / IPv6 registration
- Asia/Pacific: APNIC, APNIC / IPv6 ressource guide
- Latin America and Caribbea: LACNIC, IPv6 Registration Services, IPv6 Allocation Policy
- Africa: AfriNIC

Also a list of major (prefix length 32) allocations per local registry is available here: Ripe NCC / IPv6 allocations.

24.3.1.3. Tunnel brokers

Note: A list of available Tunnel broker can be found in the section Tunnel broker below.

- Former IPng. Tunnelbroker and IPv6 resources, now migrated to the SixXs System.
- Eckes' IPv6-with-Linux Page.
- tunnelc - a perl based tunnel client script: freshmeat.net: Project details for tunnel client SourceForge: Project Info - tunnelc (also here)
- Linux Advanced Routing & Traffic Control HOWTO, Chapter 6: IPv6 tunneling with Cisco and/or 6bone.

24.3.1.4. 6to4

- NSayer's 6to4 information
- RFC 3068 / An Anycast Prefix for 6to4 Relay Routers

24.3.1.5. ISATAP

- ISATAP (Intra-Site Automatic Tunnel Access Protocol) Information by JOIN

24.3.2. Latest news and URLs to other documents

- Lot of URLs to others documents by Anil Edathara
- go6 - The IPv6 Portal: an IPv6 online portal with a wiki-based IPv6 knowledge center, an IPv6 discussion forum, an up-to-date collection of IPv6 Events and News, free IPv6 access and services, IPv6 software applications, and much more

---

24.3.3. Protocol references

24.3.3.1. IPv6-related Request For Comments (RFCs)

Publishing the list of IPv6-related RFCs is beyond the scope of this document, but given URLs will lead you to such lists:

- List sorted by IPng Standardization Status or IPng Current Specifications by Robert Hinden
- IPv6 Related Specifications on IPv6.org

24.3.3.2. Current drafts of working groups

Current (also) IPv6-related drafts can be found here:

- IP Version 6 (ipv6)
- Next Generation Transition (ngtrans)
- Dynamic Host Configuration (dhc)
- Domain Name System Extension (dnsext)
- IPv6 Operations (v6ops)
- Mobile IP (mobileip)
- Get any information about IPv6, from overviews, through RFCs & drafts, to implementations (including availability of stacks on various platforms & source code for IPv6 stacks)

24.3.3.3. Others

- SWITCH IPv6 Pilot / References, big list of IPv6 references maintained by Simon Leinen

---

24.3.4. More information

DeepSpace6 / more interesting links

24.3.4.1. Linux related

- DeepSpace6 / (Not only) Linux IPv6 Portal - Italy (Mirror)
- IPv6-HowTo for Linux by Peter Bieringer - Germany, and his Bieringer / IPv6 - software archive
- Linux+IPv6 status by Peter Bieringer - Germany (going obsolete)
- DeepSpace6 / IPv6 Status Page - Italy (Mirror) (will superseed upper one)
- USAGI project - Japan, and their USAGI project - software archive
- Linux Optimized Link State Routing Protocol (OLSR) IPv6 HOWTO

24.3.4.2. Linux related per distribution

PLD
: PLD Linux Distribution ("market leader" in containing IPv6 enabled packages)

Red Hat
: Red Hat Enterprise Linux, Pekka Savola's IPv6 packages

Fedora Core
: Fedora Core Linux

Debian
: Debian Linux, Craig Small's IPv6 information and status

Novell/SuSE
: Novell/SuSE Linux

Mandriva
: Mandriva

For more see the IPv6+Linux Status Distributions page.

24.3.4.3. General

- IPv6.org
- 6bone
- UK IPv6 Resource Centre - UK
- WIDE project - Japan
- SWITCH IPv6 Pilot - Switzerland
- IPv6 Corner of Hubert Feyrer - Germany
- IPv6 Forum - a world-wide consortium of leading Internet vendors, Research & Education Networks...
- Playground.sun.com / IPv6 Info Page - maintained by Robert Hinden, Nokia. Get any information about IPv6, from overviews, through RFCs & drafts, to implementations (including availability of stacks on various platforms & source code for IPv6 stacks).
- 6INIT - IPv6 Internet Initiative - an EU Fifth Framework Project under the IST Programme.
- IPv6 Task Force (European Union)
- 6init - IPv6 INternet IniTiative
- IP Next Generation Overview
- IPv6: The New Version of the Internet Protocol, by Steve Deering.
- IPv6: The Next Generation Internet Protocol, by Gary C. Kessler.
- IPv6: Next Generation Internet Protocol - 3Com
- internet || site and internet2 Working Group - Presentation (HTML + PPT) from IPv6 Workshops: (Stateless Autoconfiguration, IPv6 Addressing, USAGI, Provider Independent IPv6 Addressing and other topics).
- NetworkWorldFusion: Search / Doc Finder: searched for IPv6 (102 documents found 22.12.2002)
- The Register (Search for IPv6 will result in 30 documents, 22.12.2002)
- ZDNet Search for IPv6
- TechTarget Search for IPv6
- IPv6 & TCP Resources List
- Klingon IPv6 tools, Klingon IPv6 tools (native IPv6 only access): IPv6 firewall examples, bandwith testing and portscanner

Something missing? Suggestions are welcome!

24.3.4.4. Market Research

- A Tale of Two Wireless Technology Trends: Processor Development Outsourcing and IPv6Yankee Group - 4/1/2002 - 12 Pages - ID: YANL768881
- The World Atlas of the Internet: Americas; IDATE - 2/1/2002 - 242 PAges - ID: IDT803907. Countries covered: Central America, North America, South America; List: Price: $ 3,500.00; excerpt: Panorama of Internet access markets across the globe. Market assessment and forecasts up to 2006 for 34 countries: market structure: main ISPs and market shares; number of subscribers, of ISPs.
- Early Interest Rising for IPv6 by IDC (Author); List Price: $1,500.00; Edition: e-book (Acrobat Reader); Publisher: IDC; ISBN B000065T8E; (March 1, 2002)

24.3.4.5. Patents

- Canadian Patent Database: Home, Search (Basic Search, just enter IPv6 in the search field ;-); 84 documents found 22.12.2002)
- Espacenet - European patent information: National Offices, Members of Espacenet(IPv6: 84 documents, 22.12.2002)

- Delphion Research: Patent Search Page. Basic (free) registration needed. Examples found 21.12.2002 searching for IPv6: Communicating method between IPv4 terminal and IPv6 terminal and IPv4-IPv6 converting apparatus Translator for IP networks, network system using the translator, and IP network coupling method therefor

24.3.5. By countries

24.3.5.1. Europe

- www.ist-ipv6.org: IST IPv6 Cluster, European IPv6 Research and Development Projects
- Euro6IX: European IPv6 Internet Exchanges Backbone

24.3.5.2. Austria

- IPv6@IKNnet and MIPv6 Research Group: TU Vienna, Austria (IPv6: project, publications, diploma / doctor thesis, Conference Proceedings etc.)

24.3.5.3. Australia

- Carl's Australian IPv6 Pages (old content)

24.3.5.4. Belgium

Suggestions are welcome!

24.3.5.5. Brazil

- IPv6 Summit in Brazil
- IPv6 do Brasil

24.3.5.6. China

Suggestions are welcome!

24.3.5.7. Czech

Suggestions are welcome!

24.3.5.8. Germany

- OpenBC / IPv6

24.3.5.9. France

- Renater: Renater IPv6 Project Page
- IPv6 - RSVP - ATM at INRIA
- NetBSD IPv6 Documentation

24.3.5.10. Italy

- Project6: IPv6 networking with Linux

24.3.5.11. Japan

- Linux IPv6 Users Group JP
- Yamaha IPv6 (sorry, all in japanese native ...)

24.3.5.12. Korea

- ETRI: Electronics and Telecommunications Research Institut
- IPv6 Forum Korea: Korean IPv6 Deployment Project

24.3.5.13. Mexico

- IPv6 Mexico (spain & english version): IPv6 Project Hompeage of The National Autonomous University of Mexico (UNAM)

24.3.5.14. Netherland

- SURFnet: SURFnet IPv6 Backbone
- STACK, STACK (IPv6): Students' computer association of the Eindhoven University of Technology, Netherland
- IPng.nl: collaboration between WiseGuys and Intouch

24.3.5.15. Portugal

- FCCN (National Foundation for the Scientific Computation)

24.3.5.16. Russia

- IPv6 Forum for Russia: Yaroslavl State University Internet Center

---

24.3.5.17. Switzerland

- SWITCH: The Swiss Education & Research Network

24.3.5.18. United Kingdom

- UK IPv6 Resource Center
- British Telecom IPv6 Home: BT's ISP IPv6 Trial, UK's first IPv6 Internet Exchange etc.

24.3.6. By operating systems

24.3.6.1. *BSD

- KAME project (*BSD)
- NetBSD's IPv6 Networking FAQ
- FreeBSD Ports: Ipv6
- BUGAT - BSD Usergroup Austria - www.bugat.at: FreeBSD IPv6 Tunnel (German language)

24.3.6.2. Cisco IOS

- Cisco IOS IPv6 Entry Page
- IPv6 for Cisco IOS Software, File 2 of 3: Aug 2002 -- Table of Contents: IPv6 for Cisco IOS Software; Configuring Documentation Specifics; Enabling IPv6 Routing and Configuring; IPv6 Addressing; Enabling IPv6 Processing Globally.
- Cisco Internet Networking Handbook, Chapter IPv6

24.3.6.3. Compaq

- IPv6 at Compaq - Presentations, White Papers, Documentation...

24.3.6.4. HPUX

- comp.sys.hp.hpux FAQ

24.3.6.5. IBM

- Now that IBM's announced the availability of z/OS V1.4, what's new in this release? This question was posed on 15 August 2002

24.3.6.6. Microsoft

- Microsoft Windows 2000 IPv6
- MSRIPv6 - Microsoft Research Network - IPv6 Homepage
- Getting Started with the Microsoft IPv6 Technology Preview for Windows 2000
- Internet Connection Firewall Does Not Block Internet Protocol Version 6 Traffic (6.11.2001)
- Internet Protocol Numbers (8.10.2002)
- IPv6 Technology Preview Refresh (16.10.2002)
- HOW TO: Install and Configure IP Version 6 in Windows .NET Enterprise Server (26.10.2002)
- Windows .NET Server 6to4 Router Service Quits When You Advertise a 2002 Address on the Public Interface (28.10.2002)
- msdn - Microsoft Windows CE .NET - IPv6 commands
- msdn - search for IPv6 (100 results, 22.12.2002)

24.3.6.7. Solaris

- Sun Microsystems Solaris
- Solaris 2 Frequently Asked Questions (FAQ) 1.73

24.3.6.8. Sumitoma

- Sumitomo Electric has implemented IPv6 on Suminet 3700 family routers

24.3.6.9. ZebOS

- IpInfusion's ZebOS Server Routing Software

24.3.7. IPv6 Security

- Internet Security Systems: Security Center, X-Force Database Search (21.12.2002 - 6 topics found relating to IPv6)
- NIST IPsec Project (National Institute of Standards and Technology, NIST)
- Information Security
- NewOrder.box.sk (search for IPv6) (Articles, exploits, files database etc.)
- THC IPv6 Attack Toolkit

24.3.8. Application lists

- DeepSpace6 / IPv6 Status Page (Mirror)
- IPv6.org / IPv6 enabled applications
- Freshmeat / IPv6 search, currently (14 Dec 2002) 62 projects
- IPv6 Forum / Web Links

24.3.8.1. Analyzer tools

- Wireshark (former known as *Ethereal*) is a free network protocol analyzer for Unix and Windows
- Radcom RC100-WL - Download Radcom RC100-WL protocol analyzer version 3.20

24.3.8.2. IPv6 Products

- 6wind - solutions for IPv4/IPv6 Router, QoS, Multicast, Mobility, Security/VPN/Firewall.
- Fefe's patches for IPv6 with djbdnsAug 2002 -- What is djbdns and why does it need IPv6? djbdns is a full blown DNS server which outperforms BIND in nearly all respects.
- ZebOS Server Routing Suite
- SPA Mail Server 2.21
- Inframail (Advantage Server Edition) 6.0
- HTTrack Website Copier
- CommView 5.0
- Posadis 0.50.6
- TCP Wrapper (IPv6 aware)

24.3.8.3. SNMP

- comp.protocpols.snmp SNMP FAQ Part 1 of 2

24.4. IPv6 Infrastructure

24.4.1. Statistics

- IPv6 routing table history created by Gert DÃ¶ring, Space.Net
- Official 6bone Webserver list Statisic

24.4.2. Internet Exchanges

Another list of IPv6 Internet Exchanges can be found here: IPv6 status of IXPs in Europe

---

24.4.2.1. Estonia

- TIX (tallinn interneti exchange with ipv6 support)

24.4.2.2. Europe

- Euro6IX, European IPv6 Internet Exchange Backbone

24.4.2.3. France

- French National Internet Exchange IPv6 (since 1.11.2002 active). FNIX6 provides a free and reliable high speed FastEthernet interconnection between ISP located in TeleCity Paris.

24.4.2.4. Germany

- INXS: (Cable & Wireless) Munich and Hamburg

24.4.2.5. Japan

- NSPIXP-6: IPv6-based Internet Exchange in Tokyo
- JPIX, Tokyo

24.4.2.6. Korea

- 6NGIX

24.4.2.7. Netherlands

- AMS-IX: Amsterdam Internet Exchange

24.4.2.8. UK

- UK6X: London
- XchangePoint: London

24.4.2.9. USA

- 6TAP: Chicago. Supports peerings around the globe.
- NY6IX: New York City IPv6 based Internet Exchange
- PAIX: Palo Alto

24.4.3. Tunnel broker

See also: http://www.deepspace6.net/docs/tunnelbrokers.html

24.4.3.1. Belgium

Something missing? Suggestions are welcome!

24.4.3.2. Canada

- Freenet6 - /48 Delegation, Canada Getting IPv6 Using Freenet6 on Debian Freenet6 creater

---

24.4.3.3. China

Something missing? Suggestions are welcome!

24.4.3.4. Estonia

- Estpak

24.4.3.5. Germany

- 6bone Knoten Leipzig Info bez. Hackangriff (2001)

24.4.3.6. Italy

- Comv6
- Bersafe (Italian language)

24.4.3.7. Japan

- Internet Initiative Japan (Japanese language) - with IPv6 native line service and IPv6 tunneling Service

---

24.4.3.8. Malaysia

Something missing? Suggestions are welcome!

24.4.3.9. Netherlands

- IPng Netherland - Intouch, SurfNet, AMS-IX, UUNet, Cistron, RIPE NCC and AT&T are connected at the AMS-IX. It is possible (there are requirements...) to get an static tunnel.
- SURFnet Customers

24.4.3.10. Norway

- UNINETT - Pilot IPv6 Service (for Customers): tunnelbroker & address allocation Uninett-Autoupdate-HOWTO

24.4.3.11. Spain

- Consulintel

24.4.3.12. Switzerland

- Tunnelbroker AS8758, Dolphins Network Systems (since 20.12.2002 online)

24.4.3.13. UK

- NTT, United Kingdom - IPv6 Trial. IPv4 Tunnel and native IPv6 leased Line connections. POPs are located in London, UK Dusseldorf, Germany New Jersey, USA (East Coast) Cupertino, USA (West Coast) Tokyo, Japan
- BtexacT IPv6 Tunnel Broker Service
- IPNG-UK

24.4.3.14. USA

- ESnet, USA - Energy Sciences Network: Tunnel Registry & Address Delegation for directly connected ESnet sites and ESnet collaborators.
- Hurricane Electric, US backbone; Hurrican Electric Tunnelbroker (also available under http://tunnelbroker.com/) Press Release: Hurricane Electric Upgrades IPv6 Tunnel Broker Tunnel Broker Endpoint Autoupdate, Perl Script

---

24.4.3.15. Singapore

- http://tunnel-broker.singnet.com.sg/, with NAT and IPsec option

---

24.4.3.16. More Tunnel brokers...

- Public 6to4 relay routers (MS IIE boycott!)

24.4.4. Native IPv6 Services

Note: These services are mostly only available with a valid IPv6 connection!

24.4.4.1. Net News (NNTP)

Something missing? Suggestions are welcome!

24.4.4.2. Game Server

- Quake2 over IPv6

24.4.4.3. IRC Server

Something missing? Suggestions are welcome!

24.4.4.4. Radio Stations, Music Streams

Something missing? Suggestions are welcome!

24.4.4.5. Webserver

- Peter Bieringer's Home of Linux IPv6 HOWTO

Something missing? Suggestions are welcome!

24.5. Maillists

Lists of maillists are available at:

- DeepSpace6 / Mailling Lists

Major Mailinglists are listed in following table:

Focus	Request e-mail address	What to subscribe	Maillist e-mail address	Language	Access through WWW
Linux kernel networking including IPv6	majordomo (at) vger.kernel.org	netdev	netdev (at) vger.kernel.org	English	Info, Archive
Mobile IP(v6) for Linux	Web-based, see URL	mipl	mipl (at) mobile-ipv6.org	English	Info, Archive
Linux IPv6 users using USAGI extension	usagi-users-ctl (at) linux-ipv6.org		usagi-users (at) linux-ipv6.org	English	Info / Search, Archive
IPv6 on Debian Linux			debian-ipv6 (at) lists.debian.org	English	Info/Subscription/Archive
6bone	majordomo (at) isi.edu	6bone	6bone (at) isi.edu	English	Info, Archive
IPv6 users in general	majordomo (at) ipv6.org	users	users (at) ipv6.org	English	Info, Archive
Bugtracking of Internet applications (1)	bugtraq-subscribe (at) securityfocus.com		bugtraq (at) securityfocus.com (2)	English	Info, Archive

(1) very recommended if you provide server applications.

(2) list is moderated.

Something missing? Suggestions are welcome!

Following other maillinglists & newsgroups are available via web:

- student-ipv6 (India) Description: This is the group for the Student Awareness group of IPv6 in India
- sun-ipv6-users Description: Please report problems/suggestions regarding SUN Microsystems IPng implementation
- IPv6-BITS Description: This List will co-ordinate the working of Project Vertebrae.
- linux-bangalore-ipv6 Description: The IPv6 deployment list of the Bangalore Linux User Group
- packet-switching Description: This mailing list provides a forum for discussion of packet switching theory, technology, implementation and application in any relevant aspect including without limitation LAPB, X.25, SDLC, P802.1d, LLC, IP, IPv6, IPX, DECNET, APPLETALK, FR, PPP, IP Telephony, LAN PBX systems, management protocols like SNMP, e-mail, network transparent window systems, protocol implementation, protocol verification, conformance testing and tools used in maintaining or developing packet switching systems.

- de.comm.protocols.tcp-ip Description: Umstellung auf IPv6 Source: Chartas der Newsgruppen in de.*
- Google Group: comp.protocols.tcp-ip
- Google Group: linux.debian.maint.ipv6
- Google Group: microsoft.public.platformsdk.networking.ipv6
- Google Group: fa.openbsd.ipv6

24.6. Online tools

24.6.1. Testing tools

- finger, nslookup, ping, traceroute, whois: UK IPv6 Resource Centre / The test page
- ping, traceroute, tracepath, 6bone registry, DNS: JOIN / Testtools (German language only, but should be no problem for non German speakers)
- traceroute6, whois: IPng.nl
- AAAA Lookup Checker http://www.cnri.dit.ie/cgi-bin/check_aaaa.pl
- IPv6 address analysis tool (something similar to ipv6calc's information option)

24.6.2. Information retrievement

- List of worldwide all IPv6-aggregated IP-Blocks

24.6.3. IPv6 Looking Glasses

- DRENv6 Looking Glass

24.6.4. Helper applications

- IPv6 Prefix Calculator by TDOI
- DNS record checker

24.7. Trainings, Seminars

- IPv6 Training and Workshop, AERAsec, Germany (German language only at this time)
- CIW Internetworking Professional Training CBT CD
- Training Pages, U.K. - Search for IPv6 (13 Courses, 2006-08-21)
- Erion IPv6 Training, UK

Something missing? Suggestions are welcome!

24.8. 'The Online Discovery' ...

IPv6: Addressing The Needs Of the Future by Yankee Group (Author) List Price: $595.00 Edition: e-book (Acrobat Reader) Pages: 3 (three) Publisher: MarketResearch.com; ISBN B00006334Y; (November 1, 2001)

;-) The number of copies would be interesting...

Chapter 25. Revision history / Credits / The End

25.1. Revision history

Versions x.y are published on the Internet.

Versions x.y.z are work-in-progress and published as LyX and SGML file on CVS. Because Deep Space 6 mirrors these SGML files and generate independend from TLDP public versions, this versions will show up there and also on its mirrors.

25.1.1. Releases 0.x

0.60.2
: 2007-10-03/PB: fix description of sysctl/autoconf (credits to Francois-Xavier Le Bail)

0.60.1
: 2007-06-16/PB: speling fixes (credits to Larry W. Burton)

0.60
: 2007-05-29/PB: import major contribution to Programming using C-API written by John Wenker, minor fixes

0.52
: 2007-05-23/PB: update firewalling chapter, improve document for proper SGML validation, minor bugfixes

0.51
: 2006-11-08/PB: remove broken URLs, add a new book (credits to Bryan Vukich)

0.50.2
: 2006-10-25/PB: fix typo in dhcp6 section (credits to Michele Ferritto)

0.50.1
: 2006-09-23/PB: add some URLs

0.50
: 2006-08-24/PB: check RFC URLs, fix URL to Chinese translation, finalize for publishing

0.49.5
: 2006-08-23/PB: fix/remove broken URLs

0.49.4
: 2006-08-21/PB: some review, update and enhancement of the content, replace old 6bone example addresses with the current defined ones.

0.49.3
: 2006-08-20/PB: fix bug in maillist entries, 'mobility' is now a separate chapter

0.49.2
: 2006-08-20/PB: update and cleanup of maillist entries

0.49.1
: 2006-06-13/PB: major update of mobility section (contributed by Benjamin Thery)

0.49
: 2005-10-03/PB: add configuration hints for DHCPv6, major broken URL cleanup (credits to Necdet Yucel)

0.48.1
: 2005-01-15/PB: minor fixes

0.48
: 2005-01-11/PB: grammar check and minor review of IPv6 IPsec section

0.47.1

2005-01-01/PB: add information and examples about IPv6 IPsec, add some URLs

0.47

2004-08-30/PB: add some notes about proftpd, vsftpd and other daemons, add some URLs, minor fixes, update status of Spanish translation

0.46.4

2004-07-19/PB: minor fixes

0.46.3

2004-06-23/PB: add note about started Greek translation, replace Taiwanese with Chinese for related translation

0.46.2

2004-05-22/PB: minor fixes

0.46.1

2004-04-18/PB: minor fixes

0.46

2004-03-04/PB: announce Italian translation, add information about DHCPv6, minor updates

0.45.1

2004-01-12/PB: add note about the official example address space

0.45

2004-01-11/PB: minor fixes, add/fix some URLs, some extensions

0.44.2

2003-10-30/PB: fix some copy&paste text bugs

0.44.1

2003-10-19/PB: add note about start of Italian translation

0.44

2003-08-15/PB: fix URLs, add hint on tcp_wrappers (about broken notation in some versions) and Apache2

0.43.4

2003-07-26/PB: fix URL, add archive URL for maillist users at ipv6.org, add some ds6 URLs

0.43.3

2003-06-19/PB: fix typos

0.43.2

2003-06-11/PB: fix URL

0.43.1

2003-06-07/PB: fix some URLs, fix credits, add some notes at IPsec

0.43

2003-06-05/PB: add some notes about configuration in SuSE Linux, add URL of French translation

0.42

2003-05-09/PB: minor fixes, announce French translation

0.41.4

2003-05-02/PB: Remove a broken URL, update some others.

0.41.3

2003-04-23/PB: Minor fixes, remove a broken URL, fix URL to Taiwanese translation

0.41.2

2003-04-13/PB: Fix some typos, add a note about a French translation is in progress

0.41.1

2003-03-31/PB: Remove a broken URL, fix another

0.41

2003-03-22/PB: Add URL of German translation

0.40.2

2003-02-27/PB: Fix a misaddressed URL

0.40.1

2003-02-12/PB: Add Debian-Linux-Configuration, add a minor note on translations

0.40

2003-02-10/PB: Announcing available German version

0.39.2

2003-02-10/GK: Minor syntax and spelling fixes

0.39.1

2003-01-09/PB: fix an URL (draft adopted to an RFC)

0.39

2003-01-13/PB: fix a bug (forgotten 'link" on "ip link set" (credits to Yaniv Kaul)

0.38.1

2003-01-09/PB: a minor fix

0.38

2003-01-06/PB: minor fixes

0.37.1

2003-01-05/PB: minor updates

0.37

2002-12-31/GK: 270 new links added (searched in 1232 SearchEngines) in existing and 53 new (sub)sections

0.36.1

2002-12-20/PB: Minor fixes

0.36

2002-12-16/PB: Check of and fix broken links (credits to Georg KÃ¤fer), some spelling fixes

0.35

2002-12-11/PB: Some fixes and extensions

0.34.1

2002-11-25/PB: Some fixes (e.g. broken linuxdoc URLs)

0.34

2002-11-19/PB: Add information about German translation (work in progress), some fixes, create a small shortcut explanation list, extend "used terms" and add two German books

0.33

2002-11-18/PB: Fix broken RFC-URLs, add parameter ttl on 6to4 tunnel setup example

0.32

2002-11-03/PB: Add information about Taiwanese translation

0.31.1

2002-10-06/PB: Add another maillist

0.31

2002-09-29/PB: Extend information in proc-filesystem entries

0.30

2002-09-27/PB: Add some maillists

0.29

2002-09-18/PB: Update statement about nmap (triggered by Fyodor)

0.28.1

2002-09-16/PB: Add note about ping6 to multicast addresses, add some labels

0.28

2002-08-17/PB: Fix broken LDP/CVS links, add info about Polish translation, add URL of the IPv6 Address Oracle

0.27

2002-08-10/PB: Some minor updates

0.26.2

2002-07-15/PB: Add information neighbor discovery, split of firewalling (got some updates) and security into extra chapters

0.26.1
: 2002-07-13/PB: Update nmap/IPv6 information

0.26
: 2002-07-13/PB: Fill /proc-filesystem chapter, update DNS information about depricated A6/DNAME, change P-t-P tunnel setup to use of "ip" only

0.25.2
: 2002-07-11/PB: Minor spelling fixes

0.25.1
: 2002-06-23/PB: Minor spelling and other fixes

0.25
: 2002-05-16/PB: Cosmetic fix for 2^128, thanks to JosÃ© AbÃ–lio Oliveira Matos for help with LyX

0.24
: 2002-05-02/PB: Add entries in URL list, minor spelling fixes

0.23
: 2002-03-27/PB: Add entries in URL list and at maillists, add a label and minor information about IPv6 on RHL

0.22
: 2002-03-04/PB: Add info about 6to4 support in kernel series 2.2.x and add an entry in URL list and at maillists

0.21
: 2002-02-26/PB: Migrate next grammar checks submitted by John Ronan

0.20.4
: 2002-02-21/PB: Migrate more grammar checks submitted by John Ronan, add some additional hints at DNS section

0.20.3
: 2002-02-12/PB: Migrate a minor grammar check patch submitted by John Ronan

0.20.2
: 2002-02-05/PB: Add mipl to maillist table

0.20.1
: 2002-01-31/PB: Add a hint how to generate 6to4 addresses

0.20
: 2002-01-30/PB: Add a hint about default route problem, some minor updates

0.19.2
: 2002-01-29/PB: Add many new URLs

0.19.1
: 2002-01-27/PB: Add some forgotten URLs

0.19
: 2002-01-25/PB: Add two German books, fix quote entinities in exported SGML code

0.18.2
: 2002-01-23/PB: Add a FAQ on the program chapter

0.18.1
: 2002-01-23/PB: Move "the end" to the end, add USAGI to maillists

0.18
: 2002-01-22/PB: Fix bugs in explanation of multicast address types

0.17.2
: 2002-01-22/PB: Cosmetic fix double existing text in history (at 0.16), move all credits to the end of the document

0.17.1
: 2002-01-20/PB: Add a reference, fix URL text in online-test-tools

0.17
: 2002-01-19/PB: Add some forgotten information and URLs about global IPv6 addresses

0.16
: 2002-01-19/PB: Minor fixes, remove "bold" and "emphasize" formats on code lines, fix "too long unwrapped code lines" using selfmade utility, extend list of URLs.

0.15
: 2002-01-15/PB: Fix bug in addresstype/anycast, move content related credits to end of document

0.14
: 2002-01-14/PB: Minor review at all, new chapter "debugging", review "addresses", spell checking, grammar checking (from beginning to 3.4.1) by Martin Krafft, add tcpdump examples, copy firewalling/netfilter6 from IPv6+Linux-HowTo, minor enhancements

0.13
: 2002-01-05/PB: Add example BIND9/host, move revision history to end of document, minor extensions

0.12
: 2002-01-03/PB: Merge review of David Ranch

0.11
: 2002-01-02/PB: Spell checking and merge review of Pekka Savola

0.10
: 2002-01-02/PB: First public release of chapter 1

25.2. Credits

The quickest way to be added to this nice list is to send bug fixes, corrections, and/or updates to me ;-).

If you want to do a major review, you can use the native LyX file (see original source) and send diffs against it, because diffs against SGML don't help too much.

25.2.1. Major credits

- David Ranch <dranch at trinnet dot net>: For encouraging me to write this HOWTO, his editorial comments on the first few revisions, and his contributions to various IPv6 testing results on my IPv6 web site. Also for his major reviews and suggestions.
- Pekka Savola <pekkas at netcore dot fi>: For major reviews, input and suggestions.
- Martin F. Krafft <madduck at madduck dot net>: For grammar checks and general reviewing of the document.
- John Ronan <j0n at tssg dot wit dot ie>: For grammar checks.
- Georg KÃ¤fer <gkaefer at gmx dot at>: For detection of no proper PDF creation (fixed now by LDP maintainer Greg Ferguson), input for German books, big list of URLs, checking all URLs, many more suggestions, corrections and contributions, and the German translation
- Michel Boucey <mboucey at free dot fr>: Finding typos and some broken URLs, contribute some suggestions and URLs, and the French translation
- Michele Ferritto <m dot ferritto at virgilio dot it>: Finding bugs and the Italian translation
- Daniel Roesen <dr at cluenet dot de>: For grammar checks
- Benjamin Thery <benjamin dot thery at bull dot net>: For contribution of updated mobility section
- John Wenker <jjw at pt dot com>: major contribution to Programming using C-API

25.2.2. Other credits

25.2.2.1. Document technique related

Writing a LDP HOWTO as a newbie (in LyX and exporting this to DocBook to conform to SGML) isn't as easy as some people say. There are some strange pitfalls... Nevertheless, thanks to:

- Authors of the LDP Author Guide
- B. Guillon: For his DocBook with LyX HOWTO

25.2.2.2. Content related credits

Credits for fixes and hints are listed here, will grow sure in the future

- S .P. Meenakshi <meena at cs dot iitm dot ernet dot in>: For a hint using a "send mail" shell program on tcp_wrapper/hosts.deny
- Frank Dinies <FrankDinies at web dot de>: For a bugfix on IPv6 address explanation
- John Freed <jfreed at linux-mag dot com>: For finding a bug in IPv6 multicast address explanation
- Craig Rodrigues <crodrigu at bbn dot com>: For suggestion about RHL IPv6 setup
- Fyodor <fyodor at insecure dot org>: Note me about outdated nmap information
- Mauro Tortonesi <mauro at deepspace6 dot net>: For some suggestions
- Tom Goodale <goodale at aei-potsdam dot mpg dot de>: For some suggestions
- Martin Luemkemann <mluemkem at techfak dot uni-bielefeld dot de>: For a suggestion
- Jean-Marc V. Liotier <jim at jipo dot com>: Finding a bug
- Yaniv Kaul <ykaul at checkpoint dot com>: Finding a bug
- Arnout Engelen <arnouten at bzzt dot net>: For sending note about a draft was adopted to RFC now
- Stephane Bortzmeyer <bortzmeyer at nic dot fr>: Contributing persistent configuration on Debian
- lithis von saturnsys <lithis at saturnsys dot com>: Reporting a misaddressed URL
- Guy Hulbert <gwhulbert at rogers dot com>: Send a note that RFC1924 is probably an April fool's joke
- Tero Pelander <tpeland at tkukoulu dot fi>: Reporting a broken URL
- Walter Jontofsohn <wjontof at gmx dot de>: Hints for SuSE Linux 8.0/8.1
- Benjamin Hofstetter <benjamin dot hofstetter at netlabs dot org>: Reporting a mispointing URL
- J.P. Larocque <piranha at ely dot ath dot cx>: Reporting archive URL for maillist users at ipv6 dot org
- Jorrit Kronjee <jorrit at wafel dot org>: Reporting broken URLs
- Colm MacCarthaigh <colm dot maccarthaigh at heanet dot ie>: Hint for sendfile issue on Apache2
- Tiago Camilo <tandre at ipg dot pt>: Contribute some URLs about Mobile IPv6
- Harald Geiger: Reporting a bug in how described the bit counting of the universal/global bit
- Bjoern Jacke <bjoern at j3e dot de>: Triggered me to fix some outdated information on xinetd
- Christoph Egger <cegger at chrrr dot com>: Sending note about "ip" has problems with IPv4-compatible addresses on SuSE Linux 9.0 and trigger to add a hint on 6to4-radvd example
- David Lee Haw Ling <hawling at singnet dot com dot sg>: Sending information about a tunnel broker
- Michael H. Warfield <mhw at iss dot net>: Sending note about suffix for 6to4 routers
- Tomasz Mrugalski <thomson at klub dot com dot pl>: Sending updates for DHCPv6 section
- Jan Minar <jjminar at fastmail dot fm>: Reporting minor bugs
- Kalin KOZHUHAROV <kalin at tar dot bz>: Fixing a not so well explanation
- Roel van Dijk <rdvdijk at planet dot nl>: Reporting broken URLs
- Catalin Muresan <catalin dot muresan at astral dot ro>: Reporting minor bugs
- Dennis van Dok <dvandok at quicknet dot nl>: Reporting minor bugs
- Necdet Yucel <nyucel at comu dot edu dot tr>: Reporting broken URLs
- Bryan Vukich: Reporting a broken URL
- Daniele Masini: reporting a broken iptables example

- Yao Zhao: reporting a bug in IPv6 route remove description
- Aaron Kunde: reporting a broken URL and a content related bug
- Larry W. Burton: speling fixes

25.3. The End

Thanks for reading. Hope it helps!

If you have any questions, subscribe to proper maillist and describe your problem providing as much as information as possible.