Gated

A routing daemon for Unix
User Manual

Gated Consortium

July 3, 1996

Contents

Chapter 1

Introduction

GateD is a modular software program consisting of core services, a routing database, and protocol modules supporting multiple routing protocols (RIP versions 1 and 2, DCN HELLO, OSPF version 2, EGP version 2 and BGP version 2 through 4.) GateD was first used to interconnect the NSFNET and the emerging regional networks, and to implement filtered routing based on policy. GateD allows the network administrator to control import and export of routing information by individual protocol, by source and destination autonomous system, source and destination interface, previous hop router, and specific destination address. The network administrator can specify a preference level for each combination of routing information being imported by using a flexible masking capability. Once the preference levels are assigned, GateD makes a decision on which route to use independent of the protocols involved.

GateD is designed to handle dynamic routing with a routing database built from information exchanged by routing protocols.

1.1 Syntax description conventions

File names referred to in this document will be noted with the `typewriter` style. For instance `/etc/gated.conf` is the configuration file used by **gated**. When the full path of the file is not provided the root of the path is from the root directory of the source code. If gated as been unpacked in `/usr/local/gated`, then the file `src/task.c` refers to `/usr/local/gated/src/task.c`. Commands which should be run from the operating system command interpreter will be displayed in `typewriter` style as well. `echo "Hello world"` will print `Hello world` on the screen!

As for any syntax description, keywords and special characters that the parser expects exactly are displayed using **boldface** style. Parameters are specifying in the *italic* style. Parameters shown in square brackets ('`[`' and '`]`') are used to show optional keywords and parameters. The vertical bar ('|') is used to indicate between a choice of optional parameters. Parentheses ('`(`' and '`)`') are used to group keywords and parameters when necessary.

For example, in the syntax description:

autonomoussystem *autonomous_system* [**loop** *number*]

The square brackets say that the **loop** parameter is optional. The keywords are **autonomoussystem** and **loop**. Since *autonomous_system* is in the *variable definition* style, it is a parameter that needs to be provided.

C library calls, as well as system calls will be displayed in *italic* style.

1.2 What to do now?

Gated is a very complex system which often turns out to be a source of frustration for the new users. The present section is an attempt to provide a few guidelines for people not familiar with **gated** and explains how to start with gated. Do not forget that Gated transforms your Unix box into a Internet router, and that you must be familiar with the IP protocol and IP routing protocols in order to design a correct and efficient routing scheme.

The first thing to do in to install **gated** on your system if it is not installed yet. Chapter 3 explains where to get gated, how to build and install it. It also lists advices for some operating system.

Then the configuration of **gated** is the next step. **gated** relies on a configuration file (usually `/etc/gated.conf`) to tell it what to do. Since **gated** runs routing protocols and transforms you Unix system into a router, its configuration parameters can be very complex. Chapter 4 details the syntax and semantic of a configuration file. This chapter is very long due to the numerous protocols and features **gated** can support. You may not need to read the whole chapter but just the introduction, the sections about route preference, tracing, and any section about the protocols you plan to run on your router. The policy section is also very important (section 4.4) and has applications for almost any kind of configuration.

The next stage is to start and control the execution of **gated**. **gated** command line syntax and options, and the tool to control **gated** (start, stop, reconfigure) are explained in chapter 5. This chapter also presents some tools to monitor the RIP and OSPF protocol operations.

If something goes wrong at any time of this process, or if gated is not behaving as you planed, chapter 6 tries to solve the most common problems people can experience when using gated. It won't give you all the answers to your questions but provides good leading tracks to solve your problems. Also note that section 4.29 lists a few most common configuration files, which illustrate the use of the different statements in the configuration file. Some standard router setups are presented there and can be used for a base to your configuration.

People using gated can also join a few mailing lists. The gated consortium official lists are `gated-people@gated.merit.edu` for general question about **gated** and `gated-alpha@gated.merit.edu` for questions about the alpha release of **gated**. The developers and technical people can also be reached on `gated@gated.merit.edu` or `gated-bugs@gated.merit.edu` for bug reports.

Chapter 7 is reserved for the **gated** developers or anyone involved in porting a new protocol into **gated**. It details some of the basic **gated** internal mechanisms.

1.3 Acknowledgments

The following acknowledgments and thanks apply:

Scott Brim <swb@gated.cornell.edu>, for the concept and continued administrative support; National Science Foundation for periods of financial support; Mark Fedor <fedor@psi.com> for the development and maintenance up to release 1.3.1 and his continuing advice; Dennis Ferguson <dennis@canet.ca> for BGP and significant help in redesigning portions of gated; Art Harkin <ash@hpindda.cup.hp.com> for extensive testing and many bug fixes; David Robinson <david@elroy.jpl.nasa.gov> for extensive testing and many bug fixes; Rob Coltun <rcoltun@ni.umd.edu> as the author of the UMd OSPF code and for continued support; Mike Petry <petry@ni.umd.edu> as the author of the original HELLO code; Thomas Pusateri <pusateri@netedge.com> as the author of the DVMRP, IGMP and PIM code; Igor E. Chechik <chech@kaija.spb.su> for his System V modifications; the folks at University of California at Berkeley for the routed code on which gated was originally based and continued support; the folks at IBM Austin for the original implementation of the SMUX interface; the folks at IBM Raleigh for very extensive testing (when they weren't out playing golf); Robert Hagens <hagens@ans.net> and Dan Schuh <schuh@cs.wisc.edu> as the authors of the original Wisconsin IS-IS code; Steve Heimlich <heimlich@watson.ibm.com> for IS-IS mods and current development in gated.

This documentation is a compilation of the old 3.5 HTML documentation, as well as some documentation from ISI (for the Gated Internals), and a lot of reformatting and processing from the Gated Consortium.

Chapter 2

The GateD Consortium

2.1 The GateDaemon Consortium In Brief

The Cornell Gatedaemon Project produces "GateD" (pronounced gate- dee), modular routing software used worldwide to interconnect packet-switched networks. It supports multiple IP and ISO network routing protocols. It is used by researchers, implementors, and industry as:

- a platform for new network routing protocol deployment
- a source of routing protocol reference implementations to test against
- a distribution mechanism of new protocol implementations
- a bootstrapping facility providing the core for the development of commercial products
- a means of access to leading edge network research and development

The GateDaemon Consortium presents a formal structure to support and expand the current successful collaborations already in place to develop GateD functionality. The consortium will:

- improve communication about and interchange of key technologies
- allow functionality needed by the community to be delivered on a more timely basis
- build a broader funding base for a predictable future
- streamline licensing procedures
- improve technology transfer between academic and industry

Membership is open to all organizations interested in supporting and participating in development of internetwork routing protocols. Membership is not a prerequisite for licensing of GateD source code.

Membership benefits include:

- assurance that GateD will continue to be freely available to the Internet community
- active discussions of development features and goal setting
- in-depth technical seminars on new GateD features and capabilities
- internships, where an organization can send an employee to work and learn about routing software with GateD development staff

- developer residencies, whereby an organization with an employee who is making contributions to GateD can receive Consortium funding for that work.

Members will contribute financially according to a sliding scale. There are currently four levels of membership:

- Sponsor: U.S. $50,000
- Associate: U.S. $25,000
- Affiliate: U.S. $15,000
- Academic or Startup: U.S. $5,000

To join the Merit GateD Consortium, the member must sign a Consortium agreement and pay the membership fee. For more information, contact

```
GateD Consortium
Merit Network, Inc.
4251 Plymouth Road
Ann Arbor, Mi 48105-2785

phone: (313) 764-9430
fax:   (313) 747-3185
email: consort-interest@gated.merit.edu
```

This information is also available from the `http://www.merit.edu` Gated Consortium Welcome Page.

2.2 Executive Summary

The Merit GateDaemon Project produces **GateD**, a modular software program which supports multiple routing protocols and protocol families. The project was originally supported by the National Science Foundation starting in 1987. The software was first used to interconnect the original NSFNet and the emerging "regional"networks, and later for connecting sites within the regional networks. GateD is now used by hundreds of organizations worldwide.

Merit's goal is to provide a "state-of-the-art" software platform to support the latest routing protocols for the Internet. The software provides a convenient and effective platform for routing protocol research and development. With the current focus in the Internet community on finding solutions to the scaling problems attending the rapid growth of the Internet, the need for GateD as a research and prototyping platform will continue to increase. It is used as a de facto reference implementation for many routing protocols.

The GateDaemon software can be used to bootstrap and support the production of interoperable networking products. It gives organizations access to leading edge network research and implementations without tremendous startup investments. The software is tested extensively by users throughout the Internet community before the code is released and offered in production. Because of its widespread use both as public domain software and as the heart of many commercial products, it is a powerful deployment mechanism for necessary routing enhancements needed to sustain the current growth of the Internet.

GateD software is available via the web, anonymous ftp, and gopher for individual use. It is licensed for redistribution, free of charge, to anyone, as long as they agree to return any enhancements of general interest to the project for redistribution. This approach has been overwhelmingly successful, especially in encouraging users to contribute changes and new protocol implementations. Contributions of protocol enhancements and implementations are integrated into the public distribution package. Organizations are much more willing to let their employees make their work freely available to the Internet as a whole, when they know that many other organizations will be doing so as well. In fact, the transfer of technology from academia to private industry and vice versa is well demonstrated by this project.

The demands from the user community continue to outpace the resources currently available to the project. Cornell created the GateDaemon Consortium to foster and expand the development of the GateDaemon software. Merit began a new era of fostering this work in September of 1995. Instead of adopting a licensing or royalty fee structure, the Merit Gated Consortium will continue to use membership fees to augment grant and proposal funds and provide the resources to the Internet community for the year 2000. Merit intends to extend the benefits of membership to provide additional benefits such as additional test tools to aid router software development and testing.

The consortium sponsors in-depth technical seminars on new GateD features and capabilities. Members participate in active discussions on new features and delivery goals with the GateD developers. Internships and residencies are available to member organizations who wish to send staff members to Merit to work with the development team. Improved communications about the project and Consortium activities are available both in hardcopy and electronically. In all, the Consortium invigorates and enhances the already successful collaborations of researchers and developers by making it easier for new organizations to participate.

Merit shall continue to seek support from agencies or organizations, such as NSF, whose role it is to support research, the national information infrastructure, and the welfare of the Internet as a whole. Membership in the Consortium is open to any organization in any country with an interest in the Internet or in internetwork routing protocols. Membership, while strongly encouraged, is not a prerequisite for a redistribution license. Corporations who wish to support specific enhancements to the GateD code may submit separate development projects to be considered. These projects will be evaluated on an individual basis.

Your contribution as a member in the Consortium makes it possible to deploy excellent routing protocol implementations available throughout the Internet, an important benefit to operators, developers, and users alike. GateD has proven its worth to the community over the last several years. The issues that must be addressed in the next four years are more complex than ever before. The Internet will continue to experience tremendous growth. Collaboration between industry and research is vital to successfully address the challenges that growth will bring. GateD is at the very heart of the solutions that must be created.

2.3 Licensing GateD

GateD is freely available, but redistribution is restricted. Those who wish to redistribute GateD may do so under agreement with Merit. Anyone wishing to transfer part or all of GateD to any third party must first execute the GateD Redistribution License which corresponds with the version of GateD to be redistributed. The GateD Redistribution License entails no license fee. Instead, licensees are required to submit certain modifications and enhancements of GateD to Merit for distribution to the user community. Distribution of licensee-developed enhancements is for the benefit of the GateD user community and subject only to the GateD source file restrictions and, if applicable, to the appropriate GateD Redistribution License.

Chapter 3

Installation Guide

The GateD makefile system is setup to allow the compilation of gated for different architectures using the same source tree. Each architecture uses it own object directory to store machine- dependent object files. The object directories consist of a few configured files and symbolic links for the majority of the GateD files. They are named `obj.<arch>` where `arch` is related to architecture of your system. As for the 3.6 gated version, a new auto-configuration process is provided. This feature is available on a restricted set of architecture, and you may need to use the old (before version 3.6) gated installation procedures.

3.1 Auto-configuration

This process is supported on the following system: SunOS4.1.x, SunOS5.x, BSD/OS, Ultrix4.x. For other systems you may want to use the old manual procedures described in section 3.2.

1. Change the current directory to the source (`src`) directory:

   `cd src`

2. Start the auto configuration program:

   `make autoconf`

3. Answer all the questions. When done, compile gated:

   `make`

4. Then install gated:

   `make install`

5. And clean up the mess:

   `make clean`

For those interested in the details of the process, section 7.2 explains the different stages of the configuration process. To sum up, `make autoconf` will create the object directory. The name is `obj.<arch>` where `<arch>` is the output of the command `util/configure -a`. In this object directory a `Config` file is created with all the options/parameters needed.

One can modify this `Config` to change some options. After doing that a `make config` is needed to update certain files in the object directory.

`make autoconf` will also look at a file named `Config` in the **source** (`src`) directory. This file must be used to store user preferences that are not asked by the `make autoconf` process. For instance, `make autoconf` will chose the GNU **gcc** compiler over **cc**. If one wants to force the use of **cc** the `src/Config` file should contain this single line:

```
cc        cc
```

The final `src/obj.<arch>/Config` file is a combination of the `src/Config` file and one system-generic file in `src/configs`. The files in `src/configs` should not be edited by the user.

3.2 Manual configuration

This is the usual process for any **Gated** distribution prior to version 3.6. If you want to use this process on the 3.6 or later distributions you first need to replace the generic `Makefile` in the source directory:

```
cd src
cp Makefile.org Makefile
```

and then follow the instructions:

1. Create an object directory, `src/obj.‘src/util/archtype‘`. Run `src/util/archtype` directly to display the architecture. If you will never compile GateD on a second architecture you can use `obj`. For example, on a *Sun SPARCstation* running *SunOS 4.1.1*:

   ```
   sun% src/util/archtype
   SunOS-4.1.1-sun4
   sun% mkdir src/obj.‘src/util/archtype‘
   sun% ls -d src/obj.*
   src/obj.SunOS-4.1.1-sun4/
   ```

   On *BSD 4.3 Reno* and later obj directories are supported in the format the system sources use. This support is not direct, but a hack. On these systems you will need to specify the full path to some directories, for an example see `src/configs/vangogh`.

2. Build a config file. This file is used as input to a configuration script which builds a Makefile and system specific header files.

   A description of config file options is available in `src/configs/README`.

   Examples of quite a few config files are in `src/configs`, pick the one that is closest to what you want and tailor it to your specific configuration. Install this config file in the previously created object directory with the name `Config`.

3. Configure GateD for this architecture by typing '`make config`'. This will run an awk script on your `obj.‘src/util/archtype‘/Config` file which builds a sed script. This sed script is used to edit a Makefile template. The Makefile is then run to configure architecture specific files, create symbolic links, and dependencies. `make config` must be run in the `src` directory.

4. Build GateD by typing `make` in the `src` directory or the object directory.

5. Become root and install GateD: '`make install`' in the src or object directory. Then optionally type `make install-man` in the same directory to install the man pages.

6. Make up a configuration file. Sample config files are in the conf directory and the man page explains all config options. Install the config file in `/etc/gated.conf`.

7. Take cover and .... FIRE IT UP!

3.3 Notes

3.3.1 UDP Checksums

RIP will refuse to run if it determines that UDP checksums are disabled in the kernel. Running without UDP checksums can lead to incorrect routing information being propagated, especially on serial links. This check does not help you determine if RIP packets you receive are missing a checksum, but at least it prevents you from generating these packets and calls attention to the problem.

There are two ways to enable UDP checksums. Your operating system may provide enough source to enable checksums by default. *SunOS* provides this in `/sys/netinet/in_proto.c`. Update the source and recompile the kernel.

If your operating system does not provide the relevant source, you can patch the running kernel and disk image with a sequence similar to this (as root):

```
sun# adb -k -w /vmunix /dev/mem
_udp_cksum/W 1
_udp_cksum:     0x0             =       0x1
_udp_cksum?W 1
_udp_cksum:     0x0             =       0x1
^D
sun#
```

You can probably find the proper name for the UDP checksum value with:

```
sun# nm -o /vmunix | grep udp_c
/vmunix:f80fa35c D _udp_cksum
/vmunix:f801bd10 T _udp_ctlinput
sun#
```

3.3.2 IP Multicast support

The OSPF and RIP implementations (and in the future hopefully Router Discovery) make use of IP multicasting facilities. If these facilities are not present, functionality is reduced.

Some systems ship with IP multicasting support, namely *BSDI's BSD/386 1.0*, *Sun's Solaris 2.0* and *Silicon Graphics' IRIX*. For other systems, IP multicasting support may be available (for example *SunOS 4.1.** and some versions of *Ultrix*), check the FTP directories on `ftp://gregorio.stanford.edu/vmtp-ip`, `gregorio.stanford.edu`, `ftp://parcftp.xerox.com` and `ftp://ftp.isi.edu/mbone"`.

RIP-II and OSPF specify the use of IP multicast on P2P interfaces. Due to bugs in most implementations of the IP multicast code GateD will not be able to specify it's use on these interfaces. GateD will automatically fall back to using the destination address of the P2P link. In the OSPF case, no functionality will be lost, but in the RIP-II case you will loose the ability to pass arbitrary subnet masks via these interfaces.

Another bug in IP multicast support causes multicast packets to local-wire groups to fail if there is not a default route for IP, multicast, or the specific group. As a workaround, GateD installs a route for any local-wire multicast group it uses via the loopback interface. This default is not actually used, but it avoids a kernel bug sending to these groups.

IP Multicast Routing Protocols

Gated now includes support for multicast routing protocols. It implements the router portion of the IGMP protocol to determine local group membership information. It currently supports PIM (Dense Mode) and Distance Vector Multicast Routing Protocol (DVMRP). It tries to be compatible with the 3.3 and 3.5 releases of the IP Multicast distribution from Xerox Parc.

Gated can be configured to use one of three different multicast kernel models. These are distinguished by how multicast forwarding table entries are placed in the kernel.

KRT_IPMULTI_NOCACHE This model does not provide a protocol independent multicast forwarding cache in the kernel. Therefore, it is not possible to use this type of kernel for IP Multicast Routing with Gated. This includes all releases of the IP Multicast software from Xerox PARC and Stanford University with versions 1.x and 2.x. If your kernel falls in this category, then unfortunately, you cannot use Gated for IP Multicast routing.

KRT_IPMULTI_RTSOCK Because of the DVMRP dependencies in the 1.x and 2.x releases, a new kernel was designed by some of the Gated developers that would be completely protocol independent. The design assumed that certain features typical of BSD 4.3 reno and later systems (including the routing socket) would be present. When a multicast datagram arrives for a Group,Source pair and no forwarding cache entry exists in the kernel, a request is put on the routing socket to resolve the Group,Source. The routing daemon (who listens on the routing socket) will see the request and generate a forwarding cache entry to be installed in the kernel.

Another goal of the new kernel was to eliminate the use of Virtual Interfaces (or Vifs). Instead, pseudo tunnel interfaces with their own ifnet structures were implemented. This made the software much simpler since a tunnel interface appeared to be just like and other point to point interface in the router.

The implementation also takes advantage of the radix tree that exists in BSD 4.3 Reno systems and later. The Group,Source pairs are installed as a single key in the radix tree as opposed to the more traditional hashing table found in BSD 4.3 systems. A copy of the source patches to this kernel that is applicable to most BSD 4.4 Lite systems is available from `ftp://ftp.gated.cornell.edu/pub/multicast`

Note: Not yet implemented.

KRT_IPMULTI_TTL0 With this model, the kernel also acts as a protocol independent multicast forwarding cache. Forwarding cache entries are generated on demand as data traffic is forwarded. It uses messages on the IGMP socket to communicate with the routing daemon and request that a forwarding cache entry be calculated. Examples of the use of this model include the Xerox PARC IP Multicast 3.3 and 3.5 kernel releases.

3.3.3 Running GateD on SunOS 4.0 systems

If GateD gets *sendto()* network unreachable problems when running on SunOS 4.0 systems, add `‘hostname‘` to the `ifconfig` commands for `ie0/le0/ec0` in `/etc/rc.local`. Otherwise *SunOS* has a misconception of the route to the attached network.

In an attempt to make binaries that read kernel memory compatible between different kernel architectures, Sun has created libkvm.a. Unfortunately, the dynamically loaded versions of these libraries are broken on *SunOS 4.** systems, so GateD must be statically linked. This prevents the use of a GateD binary compiled on one kernel architecture (say *sun4m*) from working on another (*sun4c*).

3.3.4 Running GateD on AIX 3.2 systems

AIX 3.2 has networking code based on *BSD 4.3 Reno*, including variable length subnet masks and the routing socket. Some of the extensions are available when the system is *not* running in *BSD 4.3* compatibility mode (see the `compat_43` variable and `no`). Among these are the ability to determine the destination address of a RIP packet (used when GateD is responding to the **ripquery** program). GateD can run in either mode with a slight loss of functionality in *BSD 4.3* compatibility mode. Make sure you compile with `-D_BSD=44`!

In order to generate a core dump useful for debugging on *AIX 3*, the default limit on the core size must be increased. This can be accomplished via the shell, or automatically when GateD is started via **gdc**. Some compilation-time configuring is necessary for this to work. Either define `GDC_CORESIZE=RLIM_INFINITY` in the `obj/Config` file, or define `GDC_RESOURCE` and use the `-c` option to `gdc` at run time.

3.3.5 Compiling GateD on systems with shared libraries

If an assertion failure occurs in `task_stdio_read()`, it is because a file descriptor was improperly closed. This can occur when the *named resolver* libraries are improperly installed in the system shared libraries. If the socket used by the shared libraries is not statically initialized to -1, file descriptor zero will be closed when GateD calls `endhostent()`. The solution is to fix the shared libraries. A workaround would be to not use any symbolic names in the config file *and* specify `options noresolv ;`.

3.3.6 Using GateD on AIX 3.1 and 3.2

Problems have been reported with `yacc` on at least some versions of *AIX 3.1* and *3.2*. On of the problems is that `yacc` does not report parse errors to the caller, resulting in GateD trying to run with an incorrect configuration. It is strongly recommended that you obtain *GNU* `bison` instead. It is available from `ftp://prep.ai.mit.edu/pub/gnu`.

3.3.7 Running HELLO and/or EGP on 4.2 based systems

If you would like to run HELLO or EGP on a 4.2 based system such as *Ultrix 1.2, 2.0, 2.1* and *SunOS 3.x* you will need to add the following code to the following modules and rebuild your kernel.

```
/sys/netinet/in.h:

#define IPPROTO_EGP 8 /* exterior gateway protocol */

#define IPPROTO_HELLO 63 /* Fuzzball HELLO protocol */

/sys/netinet/in_proto.c for {\em SunOS 3.x}:

{ SOCK_RAW,PF_INET,IPPROTO_HELLO,PR_ATOMIC|PR_ADDR,
  rip_input,rip_output,0,0,
  raw_usrreq,
  0,0,0,0,
},
{ SOCK_RAW,PF_INET,IPPROTO_EGP,PR_ATOMIC|PR_ADDR,
  rip_input,rip_output,0,0,
  raw_usrreq,
  0,0,0,0,
},

/sys/netinet/in_proto.c for {\em Ultrix 1.2, 2.0 and 2.2}:

{ SOCK_RAW,     &inetdomain,    IPPROTO_HELLO,    PR_ATOMIC|PR_ADDR,
  rip_input,    rip_output,     0,               0,
  raw_usrreq,
  0,            0,              0,               0,
  0,            0,              0,
},
{ SOCK_RAW,     &inetdomain,    IPPROTO_EGP,    PR_ATOMIC|PR_ADDR,
  rip_input,    rip_output,     0,               0,
```

```
  raw_usrreq,
  0,            0,              0,              0,
  0,            0,              0,
},
```

3.4 Interfacing to the ISODE SMUX interface

This version of GateD can do SNMP via the *ISODE 7.0* SMUX interface. If you don't already have *ISODE 7.0* you may obtain a version with many SMUX/SNMP patches applied via `ftp://gated.cornell.edu/pub/gated/isode-snmp-7.0.tar.Z` from `gated.cornell.edu` as `pub/gated/isode-snmp-7.0.tar.Z`.

Even though the *ISODE* source requires a lot of disk space, it isn't necessary to build and install all of *ISODE*, just the core distribution and the snmp code. A partial install (`inst-partial` and `inst-snmp`) are the minimum required.

GateD also supports the *ISODE 6.8* based SMUX interface on *AIX 3* systems. If you experience errors of the form:

```
Building:       gated-mib.c
mosy -s -c gated-mib.c ../mib/smi.my ../mib/mib.my ../mib/rt.my ../mib/bgp.my ..
/mib/ospf.my ../mib/rip.my
mosy 6.8 of Sat Nov 23 12:30:27 CST 1991
line 56: Warning: EXPORTS but no ModuleIdentifier
last token read was ";"
object enterprises: Warning: IMPORTS but no ModuleIdentifier
object snmpEnableAuthenTraps: Warning: IMPORTS but no ModuleIdentifier
object snmpEnableAuthenTraps: syntax error
make: *** [gated-mib.c] Error 1
```

change the default for *ipForwardInfo* to remove the second zero so it becomes:

```
DEFVAL  { { 0 } } -- 0.0
```

The mosy supplied with *ISODE 6.8* could not parse this syntax, even though it is correct.

Chapter 4

Configuration Guide

4.1 Introduction to the Gated Configuration File

4.1.1 Syntax

The gated configuration file consists of a sequence of statements terminated by a semi-colon '(`;`'). Statements are composed of tokens separated by white space, which can be any combination of blanks, tabs and newlines. This structure simplifies identification of the parts of the configuration associated with each other and with specific protocols. Comments may be specified in either of two forms. One form begins with a pound sign ('`#`') and runs to the end of the line. The other form, *C* style, starts with a '`/*`' and continues until it reaches '`*/`'.

4.1.2 Statement Grouping

The configuration statements and the order in which these statements appear divide `gated.conf` into *options statements*, *interface statements*, *definition statements*, *protocol statements*, *static statements*, *control statements*, and *aggregate statements*. Entering a statement out of order causes an error when parsing the configuration file.

Two other types of statements do not fit in these categories: *directive statements* and *trace statements*. These statements provide instructions to the parser and control tracing from the configuration file. They do not relate to the configuration of any protocol and may occur anywhere in the `gated.conf` file.

4.2 Route Preference

The *preference* is the value GateD uses to order preference of routes from one protocol or peer over another. Preference can be set in the GateD configuration files in several different configuration statements.

Preference can be set based on network interface over another, from one protocol over another, or from one remote gateway over another. Preference may not be used to control the selection of routes within an **igp**, this is accomplished automatically by the protocol based on metric. Preference may be used to select routes from the same **egp** learned from different peers or autonomous systems.

Each route has only one preference value associated with it, even though preference can be set at many places in the configuration file. Simply, the last or most specific preference value set for a route is the value used. The preference value is an arbitrarily assigned value used to determine the order of routes to the same destination in a single routing database. The active route is chosen by the lowest preference value.

Some protocols implement a second preference (*preference2*), sometimes referred to as a tie-breaker.

4.2.1 Selecting a route

Gated uses the following rules to compute the best route (or next hop) to a destination:

- The route with the best (numerically smallest) preference is preferred.
- If the two routes have the same preference, the route with the best (numerically smallest) preference2 (also known as a tie-breaker) is preferred.
- A route learned from a **igp** is preferred to a route learned from an **egp**. Least preferred is a route learned indirectly by an **igp** from an **egp**.
- If AS path information is available it is used to help determine the most preferred route.
  - A route with an AS path is preferred over one without and AS path
  - If the AS paths and origins are identical, the route with the lower metric is preferred.
  - A route with an AS path origin of **igp** is preferred over a route with an AS path origin of **egp**. Least preferred is an AS path with an **unknown** origin.
  - A route with a shorter AS path is preferred.
- If both routes are from the same protocol and AS, the one with the lowest metric is preferred.
- The route with the lowest numeric next-hop address is used.

4.2.2 Assigning preferences

A default preference is assigned to each source from which GateD receives routes. Preference values range from 0 to 255 with the lowest number indicating the most preferred route.

Table 4.1 summarizes the default preference values for routes learned in various ways. The table lists the statements (some of these are clauses within statements) that set preference, and shows the types of routes to which each statement applies. The default preference for each type of route is listed, and the table notes preference precedence between protocols. The narrower the scope of the statement, the higher precedence its preference value is given, but the smaller the set of routes it affects.

Preference Of	Defined by Statement	Default
direct connected networks	interface	0
OSPF routes	ospf	10
IS-IS level 1 routes	isis level 1	15
IS-IS level 2 routes	isis level 2	18
internally generated default	gendefault	20
redirects	redirect	30
routes learned via route socket	kernel	40
static routes from config	static	60
ANS SPF (SLSP) routes	slsp	70
HELLO routes	hello	90
RIP routes	rip	100
point-to-point interface		110
routes to interfaces that are down	interfaces	120
aggregate/generate routes	aggregate/generate	130
OSPF AS external routes	ospf	150
BGP routes	bgp	170
EGP	egp	200

Table 4.1: Default preferences for the Gated protocols

4.2.3 Sample Preference Specifications

```
interfaces {
    interface 138.66.12.2 preference 10 ;
} ;

rip yes {
    preference 90 ;
} ;

import proto rip gateway 138.66.12.1 preference 75 ;
```

In these statements the preference applicable to routes learned via RIP from gateway 138.66.12.1 is 75. The last preference applicable to routes learned via RIP from gateway 128.66.12.1 is defined in the **import** statement. The preference applicable to other RIP routes is found in the rip statement. The preference set on the interface statement applies only to the route to that interface.

4.3 Protocol Overview

4.3.1 Protocol Overview

Routing protocols determine the "best" route to each destination and distribute routing information among the systems on a network. Routing protocols are divided into two general groups: *interior* and *exterior* protocols. GateD software combines management of the interior and exterior routing protocols in one software daemon.

4.3.2 Interior Routing Protocols

Interior protocols are used to exchange reachability information within an *autonomous system* (AS). They are referred to as a class by the acronym **igp**. There are several interior protocols:

RIP

The Routing Information Protocol, Version 1 and Version 2, is the most commonly used interior protocol. RIP selects the route with the lowest metric as the best route. The metric is a hop count representing the number of gateways through which data must pass to reach its destination. The longest path that RIP accepts is 15 hops. If the metric is greater than 15, a destination is considered unreachable and GateD discards the route. RIP assumes the best route is the one that uses the fewest gateways i.e., the shortest path, not taking into account congestion or delay on route.

The RIP version 1 protocol is described in [RFC 1058]. and the RIP version 2 protocol is described in [RFC 1388].

HELLO

HELLO , another interior protocol, uses delay as the deciding factor in choosing the best route. Round-trip time is the length of time it takes a datagram to travel from the source and destination. HELLO is historically significant for the Internet as it was the protocol used among the original prototype NSFNET backbone *fuzzball* gateways. Today, like fuzzballs, HELLO is a little-used protocol.

An earlier version of the HELLO protocol is described in [RFC 891].

OSPF

Open Shortest Path First is a link-state protocol. OSPF is better suited than RIP for complex networks with many routers. OSPF provides equal cost multipath routing.

OSPF is described in [RFC 1583], the MIB is defined in [RFC 1253]. Other related documents are [RFC 1245], [RFC 1246] and [RFC 1370].

IS-IS

Intermediate System to Intermediate System (ISIS) is a link state interior gateway protocol (IGP) originally developed for routing ISO/CLNP (International Organization for Standardization/Connectionless Network Protocol) packets. The version distributed with GateD can route IP packets as well.

4.3.3 Exterior Routing Protocols

Exterior protocols are used to exchange routing information between autonomous systems. Exterior protocols are only required when an autonomous system must exchange routing information with another autonomous system. Routers within an autonomous system run an interior routing protocol like RIP. Only those gateways that connect an autonomous system to another autonomous system need to run an exterior routing protocol. There are two exterior protocols currently supported by GateD:

EGP

Exterior Gateway Protocol: Originally EGP reachability information was passed into ARPANET/MILNET "core" gateways where the best routes were chosen and passed back out to all connected autonomous systems. As the Internet moved toward a less hierarchical architecture, EGP, an exterior routing protocol which assumes a hierarchical structure, became less effective.

The EGP protocol is described in [RFC 827] and [RFC 904].

BGP

Border Gateway Protocol is replacing EGP as the exterior protocol of choice. BGP exchanges reachability information between autonomous systems, but provides more capabilities than EGP. BGP uses path attributes to provide more information about each route as an aid in selecting the best route. Path attributes may include, for example, administrative preferences based on political, organizational, or security (policy) considerations in the routing decision. BGP supports non-hierarchical topologies and can be used to implement a network structure of equivalent autonomous systems.

BGP version 1 is described in [RFC 1105], version 2 in [RFC 1163], version 3 in [RFC 1267] and version 4 in [RFC 1771]. The version 3 MIB is described in [RFC 1269]. The three documents, [RFC 1164], [RFC 1268] and [RFC 1772], describe the application of version 2, 3 and 4 in the internet. A protocol analysis of and experience with BGP version 3 are available in [RFC 1265] and [RFC 1266], [RFC 1397] talks about advertising a default route in BGP version 2 and 3. And finally, [RFC 1403] describes BGP - OSPF interaction.

4.3.4 Other Routing Protocols

Router Discovery

The Router Discovery protocol is used to inform hosts of the availability of hosts it can send packets to and is used to supplement a statically configured default router. This is the preferred protocol for hosts to run, they are discouraged from *wiretapping* routing protocols. Router Discovery is described in [RFC 1256].

4.4 Routing Policy

The *Routing Policy* represents a set of rules which defines the relation between the router and the external world in term of route exchanges and protocol interaction. To summarize, the choice of a routing policy allows the user to define:

- The list of routes the router will accept from its peers;
- The list of routes the router will propagate to its peers;
- The redistribution of routes between the different protocols that **gated** supports.

The *Routing Policy* is an very important part of the gated configuration file, and should not be neglected especially in case of the use of several different protocols on the same router (**IGP**, **EGP**, static routes, direct routes, etc...). *Routing Policy* is often referred as *Route Filtering* since defining a policy is equivalent to defining a set of filters to which to apply a route before accepting it or redistributing it.

4.4.1 Imports

The *Imports* policy defines the set of routes which is accepted. Criteria for accepting a route includes parameters like interface from which the route is received, address of the peer sending the route, autonomous system of the peer, etc... Section 4.26 explains the detail of the **import** statement for defining the incoming route filter.

4.4.2 Exports

The *Exports* policy defines the set of routes which is exported to the external world by any protocol. The criteria are the same as for the *Imports* policy.

The export policy also defines the interaction between the different protocols ran on the router. By default a given protocol will redistribute only the routes originated from that protocol. For instance RIP will announce only route learned from RIP, and BGP will announce only routes learned from the other BGP peers. The export policy will allow the protocols to exchange routes between each other, making external routes available for redistribution by internal protocols, and vice versa. The export policy is also very useful for advertising static routes (like a default route) through any protocol.

Section 4.27 explains the details of the **export** statement.

4.4.3 Route Filtering

All the routing policy statements make use of *Route Filter* to explicitly define the set of routes accepted or announced. A *Route Filter* is merely a list of routes, in term of address (prefix) and mask, to which a given route will be matched. *Route Filter* are also used on the **martians** statements.

4.5 Trace Statements

4.5.1 Trace Statements

Trace statements control tracing options. GateD's tracing options may be configured at many levels. Tracing options include the file specifications, control options, and global and protocol specific tracing options. Unless overridden, tracing options from the next higher level are inherited by lower levels. For example, BGP peer tracing options are inherited from BGP group tracing options, which are inherited from global BGP tracing options, which are inherited from global GateD tracing options. At each level tracing specifications override the inherited options.

4.5.2 Traceoptions syntax

traceoptions ["*trace_file*" [**replace**] [**size** *size* [**k**|**m**] **files** *files*]]
[*control_options*] *trace_options* [**except** *trace_options*] ;
traceoptions none;

trace_file

Specifies the file to receive tracing information. If this file name does not begin with a slash (/) the directory where gated was started in prepended to the name.

replace

Tracing should start by replacing an existing file. The default is to append to an existing file.

size *size* [**k**|**m**] **files** *files*

Limit the maximum size of the trace file to the specified size (minimum 10k). When the trace file reaches the specified size, it is renamed to `file.0`, then `file.1`, `file.2` up to the maximum number of files (minimum specification is 2).

control_options

Specifies options that control the appearance of tracing. Valid values are:

nostamp Specifies that a timestamp should not be prepended to all trace lines.

trace_options

are described in the following sections.

except *trace_options*

Used to enable a broad class of tracing and then disable more specific options.

none

Specifies that all tracing should be turned off for this protocol or peer.

4.5.3 Global tracing options

There are two types of global options (or *trace_options* in the previous syntax description), those which only affect global operations and those which have potential significance to protocols.

Global significance only

The trace flags that only have global significance are:

parse Trace the lexical analyzer and parser. Mostly used by GateD developers for debugging.

adv Trace the allocation of and freeing of policy blocks. Mostly used by the GateD developers for debugging.

symbols Used to trace symbols read from the kernel at startup. The only useful way to specify this level of tracing is via the `-t` option on the command line since the symbols are read from the kernel before parsing the configuration file.

iflist Used to trace the reading of the kernel interface list. It is useful to specify this with the **-t** option on the command line since the first interface scan is done before reading the configuration file.

Protocol Significance

The options flags that have potential significance to protocols are:

all Turn on all of the following.

general A shorthand notation for specifying both `normal` and `route`

state Trace state machine transitions in the protocols.

normal Trace normal protocols occurrences. Abnormal protocol occurrences are always traced.

policy Trace application of protocol and user-specified policy to routes being imported and exported.

task Trace system interface and processing associated with this protocol or peer.

timer Trace timer usage by this protocol or peer.

route Trace routing table changes for routes installed by this protocol or peer.

NOTE!

Not all of the above options apply to all of the protocols. In some cases their use does not make sense (for instance, RIP does not have a state machine) and in some instances the requested tracing has not been implemented (such as RIP support of the `policy` option).

NOTE!

It is not currently possible to specify packet tracing from the command line. This is because a global option for packet tracing would potentially create too much output.

When protocols inherit their tracing options from the global tracing options, tracing levels that don't make sense (such as **parse**, **adv** and packet tracing options) are masked out.

Global tracing statements have an immediate effect, especially parsing options that effect the parsing of the configuration file. Tracing values inherited by protocols specified in the configuration file are initially inherited from the global options in effect as they are parsed, unless they are overridden by more specific options. After the configuration file is read, tracing options that were not explicitly specified are inherited from the global options in effect at the end of the configuration file.

4.5.4 Packet tracing

Tracing of packets is very flexible. For any given protocol there are one or more options for tracing packets. All protocols allow use of the **packets** keyword allows for tracing *all* packets sent and received by the protocol. most protocols have other options for limiting tracing to a useful subset of packet types. These tracing options can be further controlled with the following modifiers:

detail must be specified before **send** or **recv**. Normally packets are traced in a terse form of one or two lines. When **detail** is specified, a more verbose format is used to provide further detail on the contents of the packet.

send, recv These options limit the tracing to packets sent or received. Without these options both sent and received packets will be traced.

NOTE!

Detail, if specified, must be before **send** or **recv**. If a protocol allows for several different types of packet tracing, modifiers may be applied to each individual type. But be aware that within one tracing specification the trace flags are summed up, so specifying **detail packets** will turn on full tracing for all packets.

4.6 Directive Statements

Directive statements provide direction to the GateD configuration language parser about included files and the directories in which these files reside. Directive statements are immediately acted upon by the parser. Other statements terminate with a semi-colon (;), but directive statements terminate with a newline. The two directive statements are:

%directory *directory*

Defines the directory where the include files are stored. When it is used, GateD looks in the directory identified by pathname for any included files that do not have a fully qualified filename i.e. do not begin with "/". This statement does not actually change the current the directory, it just specifies the prefix applied to included file names.

%include *filename*

Identifies an include file. The contents of the file is included in the gated.conf file at the point in the gated.conf file where the **%include** directive is encountered. If the filename is not fully qualified, i.e. does not begin with "/", it is considered to be relative to the directory defined in the **%directory** directive. The **%include** directive statement causes the specified file to be parsed completely before resuming with this file. Nesting up to ten levels is supported. The maximum nesting level may be increased by changing the definition of **FI_MAX** in `parse.h`.

In a complex environment, segmenting a large configuration into smaller more easily understood segments might be helpful, but one of the great advantages of GateD is that it combines the configuration of several different routing protocols into a single file. Segmenting a small file unnecessarily complicates routing configurations.

4.7 Options Statements

Options statements allow specification of some global options. If used, **options** must appear before any other type of configuration statement in the gated.conf file.

The options statement syntax is:

options [**nosend**]
[**noresolv**]
[**gendefault** [**preference** *preference*] [**gateway** *gateway*]]
[**syslog** [**upto**] *log_level*]
[**mark** *time*]
;

The options list can contain one or more of the following options:

gendefault [**preference** *preference*] [**gateway** *gateway*]

When **gendefault** is enabled, when a BGP or EGP neighbor is up it causes the creation of a default route with the special protocol **default**. This can be disabled per BGP/EGP group with the **nogendefault** option. By default, this route has a preference of 20. This route is normally not installed in the kernel forwarding table, it is only present so it can be announced to other protocols. If a gateway is specified, the default route will be installed in the kernel forwarding table with a next hop of the listed gateway.

Note that the use of the more general option is preferred to the use of this **gendefault** option. The **gendefault** option may go away in future releases. See section 4.28 for more information on the **generate** statement.

nosend

Do not send any packets. This option makes it possible to run GateD on a live network to test protocol interactions without actually participating in the routing protocols. The packet traces in the GateD log can be examined to verify that GateD is functioning properly. This is most useful for RIP and HELLO and possibly the SMUX SNMP interface. This option does not yet apply to BGP and is less than useful with EGP and OSPF.

noresolv

By default GateD will try to resolve symbolic names into IP addresses by using the *gethostbyname()* and *getnetbyname()* library calls. These calls usually use the Domain Name System (DNS) instead of the hosts local host and network tables. If there is insufficient routing information to send DNS queries, GateD will deadlock during startup. This option can be used to prevent these calls, and then using symbolic names will result in configuration file errors.

syslog [**upto**] *log_level*

Controls the amount of data GateD logs via syslog on systems where *setlogmask()* is supported. The available logging levels and other terminology are as defined in the `setlogmask(3)` man page. The default is equivalent to **syslog upto info**.

mark *time*

Specifying this option causes gated to output a message to the trace log at the specified interval. This can be used as one method of determining if gated is still running.

4.8 Interfaces Statement

4.8.1 Interface Syntax

interfaces {

```
        options
                [ strictinterfaces ]
                [ scaninterval time ]
                ;
        interface interface_list
                [ preference preference ]
                [ down preference preference ]
                [ passive ]
                [ simplex ]
                [ reject ]
                [ blackhole ]
                ;
        define address
                [ broadcast address ] | [ pointtopoint address ]
                [ netmask mask ]
                [ multicast ]
                ;
} ;
```

An interface is the connection between a router and one of its attached networks. A physical interface may be specified by interface name, by IP address, or by domain name, (unless the network is an unnumbered point-to-point network.) Multiple levels of reference in the configuration language allow identification of interfaces using wildcard, interface type name, or delete word address. Be careful with the use of interface names as some Unix operating systems may allow more than one address per interface. The interface_list is a list of one or more interface names including wildcard names (names without a number) and names which may specify more than one interface or address, or the token **all** for all interfaces.

options

Allows configuration of some global options related to interfaces. These are:

strictinterfaces Indicates that it is a fatal error to reference an interface in the configuration file that is not present when GateD is started and not listed in a *define* statement. Without this option a warning message will be issued but GateD will continue.

scaninterval *time* Specifies how often GateD scans the kernel interface list for changes. The default is every 15 seconds on most systems, and 60 seconds on systems that pass interface status changes through the routing socket (e.g. BSD 4.4). Note that GateD will also scan the interface list on receipt of a *SIGUSR2*.

interface *interface list*

Sets interface options on the specified interfaces. An interface list is **all** or a list of interface names (see warning about interface names), domain names, or numeric addresses. Options available on this statement are:

preference *preference* Sets the preference for routes to this interface when it is up and appears to be functioning properly. The default preference is *0*.

down preference *preference* Sets the preference for routes to this interface when GateD does not believe it to be functioning properly, but the kernel does not indicate it is down. The default value is *120*.

passive Prevents GateD from changing the preference of the route to this interface if it is not believed to be functioning properly due to lack of received routing information. GateD will only perform this check if the interface is actively participating in a routing protocol.

simplex Defines an interface as unable to hear its own broadcast packets. Some systems define an interface as simplex with the IFF_SIMPLEX flag, on others it needs to be specified in the configuration file. On simplex interfaces, packets from myself are assumed to have been looped back in software and are not used as an indication that the interface is functioning properly.

reject Specifies that the address of the interface which matches these criteria will be used as the local address when installing *reject* routes in the kernel. Should only be used with systems based on BSD 4.3 Tahoe or earlier which have installed a reject/blackhole pseudo interface.

blackhole Specifies that the address of the interface which matches these criteria will be used as the local address when installing *reject* routes in the kernel. Should only be used with systems based on BSD 4.3 Tahoe or earlier which have installed a reject/blackhole pseudo interface.

define *address*

Defines interfaces that might not be present when GateD is started so they may be referenced in the configuration file when **strictinterfaces** is defined. Possible **define** keywords are:

broadcast *address* Defines the interface as broadcast capable (e.g. Ethernet or Token Ring) and specifies the broadcast address.

pointopoint *address* Defines the interface as a pointopoint interface (e.g. SLIP or PPP) and specifies the address on the local side. The first *address* on the *define* statement references the address of the host on the **remote** end of the interface, the *address* specified after this *pointopoint* keyword defines the address on the **local** side of the interface.

An interface not defined as broadcast or pointopoint is assumed to be non-broadcast multiaccess (NBMA), such as an X.25 network.

netmask *mask* Specifies the subnet mask to be used on this interface. This is ignored on pointtopoint interfaces.

multicast Specifies that the interface is multicast capable.

4.8.2 Interface lists

An interface list is a list of references to interfaces or groups of interfaces. There are four methods available for referring to interfaces. They are listed here from most general to most specific.

all

This refers to all available interfaces.

Interface name wildcard

This refers to all the interfaces of the same type. Unix interfaces consist of the name of the device driver, like *ie*, and a unit number, like *0*, *5* or *22*. Reference to the name contain only alphabetic characters and match any interfaces that have the same alphabetic part.

For example, *ie* on a Sun would refer to all Interlan Ethernet interfaces, *le* would refer to all Lance Ethernet interfaces. But *ie* would not match *iel0*.

Interface name

This refers to a specific interface, usually one physical interface. These are specified as an alphabetic part followed by a numeric part. This will match one specific interface. But be aware that on many systems, there can be more than one protocol (i.e. *IP*) address on a given physical interface.

For example, *ef1* will match an interface named *ef1*, but not an interface named *ef10*.

Interface address

This matches one specific interface. The reference can be by protocol address (i.e. *10.0.0.51*), or by symbolic host name (i.e. *nic.ddn.mil*). Note that a symbolic host name reference is only valid when it resolves to only one address. Use of symbolic host names is not recommended.

If many interface lists are present in the config file with more than one parameter, these parameters are collected at run-time to create the specific parameter list for a given interface. If the same parameter is specified on more than one list, the parameters with the most specific interface is used.

For example, consider a system with three interfaces, *le0*, *le1* and *du0*.

```
rip yes {
    interface all noripin noripout ;
    interface le ripin ;
    interface le1 ripout ;
} ;
```

RIP packets would only be accepted from interfaces *le0* and *le1*, but not from *du0*. RIP packets would only be sent on interface *le1*.

4.8.3 IP Interface addresses and routes

The *BSD 4.3* and later networking implementations allow four types of interfaces. Some implementations allow multiple protocol addresses per physical interface, these are mostly based on *BSD 4.3 Reno* or later.

loopback This interface must have the address of **127.0.0.1**. Packets sent to this interface are sent back to the originator. This interface is also used as a catch all interface for implementing other features, such as *reject* and *blackhole* routes. Although a netmask is reported on this interface, it is ignored. It is useful to assign an additional address to this interface that is the same as the OSPF or BGP *router id*; this allows routing to a system based on the *router id* which will work if some interfaces are down.

broadcast This is a multi-access interface capable of a physical level broadcast, such as *Ethernet*, *Token Ring* and *FDDI*. This interface has an associated subnet mask and broadcast address. The interface route to an *broadcast* network will be a route to the complete subnet.

point-to-point This is a *tunnel* to another host, usually on some sort of *serial* link. This interface has a *local* address, and a *remote* address. Although it may be possible to specify multiple addresses for a *point-to-point* interface, there does not seem to be a useful reason for doing so.

The *remote* address must be unique among all the interface addresses on a given router. The *local* address may be shared among many *point-to-point* and up to one non-*point-to-point* interface. This is technically a form of the *router id* method for address-less links. This technique conserves subnets as none are required when using this technique.

If a subnet mask is specified on a *point-to-point* interface, it is only used by RIP version 1 and HELLO to determine which subnets may be propagated to the router on the other side of this interface.

non-broadcast multi-access or nbma This type of interface is multi-access, but not capable of broadcast. And example would be *frame relay* and *X.25*. This type of interface has a local address and a subnet mask.

GateD insures that there is a route available to each IP interface that is configured and up. Normally this this done by the *ifconfig* command that configures the interface; GateD does it to insure consistency.

For *point-to-point* interfaces, gated installs some special routes. If the *local* address on one or more *point-to-point* interfaces is not shared with a non-*point-to-point* interface, gated installs a route to the *local* address pointing at the *loopback* interface with a preference of 110. This insures that packets originating on this host destined for this *local* address are handled locally. OSPF prefers to route packets for the *local* interface across the *point-to-point* link where they will be returned by the router on the remote end. This is used to verify operation of the link. Since OSPF installs routes with a preference of 10, these routes will override the route installed with a preference of 110.

If the *local* address of one or more *point-to-point* interfaces is shared with a non-*point-to-point* interface, gated installs a route to the *local* with a preference of 0 that will not be installed in the forwarding table. This is to prevent protocols like OSPF from routing packets to this address across a serial interface when this system could be functioning as a *host*.

When the status of an interface changes, GateD notifies all the protocols, which take the appropriate action. GateD assumes that interfaces which are not marked *UP* do not exist. While this might not be the most correct action, it is the way things currently work.

GateD ignores any interfaces that have invalid data for the *local*, *remote* or *broadcast* addresses or the *subnet mask*. Invalid data includes zeros in any field. GateD will also ignore any *point-to-point* interface that has the same local and remote addresses, it assumes it is in some sort of *loopback test* mode.

4.9 Definition Statements

Definition statements are general configuration statements that relate to all of GateD or at least to more than one protocol. The three definition statements are **autonomoussystem**, **routerid** and **martians**. if used, **autonomoussystem**, **routerid** and **martians** must appear before any other type of configuration statement in gated.conf file.

4.9.1 Autonomous System configuration

autonomoussystem *autonomous_system* [**loops** *number*] **;**

Sets the autonomous system number of this router to be *autonomous system*. This option is required if BGP or EGP are in use. The AS number is assigned by the Network Information Center (NIC).

Loops is only for protocols supporting AS paths, such as BGP. It controls the number of times this autonomous system may appear in an AS path and defaults to 1 (one).

4.9.2 Router ID configuration

routerid *host* **;**

Sets the router identifier for use by the BGP and OSPF protocols. The default is the address of the first interface encountered by GateD. The address of a non-point-to-point interface is preferred over the local address of a point-to-point interface and an address on a loopback interface that is not the loopback address (127.0.0.1) is most preferred.

4.9.3 Martian configuration

martians {

  host *host* [**allow**] **;**
  network [**allow**] **;**
  network **mask** *mask* [**allow**] **;**

network **masklen** *number* [**allow**] **;**
default [**allow**] **;**
} ;

Defines a list of *martian* addresses about which all routing information is ignored. Sometimes a misconfigured system sends out obviously invalid destination addresses. These invalid addresses, called martians, are rejected by the routing software. This command allows additions to the list of martian addresses. See section 4.25.1 on for more information on specifying ranges. Also, the **allow** parameter may be specified to explicitly allow a subset of a range that was disallowed.

4.9.4 Sample Definition Statements

```
autonomoussystem 249 ;

martians {
    0.0.0.26
};
```

The statements in the sample perform the following functions:

- The autonomoussystem statement tells GateD to use AS number 249 for in EGP and BGP.
- The martians statement prevents routes to 0.0.0.26 from ever being accepted.

4.10 RIP

4.10.1 Description

One of the most widely used interior gateway protocols is the Routing Information Protocol (RIP). RIP is an implementation of a distance-vector, or Bellman-Ford routing protocol for local networks. It classifies routers as active and passive (silent). Active routers advertise their routes (reachability information) to others; passive routers listen and update their routes based on advertisements, but do not advertise. Typically, routers run RIP in active mode, while hosts use passive mode.

A router running RIP in active mode broadcasts updates at set intervals. Each update contains paired values where each pair consists of an IP network address and an integer distance to that network. RIP uses a hop count metric to measure the distance to a destination. In the RIP metric, a router advertises directly connected networks at a metric of 1. Networks which are reachable through one other gateway are two hops etc. Thus, the number of hops or hop count along a path from a given source to a given destination refers to the number of gateways that a datagram would encounter along that path. Using hop counts to calculate shortest paths does not always produce optimal results. For example, a path with hop count 3 that crosses three Ethernets may be substantially faster that a path with a hop count 2 that crosses two slow-speed serial lines. To compensate for differences in technology many routers advertise artificially high hop counts for slow links.

As delivered with most UNIX systems, RIP is run by the routing daemon, *routed* (pronounced route-"d"). A RIP routing daemon dynamically builds on information received through RIP updates. When started up, it issues a REQUEST for routing information and then listens for responses to the request. If a system configured to supply RIP hears the request, it responds with a RESPONSE packet based on information in its routing database. The RESPONSE packet contains destination network addresses and the routing metric for each destination.

When a RIP RESPONSE packet is received, the routing daemon takes the information and rebuilds the routing database adding new routes and "better" (lower metric) routes to destinations already listed in the database. RIP also deletes routes from the database if the next router to that destination says the route contains more than 15 hops, or if the route is deleted. All routes through a gateway are deleted if no updates are received from that gateway for a specified time period. In general, routing updates are issued every 30 seconds. In many implementations, if a gateway is not heard from for 180 seconds, all routes from that gateway are deleted from the routing database. This 180 second interval also applies to deletion of specific routes.

RIP version 2 (more commonly known as RIP II) add additional capabilities to RIP. Some of these capabilities are compatible with RIP I and some are not. To avoid supplying information to RIP I routes that could be mis-interpreted, RIP II can only use non-compatible features when its packets are multicast. On interfaces that are not capable of IP multicast, RIP I compatible packets are used that do not contain potentially confusing information.

Some of the most notable RIP II enhancements are:

Next hop The primary ones are the ability to advertise a next hop to use other than the router supplying the routing update. This is quite useful when advertising a static route to a *dumb* router that does not run RIP as it avoids having packets destined through the dumb router from having to cross a network twice.

RIP I routers will ignore next hop information in RIP II packets. This may result in packets crossing a network twice, which is exactly what happens with RIP I. So this information is provided in RIP I compatible RIP II packets.

Network Mask RIP I assumes that all subnetworks of a given network have the same network mask. It uses this assumption to calculate the network masks for all routes received. This assumption prevents subnets with different netmasks from being included in RIP packets. RIP II adds the ability to explicitly specify the network mask with each network in a packet.

While RIP I routers will ignore the network mask in RIP II packets, their calculation of the network mask will quite possibly be wrong. For this reason, RIP I compatible RIP II packets must not contain networks that would be mis-interpreted. These network must only be provided in native RIP II packets that are multicast.

Authentication RIP II packets may also contain one of two types of authentication string that may be used to verify the validity of the supplied routing data. Authentication may be used in RIP I compatible RIP II packets, but be aware that RIP I routers will ignore it.

The first method is a simple password in which an authentication key of up to 16 characters is included in the packet. If this does not match what is expected, the packet will be discarded. This method provides very little security as it is possible to learn the authentication key by watching RIP packets.

The second method is still experimental and may change in incompatible ways in future releases. This method uses the MD5 algorithm to create a crypto-checksum of a RIP packet and an authentication key of up to 16 characters. The transmitted packet does not contain the authentication key itself, instead it contains a crypto-checksum, called the *digest*. The receiving router will perform a calculation using the correct authentication key and discard the packet if the digest does not match. In addition, a sequence number is maintained to prevent the replay of older packets. This method provides a much stronger assurance that routing data originated from a router with a valid authentication key.

Two authentication methods can be specified per interface. Packets are always sent using the *primary* method, but received packets are checked with both the *primary* and *secondary* methods before being discarded. In addition, a separate authentication key is used for non-router queries.

4.10.2 RIP-I and network masks

RIP-I derives the network mask of received networks and hosts from the network mask of the interface via which the packet was received. If a received network or host is on the same natural network as the interface over which it was received and that network is subnetted (the specified mask is more specific than the *natural* netmask), the subnet mask is applied to the destination. If bits outside the mask are set it is assumed to be a host, otherwise it is assumed to be a subnet.

On point-to-point interfaces, the netmask is applied to the remote address. The netmask on these interfaces is ignored if it matches the natural network of the remote address or is all ones.

Unlike in previous releases, the zero subnet mask (a network that matches the natural network of the interface, but has a more specific, or longer, network mask) is ignored. If this is not desirable, a route filter may be used to reject it.

4.10.3 The RIP Statement

rip yes | **no** | **on** | **off** [**{**
  broadcast ;
  nobroadcast ;
  nocheckzero ;
  preference *preference* **;**
  defaultmetric *metric* **;**
  query authentication [**none** | [[**simple**|**md5**] *password*]] **;**
  interface *interface_list*
    [**noripin**] | [**ripin**]
    [**noripout**] | [**ripout**]
    [**metricin** *metric*]
    [**metricout** *metric*]
    [**version 1**]|[**version 2** [**multicast**|**broadcast**]]
    [[**secondary**] **authentication** [**none** | [[**simple**|**md5**] *password*]] **;**
  trustedgateways *gateway_list* **;**
  sourcegateways *gateway_list* **;**
  traceoptions *trace_options* **;**
}] ;

The **rip** statement enables or disables RIP. If the **rip** statement is not specified the default is **rip on ;**. If enabled, RIP will assume **nobroadcast** when there is only one interface and **broadcast** when there is more than one.

The options are as follows:

broadcast

Specifies that RIP packets will be broadcast regardless of the number of interfaces present. This is useful when propagating static routes or routes learned from another protocol into RIP. In some cases, the use of **broadcast** when only one network interface is present can cause data packets to traverse a single network twice.

nobroadcast

Specifies that RIP packets will not be broadcast on attached interfaces, even if there are more than one. If a **sourcegateways** clause is present, routes will still be unicast directly to that gateway.

nocheckzero

Specifies that RIP should not make sure that reserved fields in incoming version 1 RIP packets are zero. Normally RIP will reject packets where the reserved fields are zero.

preference *preference*

Sets the preference for routes learned from RIP. The default preference is 100. This preference may be overridden by a preference specified in import policy.

defaultmetric *metric*

Defines the metric used when advertising routes via RIP that were learned from other protocols. If not specified, the default value is 16 (unreachable). This choice of values requires you to explicitly specify a metric in order to export routes from other protocols into RIP. This metric may be overridden by a metric specified in export policy.

query authentication [**none** | (**simple**|**md5**) *password*] **;**

Specifies the authentication required of query packets that do not originate from routers. The default is **none**.

interface *interface_list*

Controls various attributes of sending RIP on specific interfaces. See the section on interface list specification for the description of the *interface_list*.

Note that if there are multiple interfaces configured on the same subnet, RIP updates will only be sent from the first one for which RIP output is configured. This limitation is required because of the way the Unix kernel operates. It will hopefully be removed in a future release.

The possible parameters are:

noripin Specifies that RIP packets received via the specified interface will be ignored. The default is to listen to RIP packets on all non-loopback interfaces.

ripin This is the default. This argument may be necessary when **noripin** is used on a wildcard interface descriptor.

noripout Specifies that no RIP packets will be sent on the specified interfaces. The default is to send RIP on all broadcast and non-broadcast interfaces when in **broadcast** mode. The sending of RIP on point-to-point interfaces must be manually configured.

ripout This is the default. This argument is necessary when it is desired to send RIP on point-to-point interfaces and may be necessary when **noripin** is used on a wildcard interface descriptor.

metricin *metric* Specifies the RIP metric to add to incoming routes before they are installed in the routing table. The default is the kernel interface metric plus 1 (which is the default RIP hop count). If this value is specified it will be used as the absolute value, the kernel metric will not be added. This option is used to make this router prefer RIP routes learned via the specified interface(s) less than RIP routes from other interfaces.

metricout *metric* Specifies the RIP metric to be added to routes that are send via the specified interface(s). The default is zero. This option is used to make other routers prefer other sources of RIP routes over this router.

version 1 Specifies that RIP packets send via the specified interface(s) will be version 1 packets. This is the default.

version 2 Specifies that RIP version 2 packets will be sent on the specified interfaces(s). If IP multicast support is available on this interface the default is to send full version 2 packets. If it is not available, version 1 compatible version 2 packets will be sent.

multicast Specifies that RIP version 2 packets should be multicast on this interface. This is the default.

broadcast Specifies that RIP version 1 compatible version 2 packets should be broadcast on this interface, even if IP multicast is available.

[secondary] authentication [none | [simple|md5] *password***]** This defines the authentication type to use. It applies only to RIP version 2 and is ignored for RIP-1 packets. The default authentication type is **none**. If a password is specified, the authentication type defaults to **simple**. The password should be a quoted string with between 0 and 16 characters.

If **secondary** is specified, this defines the secondary authentication. If omitted, the primary authentication is specified. The default is primary authentication of **none** and no secondary authentication.

trustedgateways *gateway_list*

Defines the list of gateways from which RIP will accept updates. The *gateway_list* is simply a list of host names or IP addresses. By default, all routers on the shared network are trusted to supply routing information. But if the **trustedgateways** clause is specified only updates from the gateways in the list are accepted.

sourcegateways *gateway_list*

Defines a list of routers to which RIP sends packets directly, not through multicast or broadcast. This can be used to send different routing information to specific gateways. Updates to gateways in this list are not affected by **noripout** on the interface.

traceoptions *trace_options*

Specifies the tracing options for RIP. (See Trace Statements and the RIP specific tracing options below.)

4.10.4 Tracing options

The **policy** option logs info whenever a new route is announce, the metric being announced changes or a route goes or leaves holddown.

Packet tracing options (which may be modified with **detail**, **send** or **recv**):

packets All RIP packets.

request RIP information request packets, such as *REQUEST*, *POLL* and *POLLENTRY*

response RIP *RESPONSE* packets, which is the type of packet that actually contains routing information.

other Any other type of packet. The only valid ones are *TRACE_ON* and *TRACE_OFF* both of which are ignored.

4.11 HELLO

4.11.1 The Hello Protocol

It's really better not to use HELLO unless you have a specific need for it. We plan to drop it some time around GateD 4.0.

The HELLO protocol is an interior protocol that uses a routing metric based on the length of time it takes a packet to make the trip between the source and the destination. HELLO packets carry timestamp information which allows receivers to compute the shortest delay paths to destinations. The "best" route is the route with the shortest time delay. The unit of time used in HELLO is milliseconds. If a HELLO update packet takes less than 100 milliseconds to travel between two routers, a minimum value of 100 is used for that hop. Thus on networks built of high-speed interfaces HELLO essentially defaults to using hop counts. As in any routing algorithm, HELLO cannot change routes too rapidly or it would be unstable. To avoid

Hops	HELLO metric
0	0
1	100
2	148
3	219
4	325
5	481
6	713
7	1057
8	1567
9	2322
10	3440
11	5097
12	7552
13	11190
14	16579
15	24564
16	30000

Table 4.2: Hello Metrics

instabilities, implementations of HELLO build in hysteresis and "hesitate" to change routes until they have confidence that the change will be lasting.

By default HELLO, like RIP, uses the kernel interface metric set by the *ifconfig* command to influence metric added to routes as they are installed in the routing table (*metricin*). Since the kernel interface metric is in hops, it must be translated into HELLOs millisecond metric. In order to do that, table 4.2 is used:

4.11.2 HELLO and network masks

HELLO derives the network mask of received networks and hosts from the network mask of the interface the packet via which the packet was received. If a received network or host is on the same natural network as the interface over which it was received and that network is subnetted (the specified mask is more specific than the *natural* netmask), the subnet mask is applied to the destination. If bits outside the mask are set it is assumed to be a host, otherwise it is assumed to be a subnet.

On point-to-point interfaces, the netmask is applied to the remote address. The netmask on these interfaces is ignored if it matches the natural network of the remote address or is all ones.

Unlike in previous releases, the zero subnet mask (a network that matches the natural network of the interface, but has a more specific, or longer, network mask) is ignored. If this is not desirable, a route filter may be used to reject it.

4.11.3 The Hello Statement

hello yes | **no** | **on** | **off** [**{**
        broadcast ;
        nobroadcast ;
        preference *preference* **;**
        defaultmetric *metric* **;**
        interface *interface_list*
                [**nohelloin**] | [**helloin**]
                [**nohelloout**] | [**helloout**]

```
                    [metricin metric]
                    [metricout metric] ;
          trustedgateways gateway_list ;
          sourcegateways gateway_list ;
          traceoptions trace_options ;
} ] ;
```

the **hello** statement enables or disables HELLO. If the **hello** statement is not specified the default is **hello off**. If enabled, HELLO will assume **nobroadcast** when there is only one interface and **broadcast** when there is more than one interface.

broadcast

Specifies that HELLO packets will be broadcast regardless of the number of interfaces present. This is useful when propagating static routes or routes learned from anther protocol into HELLO. In some cases, the use of **broadcast** when only one network interface is present can cause data packets to traverse a single network twice.

nobroadcast

Specifies that HELLO packets will not be broadcast on attached interfaces, even if there are more than one. If a **sourcegateways** clause is present, routes will still be unicast directly to that gateway.

preference *preference*

Sets the preference for routes learned from HELLO. The default preference is op. This preference may be overridden by a preference specified in import policy.

defaultmetric *metric* Defines the metric used when advertising routes via HELLO that were learned from other protocols. If not specified, the default value is 30000 (unreachable). This choice of values requires you to explicitly specify a metric in order to export routes from other protocols into HELLO. This metric may be overridden by a metric specified in export policy.

interface *interface_list*

Controls various attributes of sending HELLO on specific interfaces. See section 4.8.2 on interface list specification for the description of the *interface_list*.

Note that if there are multiple interfaces configured on the same subnet, HELLO updates will only be sent from first one one which HELLO output is configured. This limitation is required because of the way the Unix kernel operates. It will hopefully be removed in a future release.

The possible parameters are:

nohelloin Specifies that HELLO packets received via the specified interface will be ignored. The default is to listen to HELLO on all non-loopback interfaces.

helloin This is the default. This argument may be necessary when **nohelloin** is used on a wildcard interface descriptor.

nohelloout Specifies that no HELLO packets will be sent on the specified interfaces. The default is to send HELLO on all broadcast and non-broadcast interfaces when in **broadcast** mode. The sending of HELLO on point-to-point interfaces must be manually configured.

helloout This is the default. This argument is necessary when it is desired to send HELLO on point-to-point interfaces and may be necessary when **nohelloin** is used on a wildcard interface descriptor.

metricin *metric* Specifies the HELLO metric to add to incoming routes before they are installed in the routing table. The default is the kernel interface metric plus 1 (which is the default HELLO hop count). If this value is specified it will be used as the absolute value, the kernel metric will not be added. This option is used to make this router prefer HELLO routes learned via the specified interface(s) less than HELLO routes from other interfaces.

metricout *metric* Specifies the HELLO metric to be added to routes that are send via the specified interface(s). The default is zero. This option is used to make other routers prefer other sources of HELLO routes over this router.

trustedgateways *gateway_list*

Defines the list of gateways from which HELLO will accept updates. The *gateway_list* is simply a list of host names or IP addresses. By default, all routers on the shared network are trusted to supply routing information. But if the **trustedgateways** clause is specified only updates from the gateways in the list are accepted.

sourcegateways *gateway_list*

Defines a list of routers to which HELLO sends packets directly, not through multicast or broadcast. This can be used to send different routing information to specific gateways. Updates to gateways in this list are not affected by **noripout** on the interface.

traceoptions *trace_options*

Specifies the tracing options for HELLO. (See Trace Statements and the HELLO specific tracing options below.)

The default preference is 90. The default metric is 30000.

4.11.4 Tracing options

The **policy** option logs info whenever a new route is announced, the metric being announced changes or a route goes or leaves holddown.

Packet tracing options (which may be modified with **detail**, **send** and/or **recv**):

packets All HELLO packets

4.12 OSPF

4.12.1 The OSPF Protocol

Open Shortest Path Routing (OSPF) is a *shortest path first* (SPF) or *link-state* protocol. OSPF is an interior gateway protocol that distributes routing information between routers in a single autonomous system. OSPF chooses the least cost path as the best path. Suitable for complex networks with a large number of routers, OSPF provides equal cost multipath routing where packets to a single destination can be sent via more than one interface simultaneously. In a link-state protocol, each router maintains a database describing the entire AS topology, which it builds out of the collected link state advertisements of all routers. Each participating router distributes its local state (i.e., the router's usable interfaces and reachable neighbors) throughout the AS by flooding. Each multiaccess network that has at least two attached routers has a *designated router* and a *backup designated router*. The designated router floods a link state advertisement for the multiaccess network and has other special responsibilities. The designated router concept reduces the number of adjacencies required on a multiaccess network.

OSPF allows networks to be grouped into areas. Routing information passed between areas is abstracted, potentially allowing a significant reduction in routing traffic. OSPF uses four different types of routes, listed in order of preference: intra-area, inter-area, type 1 external and type 2 external. Intra-area paths have destinations within the same area, inter-area paths have destinations in other OSPF areas and Autonomous System External (ASE) routes are routes to destinations external to the AS. Routes imported into OSPF as type 1 routes are supposed to be from igps whose external metrics are directly comparable to OSPF metrics. When a routing decision is being made, OSPF will add the internal cost to the AS Border router to the external metric. Type 2 ASEs are used for egps whose metrics are not comparable to OSPF metrics. In this case, only the internal OSPF cost to the AS Border router is used in the routing decision.

From the topology database, each router constructs a tree of the shortest paths with itself as the root. This shortest-path tree gives the route to each destination in the AS. Externally derived routing information appears on the tree as leaves. The link-state advertisement format distinguishes between information acquired from external sources and information acquired from internal routers, so there is no ambiguity about the source or reliability of routes. Externally derived routing information (for example, routes learned from EGP or BGP) is passed transparently through the autonomous system and is kept separate from OSPF's internally derived data. Each external route can also be tagged by the advertising router, enabling a passing of additional information between routers on the borders of the autonomous system.

OSPF optionally includes *type of service* (TOS) routing and allows administrators to install multiple routes to a given destination for each type of service (e.g. low delay or high throughput.) A router running OSPF uses the destination address and the type of service to choose the best route to the destination.

OSPF intra- and inter-area routes are always imported into the GateD routing database with a preference of 10. It would be a violation of the protocol if an OSPF router did not participate fully in the area's OSPF, so it is not possible to override this. Although it is possible to give other routes lower preference values explicitly, it is ill-advised to do so.

Hardware multicast capabilities are also used where possible to deliver link-status messages.

OSPF areas are connected by the *backbone* area, the area with identifier 0.0.0.0. All areas must be logically contiguous and the backbone is no exception. To permit maximum flexibility, OSPF allows the configuration of *virtual* links to enable the backbone area to appear contiguous despite the physical reality.

All routers in an area must agree on that area's parameters. A separate copy of the link-state algorithm is run for each area. Because of this, most configuration parameters are defined on a per area basis. All routers belonging to an area must agree on that area's configuration. Misconfiguration will lead to adjacencies not forming between neighbors, and routing information might not flow, or even loop.

4.12.2 Authentication

All OSPF protocol exchanges are authenticated. Authentication guarantees that routing information is only imported from trusted routers, to protect the Internet and its users. A variety of authentication schemes can be used but a single scheme must be configured for each area. This enables some areas to use much stricter authentication than others.

OSPF protocol exchanges may be authenticated. Authentication guarantees that routing information is imported only from trusted routers, to protect the Internet and its users. There are two authentication schemes available. The first uses a simple authentication key of up to 8 characters and is standardized. The second is still experimental and uses the MD5 algorithm and an authentication key of up to 16 characters.

The simple password provides very little protection because in many cases it is possible to easily capture packets from the network and learn the authentication key. The experimental MD5 algorithm provides much more protection as it does not include the authentication key in the packet.

The OSPF specification currently specifies that the authentication type be configured per area with the ability to configure separate passwords per interface. This has been extended to allow the configuration of different authentication types and keys per interface. In addition it is possible to specify both a *primary* and a *secondary* authentication type and key on each interface. Outgoing packets use the primary authentication type, but incoming packets may match either the primary or secondary authentication type and key.

4.12.3 The OSPF Statement

```
ospf yes | no | on | off [ {
        defaults {
                preference preference ;
                cost cost ;
                tag [ as ] tag ;
                type 1 | 2 ;
        } ;
        exportlimit routes ;
        exportinterval time ;
        traceoptions trace_options ;
        monitorauthkey authkey ;
        monitorauth none | ( [ simple | md5 ] authkey ) ;
        backbone | ( area area ) {
                authtype 0 | 1 | none | simple ;
                stub [ cost cost] ;
                networks {
                        network [ restrict ] ;
                        network mask mask [ restrict ] ;
                        network masklen number [ restrict ] ;
                        host host [ restrict ] ;
                } ;
                stubhosts {
                        host cost cost ;
                } ;
                interface interface_list; [cost cost ] {
                        interface_parameters
                } ;
                interface interface_list nonbroadcast [cost cost ] {
                        pollinterval time ;
                        routers {
                                gateway [ eligible ] ;
                        } ;
                        interface_parameters
                } ;
                Backbone only:
                virtuallink neighborid router_id transitarea area {
                        interface_parameters
                } ;
        } ;
} ] ;
```

The following are the *interface_parameters* referred to above. They may be specified on any class of interface and are described under the **interface** clause.

```
enable | disable ;
retransmitinterval time ;
transitdelay time ;
priority priority ;
hellointerval time ;
```

routerdeadinterval *time* **;**
[**secondary**] **auth** (**none**|**simple**|**md5**|*auth_key*) **;**
authkey *auth_key* **;**

defaults

These parameters specify the defaults used when importing OSPF ASE routes into the gated routing table and exporting routes from the gated routing table into OSPF ASEs.

preference *preference* The preference is used to determine how OSPF routes compete with routes from other protocols in the gated routing table. The default value is 150.

cost *cost* The cost is used when exporting a non-OSPF route from the GateD routing table into OSPF as an ASE. The default value is 1. This may be explicitly overridden in export policy.

tag [**as**] *tag* OSPF ASE routes have a 32 bit tag field that is not used by the OSPF protocol, but may be used by export policy to filter routes. When OSPF is interacting with an egp, the tag field may be used to propagate AS path information, in which case the **as** keyword is specified and the tag is limited to 12 bits of information. If not specified, the tag is set to zero.

type *1* | *2* Routes exported from the GateD routing table into OSPF default to becoming type 1 ASEs. This default may be explicitly changed here and overridden in export policy.

ASE export rate

Because of the nature of OSPF, the rate at which ASEs are flooded must be limited. These two parameters can be used to adjust those rate limits.

exportinterval *time* This specifies how often a batch of ASE link state advertisements will be generated and flooded into OSPF. The default is once per second.

exportlimit *routes* This parameter specifies how many ASEs will be generated and flooded in each batch. The default is 100.

traceoptions *trace_options*

Specifies the tracing options for OSPF. (See Trace Statements and the OSPF specific tracing options below.)

monitorauthkey *authkey*

OSPF state may be queried using the *ospf_monitor* utility. This utility sends non-standard OSPF packets which generate a text response from OSPF. By default these requests are not authenticated, if an authentication key is configured, the incoming requests must match the specified authentication key. No OSPF state may be changed by these packets, but the act of querying OSPF can utilize system resources.

backbone

area *area*

Each OSPF router must be configured into at least one OSPF area. If more than one area is configured, at least one must the be *backbone*. The backbone may only be configured using the **backbone** keyword, it may not be specified as *area 0*. The backbone interface may be a *virtuallink*.

authtype *0* | *1* | *none* | *simple* OSPF specifies an authentication scheme per area. Each interface in the area must use this same authentication scheme although it may use a different *authenticationkey.* The currently valid values are **none** (*0*) for no authentication, or **simple** (*1*) for simple password authentication.

stub [**cost** *cost*] A *stub* area is one in which there are no ASE routes. If a *cost* is specified, this is used to inject a default route into the area with the specified cost.

networks The *networks* list describes the scope of an area. Intra-area LSAs that fall within the specified ranges are not advertised into other areas as inter-area routes. Instead, the specified ranges are advertised as *summary network* LSAs. If **restrict** is specified, the summary network LSAs are not advertised. Intra-area LSAs that do not fall into any range are also advertised as summary network LSAs. This option is very useful on well designed networks in reducing the amount of routing information propagated between areas. The entries in this list are either networks, or a subnetwork/mask pair. See section 4.25.1 on route filtering for more detail about specifying ranges.

stubhosts This lists specifies directly attached hosts that should be advertised as reachable from this router and the costs they should be advertised with. Point-to-point interfaces on which it is not desirable to run OSPF should be specified here.

It is also useful to assign a additional address to the loopback interface (one not on the 127 network) and advertise it as a stub hosts. If this address is the same one used as the router-id it enables routing to OSPF routers by router-id, instead of by interface address. This is more reliable than routing to one of the routers interface addresses which may not always be reachable.

interface *interface_list* [**cost** *cost*]

This form of the interface clause is used to configure a *broadcast* (which requires IP multicast support) or a *point-to-point* interface. See section 4.8.2 on interface list specification for the description of the *interface_list.*

Each interface has a *cost.* The costs of all interfaces a packet must cross to reach a destination are summed to get the cost to that destination. The default cost is one, but another non-zero value may be specified.

Interface parameters common to all types of interfaces are:

retransmitinterval *time* The number of seconds between link state advertisement retransmissions for adjacencies belonging to this interface.

transitdelay *time* The estimated number of seconds required to transmit a link state update over this interface. Transitdelay takes into account transmission and propagation delays and must be greater than 0.

priority *priority* A number between 0 and 255 specifying the priority for becoming the designated router on this interface. When two routers attached to a network both attempt to become designated router, the one with the highest priority wins. A router whose router priority is set to 0 is ineligible to become designated router.

hellointerval *time* The length of time, in seconds, between Hello packets that the router sends on the interface.

routerdeadinterval *time* The number of seconds not hearing a router's Hello packets before the router's neighbors will declare it down.

[**secondary**] **auth (none**|**simple**|**md5**|*auth_key***)** Specify the type of authentication scheme to use on this interface.

authkey *auth_key* Used by OSPF authentication to generate and verify the authentication field in the OSPF header. The authentication key can be configured on a per interface basis. It is specified by one to eight decimal digits separated by periods, a one to eight byte hexadecimal string preceded by *0x*, or a one to eight character string in double quotes.

Point-to-point interfaces also support this additional parameter:

nomulticast By default, OSPF packets to neighbors on point-to-point interfaces are sent via the IP multicast mechanism. Although, some implementations of IP multicasting for Unix have a bug that precludes the use of IP multicasting on these interfaces. Gated will detect this condition and fall back to using sending unicast OSPF packets to this point-to-point neighbor.

If the use of IP multicasting is not desired because the remote neighbor does not support it, the **nomulticast** parameter may be specified to force the use of unicast OSPF packets. This option may also be used to eliminate warnings when Gated detects the bug mentioned above.

interface *interface_list* **nonbroadcast** [**cost** *cost*]

This form of the interface clause is used to specify a **nonbroadcast** interface on a *non-broadcast multi-access* (NBMA) media. Since an OSPF *broadcast* media must support IP multicasting, a broadcast capable media, such as Ethernet, that does not support IP multicasting must be configured as a non-broadcast interface.

A non-broadcast interface supports any of the standard interface clauses listed above, plus the following two that are specific to non-broadcast interfaces:

pollinterval *time* Before adjacency is established with a neighbor, OSPF packets are sent periodically at the specified **pollinterval**.

routers By definition it is not possible to send broadcast packets to discover OSPF neighbors on a non-broadcast, so all neighbors must be configured. The list includes one or more neighbors and an indication of their eligibility to become a designated router.

virtuallink neighborid *router_id* **transitarea** *area*

Virtual links are used to establish or increase connectivity of the backbone area. The **neighborid** is the *router_id* of the other end of the virtual link. The transit *area* specified must also configured on this system. All standard interface parameters defined by the **interface** clause above may be specified on a virtual link.

4.12.4 Tracing options

In addition to the following OSPF specific trace flags, OSPF supports the **state** which traces interface and neighbor state machine transitions.

lsabuild Link State Advertisement creation

spf Shortest Path First (SPF) calculations

lsatransmit Link State Advertisement (LSA) transmission. Non implemented.

lsareceive LSA reception. Non implemented.

state State transitions. Non implemented.

Packet tracing options (which may be modified with **detail**, **send** and **recv**):

hello OSPF *HELLO* packets which are used to determine neighbor reachability.

dd OSPF Database Description packets which are used in synchronizing OSPF databases.

request OSPF Link State Request packets which are used in synchronizing OSPF databases.

lsu OSPF Link State Update packets which are used in synchronizing OSPF databases.

ack OSPF Link State Ack packets which are used in synchronizing OSPF databases.

4.12.5 The new OSPF Protocol

Gated version 3_6Alpha_2 and higher includes a new implemetation of the OSPF protocol. The configuration syntax stays the same except:

1. The **export limit**, **exportinterval**, **monitorauthkey**, **monitorauth**, **networks**, **stubhosts** statements are no longer available (they may be put back in the future).
2. The bf simple authentication scheme is not implemented.
3. The OSPF specific trace options available are: **state**, **drelection**, **db**, **spf**, **flood** As for the packet details, are available: **hello**, **dd**, **lsr**, **lsu**, **lsa**.

4.13 IS_IS

4.13.1 The IS-IS Intra-Domain Protocol

Further details of IS-IS configuration are currently described in a companion document (ISIS-config.ps) included in the gated distribution. A typical configuration may specify mode (dual, iso), system level, traceoptions, and circuits and their metrics.

IS-IS is a link state interior gateway protocol (IGP) originally developed for routing ISO/CLNP (International Organization for Standardization/Connectionless Network Protocol) packets. The version distributed with GateD can route IP packets as well. In ISO terminology a router is referred to as an "intermediate system" (IS). IS-IS intra-domain routing is organized hierarchically so that a large domain may be administratively divided into smaller areas using level 1 ISs within areas and level 2 ISs between areas. Routing between administrative domains is handled by Border Intermediate Systems (BISs) using IDRP, the inter-domain routing protocol. Level 1 systems route directly to systems within their own area and route toward a Level 2 Intermediate System when the destination system is in a different area. Level 2 Intermediate Systems route between areas and keep track of the paths to destination areas. Systems in the Level 2 subdomain route towards a destination area, or another routing domain. As with any internet routing protocol, IS-IS support for large routing domains may also include many types of individual subnetworks. These subnetworks may include point-to-point links, multipoint links, dynamically established data links as in X.25 subnetworks and broadcast subnetworks like ISO 8802 LANs.

In IS-IS all subnetwork types are treated by the subnetwork independent functions as though they were connectionless subnetworks using subnetwork convergence functions where necessary. Like OSPF, IS-IS uses a "shortest-path first" algorithm to determine routes. A congestion control component monitors and prevents buffer deadlock at each intermediate system. GateD configuration syntax allows as much auto-configuration as possible reducing the probability of error. This integration also allows the ability to specify policy for exchanging routing information with other protocols running in GateD, such as BGP, EGP, RIP and IDRP. The new IS-IS protocol supports multipath (load-split) forwarding and full injection of exterior network prefixes and attribute information with the goal of eliminating the need for any internal BGP or similar protocols. IS-IS supports static routing domain information at level 2 ISs.

The reachable address prefix indicates that any Network Service Access Points (NSAPs) which match the prefix may be reachable via the Subnet Point of Attachment (SNPA) with which the prefix is associated. Where the subnetwork to which this SNPA is connected is a general topology subnetwork supporting dynamically established data links, the prefix also has associated with it the required subnetwork addressing information, or an indication that it may be derived from the destination NSAP address. The address prefixes are handled by the Level 2 routing algorithm in the same way information about Level 1 is handled within the domain.

4.13.2 The IS-IS Statement

isis no | **dual** | **ip** | **iso** {
    level 1|2 ;
    [**traceoptions** *isis_traceoptions* ;]
    [**systemid** *6_digit_hexstring* ;]
    [**area** *hexstring* ;]
    [**set** *isis_parm value* ; ...]
    circuit *string*
        metric [**level** 1|2] 1..63;
        ...
        priority [**level** 1|2] 0..127;
        ...
        ;
    ...
} ;

This statement enables the IS-IS protocol in GateD. By default IS-IS is disabled. The **dual** option specifies that the IS-IS protocol is enabled for both ISO and IP addressing. The **isis** statement consists of an initial description of the IS and a list of statements that determine the configuration of the specific circuits and networks to be managed. Statements may appear in any order and include:

level

Indicates whether gated is running on a Level 1 (intra-area) or Level 2 (inter-area) IS. The default is Level 1.

traceoptions

are covered in the Trace Options section below.

systemid *6_digit_hexstring*

Overrides the auto-configured system ID (determined from interface addresses and corresponding netmasks.) If no system identifier is specified, the system ID portion of the first real circuit's NSAP is used. Once a system ID is set, it cannot be changed without disabling and re-enabling all of IS-IS.

area *hexstring*

IS-IS area addresses are automatically configured based on the real circuits over which IS-IS runs. Addresses specified in this statement are maintained in addition to those configured automatically from the circuits. This command is used primarily for simulation.

circuit *string*

Each **circuit** statement specifies one of the circuits the system will manage. Circuits normally correspond to UNIX interfaces, with *string* being the interface name, but simulated device names may also be specified. If the string is in the form of "simN", where N is an integer, the circuit is assumed to be a simulated circuit managed by the network simulator troll. The circuit attributes are a list of options that may appear in any order in the circuit statement.

metric

Allows specifications of Level 1 and Level 2 metrics for each circuit. Only the default metric type is supported. IS-IS metrics must be in the range 1 to 63. If no metric is set for the circuit, the default value is 63.

priority

Determines designated router election results; higher values give a higher likelihood of becoming the designated router. The level defaults to Level 1. If no priority is specified, priority is set to a random value between 0 and 127.

On a level 2 IS, to configure a circuit with a Level 1 metric of 10 and a Level 2 metric of 20, add two metric options to the circuit statement.

The default Level is 1 : the default metric is 63. The default preference for IS-IS Level 1 is 15 for IS-IS Level 2 is 18.

4.13.3 Tracing options

IS-IS configuration is most thoroughly described in a companion document (ISIS-config.ps) included in the GateD distribution. Traceoptions can be one or more of the following:

all

iih

lanadj

p2padj

lspdb

lspcontent

lspinput

flooding

buildlsp

csnp

psnp

route

update

paths

spf

events

4.14 IGMP

4.14.1 Internet Group Management Protocol (IGMP)

IGMP was primarily designed for hosts on multi-access networks to inform locally attached routers of their group membership information. This is performed by hosts multicasting IGMP Host Membership Reports. Multicast routers listen for these messages and then can exchange group membership information with other multicast routers. This allows distribution trees to be formed to deliver multicast datagrams. The original version of IGMP was defined in [RFC 1112].

Extensions to IGMP have been developed and released in later releases of the IP Multicast distribution from Xerox PARC. These extensions, known as IGMP version 2, include explicit leave messages for faster pruning and multicast traceroute messages. GATED is aware of both IGMP Version 1 and Version 2 messages. However, the multicast traceroute features have not yet been implemented. Of course, this will be done "real soon now".

4.14.2 The IGMP Statement

igmp yes | **no** | **on** | **off** [**{**
    queryinterval *sec* **;**
    timeoutinterval *sec* **;**
    interface *interface_list* **enable** | **disable**;
    traceoptions *trace_options* **;**
}] ;

The **igmp** statement enables or disables the IGMP protocol. If the **igmp** statement is not specified the default is **igmp off;** If enabled, IGMP will default to enabling all interfaces that are both broadcast and multicast capable. These interfaces are identified by the IFF_BROADCAST and IFF_MULTICAST interface flags. IGMP must be enabled before one of the IP Multicast routing protocols are enabled.

The options are as follows:

broadcast

queryinterval *sec*

This is the interval at which IGMP Host Membership Queries are sent if this router is elected the designated querier for the LAN. If not specified, the default value is 125 seconds.

timeoutinterval *sec*

This is the longest interval that a group will remain in the local group database without receiving a Host Membership Report. If not specified, the default value is 270 seconds. It is calculated as 2*(queryinterval + 10). The 10 seconds is the variation that hosts use when sending Host Membership Reports. If the value of queryinterval is specified and the timeout interval is not specified, gated will use this formula to calculate an appropriate timeout interval. However, if a timeout interval is specified, it will override any calculated value.

interface *interface_list*

Enables or disables IGMP on this interface or list of interfaces. Currently, the gated implementation of IGMP cannot detect when there are multiple addresses configured on the same physical interface. It will send IGMP messages to each logical network. This should be fixed in a future release.

The possible parameters are:

disable Specifies that IGMP packets received via the specified interface will be ignored. The default is to listen to IGMP on all interfaces.

enable This is the default. This argument may be necessary when `disable` is used on a wildcard interface descriptor.

traceoptions *trace_options*

Specifies the tracing options for IGMP. (See Trace Statements and the IGMP specific tracing options below.)

4.14.3 Tracing options

Packet tracing options (which may be modified with `detail`, `send` or `recv`):

packets All IGMP packets.

query IGMP Host Membership Query packets

report IGMP Host Membership Reports and New Host Membership Reports

leave IGMP Host Leave messages

mtrace IGMP Multicast trace route `request` and `response` packets plus Cisco multicast trace messages.

4.15 DVMRP

4.15.1 Distance Vector Multicast Routing Protocol (DVMRP)

DVMRP is the original IP Multicast routing protocol. It was designed to run over both multicast capable LANs (like Ethernet) as well as through non-multicast capable routers. In this case, the IP Multicast packets are "tunneled" through the routers as unicast packets. This replicates the packets and has an effect on performance but has provided an intermediate solution for IP Multicast routing on the Internet while router vendors decide to support native IP Multicast routing.

An earlier version of DVMRP was documented as RFC 1075 but it is not representative of the protocol in use today. The GATED version of DVMRP is compatible with the XEROX PARC releases 2.x and 3.x. It will perform pruning and grafting if the upstream neighbor supports it. It understands the capability flags introduced in XEROX PARC Release 3.5. It does not yet support the IP Multicast Traceroute or the DVMRP MIB. Stay tuned for these features.

This is a preliminary release of DVMRP. There is not currently a method to inject source networks into the DVMRP unicast routing protocol from other regions running different IP Multicast routing protocols such as PIM. This feature will be released as soon as a safe method for doing so has been tested.

4.15.2 The DVMRP Statement

dvmrp yes | **no** | **on** | **off** [**{**
    interface *interface_list* [**{**
        enable | **disable ;**
        metric *metric* **;**
        threshold *threshold* **;**
        ratelimit *rate* **;**
        advertise *network* **metric** *metric* **;**
        advertise *network* **mask** *mask* **metric** *metric* **;**
        advertise *network* **masklen** *number* **metric** *metric* **;**

```
        } ] ;
        tunnel host lcladdr local_address [ {
                enable | disable ;
                metric metric ;
                threshold threshold ;
                ratelimit rate ;
                advertise network metric metric ;
                advertise network mask mask metric metric ;
                advertise network masklen number metric metric ;
        } ] ;
        traceoptions trace_options ;
} ] ;
```

The **dvmrp** statement enables or disables the DVMRP protocol. If the **dvmrp** statement is not specified the default is **dvmrp off;** If enabled, DVMRP will default to enabling all interfaces that are multicast capable. These interfaces are identified by the IFF_MULTICAST interface flag.

Note: only one multicast routing protocol can be configured on a port at a time.

The options are as follows:

interface *interface_list*

Enables or disables DVMRP on this interface or list of interfaces. Currently, the gated implementation of DVMRP cannot detect when there are multiple addresses configured on the same physical interface. It will send DVMRP messages to each logical network. This should be fixed in a future release.

The possible parameters are:

disable Specifies that DVMRP packets received via the specified interface will be ignored. The default is to listen to DVMRP on all multicast capable interfaces.

enable This is the default. This argument may be necessary when **disable** is used on a wildcard interface descriptor.

metric This command provides a way to configure the metric on a physical port or tunnel. This metric will be added to all routes that are learned via this interface.

threshold This command provides a way to configure the threshold on a physical port or tunnel. A packet will not be forwarded out this interface unless the TTL in the packet exceeds this threshold.

ratelimit This command provides a way to limit the rate of multicast traffic on an interface. It is an option in the XEROX PARC IP Multicast kernel (Release 3.3 or greater).

advertise The `advertise` line is used to advertise a particular network out this particular interface only. This is somewhat dangerous but is necessary for joining non-DVMRP multicast regions to DVMRP regions. It can be specified by network number and netmask or network number and prefix length. If no netmask is specified, the default netmask for the class is assumed.

tunnel *host* **lcladdr** *local_address*

The tunnel statement uses the same options as the **interface** statement.

traceoptions *trace_options*

Specifies the tracing options for DVMRP. (See Trace Statements and the DVMRP specific tracing options below.)

4.15.3 Tracing options

Packet tracing options (which may be modified with **detail**, **send** or **recv**):

packets All DVMRP packets.

probe DVMRP Router Probe packets

report DVMRP Route Report packets

mapper DVMRP Neighbor and Neighbor 2 packets

prune DVMRP Prune packets

graft DVMRP Graft and Graft Ack packets

4.16 PIM

4.16.1 Protocol Independent Multicast (PIM)

PIM was designed to take advantage of two existing multicast routing protocols, DVMRP and CBT. It exhibits the behavior of a protocol in a region of dense group membership flooding multicast packets using Reverse Path Multicasting, while also taking advantage of the work done for sparse group membership in Core Based Trees. Hence, the protocol has two modes, *dense* and *sparse*. Currently, only dense mode is implemented in Gated. An overview of PIM can be found in [PIM Arch]. The detailed protocol specification is also available as [PIM Spec]. Both of these documents are considered Work in Progress.

4.16.2 The PIM Statement

```
pim yes | no | on | off [ {
        queryinterval sec ;
        routertimeout sec ;
        prunetimeout sec ;
        inactivitytimeout sec ;
        graftacktimeout sec ;
        preference {
                direct preference ;
                kernel preference ;
                ospf preference ;
                ospfase preference ;
                rip preference ;
                hello preference ;
                static preference ;
                bgp preference ;
                egp preference ;
                idrp preference ;
                isis preference ;
                slsp preference ;
        } ;
        interface interface_list [[enable] | [disable]] [mode [dense | sparse]] ;
        traceoptions trace_options ;
} ] ;
```

The **pim** statement enables or disables the PIM protocol. If the **pim** statement is not specified the default is **pim off;** If enabled, PIM will default to enabling all interfaces that are multicast capable. These interfaces are identified by the IFF_MULTICAST interface flag.

The options are as follows:

queryinterval *sec*

This is the interval at which PIM Router Queries are sent on the multicast capable LAN. PIM Router Queries are sent to the All Routers Multicast Group in order for PIM routers to find other PIM routers on a multiaccess network. If not specified, the default value is 30 seconds.

routertimeout *sec*

This is the interval that a PIM router is remembered without receiving a PIM Router Query. After this time, the router is deleted from the local database. Its value defaults to three times the queryinterval or 90 seconds when the default queryinterval is used.

prunetimeout *sec*

This is the duration that a prune will stay active for a prune sent by this router. It defaults to 180 seconds.

inactivitytimeout *sec*

This is the length of time that a forwarding cache entry will remain active once it becomes inactive, i.e. no more traffic is being forwarded for the Source, Group pair. The default value is 180 seconds.

graftacktimeout *sec*

This is the length of time this PIM router will wait for a Graft to be acknowledged before it retransmits the Graft. The default value is 60 seconds.

preference

The preference statement allows changing the preference values used in comparing protocols to determine the router that will forward multicast packets on a multiaccess LAN. These values are sent in Assert messages when more than one router is capable of forwarding the multicast packets and one detects the others duplicate packets. These values must be the same for each router on a multiaccess LAN. Therefore, the default values have been specifically coordinated to match other implementations. The default values are 0 for direct routes, 1 for static routes, 110 for OSPF and OSPFASE routes, 115 of ISIS routes, 120 for RIP routes, 130 for SLSP routes, 140 for EGP routes, 200 for BGP routes, 210 for IDRP routes, 220 for existing kernel routes, and 255 for hello routes.

interface *interface_list*

Enables or disables PIM on this interface or list of interfaces. Currently, the gated implementation of PIM cannot detect when there are multiple addresses configured on the same physical interface. It will send PIM messages to each logical network. This should be fixed in a future release.

The possible parameters are:

disable Specifies that PIM packets received via the specified interface will be ignored. The default is to listen to PIM on all multicast capable interfaces.

enable This is the default. This argument may be necessary when `disable` is used on a wildcard interface descriptor.

mode This command specifies the default mode of PIM on the interface. It is exchanged with other PIM routers in the Router Query messages and allows the router to determine if explicit joins should be sent upstream (**sparse**) or default flooding and pruning should be performed (**dense**).

traceoptions *trace_options*

Specifies the tracing options for PIM. (See Trace Statements and the PIM specific tracing options below.)

4.16.3 Tracing options

Packet tracing options (which may be modified with **detail**, **send** or **recv**):

packets All PIM packets.

query PIM Router Query packets

register PIM Register, Register Stop, and RP packets

join PIM Join/Prune packets

assert PIM Assert packets

assert PIM Graft and Graft Ack packets

4.17 EGP

4.17.1 The Exterior Gateway Protocol

The Exterior Gateway Protocol (EGP) is an exterior routing protocol used for exchanging routing information with gateways in other autonomous systems. Unlike interior protocols, EGP propagates only reachability indications, not true metrics. EGP updates contain metrics, called *distances* which range from 0 to 255. GateD will only compare EGP distances learned from the same AS.

Before EGP sends routing information to a remote router, it must establish an adjacency with that router. This is accomplished by an exchange of *Hello* (not to be confused with the HELLO protocol, or OSPF HELLO messages) and *I Heard You* (I-H-U) messages with that router. Computers communicating via EGP are called EGP *neighbors*, and the exchange of HELLO and I-H-U messages is referred to as *acquiring a neighbor*. Once the neighbor is acquired, the system *polls* the neighbor for routing information. The neighbor responds by sending an *update* containing routing information. If the system receives a poll from its neighbor, it responds with its own update packet. When the system receives an update, it includes routes from the update into its routing database. If the neighbor fails to respond to three consecutive polls, GateD assumes that the neighbor is down and removes the neighbor's routes from its database.

4.17.2 The EGP Statement

egp yes | **no** | **on** | **off**
{
    preference *preference* **;**
    defaultmetric *metric* **;**
    packetsize *number* **;**
    traceoptions *trace_options* **;**
    group

```
                    [ peeras autonomous_system ]
                    [ localas autonomous_system ]
                    [ maxup number ]
          {
                    neighbor host
                              [ metricout metric ]
                              [ preference preference ]
                              [ preference2 preference ]
                              [ ttl ttl ]
                              [ nogendefault ]
                              [ importdefault ]
                              [ exportdefault ]
                              [ gateway gateway ]
                              [ lcladdr local_address ]
                              [ sourcenet network ]
                              [ minhello | p1 time ]
                              [ minpoll | p2 time ]
                              [ traceoptions trace_options ]
                              ;
          } ;
} ] ;
```

preference *preference*

Sets the preference for routes learned from RIP. The default preference is 200. This preference may be overridden by a preference specified on the **group** or **neighbor** statments or by import policy.

defaultmetric *metric* **;**

Defines the metric used when advertising routes via EGP. If not specified, the default value is 255 which some systems may consider unreachable. This choice of values requires you to explicitly specify a metric when exporting routes to EGP neighbors. This metric may be overridden by a metric specified on the neighbor or group statements or in export policy.

packetsize *maxpacketsize*

This defines the expected maximum size of a packet that EGP expects to receive from this neighbor. If a packet larger than this value is received, it will be incomplete and have to be discarded. The length of this packet will be noted and the expected size will be increased to be able to receive a packet of this size. Specifying the parameter here will prevent the first packet from being dropped. If not specified, the default size is 8192 bytes. All packet sizes are rounded up to a multiple of the system page size.

traceoptions *trace_options*

Specifies the tracing options for EGP. By default these are inherited from the global trace options. These values may be overridden on a group or neighbor basis. (See Trace Statements and the EGP specific tracing options below.)

group

EGP neighbors must be specified as members of a **group**. A group is usually used to group all neighbors in one autonomous system. Parameters specified on the group clause apply to all of the subsidiary neighbors unless explicitly overridden on a neighbor clause. Any number of **group** clauses may specify any number of **neighbor** clauses.

Any parameters from the **neighbor** subclause may be specified on the **group** clause to provide defaults for the whole group (which may be overridden for individual neighbors). In addition, the **group** clause is the only place to set the following attributes:

peeras Identifies the autonomous system number expected from peers in the group. If not specified it will be learned dynamically.

localas Identifies the autonomous system which GateD is representing to the group. The default is that which has been set globally in the **autonomoussystem** statement. This option is usually only used when *masquerading* as another autonomous system and its use is discouraged.

maxup Specifies the number of neighbors GateD should acquire from this group. The default is to acquire all of the neighbors in the group. GateD will attempt to acquire the first **maxup** neighbors in the order listed. If one of the first neighbors is not available, it will acquire one further down the list. If after start-up GateD does manage to acquire the more desirable neighbor, it will drop the less desirable one.

neighbor *neighbor_address*

Each neighbor subclause defines one EGP neighbor within a group. The only part of the subclause that is required is the *neighbor_address* argument which is the symbolic host name or IP address of the neighbor. All other parameters are optional.

preference *preference* Specifies the preference used for routes learned from these neighbors. This can differ from the default EGP preference set in the **egp** statement, so that GateD can prefer routes from one neighbor, or group of neighbors, over another. This preference may be explicitly overriden by import policy.

preference2 *preference* In the case of a *preference* tie, the second preference, *preference2* may be used to break the tie. The default value is 0.

metricout *metric* This defines a metric to be used for all routes sent to this neighbor. The value overrides the default metric set in the **egp** statement and any metrics specified by export policy, but only for this specific neighbor or group of neighbors.

nogendefault Prevents GateD from generating a default route when EGP receives a valid update from its neighbor. The default route is only generated when the **gendefault** option is enabled.

importdefault Enables GateD to accept the default route (0.0.0.0) if it is included in a received EGP update. If not specified, the default route contained in an EGP update is ignored. For efficiency, some networks have external routers announce a default route to avoid sending large EGP update packets.

exportdefault Enables GateD to include the default route (0.0.0.0) in EGP updates sent to this EGP neighbor. This allows the system to advertise the default route via EGP. Normally a default route is not included in EGP updates.

gateway *gateway* If a network is not shared with a neighbor, **gateway** specifies a router on an attached network to be used as the next hop router for routes received from this neighbor. This option is only rarely used.

lcladdr *local_address* Specifies the address to be used on the local end of the connection with the neighbor. The local address must be on an interface which is shared with the neighbor or with the neighbor's *gateway* when the **gateway** parameter is used. A session will only be opened when an interface with the appropriate local address (through which the neighbor or gateway address is directly reachable) is operating.

sourcenet *network* Specifies the network queried in the EGP Poll packets. By default this is the network shared with the neighbor's address specified. If there is no network shared with the neighbor, one of the networks the neighbor is attached to should be specified. This parameter can also be used to specify a network shared with the neighbor other than the one on which the EGP packets are sent. This parameter is normally not needed.

p1 *time*

minhello *time* Sets the minimum acceptable interval between the transmission of EGP *HELLO* packets. The default hello interval is 30 seconds. If the neighbor fails to respond to three hello packets, GateD stops trying to acquire the neighbor. Setting a larger interval gives the neighbor a better chance to respond. Minhello is an alias for the *P1* value defined in the EGP specification.

p2 *time*

minpoll *time* Sets the time interval between polls to the neighbor. The default is 120 seconds. If three polls are sent without a response, the neighbor is declared "down" and all routes learned from that neighbor are removed from the routing database. A longer polling interval supports a more stable routing database but is not as responsive to routing changes. Minpoll is an alias for the *P2* value defined in the EGP specification.

ttl *ttl* By default, GateD sets the IP TTL for local neighbors to *one* and the TTL for non-local neighbors to 255. This option is provided when attempting to communicate with improperly functioning routers that ignore packets sent with a TTL of one.

traceoptions *trace_options* Specifies the tracing options for this EGP neighbor. By default these are inherited from group or EGP global trace options. Trace Statements and the EGP specific tracing options below.)

4.17.3 Tracing options

The **state** and **policy** options work with EGP. Packet tracing options (which may be modified with **detail**, **send** and **recv**):

packets All EGP packets

hello EGP *HELLO/I-HEARD-U* packets which are used to determine neighbor reachability.

acquire EGP *ACQUIRE/CEASE* packets which are used to initiate and terminate EGP sessions.

update EGP *POLL/UPDATE* packets which are used to request and receive reachability updates.

4.18 BGP

4.18.1 The BGP Protocol

The Border Gateway Protocol (BGP) is an exterior routing protocol used for exchanging routing information between autonomous systems. BGP is used for exchange of routing information between multiple transit autonomous systems as well as between transit and stub autonomous systems. BGP is related to EGP but operates with more capability, greater flexibility, and less required bandwidth. BGP uses *path attributes* to provide more information about each route, and in particular maintain an *AS path*, which includes the AS number of each autonomous system the route has transited, providing information sufficient to prevent routing loops in an arbitrary topology. Path attributes may also be used to distinguish between groups of routes to determine administrative preferences, allowing greater flexibility in determining route preference to achieve a variety of administrative ends.

BGP supports two basic types of sessions between neighbours, internal (sometimes refered to as IBGP) and external. Internal sessions are run between routers in the same autonomous system, while external sessions run between routers in different autonomous systems. When sending routes to an external peer the local AS number is prepended to the AS path, hence routes received from an external peer are guaranteed to have the AS number of that peer at the start of the path. Routes received from an internal neighbour will not in general have the local AS number prepended to the AS path, and hence will in general have the same AS path that the route had when the originating internal neighbour received the route from an external peer. Routes with no AS numbers in the path may be legitimately received from internal neighbours; these indicate that the received route should be considered internal to your own AS.

The BGP implementation supports three versions of the BGP protocol, versions 2, 3 and 4. BGP versions 2 and 3 are quite similar in capability and function. They will only propagate classed network routes, and the AS path is a simple array of AS numbers. BGP 4 will propagate fully general address-and-mask routes, and the AS path has some structure to represent the results of aggregating dissimilar routes.

External BGP sessions may or may not include a single metric, which BGP calls the *Multi-Exit Discriminator*, in the path attributes. For BGP versions 2 and 3 this metric is a 16-bit unsigned integer, for BGP version 4 it is a 32-bit unsigned integer. In either case smaller values of the metric are to be preferred. Currently this metric is only used to break ties between routes with equal preference from the same neighbour AS. Internal BGP sessions carry at least one metric in the path attributes, which BGP calls the *LocalPref*. The size of the metric is identical to the MED. For BGP versions 2 and 3 this metric is considered better when its value is smaller, for version 4 it is better when it is larger. BGP version 4 sessions may optionally carry a second metric on internal sessions, this being an internal version of the *Multi-Exit Discriminator*. The use of these metrics is dependent on the type of internal protocol processing which is specified.

BGP collapses routes with similar path attributes into a single update for advertisement. Routes that are received in a single update will be readvertised in a single update. The churn caused by the loss of a neighbor will be minimized and the initial advertisement sent during peer establishment will be maximally compressed. BGP does not read information from the kernel message-by-message, but fills the input buffer. It processes all complete messages in the buffer before reading again. BGP also does multiple reads to clear all incoming data queued on the socket. This feature may cause other protocols to be blocked for prolonged intervals by a busy peer connection.

All unreachable messages are collected into a single message and sent prior to reachable routes during a flash update. For these unreachable announcements, the next hop is set to the local address on the connection, no metric is sent and the path origin is set to incomplete. On external connections the AS path in unreachable announcements is set to the local AS, on internal connections the AS path is set to zero length.

The BGP implementation expects external peers to be directly attached to a shared subnet, and expects those peers to advertise next hops which are host addresses on that subnet (though this constraint can be relaxed by configuration for testing). For groups of internal peers, however, there are several alternatives which may be selected from by specifying the group type. Type *internal* groups expect all peers to be directly attached to a shared subnet so that, like external peers, the next hops received in BGP advertisements may be used directly for forwarding. Type *routing* groups instead will determine the immediate next hops for routes by using the next hop received with a route from a peer as a forwarding address, and use this to look up an immediate next hop in an IGP's routes. Such groups support distant peers, but need to be informed of the IGP whose routes they are using to determine immediate next hops. Finally, type *igp* groups expect routes from the group peers to not be used for forwarding at all. Instead they expect that copies of the BGP routes received will also be received via an IGP, and that the BGP routes will only be used to determine the path attributes associated with the IGP routes. Such groups also support distant peers, and also need to be informed of the IGP they are running with.

For internal BGP group types (and for test groups), where possible a single outgoing message is built for all group peers based on the common policy. A copy of the message is sent to every peer in the group, with possible adjustments to the next hop field as appropriate to each peer. This minimizes the computational load of running large numbers of peers in these types of groups. BGP allows unconfigured peers to connect if an appropriate group has been configured with an *allow* clause.

4.18.2 The BGP Statement

```
bgp yes | no | on | off
[ {
        preference preference ;
        defaultmetric metric ;
        traceoptions trace_options ;
        group type ( external peeras autonomous_system )
                | ( internal peeras autonomous_system )
                | ( igp peeras autonomous_system proto proto )
                | ( routing peeras autonomous_system proto proto
                        interface interface_list )
                | ( test peeras autonomous_system )
        {
                allow {
                        network
                        network mask mask
                        network masklen number
                        all
                        host host
                } ;
                peer host
                        [ metricout metric ]
                        [ localas autonomous_system ]
                        [ nogendefault ]
                        [ gateway gateway ]
                        [ preference preference ]
                        [ preference2 preference ]
                        [ lcladdr local_address ]
                        [ holdtime time ]
                        [ version number ]
                        [ passive ]
                        [ sendbuffer number ]
                        [ recvbuffer number ]
                        [ indelay time ]
                        [ outdelay time ]
                        [ keep [ all | none ] ]
                        [ analretentive ]
                        [ noauthcheck ]
                        [ noaggregatorid ]
                        [ keepalivesalways ]
                        [ v3asloopokay ]
                        [ nov4asloop ]
                        [ logupdown ]
                        [ ttl ttl ]
                        [ traceoptions trace_options ]
                        ;
                } ;
        } ] ;
external | internal | igp | test
```

The **bgp** statement enables or disables BGP. By default BGP is disabled. The default metric for announcing routes via BGP is not to send a metric.

preference *preference*

Sets the preference for routes learned from RIP. The default preference is 170. This preference may be overridden by a preference specified on the **group** or **peer** statments or by import policy.

defaultmetric *metric*

Defines the metric used when advertising routes via BGP. If not specified, no metric is propagated. This metric may be overridden by a metric specified on the neighbor or group statements or in export policy.

traceoptions *trace_options*

Specifies the tracing options for BGP. By default these are inherited from the global trace options. These values may be overridden on a group or neighbor basis. (See Trace Statements and the BGP specific tracing options below.)

4.18.3 Groups

BGP peers are grouped by type and the autonomous system of the peers. Any number of groups may be specified, but each must have a unique combination of type and peer autonomous system. There are four possible group types:

group type external peeras *autonomous_system*

In the classic external BGP group, full policy checking is applied to all incoming and outgoing advertisements. The external neighbors must be directly reachable through one of the machine's local interfaces. By default no metric is included in external advertisements, and the next hop is computed with respect to the shared interface.

group type internal peeras *autonomous_system*

An internal group operating where there is no IP-level IGP, for example an SMDS network or MILNET. All neighbors in this group are required to be directly reachable via a single interface. All next hop information is computed with respect to this interface. Import and export policy may be applied to group advertisements. Routes received from external BGP or EGP neighbors are by default readvertised with the received metric.

group type igp peeras *autonomous_system* **proto** *proto*

An internal group that runs in association with an interior protocol. The IGP group examines routes which the IGP is exporting and sends an advertisement only if the path attributes could not be entirely represented in the IGP tag mechanism. Only the AS path, path origin, and transitive optional attributes are sent with routes. No metric is sent, and the next hop is set to the local address used by the connection. Received internal BGP routes are not used or readvertised. Instead, the AS path information is attached to the corresponding IGP route and the latter is used for readvertisement. Since internal IGP peers are sent only a subset of the routes which the IGP is exporting, the export policy used is the IGP's. There is no need to implement the "don't routes from peers in the same group" constraint since the advertised routes are routes that IGP already exports.

group type routing peeras *autonomous_system* **proto** *proto* **interface** *interface_list*

An internal group which uses the routes of an interior protocol to resolve forwarding addresses. A type routing group propagates external routes between routers which are not directly connected, and computes immediate next hops for these routes by using the BGP next hop which arrived with the route as a forwarding address to be resolved via an internal protocol's routing information. In essence, internal BGP is used to carry AS external routes, while the IGP is expected to only carry AS internal routes, and the latter is used to find immediate next hops for the former.

The **proto** names the interior protocol to be used to resolve BGP route next hops, and may be the name of any IGP in the configuration. By default the next hop in BGP routes advertised to type routing peers will be set to the local address on the BGP connection to those peers, as it is assumed a route to this address will be propagated via the IGP. The *interface_list* can optionally provide a list interfaces whose routes are carried via the IGP for which third party next hops may be used instead.

group type test peeras *autonomous_system*

An extension to external BGP which implements a fixed policy using test peers. Fixed policy and *special case* code make test peers relatively inexpensive to maintain. Test peers do not need to be on a directly attached network. If GateD and the peer are on the same (directly attached) subnet, the advertised next hop is computed with respect to that network, otherwise the next hop is the local machine's current next hop. All routing information advertised by and received from a test peer is discarded, and all BGP advertiseable routes are sent back to the test peer. Metrics from EGP- and BGP-derived routes are forwarded in the advertisement, otherwise no metric is included.

Group parameters

The BGP statement has **group** clauses and **peer** subclauses. Any number of peer subclauses may be specified within a group. A group clause usually defines default parameters for a group of peers, these parameters apply to all subsidiary peer subclauses. Any parameters from the peer subclause may be specified on the group clause to provide defaults for the whole group (which may be overridden for individual peers).

4.18.4 Specifying peers

Within a group, BGP peers may be configured in one of two ways. They may be explicitly configured with a **peer** statement, or implicitly configured with the **allow** statement. Both are described here:

allow The allow clauses allows for *peer* connections from any addresses in the specified range of network and mask pairs. All parameters for these peers must be configured on the group clause. The internal peer structures are created when an incomming open request is received and destroyed when the connection is broken. For more detail on specifying the network/mask pairs, see the section on Route Filtering.

peer *host* A **peer** clause configures an individual peer. Each peer inherits all parameters specified on a group as defaults. Those default may be overridden by parameters explicitly specified on the peer subclaus.

Within each **group** clause, individual peers can be specified or a group of *potential* peers can be specified using **allow**. **Allow** is used to specify a set of address masks. If GateD receives a BGP connection request from any address in the set specified, it will accept it and set up a peer relationship.

Peer parameters

The BGP **peer** subclause allows the following parameters, which can also be specified on the **group** clause. All are optional.

metricout *metric* If specified, this metric is used as the primary metric on all routes sent to the specified peer(s). This metric overrides the default metric, a metric specified on the group and any metric specified by export policy.

localas *autonomous_system* Identifies the autonomous system which GateD is representing to this group of peers.. The default is that which has been set globally in the **autonomoussystem** statement.

nogendefault Prevents GateD from generating a default route when EGP receives a valid update from its neighbor. The default route is only generated when the **gendefault** option is enabled.

gateway *gateway* If a network is not shared with a peer, *gateway* specifies a router on an attached network to be used as the next hop router for routes received from this neighbor. This parameter is not needed in most cases.

preference *preference* Specifies the preference used for routes learned from these peers. This can differ from the default BGP preference set in the **bgp** statement, so that GateD can prefer routes from one peer, or group of peer, over othes. This preference may be explicitly overriden by import policy.

preference2 *preference* In the case of a *preference* tie, the second preference, *preference2* may be used to break the tie. The default value is 0.

lcladdr *local_address* Specifies the address to be used on the local end of the TCP connection with the peer. For *external* peers the local address must be on an interface which is shared with the peer or with the peer's *gateway* when the **gateway** parameter is used. A session with an external peer will only be opened when an interface with the appropriate local address (through which the peer or gateway address is directly reachable) is operating. For other types of peers, a peer session will be maintained when any interface with the specified local address is operating. In either case incoming connections will only be recognized as matching a configured peer if they are addressed to the configured local address.

holdtime *time* Specifies the BGP holdtime value to use when negotiating the connection with this peer, in seconds. According to BGP, if GateD does not receive a keepalive, update, or notification message within the period specified in the Hold Time field of the BGP Open message, then the BGP connection will be closed. The value must be either 0 (no keepalives will be sent) or at least 3.

version *version* Specifies the version of the BGP protocol to use with this peer. If not specified, the highest supported version is used first and version negotiation is attempted. If it is specified, only the specified version will be offered during negotiation. Currently supported version are 2, 3 and 4.

passive Specifies that active OPENs to this peer should not be attempted. GateD should wait for the peer to issue an open. By default all explicitly configured peers are active, they periodically send OPEN messages until the peer responds.

sendbuffer *buffer_size*

recvbuffer *buffer_size* Control the amount of send and receive buffering asked of the kernel. The maximum supported is 65535 bytes although many kernels have a lower limit. By default, GateD configures the maximum supported. These parameters are not needed on normally functioning systems.

indelay *time*

outdelay *time* Used to dampen route fluctuations. **Indelay** is the amount of time a route learned from a BGP peer must be stable before it is accepted into the gated routing database. **Outdelay** is the amount of time a route must be present in the gated routing database before it is exported to BGP. The default value for each is 0, meaning that these features are disabled.

keep all Used to retain routes learned from a peer even if the routes' AS paths contain one of our exported AS numbers.

analretentive Causes GateD to issue warning messages when receiving questionable BGP updates such as duplicate routes and/or deletions of non-existing routes. Normally these events are silently ignored.

noauthcheck Normally GateD verifies that incoming packets have an authentication field of all ones. This option may be used to allow communication with an implementation that uses some other form of authentication.

noaggregatorid Causes GateD to specify the routerid in the aggregator attribute as zero (instead of its routerid) in order to prevent different routers in an AS from creating aggregate routes with different AS paths.

keepalivesalways Causes gated to always send keepalives, even when an update could have correctly substituted for one. This allows interoperability with routers that do not completely obey the protocol specifications on this point.

v3asloopokay By default gated will not advertise routes whose AS path is looped (i.e. with an AS appearing more than once in the path) to version 3 external peers. Setting this flag removes this constraint. Ignored when set on internal groups or peers.

nov4asloop Prevents routes with looped AS paths from being advertised to version 4 external peers. This can be useful to avoid advertising such routes to peer which would incorrectly forward the routes on to version 3 neighbours.

logupdown Causes a message to be logged via the syslog mechanism whenever a BGP peer enters or leaves the *ESTABLISHED* state.

ttl *ttl* By default, GateD sets the IP TTL for local peers to **one** and the TTL for non-local peers to 255. This option mainly is provided when attempting to communicate with improperly functioning routers that ignore packets sent with a TTL of one. Not all kernels allow the TTL to be specified for TCP connections.

traceoptions *trace_options* Specifies the tracing options for this BGP neighbor. By default these are inherited from group or BGP global trace options.

4.18.5 Tracing options

Note that the **state** option works with BGP, but does not provide true state transition information.

Packet tracing options (which may be modified with **detail**, **send** and **recv**):

packets All BGP packets

open BGP *OPEN* packets which are used to establish a peer relationship.

update BGP *UPDATE* packets which are used to pass network reachability information.

keepalive BGP *KEEPALIVE* packets which are used to verify peer reachability.

4.19 ICMP

On systems without the BSD routing socket, gated listens to ICMP messages received by the system. Currently gated only does processing on ICMP redirect packets, but more functionality may be added in the future, such as support for the router discovery messages. Processing of ICMP redirect messages is handled by the **redirect** statement.

Currently the only reason to specify the **icmp** statement is to be able to trace the ICMP messages that gated receives.

4.19.1 The ICMP statement

icmp {
    traceoptions *trace_options* **;**
}

traceoptions *trace_options* **;**

Specifies the tracing options for ICMP. (See Trace Statements and the ICMP specific tracing options below.)

4.19.2 Tracing options

Packet tracing options (which may be modified with **detail** and **recv**):

packets All ICMP packets received.

redirect Only ICMP *REDIRECT* packets received.

routerdiscovery Only ICMP *ROUTER DISCOVERY* packets received.

info Only ICMP informational packets, which include mask request/response, info request/response, echo request/response and time stamp request/response.

error Only ICMP error packets, which include time exceeded, parameter problem, unreachable and source quench.

4.20 Redirect

4.20.1 Redirect Processing

The *redirect* code processes ICMP or ISO redirects learned by monitoring ICMP messages, or via the routing socket on systems that support it. It processes the redirect request and decides whether to accept the redirect. If the redirect is accepted, the route is installed in the gated routing table with the protocol *redirect*. Redirects are deleted from the routing table after 3 minutes.

If GateD determines that a redirect is not acceptable, it tries to figure out if the kernel forwarding table has been modified. On systems where ICMP messages are monitored this is accomplished by trying to second guess what the kernel would have done with the redirect. On systems with the routing socket, the kernel provides an indication of whether the redirect was accepted; GateD ignores redirects that were not processed.

If GateD has determined that the state of the kernel forwarding table has been changed, the necessary requests to the kernel are made to restore the correct state.

Note that on currently available systems it is not possible to disable the processing of ICMP redirects, even when the system is functioning as a router. To ignore the effects of redirects, GateD must process each one and actively restore any changes it made to the kernel's state. Because of the mechanism's involved there will be windows where the effects of redirects are present in the kernel.

By default, GateD ignores redirects when actively participating in an interior gateway protocol (RIP, HELLO, OSPF or IS-IS). It is not possible to enable redirects once they have been automatically disabled. Listening to RIP or HELLO in *nobroadcast* mode does not cause redirects to be ignored, nor does the use of EGP and BGP. Redirects must be manually configured off in these cases.

Note that in accordance with the latest IETF Router Requirements document, GateD insures that all ICMP net redirects are processed as host redirects. When an ICMP net redirect is accepted, GateD issues the requests to the kernel to make sure that the kernel forwarding table is updated to reflect a host redirect instead of a net redirect.

The redirect statement does not prevent the system from sending redirects, only from listening to them.

4.20.2 The Redirect Statement

redirect yes | **no** | **on** | **off**
[**{**
  preference *preference* **;**
  interface *interface_list*
    [**noredirects**] | [**redirects**] **;**
  trustedgateways *gateway_list* **;**
  traceoptions *trace_options* **;**
}] ;

preference

Sets the preference for a route learned from a redirect. The default is 30.

interface *interface_list*

The **interface** statement allows the enabling and disabling of redirects on an interface-by-interface basis. See the section on interface list specification for the description of the *interface_list*. The possible parameters are:

noredirects Specifies that redirects received via the specified interface will be ignored. The default is to accept redirects on all interfaces.

redirects This is the default. This argument may be necessary when **noredirects** is used on a wildcard interface descriptor.

trustedgateways *gateway_list*

Defines the list of gateways from which redirects will be accepted. The *gateway_list* is simply a list of host names or addresses. By default, all routers on the shared network(s) are trusted to supply redirects. But if the **trustedgateways** clause is specified only redirects from the gateways in the list are accepted.

traceoptions *trace_options*

There are no redirect-specific tracing options. All non-error messages are traced under the *normal* class.

4.20.3 Tracing options

There are no *Redirect*-specific tracing options. All non-error messages are traced under the *normal* class.

4.21 Router Discovery

4.21.1 The Router Discovery Protocol

The Router Discovery Protocol is an IETF standard protocol used to inform hosts of the existence of routers. It is intended to be used instead of having hosts *wiretap* routing protocols such as RIP. It is used in place of, or in addition to, statically configured default routes in hosts.

The protocol is split into two portions, the *server* portion which runs on routers, and the *client* portion that runs on hosts. GateD treats these much like two separate protocols, only one of which may be enabled at a time.

4.21.2 The Router Discovery Server

The Router Discovery Server runs on routers and announces their existence to hosts. It does this by periodically multicasting or broadcasting a **Router Advertisement** to each interface on which it is enabled. These Router Advertisements contain a list of all the router's addresses on a given interface and their preference for use as a default router.

Initially these Router Advertisements occur every few seconds, then fall back to every few minutes. In addition, a host may send a **Router Solicitation** to which the router will respond with a unicast Router Advertisement (unless a multicast or broadcast advertisement is due momentarily).

Each Router Advertisement contains a *Advertisement Lifetime* field indicating for how long the advertised addresses are valid. This lifetime is configured such that another Router Advertisement will be sent before the lifetime has expired. A lifetime of zero is used to indicate that one or more addresses are no longer valid.

On systems supporting IP multicasting, the Router Advertisements are by default sent to the all-hosts multicast address *224.0.0.1*. However, the use of broadcast may be specified. When Router Advertisements are being sent to the all-hosts multicast address, or an interface is configured for the limited-broadcast address *255.255.255.255*, all IP addresses configured on the physical interface are included in the Router Advertisement. When the Router advertisements are being sent to a net or subnet broadcast, only the address associated with that net or subnet is included.

4.21.3 The Router Discovery Server Statement

routerdiscovery server yes | **no** | **on** | **off** [**{**
    traceoptions *trace_options* **;**
    interface *interface_list*
        [**minadvinterval** *time*]
        [**maxadvinterval** *time*]
        [**lifetime** *time*]
        ;
    address *interface_list*
        [**advertise**] | [**ignore**]
        [**broadcast**] | [**multicast**]
        [**ineligible**] | [**preference** *preference*]
        ;
}] ;

traceoptions *trace_options*

Specifies the Router Discovery tracing options. (See Trace Statements and the Router Discovery specific tracing options below.)

interface *interface_list*

Specifies the parameters that apply to physical interfaces. Note a slight difference in convention from the rest of GateD, **interface** specifies just physical interfaces (such as *le0*, *ef0* and *en1*), while **address** specifies protocol (in this case IP) addresses.

Interface parameters are:

maxadvinterval *time* The maximum time allowed between sending broadcast or multicast Router Advertisements from the interface. Must be no less than *4* and no more than *30:00* (30 minutes or 1800 seconds). The default is **10:00** (10 minutes or 600 seconds).

minadvinterval *time* The minimum time allowed between sending unsolicited broadcast or multicast Router Advertisements from the interface. Must be no less than *3* seconds and no greater than **maxadvinterval**. The default is **0.75 * maxadvinterval**.

lifetime *time* The lifetime of addresses in a Router Advertisement. Must be no less than **maxadvinterval** and no greater than *2:30:00* (two hours, thirty minutes or 9000 seconds). The default is **3 * maxadvinterval**.

address *interface_list*

Specifies the parameters that apply to the specified set of addresses on this physical interfaces. Note a slight difference in convention from the rest of GateD, **interface** specifies just physical interfaces (such as *le0*, *ef0* and *en1*), while **address** specifies protocol (in this case IP) addresses.

advertise Specifies that the specified address(es) should be included in Router Advertisements. This is the default.

ignore Specifies that the specified address(es) should not be included in Router Advertisements.

broadcast Specifies that the given address(es) should be included in a broadcast Router Advertisement because this system does not support IP multicasting, or some hosts on attached network do not support IP multicasting. It is possible to mix addresses on a physical interface such that some are included in a broadcast Router Advertisement and some are included in a multicast Router Advertisement. This is the default if the router does not support IP multicasting.

multicast Specifies that the given address(es) should only be included in a multicast Router Advertisement. If the system does not support IP multicasting the address(es) will not be included. If the system supports IP multicasting, the default is to include the address(es) in a multicast Router Advertisement if the given interface supports IP multicasting, if not the address(es) will be included in a broadcast Router Advertisement.

preference *preference* The preferability of the address(es) as a default router address, relative to other router addresses on the same subnet. A 32-bit, signed, twos-complement integer, with higher values meaning more preferable. Note that *hex 80000000* may only be specified as **ineligible**. The default is **0**.

ineligible Specifies that the given address(es) will be assigned a preference of (hex 80000000) which means that it is not eligible to be the default route for any hosts.

This is useful when the address(es) should not be used as a default route, but are given as the next hop in an ICMP redirect. This allows the hosts to verify that the given addresses are up and available.

4.21.4 The Router Discovery Client

A host listens for Router Advertisements via the all-hosts multicast address (*224.0.0.1*) if IP multicasting is available and enabled, or on the interface's broadcast address. When starting up, or when reconfigured, a host may send a few Router Solicitations to the all-routers multicast address, *224.0.0.1*, or the interface's broadcast address.

When a Router Advertisement with non-zero lifetime is received, the host installs a default route to each of the advertised addresses. If the preference **ineligible**, or the address is not on an attached interface, the route is marked unusable but retained. If the preference is usable, the metric is set as a function of the preference such that the route with the best preference is used. If more than one address with the same preference is received, the one with the lowest IP address will be used. These default routes are not exportable to other protocols.

When a Router Advertisement with a zero lifetime is received, the host deletes all routes with next-hop addresses learned from that router. In addition, any router learned from ICMP redirects pointing to these addresses will be deleted. The same will happen when a Router Advertisement is not received to refresh these routes before the lifetime expires.

4.21.5 The Router Discovery Client Statement

routerdiscovery client yes | **no** | **on** | **off** [**{**
    traceoptions *trace_options* **;**
    preference *preference* **;**
    interface *interface_list*
        [**enable**] | [**disable**]
        [**broadcast**] | [**multicast**]
        [**quiet**] | [**solicit**]
        ;
}] ;

traceoptions *trace_options*

Specifies the tracing options for the *Router Discovery* protocol. (See the trace statements and the *Router Discovery* protocol specific tracing options below.)

preference *preference* **;**

Specifies the preference of all Router Discovery default routes. The default is **55**.

interface *interface_list*

Specifies the parameters that apply to physical interfaces. Note a slight difference in convention from the rest of GateD, **interface** specifies just physical interfaces (such as *le0*, *ef0* and *en1*). The Router Discovery Client has no parameters that apply only to interface addresses.

enable Specifies that Router Discovery should be performed on the specified interface(s). This is the default.

disable Specifies that Router Discovery should not be performed on the specified interface(s).

broadcast Specifies that Router Solicitations should be broadcast on the specified interface(s). This is the default if IP multicast support is not available on this host or interface.

multicast Specifies that Router Solicitations should be multicast on the specified interface(s). If IP multicast is not available on this host and interface, no solicitation will be performed. The default is to multicast Router Solicitations if the host and interface support it, otherwise Router Solicitations are broadcast.

quiet Specifies that no Router Solicitations will be sent on this interface, even though Router Discovery will be performed.

solicit Specifies that initial Router Solicitations will be sent on this interface. This is the default.

4.21.6 Tracing options

The Router Discovery Client and Server support the **state** trace flag which traces various protocol occurrences.

state State transitions

The Router Discovery Client and Server do not directly support any packet tracing options, tracing of router discovery packets is enabled via the ICMP Statement.

4.22 SNMP Statement

The Simple Network Management Protocol (SNMP) is a not a routing protocol but a network management protocol. The **snmp** statement controls whether GateD tries to contact the SNMP Multiplexing daemon to register supported variables. The SNMP daemon (usually *smuxd*) must be run independently. The **snmp** statement only controls whether GateD keeps the management software apprised of its status.

GateD communicates with the SNMP daemon via the SMUX protocol that is described in RFC 1227.

4.22.1 The SNMP statement

snmp yes | **no** | **on** | **off**
[{
    port *port* ;
    debug ;
    traceoptions *traceoptions* ;
}] ;

Reporting is enabled by specifying **yes** or **on** and disabled with **no** or **off**. The default is **on**.

port *port* By default, the SMUX daemon listens for requests on port 199. GateD can be configured to try to contact the SMUX daemon on a different port by explicitly specifying the port.

debug Specifying this option enables debugging of the ISODE SMUX code. The default is debugging disabled.

traceoptions *trace_options* Specifies the tracing options for SMUX. (See Trace Statements and the SMUX specific tracing options below.)

4.22.2 Tracing options

There are no SNMP-specific trace options. SNMP requests received via the SMUX protocol from the SNMP daemon are not handled quite like packets. The **detail**, **send**, and **recv** options are not supported.

receive SNMP requests received from the SMUX daemon and the associated responses.

register Protocol requests to register variables.

resolve Protocol requests to resolve variable names.

trap SNMP trap requests from protocols.

4.23 The Kernel Statement

While the kernel interface isn't technically a routing protocol, it has many characteristics of one, and GateD handles it similarly to one. The routes GateD chooses to install in the kernel forwarding table are those that will actually be used by the kernel to forward packets.

The add, delete and change operations GateD must use to update the typical kernel forwarding table take a non-trivial amount of time. This does not present a problem for older routing protocols (RIP, EGP), which are not particularly time critical and do not easily handle very large numbers of routes anyway. The newer routing protocols (OSPF, BGP) have stricter timing requirements and are often used to process many more routes. The speed of the kernel interface becomes critical when these protocols are used.

To prevent GateD from locking up for significant periods of time installing large numbers of routes (up to a minute or more has been observed on real networks), the processing of these routes is now done in batches. The size of these batches may be controlled by the tuning parameters described below, but normally the default parameters will provide the proper functionality.

During normal shutdown processing, GateD normally deletes all the routes it has installed in the kernel forwarding table, except for those marked with **retain**. Optionally, GateD can leave all routes in the kernel forwarding table by not deleting any routes. In this case changes will be made to insure that routes with a **retain** indication are installed in the table. This is useful on systems with large numbers of routes as it prevents the need to re-install the routes when GateD restarts. This can greatly reduce the time it takes to recover from a restart.

4.23.1 Forwarding tables and Routing tables

The table in the kernel that controls the forwarding of packets is a *forwarding table*, also know in ISO speak as a *forwarding information base*, or *FIB*. The table that GateD uses internally to store routing information it learns from routing protocols is a *routing table*, known in ISO speak as a *routing information base*, or *RIB*. The routing table is used to collect and store routes from various protocols. For each unique combination of *network* and *mask* an *active* route is chosen, this route will be the one with the best (numerically smallest) *preference*. All the active routes are installed in the kernel forwarding table. The entries in this table are what the kernel actually uses to forward packets.

4.23.2 Updating the Forwarding Table

There are two main methods of updating the kernel *FIB*, the *ioctl()* interface and the *routing socket* interface. Their various characteristics are described here.

Updating the Forwarding Table with the ioctl interface

The *ioctl()* interface to the forwarding table was introduced in *BSD 4.3* and widely distributed in *BSD 4.3*. This is a one-way interface, it only allows GateD to update the kernel forwarding table. It has several other limitations:

Fixed subnet masks The BSD 4.3 networking code assumed that all subnets of a given network had the same subnet mask. This limitation is enforced by the kernel. The network mask is not stored in the kernel forwarding table, but determined when a packet is forwarded by searching for interfaces on the same network.

One way interface GateD is able to update the kernel forwarding table, but it is not aware of other modifications of the forwarding table. GateD is able to listen to ICMP messages and guess how the kernel has updated the forwarding table in response to ICMP redirects.

Blind updates GateD is not able to detect changes to the forwarding table resulting from the use of the the *route* command by the system administrator. Use of the *route* command on systems that use the *ioctl()* interface is strongly discouraged while GateD is running.

Changes not supported In all known implementations, there is no change operation supported, to change a route that exists in the kernel, the route must be deleted and a new one added.

Updating the Forwarding Table with the routing socket interface

The *routing socket* interface to the kernel forwarding table was introduced in *BSD 4.3 Reno*, widely distributed in *BSD 4.3 Net/2* and improved in *BSD 4.4*. This interface is simply a socket, similar to a UDP socket, on which the kernel and GateD exchange messages. It has several advantages over the *ioctl()* interface:

Variable subnet masks The network mask is passed to the kernel explicitly. This allows different masks to be used on subnets of the same network. It also allows routes with masks that are more general than the natural mask to be used. This is known as classless routing.

Two way interface Not only is GateD able to change the kernel forwarding table with this interface, but the kernel can also report changes to the forwarding table to GateD. The most interesting of these is an indication that a redirect has modified the kernel forwarding table; this means that gated no longer needs to monitor ICMP messages to learn about redirects. Plus, there is an indication of whether the kernel processed the redirect, GateD can safely ignore redirect messages that the kernel did not process.

Updates visible Changes to the routing table by other processes, including the *route(8)* command are received via the routing socket. This allows GateD to insure that the kernel forwarding table is in sync with the routing table. Plus it allows the system administrator the ability to do some operations with the *route(8)* command while gated is running.

Changes supported There is a functioning *change* message that allows routes in the kernel to be atomically changed. Some early versions of the routing socket code had bugs in the change message processing. There are compilation time and configuration time options that cause delete and add sequences to be used in lieu of change messages.

Expandable New levels of kernel/GateD communications may be added by adding new message types.

4.23.3 Reading the Forwarding Table

When GateD starts up it reads the kernel forwarding table and installs corresponding routes in the routing table. These routes are called *remnants* and are timed out after a configured interval (which defaults to 3 minutes), or as soon as a more attractive route is learned. This allows forwarding to occur during the time it takes the routing protocols to start learning routes.

There are three main methods for reading the forwarding table from the kernel.

Reading forwarding table via kmem

On many systems, especially those based on BSD 4.3, GateD must have knowledge of the kernel's data structures and poke around in the kernel to read the current state of forwarding table. This method is slow and subject to error if the kernel forwarding table is updated while GateD is in the middle of reading it. This can happen if the system administrator uses the *route* command, or an ICMP redirect message is received while GateD is starting up.

Due to an oversight some systems, such as OSF/1, which are based on BSD 4.3 Reno or later, do not have the *getkerninfo()* system call described below which allows GateD to read routes from the kernel without know about kernel internal structures. On these systems it is necessary to read the kernel radix tree from the kernel by poking around in kernel memory. This is even more error prone than reading the hash based forwarding table.

Reading the forwarding table via getkerninfo/sysctl

Besides the routing socket, BSD 4.3 Reno introduced the *getkerninfo()* system call. This call allows a user process (of which GateD is one) to read various information from the kernel without knowledge of the kernel data structures. In the case of the forwarding table, it is returned to gated atomically as a series of routing socket messages. This prevents the problem associated with the forwarding table changing while GateD is in the process of reading it.

BSD 4.4 changed the *getkerninfo()* interface into the *sysctl()* interface, which takes different parameters, but otherwise functions identically.

Reading the forwarding table via OS specific methods

Some operating systems, for example SunOS 5, define their own method of reading the kernel forwarding table. The SunOS 5 version is similar in concept to the *getkerninfo()* method.

4.23.4 Reading the interface list

The kernel support subsystem of GateD is responsible for reading the status of the kernel's physical and protocol interfaces periodically. GateD detects changes in the interface list and notifies the protocols so they can start or stop instances or peers. The interface list is read one of two ways:

Reading the interface list with SIOCGIFCONF

On systems based on BSD 4.3, 4.3 Reno and 4.3 Net/2 the *SIOCGIFCONF ioctl* interface is used to read the kernel interface list. Using this method a list of interfaces and some basic information about them is return by the SIOCGIFCONF call. Other information must be learned by issuing other ioctls to learn the interface network mask, flags, MTU, metric, destination address (for point-to-point interfaces) and broadcast address (for broadcast capable interfaces).

GateD re-reads this list every 15 second looking for changes. When the routing socket is in use, it also re-reads it whenever a messages is received indicating a change in routing configuration. Receipt of a *SIGUSR2* signal also causes GateD to re-read the list. This interval may be explicitly configured in the interface configuration.

Reading the interface list with sysctl

BSD 4.4 added the ability to read the kernel interface list via the *sysctl* system call. The interface status is returned atomically as a list of routing socket messages which GateD parses for the required information.

BSD 4.4 also added routing socket messages to report interface status changes immediately. This allows GateD to react quickly to changes in interface configuration.

When this method is in use, GateD re-reads the interface list only once a minute. It also re-reads it on routing table changes indications and when a *SIGUSR2* is received. This interval may be explicitly configured in the interface configuration.

4.23.5 Reading interface physical addresses

Later version of the *getkerninfo()* and *sysctl()* interfaces return the interface physical addresses as part of the interface information. On most systems where this information is not returned, GateD scans the kernel physical interface list for this information for interfaces with IFF_BROADCAST set, assuming that their drivers are handled the same as Ethernet drivers. On some systems, such as *SunOS 4* and *SunOS 5*, system specific interfaces are used to learn this information.

The interface physical addresses are useful for IS-IS. For IP protocols, they are not currently used, but may be in the future.

4.23.6 Reading kernel variables

At startup, GateD reads some special variables out of the kernel. This is usually done with the *nlist()* (or *kvm_nlist()*) system call, but some systems use different methods.

The variables read include the status of *UDP checksum creation and generation*, *IP forwarding* and *kernel version* (for informational purposes). On systems where the routing table is read directly from kernel memory, the root of the *hash table* or *radix tree routing table* is read. On systems where interface physical addresses are not supplied by other means, the root of the *interface list* is read.

4.23.7 Special route flags

The later *BSD* based kernels support the special route flags described here.

RTF_REJECT Instead of forwarding a packet like a normal route, routes with *RTF_REJECT* cause packets to be dropped and *unreachable* messages to be sent to the packet originators. This flag is only valid on routes pointing at the *loopback* interface.

RTF_BLACKHOLE Like the *RTF_REJECT* flag, routes with *RTF_BLACKHOLE* cause packets to be dropped, but *unreachable* messages are not sent. This flag is only valid on routes pointing at the *loopback* interface.

RTF_STATIC When GateD starts, it reads all the routes currently in the kernel forwarding table. Besides interface routes, it usually marks everything else as a *remnant* from a previous run of GateD and deletes it after a few minutes. This means that routes added with the *route* command will not be retained after GateD has started.

To fix this the *RTF_STATIC* flag was added. When the *route* command is used to install a route that is not an interface route it sets the *RTF_STATIC* flag. This signals to GateD that said route was added by the systems administrator and should be retained.

4.23.8 Kernel Configuration

kernel {
        options
                [**nochange**]
                [**noflushatexit**]
                [**remnantholdtime** *time*]
                ;
        routes *number* **;**
        flash
                [**limit** *number*]
                [**type interface** | **interior** | **all**]
                ;
        background
                [**limit** *number*]
                [**priority flash** | **higher** | **lower**]
                ;
        traceoptions *trace_options* **;**
} ;

options *option_list*

Configure kernel options. The valid options are:

nochange On systems supporting the routing socket this insures that changes operations will not be performed, only deletes and adds. This is useful on early versions of the routing socket code where the change operation was broken.

noflushatexit During normal shutdown processing GateD deletes all routes from the kernel forwarding table that do not have a **retain** indication. The **noflushatexit** option prevents route deletions at shutdown. Instead, routes are changed and added to make sure that all the routes marked with **retain** get installed.

This is handy on systems with thousands of routes. Upon startup GateD will notice which routes are in the kernel forwarding table and not have add them back.

remnantholddimte *time* Normally *remnant* routes read from the kernel forwarding table at startup are timed out in three minutes or as soon as they are overridden. This option allows the interval to be configured to a value between zero and 15 minutes. Setting it to zero causes these routes to be deleted immediately.

routes *number*

On some systems kernel memory is at a premium. With this parameter a limit can be placed on the maximum number of routes GateD will install in the kernel. Normally gated adds/changes/deletes routes in interface/internal/external order, i.e. it queues interface routes first, followed by internal routes, followed by external routes, and processes the queue from the beginning. If a this parameter is specified and the limit is hit, GateD does two scans of the list instead. On the first scan it does deletes, and also deletes all changed routes, turning the queued changes into adds. It then rescans the list doing adds in interface/internal/external order until it hits the limit again. This will tend to favor internal routes over external routes. The default is not to limit the number of routes in the kernel forwarding table.

flash

When routes change, the process of notifying the protocols is called a *flash update*. The kernel forwarding table interface is the first to be notified. Normally a maximum of 20 interface routes may be processed during one flash update. The **flash** command allows tuning of these parameters.

limit *number* Specifies the maximum number of routes which may be processed during one flash update. The default is *20*. A value of *-1* will cause all pending route changes of the specified type to be processed during the flash update.

type interface | **interior** | **all** Specifies the type of routes that will be processed during a flash update. **interior** specifies that interior routes (See the definition of interior gateway protocols on section 4.3.2) will also be installed. **all** specifies the inclusion of exterior routes (See the definition of exterior gateway protocols on section 4.3.3) as well. The default is **interface** which specifies that only interface routes will be installed during a flash update.

Specifying **flash limit -1 all** causes all routes to be installed during the *flash update*; this mimics the behavior of previous versions of GateD.

background

Since only interface routes are normally installed during a flash update, the remaining routes are processed in batches in the background, that is, when no routing protocol traffic is being received. Normally, 120 routes are installed at a time to allow other tasks to be performed and this background processing is done at lower priority than flash updates. The following parameters allow tuning of these parameters:

limit *number* Specifies the number of routes which may be processed at during one batch. The default is 120.

priority flash | **higher** | **lower** Specifies the priority of the processing of batches of kernel updates in relationship to the *flash update* processing. The default is **lower** which means that flash updates are processed first. To process kernel updates at the same priority as flash updates, specify **flash**; to process them at a lower priority, use **lower**.

4.23.9 Tracing options

While the kernel interface isn't technically a routing protocol, in many cases it is handled as one. The following two symbols make sense when entered from the command line since the code that uses them is executed before the trace file is parsed.

symbols Symbols read from the kernel, by *nlist()* or similar interface.

iflist Interface list scan. This option is useful when entered from the command line as the first interface list scan is performed before the configuration file is parsed.

The following tracing options may only be specified in the configuration file. They are not valid from the command line.

remnants Routes read from the kernel when GateD starts.

request Requests by GateD to Add/Delete/Change routes in the kernel forwarding table.

The following general option and packet-tracing options only apply on systems that use the routing socket to exchange routing information with the kernel. They do not apply on systems that use the old BSD4.3 *ioctl()* interface to the kernel.

info Informational messages received from the routing socket, such as TCP lossage, routing lookup failure, and route resolution requests. GateD does not currently do processing on these messages, just logs the information if requested.

Packet tracing options (which may be modified with **detail**, **send** and **recv**):

routes Routes exchanged with the kernel, including Add/Delete/Change messages and Add/Delete/Change messages received from other processes.

redirect Redirect messages received from the kernel.

interface Interface status messages received from the kernel. These are only supported on systems with networking code derived from BSD 4.4.

other Other messages received from the kernel, including those mentioned in the **info** type above.

4.24 Static Statements

4.24.1 Static Statements

Static statements define the static routes used by GateD. A single **static** statement can specify any number of routes. The **static** statements occur after protocol statements and before control statements in the gated.conf file. Any number of **static** statements may be specified, each containing any number of static route definitions. These routes can be overridden by routes with better preference values.

static {
    (**host** *host*) | **default** |
    (*network* [(**mask** *mask*) | (**masklen** *number*)])
        gateway *gateway_list*
        [**interface** *interface_list*]
        [**preference** *preference*]
        [**retain**]
        [**reject**]
        [**blackhole**]
        [**noinstall**] **;**
    (*network* [(**mask** *mask*) | (**masklen** *number*)])
        interface *interface*
        [**preference** *preference*]
        [**retain**]
        [**reject**]
        [**blackhole**]
        [**noinstall**] **;**
} ;

host *host* **gateway** *gateway_list*

(*network* [(**mask** *mask*) | (**masklen** *number*)])

default gateway *gateway_list*

This is the most general form of the static statement. It defines a static route through one or more gateways. Static routes are installed when one or more of the *gateways* listed are available on directly attached interfaces. If more than one eligible gateways are available, they are limited by the number of multipath destinations supported (this compile time parameter is currently almost always one on Unix).

Parameters for static routes are:

interface *interface_list* When this parameter is specified, gateways are only considered valid when they are on one of these interfaces. See section 4.8.2 on interface list specification for the description of the *interface_list*.

preference *preference* This option selects the preference of this static route. The preference controls how this route competes with routes from other protocols. The default preference is 60.

retain Normally gated removes all routes except interface routes from the kernel forwarding table during a graceful shutdown. The **retain** option may be used to prevent specific static routes from being removed. This is useful to insure that some routing is available when gated is not running.

reject Instead of forwarding a packet like a normal route, *reject* routes cause packets to be dropped and *unreachable* messages to be sent to the packet originators. Specifying this option causes this route to be installed as a reject route. Not all kernel forwarding engines support reject routes.

blackhole A *blackhole* route is the same as a *reject* route except that *unreachable* messages are not supported.

noinstall Normally the route with the lowest preference is installed in the kernel forwarding table and is the route exported to other protocols. When **noinstall** is specified on a route, it will not be installed in the kernel forwarding table when it is active, but it will still be eligible to be exported to other protocols.

(*network* [(**mask** *mask*) | (**masklen** *number*)]) **interface** *interface*

This form defines a static interface route which is used for primitive support of multiple network addresses on one interface. The **preference**, **retain**, **reject**, **blackhole** and **noinstall** options are the same as described above.

4.25 Route filters and AS paths

4.25.1 Route Filtering

Route filters are a set of routes against which a given route will be matched. The resulting action of a match will depend on the use of the route filter. Routes filters are used in the **import** and **export** statements, as well as in the `martians` and other protocol definitions. This section is a generic description of the route filters and how routes are matched.

A route will match the most specific filter that applies. Specifying more than one filter with the same destination, mask and modifiers will generate an error.

4.25.2 Filtering syntax

A route filter is a list of one or more of the following entity:

network [**exact** | **refines**]
network **mask** *mask* [**exact** | **refines**]
network **masklen** *number* [**exact** | **refines**]
all
default
host *host*

These are all the possible formats for a route filter. Not all of these formats are available in all places, for instance the **host** and **default** formats are not valid for **martians**. In most cases it is possible to specify additional parameters relevant to the context of the filter. For example, on a **martian** statement it is possible to specify the **allow** keyword, on an **import** statement you can specify a preference, and on a **export** you can specify a metric.

network [**exact** | **refines**]

network **mask** *mask* [**exact** | **refines**]

network **masklen** *number* [**exact** | **refines**]

Matching usually requires both an address and a mask, although the mask is implied in the shorthand forms listed below. These three forms vary in how the mask is specified. In the first form, the mask is implied to be the natural mask of the network. In the second, the mask is explicitly specified. In the third, the mask is specified by the number of contiguous one bits.

If no additional parameters are specified, any destination that falls in the range given by the network and mask is matched, the mask of the destination is ignored. If a *natural* network is specified, the network, any subnets, and any hosts will be match. The two optional modifiers cause the mask of the destination to be considered also:

exact This parameter specifies that the mask of the destination must match the supplied mask *exactly*. This is used to match a network, but no subnets or hosts of that network.

refines Specifies that the mask of the destination must be more specified (i.e. longer) than the filter mask. This is used to match subnets and/or hosts of a network, but not the network.

all

This entry matches anything. It is equivalent to:

```
0.0.0.0 mask 0.0.0.0
```

default

Matches the **default** route. To match, the address must be the default address and the mask must be all zeros. This is equivalent to:

```
0.0.0.0 mask 0.0.0.0 exact
```

host *host*

Matches the specific host. To match, the address must exactly match the specified *host* and the network mask must be a host mask (i.e. all ones). This is equivalent to:

host **mask** 255.255.255 **exact**

4.25.3 Matching AS paths

An AS path is a list of autonomous_systems that routing information has passed through to get to this router, and an indicator of the origin of the AS path. This information can be used to prefer one path to a destination network over another. The primary method for doing this with GateD is to specify a list of patterns to be applied to AS paths when *importing* and *exporting* routes.

Each autonomous system that a route passed through prepends its AS number to the beginning of the AS path.

The origin information details the completeness of AS path information. An origin of **igp** indicates the route was learned from an interior routing protocol and is most likely complete. An origin of **egp** indicates the route was learned from an exterior routing protocol that does not support AS paths (EGP for example) and the path is most likely not complete. When the path information is definitely not complete, an origin of **incomplete** is used.

AS path regular expressions are defined in RFC 1164 section 4.2.

4.25.4 AS path matching syntax

An AS path is matched using the following syntax.

aspath *aspath_regexp* **origin any** | ([**igp**] [**egp**] [**incomplete**])

This specifies that an AS matching the *aspath_regexp* with the specified origin is matched.

4.25.5 AS path regular expressions

Technically, an AS path regular expression is a regular expression with the alphabet being the set of AS numbers. An AS path regular expression is composed of one or more AS paths expressions. An AS path expressions is composed of AS path terms and AS path operators.

AS path terms

An AS path term is one of the following three objects:

autonomous_system
.
(*aspath_regexp*)

autonomous_system Is any valid autonomous system number, from one through 65534 inclusive.

. matches any autonomous system number.

(*aspath_regexp*) parentheses group subexpressions–an operator, such as *** or *?* works on a single element or on a regular expression enclosed in parentheses

AS path operators

An AS path operator is one of the following:

aspath_term **m,n**
aspath_term **m**
aspath_term **m,**
aspath_term *****
aspath_term **+**
aspath_term **?**
aspath_term | *aspath_term*

aspath_term **m,n** a regular expression followed by *m,n* (where m and n are both non-negative integers and $m \leq n$) means at least *m* and at most *n* repetitions.

aspath_term **m** a regular expression followed by *m* (where m is a positive integer) means exactly *m* repetitions.

aspath_term **m,** a regular expression followed by *m,* (where m is a positive integer) means *m or more* repetitions.

aspath_term ***** an AS path term followed by *** means *zero or more* repetitions. This is shorthand for *0,*.

aspath_term **+** a regular expression followed by *+* means *one or more* repetitions. This is shorthand for *1,*.

aspath_term **?** a regular expression followed by *?* means *zero* or *one* repetition. This is shorthand for *0,1*.

aspath_term | *aspath_term* matches the AS term on the left, or the AS term on the right.

4.26 Route Importation

Importation of routes from routing protocols and installation of the routes in GateD's routing database is controlled by **import** statements. The format of an **import** statement varies depending on the protocol.

4.26.1 Specifying preferences

When importing a route, a preference can be assigned to the route. This preference will be used for selecting the best next hop to a destination when more than one next hop exist. The preference may be specified with one of the two following keywords:

restrict Specifies that the routes are not desired in the routing table. In some cases this means that the routes are not installed in the routing table. In others it means that they are installed with a negative preference; this prevents them from becoming *active* so they will not be installed in the forwarding table, or exported to other protocols.

preference *preference* Specifies the preference value used when comparing this route to other routes from other protocols. The route with the lowest preference available at any given route becomes the *active* route, is installed in the forwarding table, and is eligible to be exported to other protocols. The default preferences are configured by the individual protocols.

4.26.2 Route Filters

The route filters are used to explicitly list the routes the protocol is importing. All the formats allow route filters as shown below. See section 4.25.1 on route filters for a detailed explanation of how they work. When no route filtering is specified (i.e. when **restrict** is specified on the first line of a statement), all routes from the specified source will match that statement. If any filters are specified, only routes that match the specified filters will be imported. Put differently, if any filters are specified, an '**all restrict;**' is assumed at the end of the list.

network [**exact** | **refines**]
network **mask** *mask* [**exact** | **refines**]
network **masklen** *number* [**exact** | **refines**]
default
host *host*

4.26.3 Importing routes from BGP and EGP

import proto (**bgp** | **egp**) **autonomoussystem** *autonomous_system*
    restrict ;
import proto bgp | **egp autonomoussystem** *autonomous_system*
    [**preference** *preference*] **{**
    route_filter [**restrict** | (**preference** *preference*)] **;**
} ;

import proto bgp aspath *aspath_regexp*
    origin any | ([**igp**] [**egp**] [**incomplete**])
    restrict ;
import proto bgp aspath *aspath_regexp*
    origin any | ([**igp**] [**egp**] [**incomplete**])
    [**preference** *preference*] **{**
    route_filter [**restrict** | (**preference** *preference*)] **;**
} ;

The first form restrict all the route from being imported by EGP or BGP. The route are controlled by autonomous system. The second form uses a *route_filter* to explicitly list a set of routes to be accepted. **preference** can be used to set a default preference for all the routes in that set.

The two next forms are specific to BGP. They are the same as the two first ones except that the routes are controlled by by the use of an AS path, which are documented in the section 4.25.3 on Matching AS paths.

Note that EGP and BGP versions 2 and 3 only support the propagation of *natural* networks, so the *host* and *default* route filters are meaningless. BGP version 4 supports the propagation of any destination along with a *contiguous* network mask.

EGP and BGP both store any routes that were rejected implicitly by not being mentioned in a route filter, or explicitly with the **restrict** keyword in the routing table with a negative preference. A negative preference prevents a route from becoming active, which prevents it from being installed in the forwarding table, or exported to other protocols. This alleviates the need to break and re-establish a session upon reconfiguration if importation policy is changed.

4.26.4 Importing routes from an RIP, HELLO and Redirects

import proto (**rip** | **hello** | **redirect**)
    [(**interface** *interface_list*) | (**gateway** *gateway_list*)]
    restrict ;
import proto (**rip** | **hello** | **redirect**)
    [(**interface** *interface_list*) | (**gateway** *gateway_list*)]
    [**preference** *preference*] **{**
    route_filter [**restrict** | (**preference** *preference*)] **;**
} ;

The importation of RIP, HELLO and Redirect routes may be controlled by any of protocol, source interface and source gateway. If more than one is specified, they are processed from most general (protocol) to most specific (gateway).

RIP and HELLO don't support the use of preference to choose between routes of the same protocol. That is left to the protocol metrics.

These protocols do not save routes that were rejected since they have short update intervals.

4.26.5 Importing routes from OSPF

import proto ospfase [**tag** *ospf_tag*] **restrict ;**
import proto ospfase [**tag** *ospf_tag*]
    [**preference** *preference*] **{**
    route_filter [**restrict** | (**preference** *preference*)] **;**
} ;

Due to the nature of OSPF, only the importation of ASE routes may be controlled. OSPF intra- and inter-area routes are always imported into the gated routing table with a preference of 10. If a tag is specified, the import clause will only apply to routes with the specified tag.

It is only possible to restrict the importation of OSPF ASE routes when functioning as an AS border router. This is accomplished by specifying an **export ospfase** clause. Specification of an empty export clause may be used to restrict importation of ASEs when no ASEs are being exported.

Like the other interior protocols, preference can not be used to choose between OSPF ASE routes, that is done by the OSPF costs. Routes that are rejected by policy are stored in the table with a negative preference.

4.27 Route Exportation

4.27.1 The Export Statement

The **export** statement controls which routes are advertised by GateD to other systems as well as what routes are redistributed between the protocols ran on the router. Like the **import** statement, the syntax of the **export** statement varies slightly per protocol. The syntax of the **export** statement is slightly different from the syntax of the **import** statement although the meanings of many of the parameters are identical. The main difference between the two is that while route importation is just controlled by source information, route exportation is controlled by both destination and source. In other words, the **import** statement list a set of routes that a given protocol will accept *from* a given *source*. The **export** statement statement list a set of routes learned *from* a given *source*, that will be announced *to* a given *destination*. For instance you can configure BGP to *announce to* the peer on autonomous system AS123 all the routes *learned from* the RIP protocol. This is a case of inter protocol route redistribution.

Basically the outer portion of a given **export** statement specifies the destination of the routes you are exporting. The middle portion restricts the sources of the routes that you wish to consider. And the innermost portion is a route filter used to select individual routes.

Moreover a metric is set for any exported route. This metric is send to the peer by certain protocol and may be used by the peer to choose the better route to a given destination. The following section explains the detail syntax of the **export** for different protocols, as well as the use of the metric.

4.27.2 Specifying metrics

Some routing protocol can associate a metric to a route when announcing the route. The **export** statement accepts to optional keywords to specify the metric to be sent with the route or to disable the route announcement:

restrict Specifies that nothing should be exported. If specified on the destination portion of the **export** statement it specifies that nothing at all should be exported to this destination. If specified on the source portion it specifies that nothing from this source should be exported to this destination. If specified as part of a route filter it specifies that the routes matching that filter should not be exported.

metric *metric* Specifies the metric to be used when exporting to the specified destination.

4.27.3 Route Filters

A route filter can be used with the **export** statement to explicitly list all the exported routes. All the formats allow route filters as shown below. See section 4.25.1 on route filters for a detailed explanation of how they work. When no route filtering is specified (i.e. when **restrict** is specified on the first line of a statement), all routes from the specified source will match that statement. If any filters are specified, only routes that match the specified filters will be exported. Put differently, if any filters are specified, an '**all restrict ;**' is assumed at the end of the list.

network [**exact** | **refines**]
network **mask** *mask* [**exact** | **refines**]
network **masklen** *number* [**exact** | **refines**]
default
host *host*

4.27.4 Exporting routes with EGP and BGP

export proto bgp | **egp as** *autonomous system*
    restrict ;
export proto bgp | **egp as** *autonomous system*
    [**metric** *metric*] **{**
    export_list **;**
} ;

Exportation to EGP and BGP is controlled by autonomous system, the same policy is applied to all routers in the AS.

EGP metrics range from 0 to 255 inclusive with 0 being the most attractive. BGP metrics are 16 bit unsigned quantities, i.e. they range from 0 to 65535 inclusive with 0 being the most attractive. While BGP version 4 actually supports 32 bit unsigned quantities, GateD does not yet support this.

The *export_list* specifies the source of the route we want to export. The syntax of this *export_list* depend of the source protocol, and is explained in section 4.27.7.

If no export policy is specified, only routes to attached interfaces will be exported. If any policy is specified the defaults are overridden; it is necessary to explicitly specify everything that should be exports.

Note that EGP and BGP versions 2 and 3 only support the propagation of *natural* networks, so the *host* and *default* route filters are meaningless. BGP version 4 supports the propagation of any destination along with a *contiguous* network mask.

4.27.5 Exporting routes with RIP and HELLO

export proto rip | **hello**
    [(**interface** *interface_list*) | (**gateway** *gateway_list*)]
    restrict ;
export proto rip | **hello**
    [(**interface** *interface_list*) | (**gateway** *gateway_list*)]
    [**metric** *metric*] **{**
    export_list **;**
} ;

Exportation to RIP and HELLO is controlled by any of protocol, interface or gateway. If more than one is specified, they are processed from most general (protocol) to most specific (gateway).

It is not possible to set metrics for exporting RIP routes into RIP, or exporting HELLO routes into HELLO. Attempts to do this are silently ignored.

The *export_list* specifies the source of the route we want to export. The syntax of this *export_list* depend of the source protocol, and is explained in section 4.27.7.

If no export policy is specified, RIP and interface routes are exported into RIP and HELLO and interface routes are exported into HELLO. If any policy is specified the defaults are overridden; it is necessary to explicitly specify everything that should be exports.

RIP version 1 and HELLO assume that all subnets of the shared network have the same subnet mask so they are only able to propagate subnets of that network. RIP version 2 removes that restriction and is capable of propagating all routes when not sending version 1 compatible updates.

To announce routes which specify a next hop of the loop back interface (i.e. static and internally generated default routes) via RIP or HELLO, it is necessary to specify the metric at some level in the **export** clause. Just setting a default metric for e.g. RIP or HELLO is not sufficient. This is a safeguard to verify that the announcement is intended.

4.27.6 Exporting routes with OSPF

export proto osfpase [**type 1** | **2**] [**tag** *ospf_tag*]
    restrict ;
export proto osfpase [**type 1** | **2**] [**tag** *ospf_tag*]
    [**metric** *metric*] **{**
    export_list **;**
} ;

It is not possible to create OSPF intra- or inter-area routes by exporting routes from the GateD routing table into OSPF. It is only possible to export from the GateD routing table into OSPF ASE routes. It is also not possible to control the propagation of OSPF routes within the OSPF protocol.

There are two types of OSPF ASE routes, *type 1* and *type 2*, see the OSPF protocol configuration for a detailed explanation of the two types. The default type is specified by the **defaults** sub-clause of the **ospf** clause. This may be overridden by a specification on the **export** statement.

OSPF ASE routes also have the provision to carry a *tag*. This is an arbitrary 32 bit number that can be used on OSPF routers to filter routing information. See the OSPF protocol configuration for detailed information on OSPF tags. The default tag specified by the **ospf defaults** clause may be overridden by a tag specified on the **export** statement.

The *export_list* specifies the source of the routes we want to export. The syntax of this *export_list* depend of the source protocol, and is explained in section 4.27.7.

4.27.7 Specifying the route source/Route redistribution between protocols

The *export_list* used in the previous description of the **export** statement specifies the source of a set of routes based on parameter like the protocol from where the source as been learn. Indirectly the *export_list* controls the redistribution of routes between protocols. The syntax varies slightly per source protocol.

Redistributing BGP and EGP routes

proto bgp | **egp autonomoussystem** *autonomous_system*
    restrict ;
proto bgp | **egp autonomoussystem** *autonomous_system*
    [**metric** *metric*] **{**
    route_filter [**restrict** | (**metric** *metric*)] **;**
} ;

BGP and EGP routes may be specified by source autonomous system. All routes may be exported by as path, see below for more information.

Redistributing RIP and HELLO routes

proto rip | **hello**
    [(**interface** *interface_list*) | (**gateway** *gateway_list*)]
    restrict ;
proto rip | **hello**
    [(**interface** *interface_list*) | (**gateway** *gateway_list*)]
    [**metric** *metric*] **{**
    route_filter [**restrict** | (**metric** *metric*)] **;**
} ;

RIP and HELLO routes may be exported by protocol, source interface and/or source gateway.

Redistributing OSPF routes

proto ospf | **ospfase restrict ;**
proto ospf | **ospfase** [**metric** *metric*] **{**
    route_filter [**restrict** | (**metric** *metric*)] **;**
} ;

Both OSPF, and OSPF ASE routes may be exported into other protocols. See below for information on exporting by tag.

Exporting interface routes

proto direct | **static** | **kernel**
    [(**interface** *interface_list*)]
    restrict ;
proto direct | **static** | **kernel**
    [(**interface** *interface_list*)]
    [**metric** *metric*] **{**
    route_filter [**restrict** | (**metric** *metric*)] **;**
} ;

These protocols may be exported by protocol, or by the interface of the next hop. These protocols are:

direct Routes to directly attached interfaces.

static Static routes specified in a **static** clause.

kernel On systems with the *routing socket*, routes learned from the routing socket are installed in the GateD routing table with a protocol of **kernel**. These routes may be exported by referencing this protocol. This is useful when it is desirable to have a script install routes with the *route(8)* command and propagate them to other routing protocols.

Exporting aggregate routes

proto default | **aggregate**
    restrict ;
proto default | **aggregate**
    [**metric** *metric*] **{**
    route_filter [**restrict** | (**metric** *metric*)] **;**
} ;

These protocols may only be referenced by protocol.

default Refers to routes created by the **gendefault** option. It is recommended that route generation be used instead.

aggregate Refers to routes synthesized from other routes when the **aggregate** and **generate** statements are used. See the section on Route Aggregation for more information.

Filtering exported routes by AS path

proto *proto* | **all aspath** *aspath_regexp*
origin any | ([**igp**] [**egp**] [**incomplete**])
restrict ;
proto *proto* | **all aspath** *aspath_regexp*
origin any | ([**igp**] [**egp**] [**incomplete**])
[**metric** *metric*] **{**
route_filter¡/a¿ [**restrict** | (**metric** *metric*)] **;**
} ;

When BGP is configured, all routes are assigned an AS path when they are added to the routing table. For all interior routes this AS path specifies IGP as the origin and no ASes in the AS path (the current AS is added when the route is exported). For EGP routes this AS path specifies EGP as the origin and the source AS as the AS path. For BGP routes, the AS path is stored as learned from BGP.

AS path regular expressions are documented in the section on Matching AS paths.

Filtering exported routes by route Tag

proto *proto* | **all tag** *tag* **restrict ;**
proto *proto* | **all tag** *tag*
[**metric** *metric*] **{**
route_filter [**restrict** | (**metric** *metric*)] **;**
} ;

Both OSPF and RIP version 2 currently support tags, all other protocols always have a tag of zero. The source of exported routes may be selected based on this tag. This is useful when routes are classified by tag when the are exported into a given routing protocol.

4.27.8 Example of route filtering or redistribution

The example will concern the export of routes to a BGP peer. We want to announce to the BGP peer on autonomous system AS123 the following routing information:

- All the BGP routes learned from the peer on autonomous system AS999 (an other BGP peer).
- The static default route we have configured with a **static** clause.
- A certain set of routes obtained from your internal network through the RIP protocol.

The configuration file will look like:

```
# This control the BGP exports to the on autonomous system AS123

export proto bgp as AS123 {

        # The BGP route learned from AS999

        proto bgp as AS999 { all; };

        # The static default route
```

```
        proto static { default; };

        # A certain set of our RIP routes

        proto rip {
                1.2.3 masklen 24;
                1.2.4 masklen 24;
                192.1.0;
        };
};
```

More examples are provided in section 4.29.

4.28 Route Aggregation

Route aggregation is a method of generating a more general route given the presence of a specific route. It is used, for example, at an autonomous system border to generate a route to a network to be advertised via EGP given the presence of one or more subnets of that network learned via RIP. Older versions of GateD automatically performed this function, generating an aggregate route to a *natural* network (using the old Class A, B and C concept) given an interface to a subnet of that *natural* network. However that was not always the correct thing to do, and with the advent of classless inter-domain routing it is even more frequently the wrong thing to do, so aggregation must be explicitly configured. No aggregation is performed unless explicitly requested in an **aggregate** statement.

Route aggregation is also used by regional and national networks to reduce the amount of routing information passed around. With careful allocation of network addresses to clients, regional networks can just announce one route to regional networks instead of hundreds.

Aggregate routes are not actually used for packet forwarding by the originator of the aggregate route, only by the receiver (if it wishes). A router receiving a packet which does not match one of the component routes which led to the generation of an aggregate route is supposed to respond with an ICMP *network unreachable* message. This is to prevent packets for unknown component routes from following a default route into another network where they would be forwarded back to the border router, and around and around again and again, until their TTL expires. Sending an unreachable message for a missing piece of an aggregate is only possible on systems with support for reject routes.

A slight variation of aggregation is the generation of a route based on the existence of certain conditions. This is sometimes known as the *route of last resort*. This route inherits the next hops and aspath from the contributor specified with the lowest (most favorable) preference. The most common usage for this is to generate a default based on the presence of a route from a peer on a neighboring backbone.

4.28.1 Aggregation and Generation syntax

aggregate default
    | (*network* [(**mask** *mask*) | (**masklen** *number*)])
    [**preference** *preference*] [**brief**] **{**
    proto [**all** | **direct** | **static** | **kernel** | **aggregate** | *proto*]
        [(**as** *autonomous system*) | (**tag** *tag*)
            | (**aspath** *aspath_regexp*)]
        restrict ;
    proto [**all** | **direct** | **static** | **kernel** | **aggregate** | *proto*]
        [(**as** *autonomous system*) | (**tag** *tag*)

```
                    | ( aspath aspath_regexp ) ]
          [ preference preference ] {
          route_filter [ restrict | ( preference preference ) ] ;
     } ;
} ;

generate ( default | network [ ( mask mask ) | ( masklen number ) ] )
     [ preference preference ] {
          [ ( as autonomous system ) | ( tag tag )
                    | ( aspath aspath_regexp ) ]
          restrict ;
     proto [ all | direct | static | kernel | aggregate | proto ]
          [ ( as autonomous system ) | ( tag tag )
                    | ( aspath aspath_regexp ) ]
          [ preference preference ] {
          route_filter [ restrict | ( preference preference ) ] ;
     } ;
} ;
```

Routes that match the route filters are called *contributing* routes. They are ordered according to the aggregation preference that applies to them. If there are more than one contributing routes with the same aggregating preference, the route's own preferences are used to order the routes. The preference of the aggregate route will be that of contributing route with the lowest aggregate preference.

preference *preference*

Specifies the preference to assign to the resulting aggregate route. The default preference is 130.

brief

Used to specify that the AS path should be truncated to the longest common AS path. The default is to build an AS path consisting of SETs and SEQUENCEs of all contributing AS paths.

proto *proto*

In addition to the special protocols listed, the contributing protocol may be chosen from among any of the ones supported (and currently configured into) GateD.

as *autonomous_system*

Restrict selection of routes to those learned from the specified autonomous system.

tag *tag*

Restrict selection of routes to those with the specified tag.

aspath *aspath_regexp*

Restrict selection of routes to those that match the specified AS path.

restrict

Indicates that these routes are not to be considered as contributors of the specified aggregate. The specified protocol may be any of the protocols supported by GateD.

route_filter

See below.

A route may only contribute to an aggregate route which is more general than itself; it must match the aggregate under its mask. Any given route may only contribute to one aggregate route, which will be the most specific configured, but an aggregate route may contribute to a more general aggregate.

4.28.2 Route Filters

All the formats allow route filters as shown below. See section 4.25.1 on route filters for a detailed explaination of how they work. When no route filtering is specified (i.e. when **restrict** is specified on the first line of a statement), all routes from the specfied source will match that statement. If any filters are specified, only routes that match the specified filters will be considered as contributors. Put differently, if any filters are specified, an '**all restrict;**' is assumed at the end of the list.

network [**exact** | **refines**]
network **mask** *mask* [**exact** | **refines**]
network **masklen** *number* [**exact** | **refines**]
default
host *host*

4.29 Examples

4.29.1 Passive RIP listener

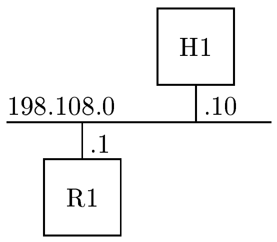

Figure 4.1: *Passive RIP listener*

The host H1 listen to RIP packets, but do not generate any routing announcement. To be used on a single connected workstation. A default route to the router R1 is set for safety which will be kept into the kernel routing table when gated exists (use of the **retain** statement). The logging in done into the file `/tmp/gated.log`, maximum size of 100kB, rotated once.

```
#
#  This configuration runs RIP in quiet mode, it only listens to
#  packets, no matter how many interfaces are configured.
#

traceoptions "/var/spool/gated/gatedlog" size 10k files 2 general;
```

```
rip yes {
        nobroadcast ;
};

static {
        0.0.0.0 gateway 198.108.0.1 retain;
};
```

4.29.2 RIP internal router

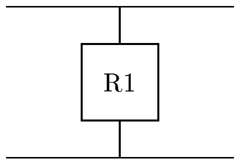

Figure 4.2: *RIP internal router*

In this example a router with several interface runs only RIP. RIP will accept any RIP route from any router, and will export to any interface all RIP routes as well as attached networks.

```
#
# Just enable RIP
#
rip yes;
```

4.29.3 RIP 2 with subnetted network

This this the same example as the previous one, except that a static default route is set and announced to the other RIP routers, and all subnets of 192.168.1 are aggregated into a natural class C network and announced to the other RIP routers.

```
rip yes;

# Define a default route

static {
        default gateway 192.168.1.18; #ether address of WORLD router
};

# Create the aggregate 192.168.1.0/24 from the attached subnetwork
# 192.168.1.0/28 and 192.168.1.16/28

aggregate 192.168.1.0 mask 255.255.255.0 {
        proto direct {
                192.168.1.0 mask 255.255.255.240;
                192.168.1.16 mask 255.255.255.240;
```

```
        };
};

# RIP exportation

export proto rip metric 2 {
        # The aggregate (class C net)
        proto aggregate {
                192.168.1.0 mask 255.255.255.0;
        };
        # The static route
        proto static {
                default;
        };
};
```

4.29.4 BGP-RIP router

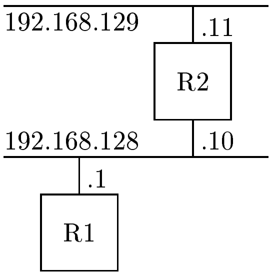

Figure 4.3: *RIP-BGP router*

The router R2 (in the autonomous system 1234) has a BGP session with the router R1 (in the autonomous system 2021). R2 will accept only the default route from R1, and will announce to R1 all it internal route got from RIP. R2 will also redistribute the default route to the internal routers through RIP.

```
#
# My autonomous system
#
autonomousystem 1234;

#
# RIP enable except on the DMZ (192.168.128)
#
rip yes {
        interface le1 noripin noripout;
};

#
# BGP session with the ISP
```

```
#
bgp yes {
        group type external peeras 2021 {
                peer 192.168.128.1;
        };
};

#
# Import only default from the ISP
#
import proto bgp {
        default;
};

#
# Export our routes to the ISP
#
export proto bgp {
        proto rip {
                all;
        };
};

#
# Redistribute default to our networks via RIP
#
export proto rip metric 2 {
        proto bgp as 2021 {
                default;
        };
        proto direct {
                all;
        };
};
```

4.29.5 BGP-RIP router with aggregation

This is the same example as the previous, but R2 will perform some aggregation of internal routes and announces only the aggregate to the ISP router.

```
#
# My autonomous system
#
autonomoussystem 1234;

#
# RIP enable except on the DMZ (192.168.128)
#
rip yes {
        interface le1 noripin noripout;
};
```

```
#
# BGP session with the ISP
#
bgp yes {
        group type external peeras 2021 {
                peer 192.168.128.1;
        };
};

#
# Import only default from the ISP
#
import proto bgp {
        default;
};

#
# Export our routes to the ISP
#
export proto bgp {
        proto aggregate { all; };
        proto rip { all; };
};

#
# Redistribute default to our networks via RIP
#
export proto rip metric 2 {
        proto bgp {
                default;
        };
        proto direct {all; };
};

#
# Aggregate our 4 class C nets into a /22 net. Two are directly connected nets.
#
aggregate 192.168.128.0 masklen 22 {
        proto direct {
                192.168.128.0 masklen 24 ;
                192.168.129.0 masklen 24 ;
        };
        proto rip {
                192.168.130.0 masklen 24 ;
                192.168.131.0 masklen 24 ;
        };
};
```

4.29.6 Sample Multicast Router Configurations

As with unicast routing protocols, there are a number of multicast routing protocols, each with its own advantages and disadvantages. In GateD, support for DVMRP, MOSPF, and Dense mode and Sparse mode PIM are anticipated. Currently, only Dense mode PIM is completed. All of these protocols rely on IGMP.

This configuration simply enables IGMP and PIM on all interfaces. The default mode for PIM is dense mode.

```
#
# Turn on IGMP
#
igmp yes ;

#
# Turn on PIM
#
pim yes ;
```

If you are running PIM in the presence of tunneled DVMRP and the GateD multicast kernel then PIM will try to run over the tunnel pseudo interfaces. They should be disabled as follows:

```
igmp yes ;

pim yes {
        interface tu disable ;
};
```

An example of a DVMRP host with one tunnel. This host also runs RIP, and does a lot of extensive logging:

```
traceoptions "/tmp/gated.log" replace  all;

rip yes;

kernel {
        traceoptions all request;
};

igmp on {
        traceoptions all packets;
};

dvmrp on {
        #
        # Tunnel definition. First remote address, then local address
        #
        tunnel 198.108.60.64 lcladdr 192.231.253.50;
        traceoptions all packets;
};

static {
        default gateway 192.231.253.2  retain;
};
```

Chapter 5

Operation Guide

5.1 Starting gated

5.1.1 Starting GateD at boot time

GateD is usually started from `/etc/rc.local` at system startup (boot) time. GateD is preferably started after the partition it resides in is mounted and after *syslog(8)* has been started. It is acceptable to start GateD **before** the interfaces have been enabled with *ifconfig(8)*, but it may be started later if desired. GateD must be started before starting any programs that require routing information, for example NFS mounts from systems on different networks or *xntpd(8)*.

5.1.2 Starting GateD directly

Usually a few lines like the following is added to `/etc/rc.local` to start GateD:

```
if [ -f /etc/gated.conf -a -f /usr/local/sbin/gated ]; then
    /usr/local/sbin/gated ; echo -n "gated "
fi
```

5.1.3 Starting GateD with Gdc

Starting GateD with Gdc is required when some system resources are inadequate to run gated. Again, a few lines need to be added to `/etc/rc.local`:

```
if [ -f /etc/gated.conf -a -f /usr/local/bin/gdc ]; then
    /usr/local/bin/gdc [flags] start ; echo -n "gated(via gdc) " ;
fi
```

Where *flags* are any **gdc** options.

5.1.4 Controlling GateD

GateD is normally controlled via the **gdc** utility, but may be controlled directly. See section 5.2 and section 5.3 for more information.

5.1.5 Checking GateD status

Using **gdc** it is possible to obtain a status dump from GateD. It is also possible to query GateD for information about the RIP and OSPF protocols with the **ripquery** and **ospf_monitor** utilities. See section 5.5 and section 5.4 for more information.

5.2 The Gated command

5.2.1 Command line options

gated [**-c**] [**-C**] [**-n**] [**-N**] [**-t** *trace_options*] [**-f** *config_file*] [*trace_file*]

The command-line options are:

-c

Specifies that the configuration file will be parsed for syntax errors and then **gated** will exit. If there were no errors, **gated** will leave a dump file in `/usr/tmp/gated_dump`. **Gated** does not need to be run as the super user to use the **-c** option but it may not be possible to read the kernel forwarding table and interface configuration if not run as super user. The **-c** option implies *-tgeneral.* All **traceoption** clauses in the configuration file will be ignored.

-C

Specifies that the configuration file will just be parsed for syntax errors. **gated** will exit with a status 1 if there were any errors and 0 if there were not. **Gated** does not need to be run as the super user to use the **-C** option but it may not be possible to read the kernel forwarding table and interface configuration if not run as super user.

-n

Specifies that **gated** will not modify the kernel forwarding table. This is used for testing **gated** configurations with actual routing data.

-N

Specifies that **gated** will not daemonize. Normally, if tracing to stderr is not specified **gated** will daemonize if the parent process ID is not 1. This allows the use of an `/etc/inittab`-like method of invoking **gated** that does not have a PID of 1.

-t *Trace_options*

Specifies a comma separated list of trace options to be enabled on startup. If no flags are specified, **general** is assumed. No space is allowed between this option and its arguments.

This option must be used to trace events that take place before the config file is parsed, such as determining the interface configuration and reading routes from the kernel.

See section 4.5 for valid trace options and a more detailed explanation of tracing.

-f *config_file*

Use an alternate config file. By default, **gated** uses `/etc/gated.conf`.

If a trace file is specified on the command line, or no trace flags are specified on the command line, **gated** detaches from the terminal and runs in the background. If trace flags are specified without specifying a trace file, **gated** assumes that tracing is desired to stderr and remains in the foreground.

5.2.2 Signal processing

The following signals may be used to control **gated**:

SIGHUP Re-read configuration.

A SIGHUP causes **gated** to reread the configuration file. **Gated** first performs a clean-up of all allocated policy structures. All BGP and EGP peers are flagged for deletion and the configuration file is re-parsed.

If the re-parse is successful, any BGP and EGP peers that are no longer in the configuration are shut down, and new peers are started. **Gated** attempts to determine if changes to existing peers require a shutdown and restart. OSPF is not capable of reconfiguring, it is shutdown and restarted during a reconfiguration. This may have an adverse impact on the routing system.

It should also be possible to enable/disable any protocol without restarting **gated**.

SIGINT Snap-shot of current state.

The current state of all **gated** tasks, timers, protocols and tables are written to `/usr/tmp/gated_dump`.

On systems supporting *fork()*, this is done by forking a subprocess to dump the table information so as not to impact **gated**'s routing functions. On systems where memory management does not support copy-on-write, *fork()* will cause the **gated** address space to be duplicated; this may be cause a noticeable impact on the system. On system not supporting *fork()*, the main process immediately processes the dump, which may impact **gated**'s routing functions.

SIGTERM Graceful shutdown.

On receipt of a **SIGTERM**, **gated** attempts a graceful shutdown. All tasks and protocols are asked to shutdown. Most will terminate immediately, the exception being EGP peers which wait for confirmation. It may be necessary to repeat the **SIGTERM** once or twice if it this process takes too long.

All protocol routes are removed from the kernel forwarding table on receipt of a **SIGTERM**. Interface routes, routes with *RTF_STATIC* set (from the route command where supported) and static routes specifying *retain* will remain. To terminate **gated** with the exterior routes intact, use **SIGKILL**.

SIGUSR1 Toggle tracing.

On receipt of a **SIGUSR1**, **gated** will close the trace file. A subsequent SIGUSR1 will cause it to be reopened. This will allow the file to be moved regularly.

It is not possible to use **SIGUSR1** if a trace file has not been specified, or tracing is being performed to stderr.

SIGUSR2 Check for interface changes.

On receipt of a **SIGUSR2**, **gated** will rescan the kernel interface list looking for changes.

5.2.3 Files

Many of default filenames listed below contain the string `%s`, which is replaced by the name with which gated is invoked. Normally this is *gated*, but if invoked as *gated-test*, *gated* will by default look for `/etc/gated-test.conf`. These paths may all be changed at compilation time.

`/usr/tmp/gated_dump` Where **gated** writes status information. The default is `/usr/tmp/%s_dump`. Another common path is `/var/tmp/%s_dump`.

`/etc/gated.conf` Where **gated** looks for its configuration file. The default is `/etc/%s.conf`.

`/etc/gated.pid` Where **gated** writes its process id (PID). The default is `/etc/%s.pid`, but `/var/run/%s.pid` is common.

5.3 gdc

5.3.1 Synopsis

gdc [**-q**] [**-n**] [**-c** *coresize*] [**-f** *filesize*] [**-m** *datasize*] [**-s** *stacksize*] [**-t** *seconds*] *command*

Gdc provides a user-oriented interface for the operation of the *gated (8)* routing daemon. It provides support for starting and stopping the daemon, for the delivery of signals to manipulate the daemon when it is operating, for the maintenance and syntax checking of configuration files, and for the production and removal of state dumps and core dumps.

Gdc can reliably determine *gated*'s running state and produces a reliable exit status when errors occur, making it advantageous for use in shell scripts which manipulate *gated*. Commands executed using **gdc** and, optionally, error messages produced by the execution of those commands, are logged via the same *syslogd(8)* facility which *gated* itself uses, providing an audit trail of operations performed on the daemon.

If installed as a setuid root program **gdc** will allow non-root users who are members of a trusted group (by default the **gdmaint** group) to manipulate the routing daemon while denying access to others. The name of the user is logged along via *syslogd(8)* along with an indication of each command executed, for audit purposes.

The command-line options are:

-n

Run without changing the kernel forwarding table. Useful for testing, and when operating as a route server which does no forwarding.

-q

Run quietly. With this option informational messages which are normally printed to the standard output are suppressed and error messages are logged via *syslogd(8)* instead of being printed to the standard error output. This is often convenient when running **gdc** from a shell script.

-t *seconds*

Specifies the time in seconds which **gdc** will spend waiting for *gated* to complete certain operations, in particular at termination and startup. By default this value is set to 10 seconds.

These additional command-line options may be present, depending on the options used to compile **gdc**:

-c *coresize*

Sets the maximum size of a core dump a *gated* started with **gdc** will produce. Useful on systems where the default maximum core dump size is too small for *gated* to produce a full core dump on errors.

-f *filesize*

Sets the maximum file size a *gated* started with **gdc** will produce. Useful on systems where the default maximum file dump size is too small for *gated* to produce a full state dump when requested.

-m *datasize*

Sets the maximum size of the data segment of a *gated* started with **gdc**. Useful on systems where the default data segment size is too small for *gated* to run.

-s *stacksize*

Sets the maximum size of stack of a *gated* started with **gdc**. Useful on systems where the default maximum stack size is too small for *gated* to run.

The following commands cause signals to be delivered to *gated* for various purpose:

COREDUMP

Sends an abort signal to *gated*, causing it to terminate with a core dump.

dump

Signal *gated* to dump its current state into the file `/usr/tmp/gated_dump`.

interface

Signal *gated* to recheck the interface configuration. *gated* normally does this periodically in any event, but the facility can be used to force the daemon to check interface status immediately when changes are known to have occurred.

KILL

Cause *gated* to terminate ungracefully. Normally useful when the daemon has hung.

reconfig

Signal *gated* to reread its configuration file, reconfiguring its current state as appropriate.

term

Signal *gated* to terminate after shutting down all operating routing protocols gracefully. Executing this command a second time should cause *gated* to terminate even if some protocols have not yet fully shut down.

toggletrace

If *gated* is currently tracing to a file, cause tracing to be suspended and the trace file to be closed. If *gated* tracing is current suspended, cause the trace file to be reopened and tracing initiated. This is useful for moving trace files.

By default *gated* obtains its configuration from a file normally named `/etc/gated.conf`. The *gdc* program also maintains several other versions of the configuration file, in particular named:

`/etc/gated.conf+` The *new* configuration file. When **gdc** is requested to install a new configuration file, this file is renamed `/etc/gated.conf`.

`/etc/gated.conf-` The *old* configuration file. When **gdc** is requested to install a new configuration file, the previous `/etc/gated.conf` is renamed to this name.

`/etc/gated.conf--` The *really old* configuration file. **Gdc** retains the previous *old* configuration file under this name.

The following commands perform operations related to configuration files:

checkconf Check `/etc/gated.conf` for syntax errors. This is usefully done after changes to the configuration file but before sending a **reconfig** signal to the currently running *gated*, to ensure that there are no errors in the configuration which would cause the running *gated* to terminate on reconfiguration. When this command is used, **gdc** issues an informational message indicating whether there were parse errors or not, and if so saves the error output in a file for inspection.

checknew Like **checkconf** except that the *new* configuration file, `/etc/gated.conf+`, is checked instead.

newconf Move the `/etc/gated.conf+` file into place as `/etc/gated.conf`, retaining the older versions of the file as described above. **Gdc** will decline to do anything when given this command if the *new* configuration file doesn't exist or otherwise looks suspect.

backout Rotate the configuration files in the *newer* direction, in effect moving the *old* configuration file to `/etc/gated.conf`. The command will decline to perform the operation if `/etc/gated.conf-` doesn't exist or is zero length, or if the operation would delete an existing, non-zero length `/etc/gated.conf+` file.

BACKOUT Perform a **backout** operation even if `/etc/gated.conf+` exists and is of non-zero length.

modeconf Set all configuration files to mode 664, owner root, group gdmaint. This allows a trusted non-root user to modify the configuration files.

createconf If `/etc/gated.conf+` does not exist, create a zero length file with the file mode set to 664, owner root, group gdmaint. This allows a trusted non-root user to install a new configuration file.

The following commands provide support for starting and stopping *gated*, and for determining its running state:

running Determine if *gated* is currently running. This is done by checking to see if *gated* has a lock on the file containing its pid, if the pid in the file is sensible and if there is a running process with that pid. Exits with zero status if *gated* is running, non-zero otherwise.

start Start *gated*. The command returns an error if *gated* is already running. Otherwise it executes the *gated* binary and waits for up to the delay interval (10 seconds by default, as set with the **-t** option otherwise) until the newly started process obtains a lock on the pid file. A non-zero exit status is returned if an error is detected while executing the binary, or if a lock is not obtained on the pid file within the specified wait time.

stop Stop *gated*, gracefully if possible, ungracefully if not. The command returns an error (with non-zero exit status) if *gated* is not currently running. Otherwise it sends a terminate signal to *gated* and waits for up to the delay interval (10 seconds by default, as specified with the **-t** option otherwise) for the process to exit. Should *gated* fail to exit within the delay interval it is then signaled again with a second terminate signal. Should it fail to exit by the end of the second delay interval it is signaled for a third time with a kill signal. This should force immediate termination unless something is very broken. The command terminates with zero exit status when it detects that *gated* has terminated, non-zero otherwise.

restart If *gated* is running it is terminated via the same procedure as is used for the **stop** command above. When the previous *gated* terminates, or if it was not running prior to command execution, a new *gated* process is executed using the procedures described for the **start** command above. A non-zero exit status is returned if any step in this procedure appears to have failed.

The following commands allow the removal of files created by the execution of some of the commands above:

rmcore Removes any existing *gated* core dump file.

rmdump Removes any existing *gated* state dump file.

rmparse Removes the parse error file generated when a **checkconf** or **checknew** command is executed and syntax errors are encountered in the configuration file being checked.

5.3.2 Related files

Many of default filenames listed below contain the string `%s`, which is replaced by the name with which gated is invoked. Normally this is *gated*, but if invoked as *gated-test*, **gated** will by default look for `/etc/gated-test.conf`. These paths may all be changed at compilation time.

`/etc/gated` the *gated* binary. Another popular location is `/usr/local/sbin/gated`.

`/etc/gated.conf` current *gated* configuration file.

`/etc/gated.conf+` newer configuration file.

`/etc/gated.conf-` older configuration file

`/etc/gated.conf--` much older configuration file

`/etc/gated.pid` where *gated* stores its pid, the default is `/etc/%s.pid`. Another popular location is `/var/run/%s.pid`.

`/usr/tmp/gated_dump` *gated*'s state dump file, the default is `/usr/tmp/%s_dump`. Another popular location is `/var/tmp/%s_dump`.

`/usr/tmp/gated_parse` where config file parse errors go, the default is `/usr/tmp/%s_parse`. Another popular location is `/var/tmp/%s_parse`.

`/usr/tmp` where *gated* drops its core file. Another popular location is `/var/tmp`. The core file is usually `core`, but some systems use `core.gated`.

5.3.3 Notes

Many commands only work when *gated* is installed in the system directory it was configured with.

There is not yet any way to tell **gdc** about systems which name their core dump other than `core` (`core.gated` is a less common possibility).

5.4 OSPF monitor

5.4.1 Synopsis

ospf_monitor *mon_db_file*

Use the **ospf_monitor** command to query OSPF routers. The **ospf_monitor** command operates in interactive mode. It allows the user to query the various OSPF routers to provide detailed information on IO statistics, error logs, link-state data bases, AS external data bases, the OSPF routing table, configured OSPF interfaces, and OSPF neighbors.

Mon_db_file is the complete pathname of a database composed of records configuring destinations for **ospf_monitor** remote commands. Each destination record is a single-line entry which lists the destination IP address, the destination hostname, and an OSPF authentication key (if authentication is activated by the destination). Since authentication keys may be present in the destination records, it is recommended that general access to this database be restricted.

Refer to [RFC 1583] (OSPF Specification, version 2) for details about OSPF database and packet formats.

5.4.2 Commands

Upon entering interactive mode, **ospf_monitor** presents the `'[ # ] dest command params >'` prompt, at which you can enter any of **ospf_monitor**'s interactive commands. Interactive commands can be interrupted at any time via a keyboard interrupt. Note that the command line length must be less than 200 characters.

Local Commands

? Display all local commands and their functions.

?R Display all remote commands and their functions.

d Display all configured destinations. This command displays *dest_index* , the IP address, and the hostname of all potential **ospf_monitor** command destinations configured in *mon_db_file*.

h Display the command history buffer showing the last 30 interactive commands.

x Exit the **ospf_monitor** program.

@ *remote_command* Send *remote_command* to the same (previous) destination.

@*dest_index remote_command* Send *remote_command* to configured destination *dest_index*.

F *filename* Send all **ospf_monitor** output to *filename*.

S Send all **ospf_monitor** output to stdout.

Remote Commands

a *area_id type ls_id adv_rtr* Display link state advertisement. *Area_id* is the OSPF area for which the query is directed. *adv_rtr* is the router-id of the router which originated this link state advertisement. *Type* specifies the type of advertisement to request and should be specified as follows:

- **1** Request the router links advertisements. They describe the collected states of the router's interfaces. For this type of request, the *ls_id* field should be set to the originating router's Router ID.
- **2** Request the network links advertisements. They describe the set of routers attached to the network. For this type of request, the *ls_id* field should be set to the IP interface address of the network's Designated Router.
- **3** Request the summary link advertisements describing routes to networks. They describe inter-area routes, and enable the condensing of routing information at area borders. For this type of request, the *ls_id* field should be set to the destination network's IP address.
- **4** Request the summary link advertisements describing routes to AS boundary routers. They describe inter-area routes, and enable the condensing of routing information at area borders. For this type of request, the *ls_id* field should be set to the Router ID of the described AS boundary router.
- **5** Request the AS external link advertisements. They describe routes to destinations external to the Autonomous System. For this type of request, the *ls_id* field should be set to the destination network's IP address.

c Display cumulative log. This log includes input/output statistics for monitor request, hello, data base description, link-state request, link-state update, and link-state ack packets. Area statistics are provided which describe the total number of routing neighbors and number of active OSPF interfaces. Routing table statistics are summarized and reported as the number of intra-area routes, inter-area routes, and AS external data base entries.

e Display cumulative errors. This log reports the various error conditions which can occur between OSPF routing neighbors and shows the number of occurrences for each.

h Displays the next hop list. This is a list of valid next hops mostly derived from the SPF calculation.

l [*retrans*] Display the link-state database (except for ASE's). This table describes the routers and networks making up the AS. If *retrans* is non-zero, the retransmit list of neighbors held by this lsdb structure will be printed.

A [*retrans*] Display the AS external data base entries. This table reports the advertising router, forwarding address, age, length, sequence number, type, and metric for each AS external route. If *retrans* is non-zero, the retransmit list of neighbors held by this lsdb structure will be printed.

o [*which*] Display the OSPF routing table. This table reports the AS border routes, area border routes, summary AS border routes, networks, summary networks and AS external networks currently managed via OSPF. If *which* is omitted, all of the above will be listed. If specified, the value of *which* (between 1 and 63) specifies that only certain tables should be displayed. The appropriate value is determined by adding up the values for the desired tables from the following list:

1 Routes to AS border routers in this area.

2 Routes to area border routers for this area.

4 Summary routes to AS border routers in other areas.

8 Routes to networks in this area.

16 Summary routes to networks in other areas.

32 AS routes to non-OSPF networks.

I Display all interfaces. This report shows all interfaces configured for OSPF. Information reported includes the area, interface IP address, interface type, interface state, cost, priority, and the IP address of the DR and BDR for the network.

N Display all OSPF routing neighbors. Information reported includes the area, local interface address, router ID, neighbor IP address, state, and mode.

V Display Gated version information.

5.5 ripquery

5.5.1 Synopsis

ripquery [**-1**] [**-2**] [-[**a5**] *authkey*] [**-n**] [**-N** *dest*[/*mask*] [**-p**] [**-r**] [**-v**] [**-w** *time*] *gateway ...*

Ripquery is used to request all routes known by a RIP gateway by sending a RIP request or POLL command. The routing information in any routing packets returned is displayed numerically and symbolically. **Ripquery** is intended to be used as a tool for debugging gateways, not for network management. SNMP is the preferred protocol for network management.

Ripquery by default uses the RIP POLL command, which is an undocumented extension to the RIP specification supported by *routed* on *SunOS 3.x* and later and by *gated* 1.4 and later. The RIP POLL command is preferred over the RIP REQUEST command because it is not subject to Split Horizon and/or Poisoned Reverse. See [RFC 1058] and [RFC 1388] for more information.

5.5.2 Options

-1

Send the query as a version 1 packet.

-2

Send the query as a version 2 packet (default).

-[**a5**] *authkey*

Specifies the authentication password to use for queries. If *-a* specified, an authentication type of SIMPLE will be used, if *-5* is specified, an authentication type of MD5 will be used, otherwise the default is an authentication type of NONE. Authentication fields in incoming packets will be displayed, but not validated.

-n

Prevents the address of the responding host from being looked up to determine the symbolic name.

-N *dest*[/*mask*]

Specifies that the query should be for the specified dest/mask instead of complete routing table. The specification of the optional mask implies a version 2 query. Up to 23 requests about specific destinations may be include in one packet.

-p

Uses the RIP POLL command to request information from the routing table. This is the default, but is an undocumented extension supported only by some versions of *SunOS 3.x* and later versions of *gated*. If there is no response to the RIP POLL command, the RIP REQUEST command is tried. *Gated* responds to a POLL command with all the routes learned via RIP.

-r

Used the RIP REQUEST command to request information from the gateway's routing table. Unlike the RIP POLL command, all gateways should support the RIP REQUEST. If there is no response to the RIP REQUEST command, the RIP POLL command is tried. *Gated* responds to a REQUEST command with all the routes he announces out the specified interface. Due to limitations in the UDP interface, on systems based on *BSD 4.3 Reno* or earlier, REQUESTs respond about the interface used to route packets back to the sender. This can be avoided by running **ripquery** on the host being queried.

-v

Version information about **ripquery** is displayed before querying the gateways.

-w *time*

Specifies the time in seconds to wait for the initial response from a gateway. The default value is 5 seconds.

5.6 Gated Interactive Interface

The *Gated Interactive Interface (GII)* provides an interactive interface to a running Gated daemon which can be used to query internal gated variables. This interface, which is implemented like any other protocol in Gated, accepts *telnet* connections to port 616 (C.f. [RFC 854] for a description of the telnet protocol), and, after user identification, answers any query sent as ASCII commands. The commands include querying about the memory, routing table, interface list and other internal parameters.

5.6.1 Configuration

GII will identify the user by using the Unix password of the *gii* ID of the system. In other words, when using the GII interface, one must create a 'gii' account and set up its password that will be used when connecting to the GII.

5.6.2 Connection to the GII

Users open a telnet connection to the machine running gated, on TCP port 616. They identify themselves using a simple password scheme, and get the **Gated** prompt. The GII is them ready to accept queries. Here is an example of such a session.

```
(lpj@obelix: 53) telnet radb2.ra.net 616
Trying 198.108.0.9...
Connected to radb2.ra.net.
Escape character is '^]'.
Password?
100 Gated Interactive Interface. Version R3_5Alpha_10
GateD> sh ip
100 HELP: The possible subcommands are:
100     route [x.x.x/len]: Show info about IP routes
100     walkup x.x.x/len: Show less specific routes
100     walkdown x.x.x/len: Show more specific routes
GateD>
```

5.6.3 Getting help

The top level command **help** provides a list of available commands. Commands may be followed by subcommands. To get help about the subcommand, just type the main command without subcommand. GII will display a list of available subcommands. Commands may be abbreviated when no confusion is possible.

5.6.4 Gated Internals

The following command will display parameters related to the internal structure of gated:

show version shows the version of the running Gated.

show kernel shows the type of kernel of the host, including what features are supported (Reject routes, Multicast, UDP checksums, etc...).

show memory shows the memory usage, divided by memory block structures.

show task shows the running tasks in Gated.

5.6.5 Interfaces

The following command will display parameters related to the interfaces:

show interface [name|index] Without parameter this command lists all the interfaces of the system. If one interface name or index is given as an argument, all parameters concerning this interface are displayed.

5.6.6 Routing table

The following command will display parameters related to the routing table:

show ip route [x.x.x/len] Without parameter this command prints the size of the IP routing table. With an argument which must be a prefix number and mask length, it displays the complete information about the given route like number of announcements, next hop, AS path, active route, etc...

show ip walkup x.x.x/len List all the routes less specific than *x.x.x/len*, i.e the aggregates which contain *x.x.x/len*.

show ip walkdown x.x.x/len Lists all the routes more specific than *x.x.x/len*, i.e all the components of the aggregate *x.x.x/len*. **show ip walkdown 0/0** will display the whole routing table. **ˆC** can be used to stop the listing.

Some examples:

```
GateD> sh ip walkdown 0/0
100 Sta         0.0.0.0/0  192.231.253.2   IGP (Id 1)
100 Sta             127/8  127.0.0.1       IGP (Id 1)
100 RIP         141.211/16 192.231.253.2   IGP (Id 1)
100 RIP         141.212/16 192.231.253.2   IGP (Id 1)
100 RIP         141.213/16 192.231.253.2   IGP (Id 1)
100 RIP         141.214/16 192.231.253.2   IGP (Id 1)
100 Dir     192.231.253/24 192.231.253.50  IGP (Id 1)
100 Dir         198.108/24 198.108.0.9     IGP (Id 1)
100 RIP      198.108.60/24 192.231.253.2   IGP (Id 1)
GateD>
GateD> sh ip walkdown 141/8
100 RIP         141.211/16 192.231.253.2   IGP (Id 1)
100 RIP         141.212/16 192.231.253.2   IGP (Id 1)
100 RIP         141.213/16 192.231.253.2   IGP (Id 1)
100 RIP         141.214/16 192.231.253.2   IGP (Id 1)
GateD>
GateD> sh ip walkup 192.231.253/24
100 Dir     192.231.253/24 192.231.253.50  IGP (Id 1)
100 Sta         0.0.0.0/0  192.231.253.2   IGP (Id 1)
GateD>
GateD> sh ip ro 198.108.60/24
100 Route 198.108.60 - 255.255.255 entries 1 Announced 1 Depth 0 <>
100 * RIP 192.231.253.2   192.231.253.2   4/0     0   10:25:13 IGP (Id 1)
```

5.6.7 DVMRP

show dvmrp mfc List all Multicast forwarding Cache defined.

show dvmrp targets List all the DVMRP targets (interfaces).

5.6.8 Other

quit to close the GII session.

5.6.9 OSPF

These commands are only available for the new OSPF shipped with gated version 3_6Alpha_2 and higher.

show ospf global Show general OSPF parameter like Router ID, preference, flags, etc...

show ospf interface [**x.x.x.x**] Without parameter shows the list of interfaces configured for OSPF, and their parameter (Timing interval, states, etc...). With an IP address as a parameter it shows the list of neighbours on that interface and their parameters and states.

show ospf area [**x.x.x.x**] Without parameter shows the list of area configured on that router. with an area number as a parameter, it show the interfaces for that area as well as the full dump of the link state database. (area number are in the same format as IP addresses)

Chapter 6

Trouble Shooting and Questions

- *A default route is defined in the configuration file, but the IGP refuses to export it.*

  Make sure to explicitly configure the **igp** to export the default route by using the **export** statement. For instance if the **igp** is RIP, the configuration file should contain something like:

```
export proto rip {
        proto static {
                default;
        };
};
```

- *The internal network is a subnetted class C network. The IGP or EGP refuses to export the class C network but only send the individual subnets.*

  If the **igp** or **epg** is a classless protocol (RIP 2, BGPv4), the protocol won't announce the class C unless explicitly configured to do so. The combined use of the **aggregate** and **export** statements should do the trick. See section 4.29.3 for a detailed example.

- *I've installed the Multicast patches for SunOS 4, but Gated does not seem to consider the interfaces as multicast. (Or gated still refuses to run RIP or OSPF in multicast mode)*

  One problem is that, when using gcc to build gated, the C header files are not patched correctly. gcc uses it own headers in `/usr/local/lib/gcc/<architecture>/<version>/include` instead of the normal `/usr/include`, and those files do not get patched for multicast support. Therefore, gated won't consider any interface as multicast.

  One solution is to copy the patched header files from `/usr/include` to `/usr/local/lib/gcc/<architecture>/<version>/include`.

- *I get those wierd 'VARIABLE MASKS NOT SUPPORTED' messages in my log files when gated starts.*

  That's means that your kernel does not support variables network mask (also refered as CIDR – C.f. [RFC 1518]). For instance SunOS4.1 does not support them. As long as you use only classfull IP addresses (the old class A, B and C network, C.f. RCF 791), or RFC791-type subnetting, everything should be OK, just ignore the message. If you want to do CIDR, you need to get another OS, this problem is related to the kernel, not gated!

- *When compiling, the C compiler complains about the lack of the 'objects.h' and 'smux.h' files.*

  That's because you are trying to build gated with the ISODE SMUX support (i.e. you can use SNMP/SMUX to query gated). In order to to that you need to install the ISODE package first. (C.f.

the INSTALL file for details). If you don't need SNMP you can just remove the 'isode_snmp' protocol from the list of protocols in your Config file (A few Config files distributed with gated have SNMP configured by default)

- *Gated works fine for a few minutes and then generates this error:* **if_rtdown: DOWN route for interface xxx0 1.2.3.4/255**

  That may be due to the fact that GateD times out the interface on which you are running an IGP if the interface does not receive any IGP packets (iether because the interface does not operate properly, or because there is no other router on that segment). A solution to that problem is to define the interface as *passive* by using the **passive** attribute in the **interfaces** section of the configuration file. For instance:

```
interfaces {
        interface xxx0 passive;
};
```

Chapter 7

Developer's Guide

7.1 Overview

Among other things, Gated is a portable routing protocol development platform, that factors out (and separately provides) functionality common to most protocol implementations. This functional factorization has three important advantages: 1) it simplifies the task of a routing protocol implementor, 2) when two or more routing protocols instances (i.e. participants in routing information interchange) are run on the same router, it enables harmonious sharing of computation and memory resources, and 3) it isolates the machine dependent part of protocol implementations to enable portability.

In the abstract, Gated models the operation of a human-configurable router as comprising (in the steady state) of the following primitive operations (in some order): receive protocol messages from neighbor routing entities, perform route computations, update one's own forwarding information base (FIB), and send protocol messages to neighbors.

Based on this model, we can factor out the following general mechanisms required, to varying degrees, by all routing protocol implementations:

- Mechanisms for scheduling protocol computation.
- Memory management mechanisms.
- Neighbor communication management mechanisms.
- Route storage mechanisms.
- Configuration and reconfiguration support.

Some of these mechanisms are provided by classical operating systems (such as UNIX). However, the underlying operating system's mechanisms can be inefficient for routing protocol implementations.

For this reason, Gated implements its own, highly optimized versions of these mechanisms. Gated, in its role as a routing daemon for BSD-based systems, was also built to run more than one protocol instance in a single box; using GateD's common substrate, these protocol instances could be made to share processor and memory resources more effectively.

Following is an overview of each of these mechanisms.

7.1.1 Mechanisms for Scheduling Protocol Computations

A single Gated daemon may run multiple routing protocol instances. It must be able to fairly allocate computational resources to these different routing protocol instances (e.g. a BGP instance and an OSPF instance). To this end, Gated provides support for *tasks*, *timers*, and *jobs*.

A **task**, is the thread of execution of a sequence of related primitive operations (such as reading a message, processing the message). These related operations can be scheduled independently of (i.e. arbitrarily interleaved with) the operation of other protocol instances or other operations within the same protocol instance.

Different protocol implementations use tasks differently. The simplest way to use tasks is to allocate a single task to all computation that happens within a protocol. Some, like BGP which maintains a transport connection to each individual peer, assign a task to protocol activity associated with the peer .

GateD's **timer** abstraction allows protocol instances to express periodic or time-delayed computations. An example of such a computation is the monitoring of the state of a peer connection, or sending keep-alive messages in BGP.

GateD's **job** abstraction allows a protocol instance to schedule some computation for some convenient time in the future. The OSPF implementation, for instance, schedules an SPF computation using jobs.

7.1.2 Memory Management Mechanisms

Gated uses the underlying operating system's memory allocator for its memory needs. However, it uses its own, highly space-optimized, allocator for descriptor blocks (e.g. the task state block, other protocol control blocks and so on).

7.1.3 Communication Mechanisms

For routing protocol message transport (unreliable datagram, or reliable byte-stream connections), protocol implementations almost always use the underlying operating system interface. Some earlier protocol implementations used GateD's interface to unreliable datagram transport.

To simplify protocol implementations, Gated provides abstract data types for physical interfaces, and routing neighbor gateways. These data types encapsulate information about these physical entities commonly used by different protocols.

7.1.4 Route Storage Mechanisms

Another common function of routing protocol implementations is the storage and processing of routes, and updating the router's forwarding information base. Gated provides a central routing database, in which protocol instances running on the same router store their reachability information. The module providing this functionality also directly updates the kernel's routing table.

Gated's AS Path Module for efficient storage and processing of BGP AS paths can also be included here. This module provides an easily processable representation for BGP aspaths.

7.1.5 Configuration Mechanisms

Finally, Gated provides abstract data types that encapsulate configuration information common to most protocol implementations. These abstract data types encapsulate such things as autonomous system numbers that the entity is running in, policy specifications and so forth.

7.2 Compilation Configuration Process

7.2.1 Organization

The *gated* building process allows the compilation of gated for different architectures using the same source tree. Each different system can use a subdirectory of the `src` directory to compile the source files into architecture dependent object files.

Moreover the list of files to be compiled, and their content are dependent on the list of protocols the user chooses to configure in his `Config` file.

The *'make config'* process is in charge of configuring the object directory with the set of files needed for a certain configuration, as well as creating certain files used during the building process.

7.2.2 The *make config* process

The *'make config'* process creates the following files:

- `obj/defines.h` contains a list of `#define` clauses to be included in all the C source code. The value of those `#define` are generated from the **options** statements in the `Config` file as well as from the **protocol** statements.
- `obj/paths.h` contains a list of paths for different files *gated* uses, like the config file, dump file, etc... It is generated from `src/paths.h` by substitution (C.f. section 7.2.4).
- `obj/version.c` contains a list of version numbers for each source file as well as for the *gated* release. The version numbers are embedded in C string constants to be hardcoded in the *gated* binary.
- `obj/parser.y` is the real *yacc* source file to generate the parser for *gated* for the selected protocols. It is generated from `src/parser.y` by substitution (C.f. section 7.2.4).
- `obj/signames.c` is the C declaration of the `sys_signame[]` variable if the system does not support it. `signames.c` is generated from `<signal.h>` only if the NEED_SIGNAME option is defined in the `Config` file.
- `obj/Makefile` is the **make** rule file used to build *gated*. It is generated from `util/Makefile.temp` by substitution (C.f. section 7.2.4) and contains only the rules for the configured source files.
- `obj/*.c obj/*.h obj/*.y obj/*.l` A set of symbolic link to the source files in `src/`.

The files needed for this configuration process are:

- `util/configure.pl` is a **perl** script doing the configuration.
- `util/Makefile.templ` is a template for the Makefile.
- `util/files` is the list of files to compile. The list is sorted by protocols.
- `parser.y` is the full yacc source file with support for all protocols.
- `*.c *.h` are all the source files in the `scr/` directory.

To summarize, when executing `make config` the `util/configure.pl` script is first run. This script will:

- find out about the current directory and the object directory used for that system,
- generate all the files previously listed in the object directory.

Then a `make config` will be executed in the object directory. This process will:

- clean up the object directory,
- create `parser.c` and `parser.h` by running **yacc** or **bison** on `parser.y`.
- create the symbolic links,
- compute the file dependencies by running the `mkdep` command.

7.2.3 Make's targets list

The object directory *Makefile* defines the following targets for any program to be built. Other can be added on a case by case rule.

- *all::* Default target when executing `make`.
- *lint::* Used when checking the code with **lint(1)**.
- *install::* Used to install the software's executable.
- *install-man::* Used to install the software's manual pages and documentation
- *clean::* Used to clean up the directory.
- *cleanall::* A better clean than *clean::*. *clean::* removes only object files. *cleanall::* removes any file which can be generated from the original distribution.

7.2.4 File filtering

Some files in the distribution package are run through a filter to obtain a version which only supports of the user configured protocols. For example if only **rip** and **bgp** are needed then the resulting file will only contain the code for RIP and BGP, which makes the final binary smaller. The general syntax for filtering in or out some part of the code is similar to the C processor. The syntax is:

@BEGIN [**NOT**] *SYMBOL* [| *SYMBOL2*...]
text...
@END

The text in between **@BEGIN** and **@END** will be kept if the symbol *SYMBOL* (or *SYMBOL2*...) is defined. Those symbols are any variable, protocol or option defined in the Config file. They include program names (**cc**, **lex**...), directory or file names (**path_config**, **mandir**, ...), options (**NEED_STRERROR**, **POSIX_SIGNALS**, ...) and protocols (**bgp**, **rip**, ...). For the later the symbol to add after **@BEGIN** is **PROTO_BGP** for **bgp**, **PROTO_RIP** for **rip**, etc... (See the `%proto` definition in `util/configure.pl` for the equivalence between protocol names and symbols).

Also some generic names (as the dump file name, configuration file name, etc...) are replaced by their real value chosen by the user. If the source file contains the tokens **@(symbol)** then this token will be replaced by the value of **symbol** defined in the `Config` file. For instance, if the `Config` file contains

```
path_config             /etc/\%s.conf
```

and the file `paths.h` contains

```
#define _PATH_CONFIG   "@(_path_config)"
```

then the resulting file will be

```
#define _PATH_CONFIG   "/etc/\%s.config"
```

[**Note:** trailing '_' are ignored and substitution is case insensitive.]

7.2.5 Adding a new protocol

The following are recommendation to install a new protocol into gated:

- Define a name for the protocol to be used in the Config file, and add this name to the `%proto` definition in `util/configure.pl`. Also, if any new option is needed in the Config file add the default value to the `%options` table. You also may want to look at the `%dirPath` (directories and files used to configure the Makefile), `runtimePath` (directories and files to be hardcoded in `paths.h`), `%miscParam` (some more parameters for the Makefile), `%unixCmd` (Unix commands for the Makefile), `%cmdFlags` (flags for the Unix commands).
- Add all the new source files (C, yacc, etc... code) into the `util/files` file using the syntax defined in section 7.2.4.
- Modify `util/Makefile.temp` to add any new target.
- Update `parser.y` to add support for the new protocol configuration into the gated parser. Conditional compilation can be done by using the syntax defined in section 7.2.4.

7.3 Protocol Scheduling

In the abstract, we can model the operation of a participant in routing protocol exchange and computation as a loop consisting of the following primitive operations: read a protocol message, perform a route computation (this computation may trigger other sub-computations or cause a computation to be scheduled for later execution), write a protocol message.

Gated provides a common substrate, which includes abstractions such as tasks, timers and jobs, for implementing these primitive operations efficiently. A *task* represents a separately schedulable thread of protocol computation. A *timer* represents an event scheduled for a future instant, which causes some non-preemptable computation to be performed at that instant. This computation is associated with a specific task. A *job* (which can either be foreground or background) represents some non-preemptable protocol computation that can be scheduled anytime in the future.

7.3.1 Tasks in Gated

Since Gated provides a platform for prototyping and operating multiple routing protocol instances in a single entity (e.g. a router), it must be able to harmoniously manage primitive operations belonging to different routing protocol instances (e.g. a BGP instance and an OSPF instance). To do this, Gated provides a **task**, which is the thread of execution of a sequence of related primitive operations. These related operations can be scheduled independently of (i.e. arbitrarily interleaved with) the operation of other protocol instances or other operations within the same protocol instance. Tasks are roughly equivalent to co-routines (tasks are scheduled non-preemptively).

Different protocol implementations use tasks differently. The simplest way to use tasks is to allocate a single task to all computation that happens within a protocol. OSPF and RIP have a single task in whose context all protocol computation is performed. BGP, which maintains a transport connection to each individual peer, assigns a task to all protocol activity (message processing, route computation etc.) associated with the peer (or a peer group).

In the current Gated implementation, tasks do *not* correspond to separate UNIX processes (with the exception of the system trace dump task).

Task Data Structures

The central data structure in the task module is the task struct. It is a descriptor for all the processing state associated with a particular sequence of primitive operations. Such state includes pointers to protocol specific information (such as the protocol implementation state block, e.g. the *bgpPeer* struct associated with this block). A task is usually associated with a UNIX socket; the task structure contains the socket descriptor.

In addition, the task structure contains pointers to protocol specific routines for performing different primitive operations that the task module supports. These primitive operations are:

- Reading one or more routing protocol messages.
- Writing one or more routing protocol messages.
- *Flashing* route changes to the forwarding table.
- Scheduling a computation for a specified time in the future.
- Scheduling a computation for some time in the future.

For example, when it is ready to read update messages from a peer, the BGP protocol implementation registers a pointer to *bgp_recv_v4_update()*. with the corresponding peer's task. Then, when data actually arrives on a socket, the task module simply calls this handler to perform the protocol specific read.

To notify protocols of Gated reconfiguration, tasks also maintain pointers to protocol specific notification handlers (e.g., the protocol specific handler to be called before reconfiguration, after reconfiguration and so on).

Gated individually prioritizes each of these operations. At any given instant, pending timers have highest priority, followed by foreground jobs, reads from sockets, protocol specific exception handling, writes to sockets, and finally background jobs. Gated maintains a global list of tasks (*task_read_queue*, *task_write_queue* etc.) waiting to perform these operations; the task structure also contains linked list pointers for placement on each of these queues. (Actually, jobs and timers are queued in separate data structures. That's because a task may have more than one timer or job pending).

Task Module Entry Points

Protocol instances normally create a task by first allocating a task data structure for the task (by calling *task_alloc()*), instantiating the various field of the task structure, and then calling *task_create()* to insert the task structure into a global list of tasks. This insertion takes place in priority order (tasks are assigned a static priority). This latter function also creates a socket for use by the task for reads and writes.

Operations on the socket associated with the task include *task_close()* (to close the socket associated with the task, and remove it from the global read and write queues), *task_set_socket()* and *task_reset_socket()*.

Storage associated with the task data structure is removed when *task_delete()* is called. This frees up any other resources (timers, jobs) associated with the task and removes the task from any other lists to which it may belong.

Some task module functions are used to notify each individual task of events of global interest. Some of these are processed in task priority order. Thus, *task_reconfigure()* is called to force all protocol instances to re-read the configuration files. Similarly, *task_ifachange()* and *task_iflchange()* are used to notify protocol instances of changes in the status of one or more network interfaces. And, *task_terminate()* is used to notify a protocol instance of GateD's intention to wind up operations (e.g. in response to operator action). Also, *task_newpolicy()* is used to inform interested protocol modules of routing policy configuration changes. Finally, *task_flash()* calls each protocol instance's flash handler when any change happens to GateD's global route database.

Once a task has been created, individual protocol instances may schedule reads on the task socket using *task_set_recv()*, schedule blocked writes to the socket using *task_set_write()* (when a protocol instance has some data to send on a socket, it attempts to write it directly; if that blocks, it calls this function). No protocol implementation seems to use the socket exception feature.

From the operating system, task reads go directly into a global *task_recv_buffer* (Gated can do this because tasks are non-preemptive). It is then the protocol instance's responsibility to consume all the data in the buffer. Similarly, in preparing to write data, protocol instances formulate messages in a single global *task_send_buffer*; if the write blocks, it is the protocol instance's responsibility to spool the data elsewhere.

Task Processing and the Gated Main Loop

All protocol processing in Gated is scheduled in its *main()* loop. As described above, the main loop prioritizes different computations and repeatedly loops to discover which computations have been enabled since the last time the loop was traversed.

Basically, the main loop does a *select()* on all socket descriptors used for reading, writing, or exception handling. It falls into the *select()* when no timers are scheduled and the job queues are empty. Gated stays in the *select()* until the next scheduled timer or 8 seconds, whichever is earlier. After coming out of the select, it processes "high priority" timer events (see the timer section). If any operator actions occurred (in the form of UNIX signals), it processes these (from *task_process_signals()*) and causes any queued foreground jobs to run before returning to the *select()*. Otherwise, it performs socket reads and writes (from *task_process_sockets()*), pending timers and background jobs in that order.

Thus, a protocol's message read and write operations, as well as timers and jobs get scheduled from GateD's main loop. Only the flash routines are performed whenever (as a result of a read) GateD's routing database gets updated.

7.3.2 Timers in Gated

A timer is Gated schedules a particular computation for a specified future instant. The Gated timer module allows specification of both one-shot timers and interval timers. Time specification granularity is 1 second. Timer expiry is governed by system load; thus, a timer may not expire at exactly the scheduled interval since its last expiry.

Gated allows protocols to specify higher priority timers (these have a greater probability of correct expiry compared to "normal" timers). Gated also allows specification of drifting timers: that is, depending on how the timer is initialized, the next timer expiry is calculated based on when the timer actually expired last and the interval. It is also possible to configure the timer to be non-drifting: that is, timer expiry is scheduled at exactly some integral multiple intervals after the timer was first scheduled.

Protocols use timers for periodic tasks such as monitoring connection status, timing out connection opens and so on.

In the current Gated implementation, timer expiry is checked by polling all active timers in the main loop; operating system facilities for timer expiry notification are *not* used.

Timer data structures

The central data structure is the timer descriptor block *task_timer*. This includes timer parameters (interval, start time) as well as state describing the action to perform when timer expiry happens.

These *task_timer* data structures are queued into one of three global queues maintained by the task module: a list of high priority timers, a list of active timers and a list of inactive timers (the last is used when protocols temporarily need to idle timer expiry). The first two lists are sorted in increasing order of next expiry, to ease processing.

High priority timers are used by some protocols like OSPF for timing their hello deliveries. BGP uses "normal" timers for all its timing needs: keepalives, connection retries and so on.

Timer Module Entry Points

The function *task_timer_create()* creates a timer with the specified parameters and inserts it into the appropriate global list of timers. Conversely, *task_timer_delete()* removes a timer from its queue and frees any resources that the timer holds.

Also, *task_timer_reset()* is used to idle the timer (i.e. move it into the inactive list). *task_timer_uset()* is used to reschedule an idle timer or a timer already in progress.

Timer Module Processing

The primary timer handling is performed in the *task_timer_dispatch()* and *task_timer_hiprio_dispatch()* routines. These simply traverse the list of timers, call the appropriate handlers, recompute the next interval at which the timer should fire and reschedule the timer for then. These are called from the main loop.

7.3.3 Jobs in Gated

A Gated *job* represents a unit of computation that can be scheduled for a time later than the present. Jobs are either foreground or background. Foreground jobs are time-limited computations run when it is safe to modify the Gated routing database and are typically called from high priority timer handlers or flash handlers. Background jobs are longer running computations scheduled when no timers or I/O is pending. Each background job has a priority between zero and seven; background jobs are scheduled in priority order.

For example, BGP uses a foreground job to close a peer connection if an error is observed when flushing routes to that peer. This cannot be performed correctly from the task's write routine since there may be reads queued up or timers scheduled for the task. Similarly, the OSPF implementation sets a timer after an SPF computation to ensure that the SPF computation is not done within that time period; at the expiry of the timer, it sets up a background task to run the next SPF computation.

Job data structures

The descriptor block for a single job, *task_job*, contains pointers to the task for which this job is scheduled, the routine encompassing the computation to be performed and a single argument to pass to that routine.

The Gated module maintains a global list of foreground jobs (ordered FIFO) and a list of background jobs (ordered by priority). In addition, there exists a single array of pointers to the locations in the background queue marking ends of blocks of *task_job* structs with the same priority.

Job Entry Points

Protocol instances call *task_job_create()* to allocate a *task_job* structure and insert it into the appropriate list. They call *task_job_delete()* to do the opposite. Foreground jobs are automatically deleted after completion. Background jobs must be explicitly deleted.

Job Processing

The Gated main loop calls *task_job_fg_dispatch()* to schedule computation of foreground jobs. This runs all foreground jobs to completion. When possible, the Gated main loop calls *task_job_bg_dispatch()* to schedule computation of foreground jobs. Unlike the former, this function simply removes the first, highest-priority background job from the list of background jobs, schedules its execution and returns.

7.4 Memory management

Gated uses the operating system memory management functionality for some situations. However, to optimize memory usage for the most common case of allocating control blocks, Gated provides its own memory management routines. The memory management code is part of the task module probably for historical reasons.

Protocol modules (and other support modules as well) allocate memory in Gated in two distinctly different ways. For dynamic allocation of fixed size descriptors, Gated uses the memory allocator described below. For variable sized blocks (e.g. the BGP attribute descriptor structure *as_path*), Gated uses *malloc()*. (Actually, Gated optimizes this latter allocation as well, using *malloc()* only for the uncommon case and allocating fixed sized blocks for the common case).

The fixed block allocation model is optimized for the case when a module repeatedly requests the same sized block. The module first contacts the memory manager to obtain a descriptor block (called a *task_block*, see below) which indicates the size of blocks it will request in the future. To later allocate a block, it calls the memory manager with this descriptor block; the memory manager returns a pointer to the appropriately sized block.

7.4.1 Memory Management Data Structures

The fixed block allocator has two main data structures, a *task_block* and a *task_size_block*. The *task_block*, which is the descriptor block referred to above, contains size of each allocation request and some bookkeeping data.

A *task_size_block* is a descriptor for a block allocation of a particular size. It contains a linked-list of *task_block*s for blocks of that size. In addition, it contains a pointer to a free-list of blocks of that particular size.

7.4.2 Memory Management Entry Points

Protocol modules register with the memory manager by calling *task_block_init()* and specifying the size of allocations it expects to make with this registration.

Macros *task_block_alloc()* and *task_block_free()* respectively allocate and free a block out of the requested size. The memory allocation strategy is explained below.

Some protocol modules allocate data in units of pages directly. For example, the task module allocates its send and receive buffers this way. These modules may explicitly free these pages by calling *task_block_reclaim()*. These pages go into a global free list, out which the memory manager may allocate data for its block allocator.

7.4.3 Memory Management Internals

The internals of the memory manager can be explained with the aid of the following example. Suppose a protocol module registers with the memory manager for a block allocation of 16 bytes. The memory manager returns to the protocol module a *task_block* descriptor, after linking that descriptor to the *task_size_block* descriptor for 16 byte blocks.

The latter descriptor maintains a linked list of free 16 byte blocks. Freed blocks are linked in a singly-linked list (even though the pointer to a free block is cast into the head of a doubly-linked list, only the forward pointer is used in the free list). Whenever the free list for a *task_size_block* is empty, the memory manager allocates a page, then carves up the page into a linked list of appropriately sized blocks (the fragment of the page that doesn't fit into one block, called a *runt*, is linked into a free list corresponding to its size).

Allocations are done out of this free list. When a protocol module frees up a block, the freed block goes back on the free list.

7.5 Neighbor communication

Every routing protocol instance communicates with one or more neighbor for exchange of routing information. Gated provides an interface for protocol implementations to send/receive routing messages, and some abstract data types for physical entities involved in such communication (e.g. the neighbor itself, the physical interface over which the Gated instance talks to the router).

7.5.1 Interface to Data Transport Mechanisms

Most protocol implementations directly access the operating system interface to send and receive routing messages to and from neighbors. Thus, BGP directly reads the socket associated with its TCP connections to its peers.

However, some older protocol implementations appear to use the functions *task_receive_packet()* and *task_send_packet()* respectively. While ideal for protocols that use unreliable datagrams for sending routing protocol messages, these functions do not suffice for protocol that use stream-based transports.

7.5.2 Neighbor Gateway Module

Gated provides an abstract data type to encapsulate all useful information pertaining to a routing protocol neighbor.

The basic data structure here is the *gw_entry*, one per protocol and (possibly) one per peer, which stores the neighbor's address, some state information, the list of routes heard from the neighbor and the import and export policies pertinent to the neighbor.

The parser calls *gw_locate()* when reading from the configuration file, to allocate a *gw_entry* to the configured neighbor.

7.6 Routes storage mechanism

This component of the Gated core functionality provides a central repository for different protocol instances' currently preferred routes to destinations. It also provides an efficient way to store and process BGP route attribute information.

7.6.1 Gated Route Database

Gated provides a central repository for storing one or more of a protocol instance's preferred routes to a given destination. The route database module embodies a set of rules which select which protocol instance's route (and, if a protocol instance stores more than one route which one of these) to a same destination should be chosen.

Different protocol modules use the route database in different ways. OSPF maintains its link state database separately and instantiates routes into the routing database when it performs a link state computation. From OSPF, there can be only one route to a given destination. RIP and BGP add routes received in updates from peers; so the routing database may contain as many BGP routes to a destination as there are peers.

A sequence of atomic changes to the route database are made by explicitly locking the database. Whenever the database is unlocked, the route database module informs all protocol instances of all changes that have taken place; this can result in route update messages or unreachables being sent.

A detailed description of the data types, macros and function definitions of the route database may be found here.

Route Database Data Structures

The main data structure in the route database is the *rt_entry*. This is a placeholder for route-specific information for a particular destination/mask pair: next-hop, metric, preference, AS path and so on.

All routes to the same destination/mask pair are chained together in a doubly-linked list. The head of this list is a *rt_head* structure. A *rt_entry* contains a back pointer to this structure. The *rt_head* also contains a pointer to the "active" (i.e. most preferred) route to this destination/mask, a pointer to the previously "active" route and other information.

The Gated route database maintains a single radix tree for all the routes in the database (actually, one per address family). Tree searches are keyed on destination/mask pairs. The internal nodes of this tree are *radix_nodes*, and each internal node points to a *rt_head* structure.

A figure helps explain the routing database data structures better.

Some other data structures are of interest to protocol implementations. The *rt_list* structure contains a list of changes that have occurred to the database; this list is communicated to all protocol instances every time a change occurs. The *rt_parms* is used to temporarily store route parameters when parsing routing protocol messages; a *rt_entry* is later created for the route.

When Gated selects the "active" route, it notifies all protocol instances (this corresponds to an OSPF instance, or a BGP connection) of the new "active" route. A protocol instance's export policy may allow it to announce that route. The *rt_bits* field encodes the protocol instances that are currently announcing a particular route. This field is a bit mask; each protocol instance is allocated a "route bit" from a global bitmask – if a particular bit in the *rt_bits* bitmask is set, it means that the protocol instance allocated that bit is announcing the route.

Protocol instances maintain their own maps of routes they have stored in the routing database. Given a *rt_entry*, how do protocol instances find out which of their own data structures contain references to the *rt_entry*? Such protocol specific data is stored in a byte array in the route entry's *rt_head*. The position of this data within the byte array is determined by the protocol's "route bit". To each protocol instance, Gated statically allocates block of locations in this byte array where it may store protocol specific data. Gated maintains a global bitmask of allocated locations in the byte array (in the *rttsi_map*) as well as a mapping from a protocol's route bit to the allocated locations (*rtbit_info*). These data structures are better explained using the following figure.

Route Database Entry Points

Protocol instances call *rt_add()* to add a route with the specified parameters and to the specified destination/mask pair to the route database. This function allocates a *rt_head* and inserts that into the radix tree (if necessary, when no other routes to the same destination exist).

The function *rt_delete()* logically deletes a route from the route database. Resources allocated to the route are not released immediately, pending notification of the delete to all protocol instances that may be announcing the route.

The functions *rttsi_alloc()*, *rttsi_set()*, and *rttsi_get()*, respectively allocate a contiguous block of bytes for storing protocol-specific data in a *rt_head* entry, store and retrieve protocol instance specific data from a specified *rt_head*.

When a protocol instance receives an update for a route previously sent from a peer, it calls *rt_change_aspath()* to update the route entry with the new information. As an optimization, if the route in question is currently "active", Gated stores the changes separately and applies them when the route is re-advertised.

The function *rthlist_all()* walks through a specified address family's radix tree and returns a list of all *rt_head* entries in the tree. Variant functions exist for returning the list of all *rt_entry* elements in the tree, or the list of those specifying a certain criterion.

Finally, *rt_new_policy()* is called when a re-configuration occurs, and *rt_flash_update()* is queued by *rt_close* and called from the task scheduler to notify the kernel routing table and all protocol instances of the most recent routing table changes.

Route Database Internals

The functions *rt_table_add()*, *rt_table_locate()*, *rt_table_delete()*, respectively add, find, and delete a *rt_head* entry from a radix tree, given a destination address and mask. These are standard radix tree operations.

The functions prefixed by *rt_event_* flag a *rt_entry* with a specified state change. Actions related to a state change (e.g. when a route is deleted, a new active route must be computed) are also performed here.

The function *rt_insert()* encodes the routing database's rules on selecting the “active” route. The route with the highest preference is selected for the “active” route. To break ties, routes with shorter AS paths are preferred. To break ties among these, routes with the lowest next-hop addresses are chosen.

7.6.2 Gated AS Path Module

The Gated AS path module provides a facility to parse and otherwise manipulate BGP attributes. It is ostensibly separated from the BGP implementation under the expectation that other protocols (IS-IS) will also need similar facilities in the future.

In the abstract, given a set of attributes (origin, AS path and optional transitive attributes), this facility returns a descriptor for those attributes. This module also maintains the invariant that identical attribute sets share the same descriptor.

A detailed description of the data types, macros and function definitions of the AS path module may be found here.

AS Path Data Structures

The main data structure of the AS path module is an attribute descriptor block *as_path*. This is a variable sized block containing the origin attribute, the AS path, and any optional transitive attributes found in a route's attribute list.

If two routes have the same attributes, they share the same *as_path* struct. The descriptor contains a reference count of the number of routes sharing an AS path.

An AS path referenced by one or more routes is stored in a global hash table *path_list*, for quick retrieval. The hash value is a function of the AS path length, the origin attribute, and the length of the optional transitive attributes.

One other data structure of interest internally is the *as_path_list* structure. This is used to aggregate a number of different routes; a list of these structures denotes the AS paths of the routes contributing to the aggregate.

AS Path Module Entry Points

The primary function of the AS path module is performed by two routines *aspath_attr()* and *aspath_format_v4()*. Given a pointer to a BGP update message containing an attribute list, *aspath_attr()* returns a pointer to a *as_path* struct having the same attributes. *aspath_format_v4()* performs the opposite function: given an *as_path* struct, it formulates a BGP update message attributes part using the contents of the *as_path* descriptor.

Whenever routing updates are received and the Gated routing database changes, Gated updates its forwarding table. Before it does so, it tries to aggregate the route changes, if possible. *aspath_do_aggregation()* is used to compute the AS path of the aggregate route.

Finally, *aspath_rt_build()* is used to associate an AS path descriptor with a *rt_entry*. This simply ups the reference count in the common case.

AS Path Module Internals

The AS path hash table manipulation is relatively straightforward. *aspath_find()* implements the invariant of having matching attribute sets be represented by a single descriptor.

AS path memory allocation uses the fixed block allocator for the two common cases of attribute blocks being either smaller than 32 bytes or 64 bytes, and uses *malloc()* otherwise.

The Gated implementation allows a router running Gated to belong to a number of different ASs. An AS path may have been advertised to BGP instances running in different ASs on the same router. When this router advertises the route, it must correctly include all the local ASs which received this advertisement and passed it on. The AS path module maintains an array of ”local” AS numbers. The *aslocal_*()* set of functions is used to manipulate this array. The *as_path* descriptor maintains a bitmask of local ASs (a bit position in the bitmask indicates the index into the local AS array) which received this AS path. When the AS path module formulates an update message, it ensures that it lists all the local ASs relevant to the AS path as an AS set.

7.6.3 Routing database module type declarations

Type *rt_changes*

This structure is part of the *rt_head* structure. The latter heads a list of routes and contains a pointer to the active route being advertised to this destination. This structure contains changes (in metric, preference, AS path etc) if any, that have occurred in the active route.

type *rt_aggr_entry*

When a protocol aggregates a number of different destinations, the routes to be aggregated are stored in a doubly-linked list. This element forms the head of this doubly-linked list. It contains a back-pointer to the *rt_entry* for this route.

type *rt_aggr_head*

The head of a list of *rt_aggr_entry*s. This also contains the AS paths of the contributing routes.

type *rt_head*

The routing database in gated is a radix tree, each internal node of which contains a pointer to an *rt_head* structure. This structure contains a destination address and a doubly linked list of *rt_entry*s for routes to this destination. This structure also contains a back pointer to the internal node in the radix tree.

type *rt_entry*

This structure contains one of the routes known to the GateD routing database to a given destination. The information contained in this descriptor includes how the route was learned (IGP/EGP etc.), the next hop to the destination, the metric/preference value, the AS paths and so on.

type *rt_parms*

A temporary structure used to hold route parameters until all sanity checks have been done and the route has been added to the database.

type *rt_parms*

Each *rt_head* entry has a linked list of target specific information (tsi). This is a linked list of 16byte chunks of data which is route and protocol specific (for instance, the tsi data for a BGP route points to the corresponding bgp_rto_entry that contains this route). The tsi's for each protocol are stored sequentially; the *rttsi*()* set of routines is used to access this data.

type *rt_list*

Route flash updates to be processed are stored in this structure. An *rt_list* structure is a descriptor block at the head of a page. Descriptor blocks to routes to be processed are store in the page after this descriptor block. Subsequent pages are chained in a list through the *rtl_next* field in the descriptor.

7.6.4 Routing database module macros

macro *RTBIT_**

Each protocol or protocol peer (in peer-based protocols) is assigned a bit. The *rtbit_map* array contains a pointer to protocol specific state for each protocol. Each *rt_entry* contains a bit mask which specifies which protocol(s) this route was learned from. These macros manipulate this bitmask. The GateD routing database module maintains a mapping from bit number to the task that has the bit registered with the module.

macro *RT_ALLRT(_REV)*

Macro to sequentially traverse, from the beginning (resp. end), a linked list of *rt_entry*s.

macro *RT_ALLRT(_REV)_END*

The corresponding macros for ending the loop traversal.

macro *RT_LIST_**

These macros manipulate the *rt_list*s containing flash updates. These include macros to traverse the list one element at a time as well as macros to allocate the list.

7.6.5 Routing database module function definitions

function *rttsi_alloc()*

The *rttsi_map* array is a bitmask of all the tsi positions currently allocated. We simply find the first block of unallocated data bytes and proceed to allocate from there. This ensures that, for each protocol, we allocate tsi's contiguously. This function sets the *index* field in the *rtbit_map* array entry accordingly.

function *rttsi_get()*

Traverse a tsi list in a given *rt_head* structure and find out the data corresponding to the specified *rtbit*. Copy the data out onto the specified buffer.

function *rttsi_set()*

Traverse a tsi list in a given *rt_head* structure and write the specified data to the tsi location specified by the *rtbit*.

function *rttsi_reset()*

Traverse a tsi list in a given *rt_head* structure and reset the data bytes at the tsi location specified by the *rtbit*.

function *rttsi_release()*

Walk through a tsi list in the specified *rt_head* structure and release each block of tsi data.

function *rttsi_free()*

This changes the bitmap of tsi allocations to reflect the fact that the specified rtbit_map entry is no longer represented in the bitmap.

function *rth_remove()*

This removes all the resources associated with the specified *rt_head* including the tsi list and the radix tree node to which the *rt_head* belongs. The list of routes themselves are taken care of elsewhere.

function *rth_locate()*

For a given destination/mask pair, call the tree search routine to find a matching internal node with a non-null external data. If one exists, return the external data. Otherwise, allocate an *rt_head* entry, initialize it and insert it into the radix tree.

function *rtchanges_free()*

Free the resources associated with an *rt_changes* structure. These include any AS paths and address information.

function *rt_free()*

Free the resources (AS path, *rt_head* if necessary) for the given *rt_entry* structure.

function *select_active()*

Called whenever a list of routes to a destination changes (i.e. either a route is added or one is removed or both). This function selects the first route in the list of routes pointed to by the specified *rt_head*. This is used as the active route unless a holddown protocol is advertising the currently active route.

function *rt_remove()*

Remove a route entry from the list to which it belongs. Be sure to decrement the count of routes in the queue head.

function *rt_remove()*

In a list of routes, the routes are ordered active first, followed by hidden and "deleted" routes. To insert a "deleted" route, add to end of list. To insert a hidden route, search list from the back until you find an appropriate location to insert. Finally, to insert an active route, insert in order of preference. For routes of the same preference, aspath lengths and numerical origin IDs are used to break ties.

function *rt_event_*()*

This set of functions is called when a route state transition happens. Examples of such transitions include a route becoming active/inactive, routes changing preferences or next-hops etc. The action taken depends on the state (or combination of states) the route is currently in, and usually involves setting the state flag appropriately and (in some cases) could also involve re-queuing the route and recomputing the new active route.

function *rt_new_policy()*

Called when gated is initialized or after a reconfiguration. Remove any existing flash list and build a complete flash list from the entire routing database. Update the kernel routing table.

function *rt_flash_update()*

Called when gated is shutting down. Simply process the existing flash list by recomputing the aggregate list and flashing the kernel routing table with the given list.

function *rtbit_alloc()*

The *rtbit_map* array contains a map of bits allocated to a protocol or protocol peer. Traverse through this array to find the first unallocated bit and allocate it to this protocol. Also allocate a tsi block from the *rttsi_map* for this bit number and for the specified data size.

function *rtbit_free()*

Free the specified *rtbit_map* entry.

function *rtbit_set()*

Given a *rt_entry*, set the specified protocol bit to indicate that this protocol or protocol instance is announcing this route.

function *rtbit_reset()*

Reset the specified protocol bit in the specified route entry.

function *rtbit_reset_all()*

Reset the specified protocol bit in all existing routes. Make sure to clear out the tsi fields in each route header to which such route entries may belong.

function *rt_locate*()*

Given a destination address and mask, locate the *rt_entry* corresponding to this destination and additionally matching certain criteria (e.g. matching state and protocol, gateway entry etc.). Traverse the radix tree, find the matching *rt_head*, and traverse the *rt_list* until you find the corresponding *rt_entry*. Appears to be used in RIP and OSPF, not in BGP.

function *rt_lookup()*

Similar to *rt_lookup*()*, this function finds route to a destination/mask matching a set of criteria. However, it scans the entire routing table for aggregates to this destination and finds the most specific aggregate matching this criteria. Seems to be used mostly in the SNMP code.

function *rt_add()*

Add a route to the GateD routing database, given the various parameters to the route in a *rt_parms* struct. This function is called when updates are received in BGP, for instance. Simply locate the head of the queue into which this *rt_entry* should go into. Allocate a *rt_entry* copy the parameters (preference, metrics etc) into it. Add the *rt_entry* to the list of routes advertised by the specified router. Allocate an AS path to the route if necessary. Finally, call *rt_insert()* to insert the route into the list in preference order.

function *rt_change_aspath()*

Inaptly named, called when an update is received for any existing route and one or more of the route's attributes (metric, AS path, preference) might have changed. Most complicated processing in this case: is when a next-hop (or set of next-hops for a multi-path) has changed.

function *rt_delete()*

Called when an unreachable has been received. This simply removes the route from its gateway list, and marks it for deletion (which has to be done carefully if it is on a flash list, for example).

function *rt_default*()*

A configuration may specify a default route with a specified next hop and preference. These set of routines are used to add/delete or reinitialize the default route after a re-configuration.

7.7 Configuration support

7.7.1 Gated Policy Module

The Gated policy module provides a mechanism for storing Gated policy descriptions. The gated configuration mechanism enables specification of import and export policy; the policy module stores these specifications in a manner that enables quick policy conformance testing.

Abstractly, for a given protocol, Gated allows the import and export policies to each be specified by a set of *policy elements* (route parameter, (destination/mask, preference), (destination/mask, preference),...). Route parameter is protocol dependent: for example, BGP policy statements can be expressed in terms of AS number and an AS path regular expression, RIP policy statements can be expressed in terms of an interface list or router address list, and OSPF policy statements can be expressed in terms of route tags. The semantics of a policy element are as follows: if a route contains the route parameter specified in the policy element (e.g., the AS path matches the regular expression specified, or the AS number matches the peer from which the route was received), then the route is assigned the preference associated with the destination/mask that it best matches.

The policy module implements these policy rules in a time and space efficient manner. A detailed description of the data structures, macros and functions used in the policy module may be found here.

Policy Module Data Structures

The fundamental data structure in the Gated policy module is the *adv_entry*. The *adv_entry* is used to represent both a single policy element and the list of (destination/mask, preference) tuples within a policy element. The *dest_mask* structure within an *adv_entry* is used to store the destination address and mask in a tuple. A discriminator field is used to distinguish what the *adv_entry* contains.

To faster match a route's destination address and mask with the list of destination address mask pairs in a policy element, Gated builds a radix tree from this list. An internal node of this radix tree is the *dest_mask_internal* structure.

An import or export list is an ordered list of these policy elements. Import or export lists are stored in the protocol descriptor blocks. For instance, BGP maintains a per peer policy list in the *gw_entry* data structure associated with the peer. In addition, for BGP policy statements restricting importation of routes based on AS path regular expression, BGP maintains a global policy list.

This figure pictorially depicts the relationship between these different data structures.

Policy Module Entry Points

To compute route preferences based on policy or to filter route announcements, protocol modules use the *import()* and *export()* entry points. Given a policy list, these functions find the first (if any) policy element in the list which matches one of the given routes' parameters. It then finds out the best match destination/mask in the corresponding policy element's list of (destination/mask, preference) tuples.

Gated creates the radix trees from the list of (destination/mask, preference) tuples when the configuration files are parsed. To insert an element of this list into the radix tree formed, the parser code *adv_destmask_insert()*. After the entire tree is built for a given policy element, the parser calls *adv_destmask_finish()* to adjust tree pointers for fast tree traversal.

Policy Module Internals

Given a destination address and mask, the tree traversal function *adv_destmask_match().* finds the corresponding (destination/mask, preference) tuple in the specified list of such tuples. This is used both in the *import()* and *export()* functions.

Bibliography

[PIM Arch] Internet Draft, *"Protocol Independent Multicast (PIM): Motivation and Architecture"*, S. Deering, D. Estrin, D. Farinacci, V. Jacobson, C. Liu, L. Wei.

[PIM Spec] Internet Draft, *"Protocol Independent Multicast (PIM): Protocol Specification"*, S. Deering, D. Estrin, D. Farinacci, V. Jacobson, C. Liu, L. Wei.

[RFC 827] Request For Comment 1388, *"Exterior Gateway Protocol EGP"*, 10/01/1982, E. Rosen.

[RFC 854] Request For Comment 854, *"Telnet Protocol specification"*, 05/01/1983, J. Postel, J. Reynolds.

[RFC 891] Request For Comment 891, *"DCN local-network protocols"*, 12/01/1983, D. Mills.

[RFC 904] Request For Comment 904, *"Exterior Gateway Protocol formal specification"*, 04/01/1984, International Telegraph and Telephone Co, D. Mills.

[RFC 1058] iequest For Comment 1058, *"Routing Information Protocol"*, 06/01/1988, C. Hedrick.

[RFC 1105] Request For Comment 1105, *"Border Gateway Protocol BGP"*, 06/01/1989, K. Lougheed, Y. Rekhter.

[RFC 1112] Request For Comment 1112, *"Host Extensions for IP Multicasting (IGMP)"*, S. Deering.

[RFC 1163] Request For Comment 1163, *"A Border Gateway Protocol (BGP)"*, 06/20/1990, K. Lougheed, Y. Rekhter.

[RFC 1164] Request For Comment 1164, *"Application of the Border Gateway Protocol in the Internet"*, 06/20/1990, J. Honig, D. Katz, M. Mathis, Y. Rekhter, J. Yu.

[RFC 1245] Request For Comment 1245, *"OSPF Protocol Analysis"*, 08/08/1991, J. Moy.

[RFC 1246] Request For Comment 1246, *"Experience with the OSPF Protocol"*, 08/08/1991, J. Moy.

[RFC 1253] Request For Comment 1253, *"OSPF Version 2 Management Information Base"*, 08/30/1991, F. Baker, R. Coltun.

[RFC 1256] Request For Comment 1256, *"ICMP Router Discovery Messages"*, 09/05/1991, S. Deering.

[RFC 1265] Request For Comment 1265, *"BGP Protocol Analysis"*, 10/28/1991, Y. Rekhter.

[RFC 1266] Request For Comment 1266, *"Experience with the BGP Protocol"*, 10/28/1991, Y. Rekhter.

[RFC 1267] Request For Comment 1267, *"A Border Gateway Protocol 3 (BGP-3)"*, 10/25/1991, K. Lougheed, Y. Rekhter.

[RFC 1268] Request For Comment 1268, *"Application of the Border Gateway Protocol in the Internet"*, 10/25/1991, P. Gross, Y. Rekhter.

[RFC 1269] Request For Comment 1269, *"Definitions of Managed Objects for the Border Gateway Protocol (Version 3)"*, 10/26/1991, J. Burruss, S. Willis.

[RFC 1370] Request For Comment 1370, *"Applicability Statement for OSPF"*, 10/23/1992, Internet Architecture Board.

[RFC 1388] Request For Comment 1388, *"RIP Version 2 Carrying Additional Information"*, 01/06/1993, G. Malkin.

[RFC 1397] Request For Comment 1397, *"Default Route Advertisement In BGP2 And BGP3 Versions Of The Border Gateway Protocol"*, 01/13/1993, D. Haskin.

[RFC 1403] Request For Comment 1403, *"BGP OSPF Interaction"*, 01/14/1993, K. Varadhan.

[RFC 1518] Request For Comment 1518, *"Classless Inter-Domain Routing (CIDR): an Address Assignment and Aggregation Strategy"*, September 1993, V. Fuller, T. Li, J. Yu, K. Varadhan.

[RFC 1583] Request For Comment 1583, *"OSPF Version 2"*, 03/23/1994, J. Moy.

[RFC 1771] Request For Comment 1771, *"A Border Gateway Protocol 4 (BGP-4)"*, 03/21/1995, Y. Rekhter, T. Li.

[RFC 1772] Request For Comment 1772, *"Application of the Border Gateway Protocol in the Internet"*, 03/21/1995, Y. Rekhter, P. Gross.

Index