TE WHARE WĀNANGA O TE ŪPOKO O TE IKA A MĀUI
VICTORIA
UNIVERSITY OF WELLINGTON

Developing the Fringe Routing Protocol

by
Deb Shepherd

a thesis submitted to Victoria University of Wellington
in fulfilment of the requirements for the degree of
Masters of Engineering
in
Network Engineering

2011

extending the work begun by
Don Stokes
Knossos Networks Ltd

supervised by
Andy Linton and Ian Welch
Victoria University of Wellington

for ian

an engineer of the old–fashioned kind
wish you could read this

with thanks to

Andy Linton
Victoria University of Wellington
for your light hand of supervision

DonStokes
Knossos Networks

Ian Welch
Victoria University of Wellington

Peter Komisarczuk
formerly of *Victoria University of Wellington*
now at *Thames Valley University*

John Rumsey
Knossos Networks

and

Stephen Marshall
husband extraordinaire
for much more than can be listed here

Table of Contents

Chapter Five

Chapter Six

Developing the Fringe Routing Protocol

Abstract

An ISP style network often has a particular traffic pattern not typically seen in other networks and which is a direct result of the ISP's purpose, to connect internal clients with a high speed external link. Such a network is likely to consist of a backbone with the clients on one 'side' and one or more external links on the other. Most traffic on the network moves between an internal client and the external world via the backbone.

But what about traffic between two clients of the ISP? Typical routing protocols will find the 'best' path between the two gateway routers at the edge of the client stub networks. As these routers connect the stubs to the ISP core, this route should be entirely within the ISP network. Ideally, from the ISP point of view, this traffic will go up to the backbone and down again but it is possible that it may find another route along a redundant backup path.

Don Stokes of Knossos Networks has developed a protocol to sit on the client fringes of this ISP style of network. It is based on the distance vector algorithm and is intended to be subordinate to the existing interior gateway protocol running on the ISPs backbone. It manipulates the route cost calculation so that paths towards the backbone become very cheap and paths away from the backbone become expensive. This forces traffic in the preferred direction unless the backup path 'shortcut' is very attractive or the backbone link has disappeared.

It is the analysis and development of the fringe routing protocol that forms the content of this ME thesis.

1. Introduction

The basic underlying architecture and protocols of the Internet have been developed and successfully used for several decades. As the system evolves, it is inevitable that new ideas and concepts emerge requiring new ways of doing things, and that better ways of handling existing concepts appear. Such improvements are typically created in order to streamline or change the behaviour of one small part of the larger system, often in fairly narrow circumstances.

This project examines a new protocol that does exactly that. The Fringe Routing Protocol (FRP) is designed by Don Stokes of Knossos Networks (knossos.net.nz), a commercial ISP in downtown Wellington, New Zealand. Don devised FRP to solve a particular issue common to ISP style networks — an ISP tends to want traffic between it's connected client stub networks to move via the core backbone rather than via any alternative back routes added in to the network for reliability and redundancy.

An ISP designs their network primarily to connect their customers to the outside world. The core of the network is a highspeed, high capacity backbone capable of delivering traffic for the ISP itself and for its customers, quickly and efficiently. The outer fringe will have upstream (external) gateway connections to multiple layer–2 providers for redundancy and robustness. The inner fringe holds a multitude of internal client gateways and routers connecting layer–3 client stub networks to the backbone.

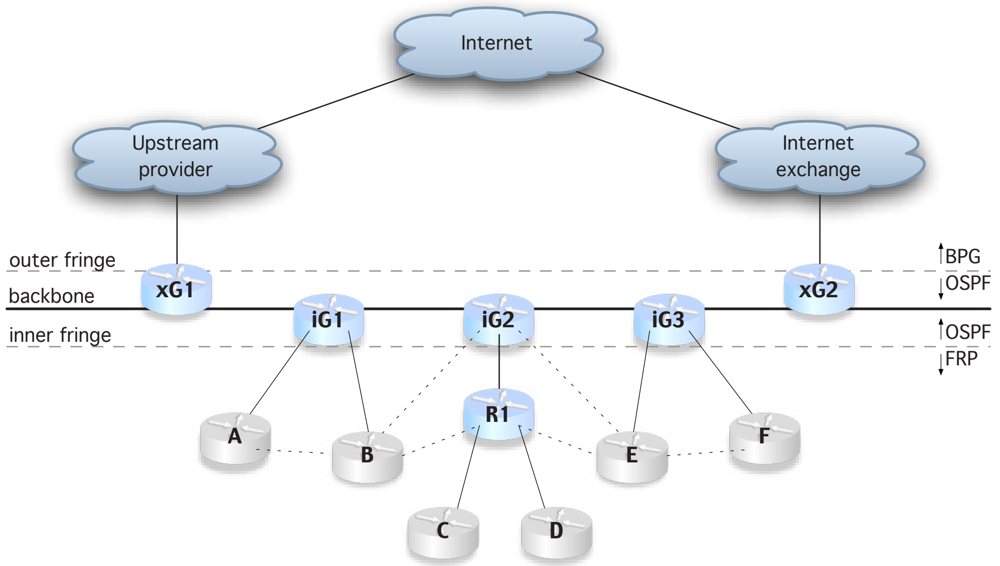


Figure 1.1
showing a simplified ISP network
ISP infrastructure is blue, client stub networks are grey
default routes are solid lines, backup routes are dotted
external gateway are labeled xG, internal (client) gateways are labeled iG,
routers are labeled R, client stub networks are labeled A–F

The nature of an ISP network — to provide connectivity to customers — means that the main flow of customer generated traffic through the network core is either from the outside world to the client's network, or from the client to the outside world.

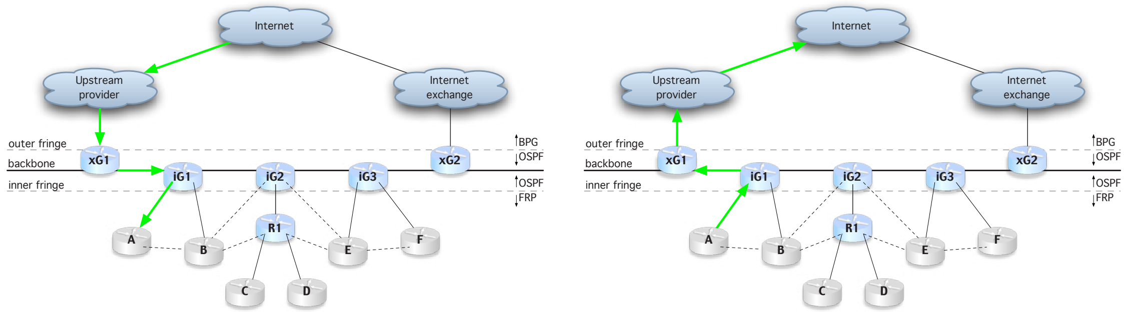


Figure 1.2
the main flow of client based network traffic through an ISP network

Considerably less traffic is likely to be generated between individual ISP customers and the traffic that is generated stays entirely within the ISP's network. This means that the ISP is able to impose some level of control over the routing of that traffic. The left hand diagram in figure 1.3 shows the preferred route for routing traffic between customer A and customer B, but A and B are also connected by a backup link, allowing traffic to take that path if it chooses.

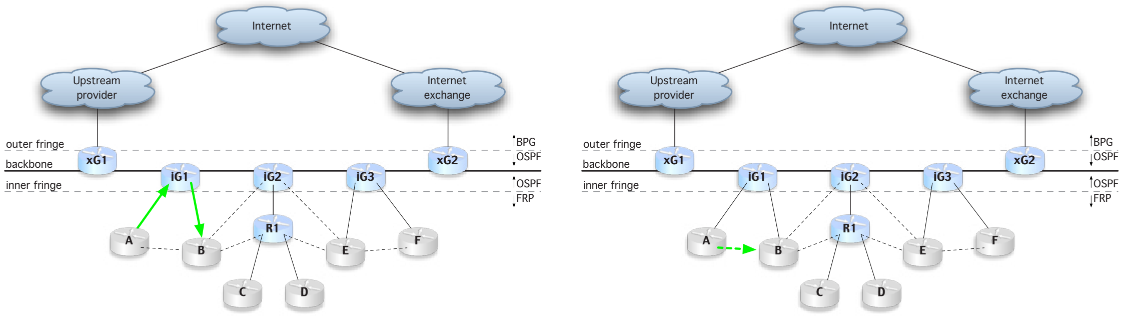


Figure 1.3
traffic between clients of the same ISP

This is the scenario that FRP is designed to address, a lightweight solution to routing traffic at the inner fringe of the ISP network and an alternative to the commonly used but more heavyweight protocols such as the Border Gateway Protocol (BGP) or Open Shortest Path First (OSPF).

An initial implementation of the fringe routing protocol has been developed at Knossos Networks as a proof of concept. That version was written, from scratch, specifically for the Knossos network and is currently running successfully.

This ME project involved taking FRP and creating a new, independent, implementation to complement the first, this time in a completely different development environment. Of the number of possible choices, the *Quagga software routing suite* was mandated. The process was to take the initial version as a reference and reverse engineer it with the aid of the accompanying notes. A specification of the protocol was then developed, which became the basis for the development of the *Quagga* FRP routing daemon.

In order to create a new *Quagga* protocol daemon, the intricate complexities of the *Quagga* suite need to be understood. It was necessary to also reverse engineer a number of the other *Quagga* daemons while building the new one as this process is not well documented.

2. Background

The project of developing a working version of the fringe routing protocol starts with the provided reference implementation and accompanying notes (see Appendix A). The brief set of notes provided is sparse but concise, the barest beginnings of a specification, and although padded out with a couple of face–to–face sessions, initial lack of knowledge limited the usefulness of these conferences.

The notes are essentially complete in that they address all the significant architectural components of the protocol and contain all the necessary detail. But understanding needs to be teased out, there being no explanation as to why elements are necessary or why certain factors need to be handled in a certain way. It is very much an architectural description of intent rather than an engineering framework for implementation. Included are a brief list of goals, the basics of the route forwarding algorithm, a short explanation on the importance of gateways, the message formats, a concise description of the batch processing, and an example of how the sequence numbering works.

So the initial part of the project became the process of understanding the sources provided, and an attempt at a first draft of a specification was written. The draft specification was created by reverse engineering the prototype to the point where the new version could be created from the specification alone, using it as a buffer between the code of the two separate implementations.

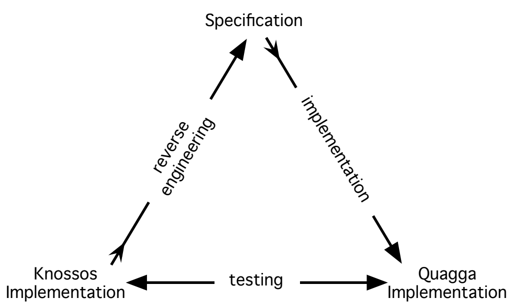


Figure 2.1
The ideal FRP development process

Reference implementation

The Knossos Networks reference implementation (see Appendix B) is a stand–alone system, combining the implementation of a basic, single threaded, Unix based router with the implementation of the fringe routing protocol. The code generally tries to keep these two functions separate but sometimes the two do become combined. At times the notes and the code do not match and the implementation significantly expands on, or even introduces new functionality.

The first step to understanding the workings of the system was to deconstruct it and fit it to the description of the protocol provided. The code is spread across two header and three code files. The main program loop plus supporting functions resides in a file named `main.c`, which is in turn supported by `route.c` and `parse.c`.

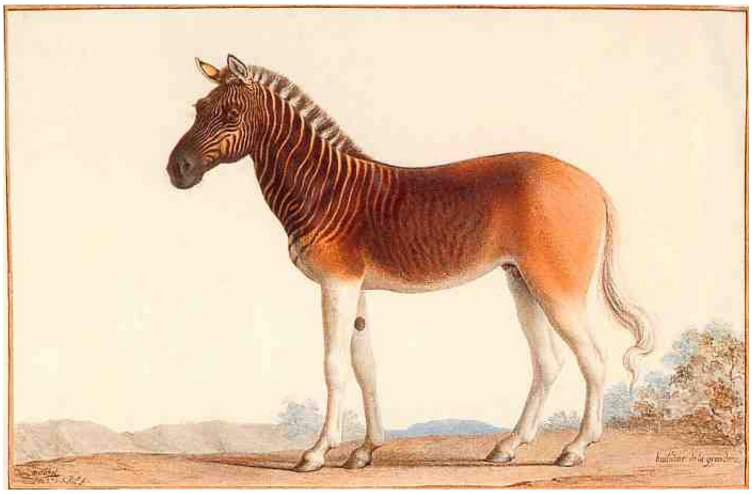
One header contains all defines, variables and data structures for the router in general. The other specifies those needed explicitly for the fringe routing protocol making it an invaluable resource. It was used extensively to define and build the required message and packet formats and many of the data structures. The main program loop is not multi–threaded so everything happens in a linear sequence, repeated indefinitely. This is one of the major differences between the implementations as the *Quagga* version is multi–threaded.

The essence of the FRP decision algorithm is in `route.c`. This only contains two functions, one to add an IP route to the routing table and another labelled as the 'main routing engine' [58]. `parse.c` handles numerous additional functions including adding peers, handling addresses, handling access control lists and parsing the configuration file.

The Quagga Software Routing Suite

One of the major outcomes of the FRP project is the development of a fringe routing protocol routing daemon using the *Quagga software routing suite*. *Quagga* is a popular open source alternative to *Cisco* or *Juniper* routers. The major difference, apart from price, is that *Cisco* and *Juniper* produce highly optimised dedicated hardware routers whereas *Quagga* is a software suite that allows a standard Unix based desktop machine to operate as a router, using a very similar command structure to *Cisco* routers. As the ability to hook an entirely new routing protocol seamlessly into the inner workings of the router was required, the open source route was an obvious choice — access to *Cisco* or *Juniper* at that level is virtually impossible.

The selection of *Quagga* from among the available choices was mandated. This springs from the fact that not only do *Knossos Networks* use *Quagga* in house, the Wellington Internet exchange (WIX) and *CityLink*, the company that run it, utilise *Quagga* extensively on small routers at the edge of their network.



The quagga (Equus quagga quagga) is an extinct subspecies of the plains zebra — Wikipedia (http://en.wikipedia.org/wiki/Quagga) image is in the public domain

Quagga is (a very much alive) fork of the older (and now essentially extinct) GNU Zebra project. Zebra began in 1996 under the auspices of Kunihiro Ishiguro [http://www.zebra.org/history.html] with the intention of creating a modular, TCP/IP based software routing engine made freely available under the GNU General Public License [http://www.zebra.org/what.html]. Zebra's modularity is achieved by creating a distinct daemon for each of the supported protocols — RIPv1, RIPv2 and OSPFv2 and BGP-4. This allows individual protocols to be modified separately, taking down the affected process while leaving the rest of the system on line. Likewise, the failure of any one daemon will only take out that individual process [http://www.zebra.org/features.html].

In 2003, *Zebra* forked into two separate projects, the original *Zebra* and a new project called *Quagga* [http://www.quagga.net/news2.php?y=2003&m=8]. The last *Zebra* release was zebra-0.95a in late 2005 [http://www.zebra.org/index.html]. As of early 2011, the latest *Quagga* release is quagga-0.99.17 [www.quagga.net].

Quagga is written in the C programing language and is firmly based on the *Zebra* model. It currently supports the BGP-4 (rfc1657, 1771, 1965, 1997, 2545, 2796, 2842, 2858), ISIS, OSPFv2 & v3 (rfc1850, 2328, 2370, 3101), OSPF6 (rfc2740), RIPv1 & v2 (rfc1058, 1724, 2082, 2453), and RIPng (rfc2080) routing protocols [http://www.quagga.net/docs/docs-multi/Supported-RFC.html]. Quagga is available for various Unix and Linux operating systems including FreeBSD, NetBSD, Debian, Ubuntu, Gentoo, and MacOSX. [www.quagga.net/about.php]

Somewhat surprisingly given that it is an open source project, *Quagga* proved to be a stable software platform with a well written development environment. The area in which it failed to deliver was the one which became a defining factor for this project. *Quagga* completely lacks clear, useful documentation. While the code is, on the whole, well written and well commented, it is a large complex system which is extremely difficult to understand sufficiently well to start to develop within it.

FRP in Quagga

Implementation started by experimenting with turning a copy of an existing Quagga protocol daemon into something that pretended to be a FRP daemon. Although this was useful, the next step was to start again and build an empty shell of a daemon that worked correctly inside Quagga but did nothing beyond that. The FRP implementation was then added to the shell.

As already mentioned, the intention was to develop the daemon entirely from the newly written specification but this proved to be impossible and, eventually, reference to the code of the prototype version became necessary in order to answer the many questions that arose. However, only one small piece of the prototype code was included in the Quagga daemon, the rest is written from scratch.

Tools

This project was conceived and developed entirely in a Unix operating system environment and ideally FRP should run on any 'flavour' of Unix. In the development of the FRP protocol to this point in time, a number of different Unix platforms have been used. The original implementation was developed and currently runs on FreeBSD. In the scope of the current project, this original version has been ported to NetBSD and to MacOSX, simply for convenience.

One of the problems with understanding and developing in a computer network environment is the fact that it is not possible to actually see the data that is travelling through the network. Tools are required to look at different parts of the environment at different points in time to see what is actually happening. Unix supplies a good range of useful command line tools including *netstat*, *route*, *ip*, and *ifconfig*, which are good for looking up information on sockets, interfaces, addresses, protocols and the kernel routing table. *ping* is useful for checking that machines acting as routers are up and visible and that traffic is actually routing through the network. *tcpdump* [http://www.tcpdump.org] captures network traffic over a period of time, printing out useful information and providing the ability to trace packets and analyse the data they contain.

WireShark [http://www.wireshark.org] is similar to *tcpdump* but comes as a standalone package with a graphical user interface and the ability to visually drill down through the contents of a packet. *WireShark* helpfully sections up each packet into its component headers and payload. It also makes traffic filtering easier than in *tcpdump*. *WireShark* is a cross platform, Unix based open source application using *tcpdump*'s *pcap* library to handle packet capture.

FRPsniffer

It became clear however, that it would be useful to have a packet sniffer tool that collected only FRP packets and knew how to correctly extract data from them. From this idea, *FRPsniffer* was designed and implemented — which also provided a useful first attempt at understanding and working with the FRP message and packet structure. Although *tcpdump* and especially *WireShark* are useful tools, this dedicated FRP traffic capture tool became more useful still. It was extensively used to watch FRP behaviour and the traces shown later in this document all come from *FRPsniffer*.

FRPsniffer is written in C and compiled with the MacOSX Xcode compiler. As this is essentially a Unix GNU Compiler Collection (gcc) compiler, the code is portable to any Unix based environment. The sniffer is written around the same library as *tcpdump* and *WireShark*, *tcpdump*'s *libcap* [http://www.tcpdump.org]. This is an open source C/C++ library providing a packet capture application programming interface (API). *FRPsniffer* is hardcoded to promiscuously collect all ethernet based UDP traffic on FRP's port and to display it broken down into FRP's packet and message structure.

Resources and documentation

To continue the theme of paucity in documentation, the standard technique of conducting a literature search for academic, peer reviewed papers proved fruitless. The following is a brief look at the sources that proved to be the most useful and the most regularly consulted during the completion of this project. For a complete list, refer to the bibliography on page 107.

General background

In order to even start on this project, a good general knowledge of computer networks and their architecture needed to be moved on to the next level and the inevitable gaps needed to be plugged. A couple of general networking texts

Kurose, J. F. and Ross, K. 2002 Computer Networking: a Top-Down Approach Featuring the Internet. 2nd. Addison-Wesley Longman Publishing Co., Inc.

Tanenbaum, A. S. 1985 Computer Networks. Prentice Hall PTR.

provided the starting point, followed by several more detailed works including

Perlman, R. 2000 Interconnections (2nd Ed.): Bridges, Routers, Switches, and Internetworking Protocols. Addison-Wesley Longman Publishing Co., Inc.

Stallings, W. 1993 Data and Computer Communications. 4th. Prentice Hall PTR.

Routing

The purpose of this project is the development of a routing protocol. A number of books provide good, general background reading on the topic of routing but the most useful was

Parkhurst, W. 2004 Routing First-Step. Cisco Press.

which provided a solid base to build on. The other useful volumes are

Beijnum, P. I. and Beijnum, I. V. 2002 BGP. O'Reilly & Associates, Inc.

Malhotra, Ravi. 2002. IP routing. O'Reilly Media, Inc.

Medhi, D. and Ramasamy, K. 2007 Network Routing: Algorithms, Protocols, and Architectures. Morgan Kaufmann Publishers Inc

Moy, J. T. 1998 OSPF: Anatomy of an Internet Routing Protocol. Addison-Wesley Longman Publishing Co., Inc.

The definitive source of information on routing protocols and the topics surrounding them is the Internet Engineering Task Force (IETF) Request for Comments (RFC) series.

Carpenter, B., Ed., "Architectural Principles of the Internet", RFC 1958, June 1996.

Bush, R. and D. Meyer, "Some Internet Architectural Guidelines and Philosophy", RFC 3439, December 2002.

provide general conceptual discussion on Internet design and Architecture, while

Hedrick, C., "Routing Information Protocol", RFC 1058, June 1988.

gives a very full and useful description of the distance vector algorithm and realities of implementing it, and

Fuller, V. and T. Li, "Classless Inter-domain Routing (CIDR): The Internet Address Assignment and Aggregation Plan", BCP 122, RFC 4632, August 2006.

contains critical information on current IPv4 address space usage.

Bradner, S., "The Internet Standards Process -- Revision 3", BCP 9, RFC 2026, October 1996.

Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

Postel, J. and J. Reynolds, "Instructions to RFC Authors", RFC 2223, October 1997.

has useful information on how to write and submit a RFC or any other documentation and, although obsolete,

Hinden, R., "Internet Engineering Task Force Internet Routing Protocol Standardization Criteria", RFC 1264, October 1991.

is interesting in that it still has some useful insight into what should be in a RFC for a routing protocol.

FRP

Specific information on the FRP protocol was obtained from the supplied specification notes and the code of the original implementation, both written by Don Stokes of Knossos Networks. Additional information came from talking to Don and from listening to him present the FRP protocol at the New Zealand Network Operators Conference in Wellington, February 2011 (see Appendix B).

Unix networking

The creation of a routing daemon for a Unix environment required a good working knowledge of the Unix socket API and how it works. Two sources proved to be invaluable. The book

W. R. Stevens *et al*, *Unix network programming: the sockets networking API*, 3rd ed. Boston, MA: Addison–Wesley, 2004.

provided a good, in depth recap of the topic and filled in many gaps but it was the online resource

B. J. Hall. (2009). *Beej's guide to network programming: using Internet sockets (version 3.0.14).* [Online]. Available: http://beej.us/guide/bgnet/

that proved the most valuable as it provided succinct descriptions of the different system calls, functions and structures in a logical manner that made it a very easy to use reference manual.

Quagga

The Quagga software routing suite package is complex and difficult to come to terms with. The Quagga package comes with a manual

K. Ishiguro *et al*. (2006). *Quagga: a routing software package for TCP/IP networks (Quagga 0.99.4)*. [Online]. Available: http://www.quagga.net/docs.php

which concentrates on installing, building and configuring Quagga plus providing a list of user commands. It also contains useful (if basic) sections on Quagga's architecture, supported RFCs and the Zebra protocol.

When it came to the *Quagga* daemon itself, the best resource was the actual program code

Available: http://www.quagga.net/download/

and the comments contain within it. The comments are generally adequate, although often brief and sometimes incomplete Two additional documents, although unfinished and very dated, provided invaluable background material and advice. The first,

Y. Uriarte. (2001). *Zebra for dummies*. [Online]. Available: http://www.quagga.net/zhh.html

proved particularly useful as it provides a 'how to' guide to creating a new Quagga routing daemon from scratch, including information on mandatory inclusions that is not available from any other source. The second,

Pilot. *Zebra hacking notes: for GNU Zebra 0.93b (rev 1.6).* [Online]. Available: http://quagga.net/faq/zebra-hacking-guide.txt

is more a list of notes, not as helpful as the Dummies guide but useful as an addendum to it.

Network packet sniffing

When writing the FRPsniffer support tool to collect and display traffic and packet contents in order to watch FRP actually running, the following article was very useful

L. M. Garcia, "Programming with libpcap: sniffing the network from our own application," in *Hackin9*, vol. 3, no. 2, pp. 38-46, 2008.

as was this documentation from the *tcpdump and libpcap* website

T Carstens. *Programming with pcap* [Online]. Available: http://www.tcpdump.org/pcap.html

3. Specifying the Fringe Routing Protocol

This chapter defines and specifies the fringe routing protocol. It looks at the goals it is expected to achieve, and the network environment it is designed to work in. The components of the protocol are specified: the importance of gateways; route forwarding; router configuration; message formats; and packet exchange.

The fringe routing protocol aims to send client to client traffic in an ISP style network up to the backbone and down again rather than across secondary backup links — that is from A to iG1 to B, rather than directly from A to B. To understand FRP however, it is necessary to move from looking at traffic delivery in the forwarding plane — how actual traffic is directed around the network — to studying the advertisement of preferred, or 'best' routes between routers in the control plane.

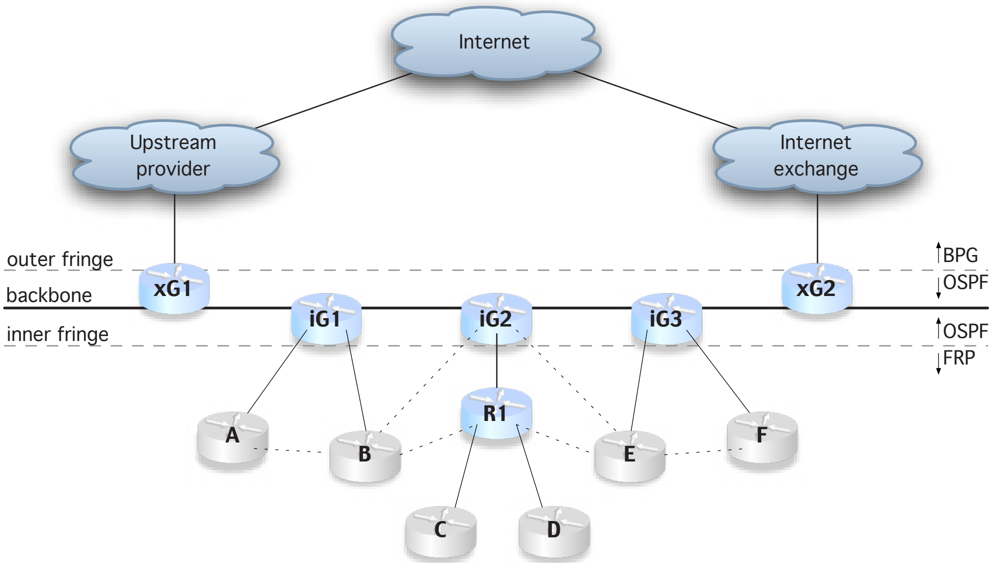


Figure 3.1
showing a simplified ISP network
ISP infrastructure is blue, client stub networks are grey
default routes are solid lines, backup routes are dotted
external gateway are labeled xG, internal (client) gateways are labeled iG,
routers are labeled R, client stub networks are labeled A–F

A primary concept of FRP is that of gateways. A FRP gateway is one that connects a customer stub network to the ISP backbone, so iG1, iG2 and iG3 are all potential FRP gateways. Each FRP router must have a single designated gateway and prefers to only advertise routes it has discovered if they lead towards its gateway. The only exception is if a link has gone down or if an alternative route is extremely cheap. The result of this carried through to the forwarding plane is that traffic is generally delivered up to the backbone, via the FRP gateway, and back down again.

The goals of the Fringe Routing Protocol

The primary goal of the fringe routing protocol is to simplify the management of an ISP style public network. The protocol achieves this by having the following characteristics:

1. Traffic is routed, by preference, towards the backbone via a designated gateway. Traffic is only routed away from the backbone in exceptional circumstances.
2. FRP is intended to be subordinate to the existing IGP, which is still used at the network core. FRP supplies reduced routing complexity at the network's inner fringe.
3. FRP attains reduced complexity by operating as a lightweight protocol. It achieves this by:
    a. remaining quiescent until an FRP gateway route to the backbone is established;
    b. and therefore only updating the kernel routing table or advertising routes if there is a path to a gateway;
    c. keeping routing table sizes to a minimum;
    d. keeping traffic down to keepalives if there are no route updates to be exchanged.
4. FRP must work on public access networks and so therefore must be secure. This security is based on a configured peer relationship.

FRP gateways

A key part of FRP is the use of gateways. FRP is designed to forward traffic up on to the ISP backbone for delivery — to destinations beyond the ISP network, to the ISP itself, or to another client of the ISP. FRP facilitates this delivery by utilising FRP designated gateways sitting on the backbone.

Every router using the fringe routing protocol needs to discover the 'best' path to one of the FRP backbone gateways. This becomes the router's designated gateway. It shares knowledge of this gateway, including the path to it, with its peers using special gateway messages. In the context of FRP, a peer is any directly connected router designated as such in the host's configuration.

Establishing gateway routes

FRP specifies that if a router has no gateway, then it is quiescent. However routers are only told if they themselves are a gateway, not what their 'nearest' or 'best' gateway is. Therefore any router that is designated as a FRP gateway needs to tell its peers that they can use it as such. This allows any connected peer on the FRP network to eventually compile a path to gateway for themselves. The newly discovered gateway and its path are then shared with peers and the propagation continues.

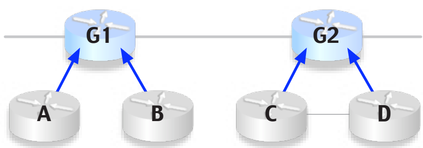


Figure 3.2
A and B gateway via G1
C and D gateway via G2

In figure 3.2, two FRP routers, G1 and G2, come on line and discover that they are gateways. They do not acknowledge each other as potential gateways because their own gateway route, being 0, is cheaper.

G1 and G2 duly inform each of their peers, A, B, C, and D, of this status so that each peer now has a path to gateway set. A and B set their gateway route via G1. C and D set theirs via G2.

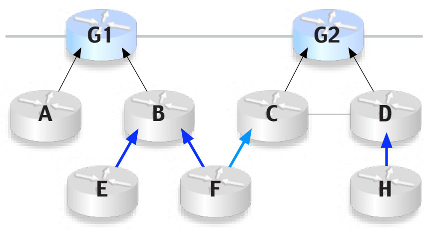


Figure 3.3
B advertises G1 to E and F
C advertises G2 to F
D advertises G2 to H
E gateway via G1, nexthop B
F chooses say G1, nexthop B over G2, nexthop C
H gateway via G2, nexthop D

In figure 3.3, B now advertises G1 to E and F, while C advertises G2 to D and F, and D advertises G2 to C and H. Consequently, E has a gateway route via G1 with a nexthop of B and H has a gateway route via G2 with a nexthop of D.

Both C and D ignore the advertisements from each other as they already have a cheaper route to G2.

F is given two gateway routes to choose from, one via G1, nexthop B and another via G2, nexthop C. F needs to pick the cheapest of these two — let's say F, B, G1.

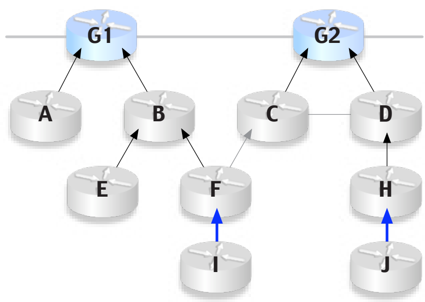


Figure 3.4
F advertises G1 to I
H advertises G2 to J
I gateway via G1, nexthop F
J gateway via G2, nexthop H

Next F advertises G1 to I, because it is cheaper than G2, and H advertises G2 to J.

I now has a gateway route via G1, nexthop F, and J now has a gateway route via G2, nexthop H.

Convergence is reached, leaving the setting of gateways complete, until a change in network topology forces a node to re–advertise and the process begins again.

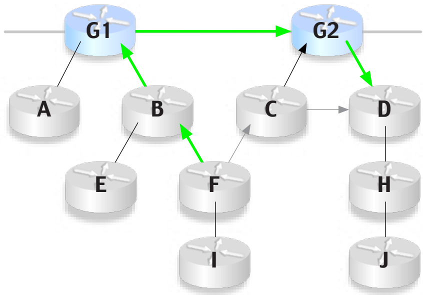


Figure 3.5
F to D via B, G1, G2, D rather than via C, D

FRP traffic is weighted to prefer gateway routes over other routes, so F will prefer to send traffic for D via F, B, G1, G2, D rather than F, C, D. In this example, route F, C, D will be seen as a redundant or back route because FRP prefers to pass traffic towards the gateway and F's gateway nexthop is B.

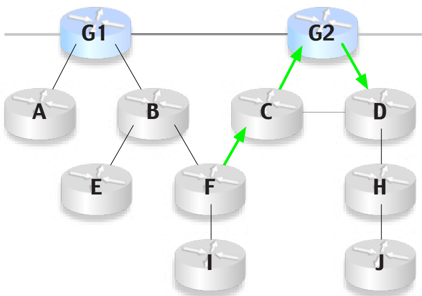


Figure 3.6
F to D via B, G1, G2, D rather than via C, D unless C, D is very cheap

There are only two reasons that F will route traffic to D via C. One is if the route from F to C to D is very cheap, an unlikely occurrence because then the cheapest gateway is more likely to be G2. The second is if the gateway weighting changes so that G2, nexthop C, becomes F's gateway route. The route from F to D then becomes F, C, G2, D.

Exchanging gateway routes

An FRP router is told in its configuration if it is a gateway or not. On start up, it works through the list of peers it is provided with and communicates with each one. If the router is a gateway, it uses this initial exchange to inform its peers of its gateway status and they adjust their path to gateway appropriately, sharing any new gateway routes with all their peers.

This interaction results in all 'alive' peers exchanging their current path to gateway. Routers that are not a gateway use these to reassess the cheapest gateway path for themselves, then sharing this with all their peers. Gateway route exchanges are triggered whenever a router changes their gateway status or recomputes a new path to gateway.

Avoiding loops in gateway paths

A common problem experienced by routing protocols is the creation of loops in the route paths discovered. This same issue can potentially occur in FRPs path to gateway discovery. It is to avoid this problem that a FRP host sends its entire gateway path to its peers. On receiving a path to gateway message, each router checks to see if its own IP address is in the path list. If it is, then the gateway path is excluded as a "*router will not learn a path that contains itself*" [58]. This avoids counting to infinity by ensuring that loops are not formed.

FRP route forwarding

The primary function of any router, or host, is to develop a forwarding table to facilitate the correct and efficient delivery of network traffic at the forwarding plane level. The forwarding plane encompasses the 'front end' of the router, the mechanism by which it accepts a packet for delivery and decides, by consulting its forwarding table, which of its peers is the 'nexthop' in the chain of routers the packet will pass through to reach its destination.

The primary function of any routing protocol is to provide the host with a routing table containing the 'best' routes according to that routing protocol. A host may employ more than one routing protocol, so the forwarding table is built by combining all the routing tables submitted by all the different protocols, and choosing the 'best' of 'the best' routes. This 'back end' of the router is the control plane.

FRP is designed to be subordinate to the existing interior gateway protocol (IGP) (goal 2, see page 20), typically OSPF or IS-IS, with the intention of simplifying route advertisement at the inner fringe of the network. The core of the network still requires the heavyweight complexity of the existing well–known, well–understood and well–tested protocols. However, the potentially large number of routes produced by these protocols do not necessarily need to be passed on to the client stub networks at the fringe when a single default route may be all that is necessary. So FRP takes over from OSPF et al, establishing a single gateway and telling the relevant client network to push all traffic towards it (goal 1, see page 20).

An issue that has not yet been addressed is where the FRP should sit in the table of administrative distance that is utilised to allow a router to decide between different routes to the same destination provided by different routing protocols. For the purposes of testing, FRP was given an arbitrarily high rank so that it automatically trumped every other protocol. On reflection, it seems that a high position on the list is perhaps the correct answer, given that FRP is designed to simplify the number of routes at the fringe. This does, however, stand out against the idea of FRP being subordinate to the core network IGP.

Protocol	Administrative distance
Directly connected route	0
Static route	1
External BGP	20
OSPF	110
IS-IS	115
RIP	120
Internal BGP	200

Figure 3.7
An abbreviated look at the administrative distance table [18]

In common with many other routing protocols, FRP builds a routing table, or routing information base (RIB), by selecting the ‘best’ route to each known destination from among all the routes provided by all its neighbours, or peers. This table of ‘best’ routes is then shared with all peers, excluding in each case any routes that the peer is nexthop for. Meanwhile, the host’s peers are still passing on their ‘best’ routes, triggering changes in the host’s RIB as new and ‘better’ routes are supplied until convergence is reached — that is all hosts have completed their RIBs and no more routing information is passed on. From this point, routes are only exchanged when changes to the network topology are discovered and shared.

Where FRP differs to other protocols is in the mechanisms it employs to build the RIB and in how it distributes routes to peers, in order to achieve the goals specified in its design.

Quiescence

A FRP router remains quiescent until it is able to determine a path to a FRP gateway (goal 3a, see page 20). A quiescent router exchanges configuration information and receives gateway path and route update information from other peers but does not send out routing updates until a gateway path is established. Should a node lose its gateway path and not be able to establish a new one, it returns to a quiescent state.

Establishing routes to destinations

FRP employs a distance vector algorithm to decide which route to a destination is the 'best'. Distance vector is iterative, distributed and self terminating. It is also asynchronous in that peers to not have to synchronously exchange data, although FRP overrides this in its message exchange process. A distance vector based protocol receives routing information from directly attached peers and then redistributes new routing information back to its peers.

The distance vector algorithm specifies that a node establish a route for each known peer, setting the cost to the cost of the link between them and setting the nexthop as that peer. This information is shared with all peers. Using the similar information delivered from those peers, identify any new destinations and create a route to them, specifying the peer that supplied the route as the nexthop. The cost of the route is calculated by added the cost of the link to the nexthop peer to the cost of the route as advertised. If more than one peer offers a route to a destination, select the one with the lowest cost. Share this new information with all peers. Continue to update and share whenever new routing information is received or whenever the network topology changes.

FRP modifies the behavior of the algorithm by adding the criteria that traffic should be preferentially forwarded towards the host's gateway. This creates the need for a more complex check to determine the 'best' route and brings the FRP decision algorithm in to play. FRP peers exchange information on the cost of the link between them, using this to assign a cost to each hop and so building up a metric for each route, as per the normal distance vector method. Additional to this is the metric associated with the host's gateway route and a flag that is added to each route that leads towards that gateway.

For a route to be included in the host's forwarding table, and therefore to be used as a route for delivering traffic, it must be cheaper than the combined gateway costs of the two nodes. FRP prefers to route traffic via the gateways and the backbone rather than over routes between peers. As a consequence, very few routes will be included in the FRP forwarding table as the FRP algorithm ensures that the gateway costs are almost always lower. Therefore, most traffic is directed towards the host's gateway and the backbone.

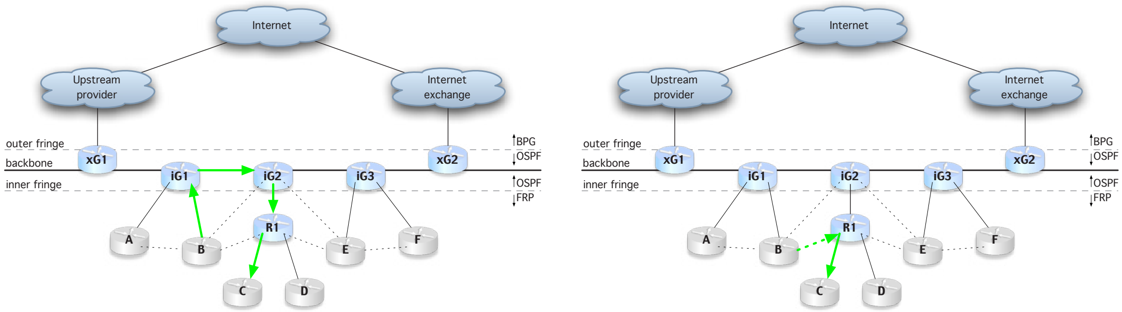


Figure 3.8
The example below illustrated

For example:

B's gateway is iG1, C's gateway is iG2 via R1. Should B route traffic for C via its designated gateway, across the backbone, and down to C (B, iG1, iG2, R1, C); or is it cheaper to send it via the backroute (B, R1, C). FRP prefers the first.

The FRP algorithm

A route leading away from the gateway is included in the RIB if:

`route-cost + link-cost < route-gw-cost + target-gw-cost + is-gw-route`

where:

- route-cost is the accumulated sum of the costs of intervening links
- link-cost is the cost of the link to remote-node
- route-gw-cost is the cost from the route's originating node to its gateway, as expressed in the routing update
- target-gw-cost is the cost from the remote-node to its gateway, as expressed in its link advertisements
- is-gw-route equals 1 if the route is being passed toward the gateway, 0 otherwise

[57]

Note that the cost of traversing the backbone is 0.

This same algorithm is also used to determine if a route should be advertised.

Split horizon

Having placed a route in the RIB, when should a FRP router advertise that route to its peers? The first caveat is that a route is never advertised back to the peer that provided it. This is the standard distance vector technique of avoiding counting to infinity — two peers each specifying that the other is the nexthop of a particular route.

Advertising routes

The second caveat is that the FRP algorithm is applied using the host's knowledge of the peers gateway cost. The route is only forwarded to the peer if the calculation indicates that it is cheaper for the peer to use that route than it is to send traffic via its own designated gateway.

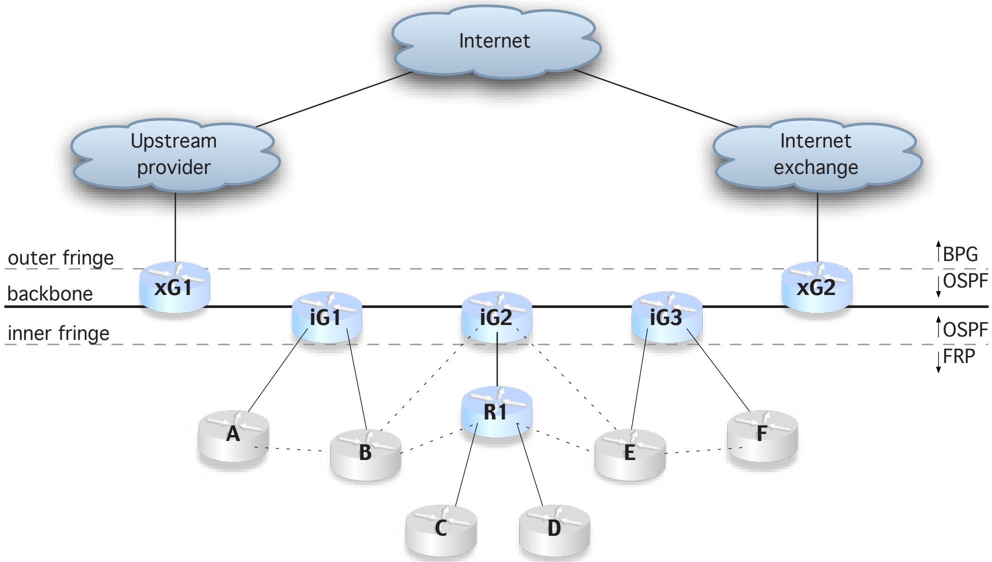


Figure 3.9
The example below illustrated

For example:

R1 is a router on the ISP fringe. It is not a gateway — its gateway is iG2 and its peers are B, C, D, E.

B's gateway is iG1, C&D's gateway is iG2 via R1, E's gateway is iG3.

Does B pass the route B,A to R1?

```
route-cost + link-cost < route-gw-cost + target-gw-cost + is-gw-route
B,A + B,R1 < A,iG1 + R1,iG2 + not-a-gateway
```

The hop count only scenario, assume all links = 1.

```
1 + 1 < 1 + 1 + 0
```

The weighted links scenario, assume solid (preferred) routes = 1 and dotted (redundant) routes = 2.

```
2 + 2 < 1 + 1 + 0
```

Does R1 pass route R1,B,A to C?

```
R1,B,A + R1,C < A,iG1 + C,R1,iG2 + not-a-gateway
```

The hop count only scenario, assume all links = 1.

```
2 + 1 < 1 + 2 + 0
```

The weighted links scenario, assume solid (preferred) routes = 1 and dotted (redundant) routes = 2.

```
4 + 1 < 1 + 2 + 0
```

In both cases, the hop count comes out equal so the backbone route is the favoured one, although another hop between R1 and its gateway would mean that the redundant route was preferred. With the link weighting, the backbone route is clearly favoured over the alternative.

FRP router configuration

A router requires a certain amount of data available to initially get up and running. Typically this might include address, interface and network information; the rules for accessing and communicating with neighbours or peers; output locations for debugging, status and logging data; and in the case of FRP, gateway information.

Configuration file

Each FRP router needs a configuration file. It stores the information required during the start up process and is where the mandatory and default operating values are set. Although each individual implementation of FRP will need to handle configuration differently in line with the system being developed for, the file that came with the prototype implementation provides an indication of what is important to FRP. It is clear that this example is a work in progress as different methods, some of which are commented out (#), are used to specify the same data. The original prototype of FRP sets all configurable values in this file, these values can not be changed once the router is up and running.

```
gateway always
debug 2
pidfile frpd.pid
#secret fred
statusfile frpd.status
#acl peers permit ip 192.168.151.0/24
#acl accepts permit ip 192.168.0.0/16-24
#listen 192.168.151.110 secret fred poll 0.5 retry 0.1 fail 2
#listen 192.168.151.110 accept-connect peers
#listen 192.168.151.110 accept accepts
listen 192.168.151.110
#peer 192.168.151.91 secret fred
#peer 1.1.1.2
override static [57]
```

In this example, the basic FRP information is

- `gateway always`, indicating that this router is a gateway
- `secret fred`, setting the secret used by this router
- `listen 192.168.151.110 secret fred poll 0.5 retry 0.1 fail 2`, setting the local address, secret, poll time, retry time and fail time
- `listen 192.168.151.110`, the address of this router
- `peer 192.168.151.91 secret fred`, a peer known to this router — its address and secret

The configuration file lines

```
#acl peers permit ip 192.168.151.0/24
#acl accepts permit ip 192.168.0.0/16-24
#listen 192.168.151.110 accept-connect peers
#listen 192.168.151.110 accept accepts
```

indicate that the original implementation allowed the use of access lists to control which peers a router can communicate with. Given that the use of 'secrets' mandates the explicit setting up of peers by an administrator (see Peers and Secrets below), this may not actually be necessary.

Some output information is also specified to support multiple levels of debugging and the creation of two output files, one is used to store the process id and the other a complete copy of the entire FRP routing table.

User commands

The original prototype of FRP sets all configurable values in the configuration file. The *Quagga* implementation requires the mandatory basics to be set there but other mandatory and optional values are pre–set as defaults in the daemon code. These values may be reset either within the configuration file or via a user command line tool while the router is up and running.

Consequently, the *Quagga* implementation of FRP needs to identify the list of configuration settings required, the default value for each setting, and whether the setting is to be turned in to a command to allow an administrator to live change the setting.

Peers and secrets

FRP peers need to be known and set up in advance. An access control list can be set up to specify which addresses are acceptable as peers but the real restraint is the need to know each individual peer's specific security information before any messages can be exchanged. Therefore each peer requires a secret to be stored in the configuration file and loaded on start up, or provided via the command line with all the other information if a peer is added while the router is up and running.

The only way to achieve this is for the secret to be set manually at each end. Consequently the secret is never sent via the network except as part of the security hash used to encode and decode packets, a good security feature if the secrets are exchanged via some trusted and secure means. This is helped by the fact that FRP is designed to be used in isolated pockets within confines of an ISP network so it may often be the case that one administrator is setting all the secrets anyway, and even if there are more than one, then they probably still work for the same organisation.

Other information also needs to be stored for each peer — the peer's address, the cost of the link between router and peer, and the poll and retry times to be set for that peer.

Sequence numbers

Each FRP router contains a random number generator that provides a sequence number each time a new conversation is started. A router keeps track of the sequence currently in use with each peer (`lseq`), incrementing the previously used number by one before embedding it in a new packet.

Each host also maintains a copy of the peer's current sequence number (`rseq`), and the two are used in tandem to catch gaps where packets have gone missing and to ensure that packets are processed in the correct order. When a packet arrives from a peer, the sequence numbers are checked to ensure that the expected number is in fact the one used. If this is not the case, then a problem has obviously occurred and the packet is not processed.

A packet arriving from a peer stating that the host's latest sequence number is 0 indicates the start of a new conversation with that peer. Consequently, if a sequence number ever naturally wraps to 0, it is automatically reset to 1.

FRP message formats

Often the concept of a 'packet' and a 'message' is essentially the same but FRP clearly differentiates between the two. This is a direct result of the fact that FRP allows multiple messages to be sent in a single packet, potentially reducing the amount of traffic flowing between FRP routers but increasing the complexity of building and processing packets.

A single FRP message begins with a message header providing the length of the entire message in bytes and a code indicating what type of message this is. The header is followed by the fields specific to that message. A FRP packet consists of a packet header containing security and sequencing information followed by zero or more messages in the above format. Packets containing just the packet header and no messages are only used in two special circumstances; generally at least one message is attached. Some message types may occur more than once in a packet.

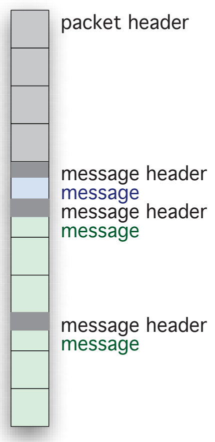


Figure 3.10
A representation of a packet containing multiple messages

Although there is no requirement for most messages to be sent in any particular order, the reality is that the linear execution of code tends to ensure that they are. Each message is however a complete and self-sufficient bundle and once extracted, can be processed independently.

The FRP packet header

The packet header sits at the beginning of each packet sent by an FRP router to one of its peers. There is exactly one packet header per packet sent, regardless of how many messages the packet contains.

Size: 128 bits / 16 bytes

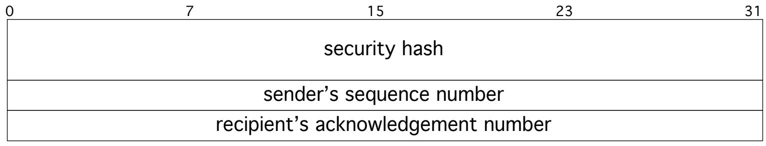


Figure 3.11
Specification of a packet header

security hash	*64 bits / 8 bytes* a hash of the local router's secret plus the contents entire packet
sender's sequence number	*32 bits / 4 bytes* the current sequence number of the local (sending) router
recipient's acknowledgement number	*32 bits / 4 bytes* the latest received sequence number from the remote router that is at the other end of the current conversation

FRP message headers

A FRP packet can contain multiple messages. Each message must be prefixed with exactly one message header specifying the type and length of the message.

Size: 16 bits / 2 bytes

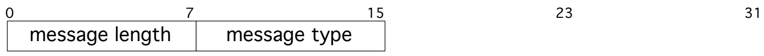


Figure 3.12
Specification of a message header

message length	*8 bits / 1 byte* the actual length of this message, including the message header, in bytes
message type	*8 bits / 1 byte* the type of the message that follows specified in one of the following hexadecimal codes: `0x01` control message `0x41` IPv4 configuration message `0x42` IPv4 path to gateway message `0x43` IPv4 route update message `0x61` IPv6 configuration message `0x62` IPv6 path to gateway message `0x63` IPv6 route update message

Session control messages

Control messages inform the receiving peer about the type of conversation in progress.

Size: 32 bits / 4 bytes

0–7	8–15	16–23	24–31
message length	message type	control type	control parameters

Figure 3.13
Specification of a control message

message header: message length	4
message header: message type	`0x01`
control type	the type of this control message is specified using one of the following numerical codes: 1 POLL 2 ACK 3 NAK POLL is used to send a message checking that a peer that has not been heard from for a period of time is still alive ACK is used to indicate that the local router received the last message of a conversation from the remote peer and is now closing the conversation down NAK is used to indicate receipt of a malformed packet
control parameters	not currently used

Static configuration messages

Configuration messages tell other peers about the state of the sending router. One is always sent at start up and then again at any other time that the router state changes.

Size: 96 bits / 12 bytes

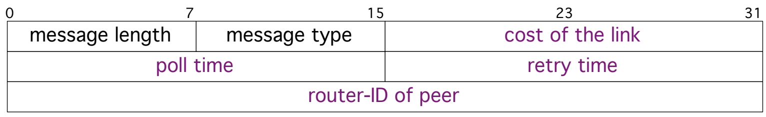


Figure 3.14
Specification of an IPv4 configuration message

message header: message length	`12`
message header: message type	`0x41`
cost of link	*16 bits / 2 bytes* the cost of the link between the local router and the remote peer
poll time	*16 bits / 2 bytes* the amount of time to wait without hearing from the remote peer before sending a POLL control message to see if the peer is still alive
retry time	*16 bits / 2 bytes* the amount of time to wait for an acknowledgement from the remote peer before resending the last packet
router-ID of peer	*32 bits / 4 bytes* the unique id of the remote peer, typically it's IPv4 address

Size: 192 bits / 24 bytes

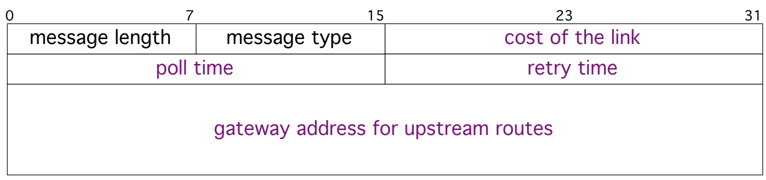


Figure 3.15
Specification of an IPv6 configuration message

message header: message length	`24`
message header: message type	`0x61`
cost of link	*16 bits / 2 bytes* the cost of the link between the local router and the remote peer
poll time	*16 bits / 2 bytes* the amount of time to wait without hearing from the remote peer before sending a POLL control message to see if the peer is still alive
retry time	*16 bits / 2 bytes* the amount of time to wait for an acknowledgement from the remote peer before resending the last packet
gateway address to use	*128 bits / 16 bytes* the unique id of the remote peer, typically it's IPv6 address

Path to gateway messages

If a FRP router is not told that it is a designated gateway router, it needs to find out the path to its nearest gateway. The peers of the router pass it 'path to gateway' messages telling it of the best path currently known by that peer. The local router can use this to choose the best gateway path for itself, which is then passed on to all its own peers (except the one who originally told it about the gateway path).

Size: from 64 to 2016 bits / from 8 to 252 bytes

0–7	8–15	16–23	24–31
message length	message type	cost from peer to gateway	
path from peer to gateway [1]			
...			
path from peer to gateway [n] (1 ≤ n ≤ 62)			

Figure 3.16
Specification of an IPv4 path to gateway message

message header: message length	`1+1+2+4 + 0` to `244` in 4 byte increments (ie: between `0` and `61` additional IPv4 addresses)
message header: message type	`0x42`
cost from peer to gateway	*16 bits / 2 bytes* cost of the link between the advertising router and it's gateway if the advertising router is a gateway, the cost is 0 if the advertising router does not know of a gateway, the cost is infinity (which is `0xffff` in FRP)
path from peer to gateway	*32 bits / 4 bytes x n (1 $\leq n \leq$ 62)* a list of 1 to 62 IPv4 addresses specifying the path taken by this router to reach it's designated gateway the first address in the list is always this router's address

Size: from 160 to 7968 bits / from 20 to 996 bytes

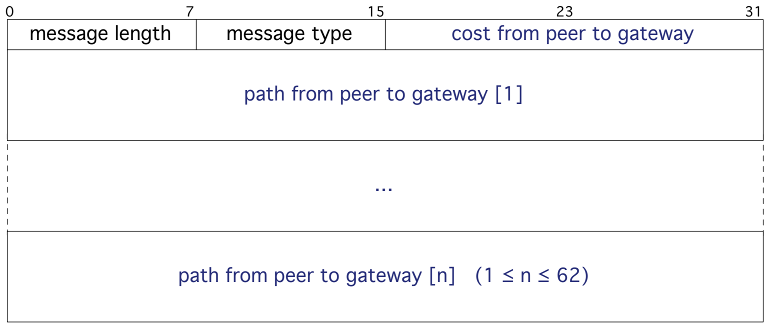


Figure 3.17
Specification of an IPv6 path to gateway message

message header: message length	`1+1+2+16 + 0` to `980` in 16 byte increments (ie: between `0` and `61` additional IPv6 addresses)
message header: message type	`0x62`
cost from peer to gateway	*16 bits / 2 bytes* cost of the link between the advertising router and it's gateway if the advertising router is a gateway, the cost is `0` if the advertising router does not know of a gateway, the cost is infinity (which is `0xffff` in FRP)
path from peer to gateway	*128 bits / 16 bytes x n (1 ≤ n ≤ 62)* a list of 1 to 62 IPv6 addresses specifying the path taken by this router to reach it's designated gateway the first address in the list is always this router's address

Route update messages

Once the local router establishes a path to a gateway, it starts to exchange routing information with other FRP peers using route update messages. It utilises the information provided by peers via these messages to build and rebuild its own routing table, selecting the best route to each known destination to store and to share with its peers.

Size: 96 bits / 12 bytes

0 – 7	8 – 15	16 – 23	24 – 31
message length	message type	update type flag	prefix length
route cost		cost from originator to gateway	
IP prefix			

Figure 3.18
Specification of an IPv4 route update message

message header: message length	`12`
message header: message type	`0x43`
update type flags	*8 bits / 1 byte* the update flags are specified by setting one or more of the following bits: `0x01` begin `0x02` commit `0x04` null `0x08` update `0x10` delete `0x80` gateway
prefix length	*8 bits / 1 byte* length of the prefix mask (a number between 0 and 32)
route cost	*16 bits / 2 bytes* cost of the new route being advertised
cost from originator to gateway	*16 bits / 2 bytes* cost of the link between the advertising router and it's gateway
IP prefix	*32 bits / 4 bytes* destination address or prefix

Size: 96 bits / 12 bytes

0 – 7	8 – 15	16 – 23	24 – 31
message length	message type	update type flag	prefix length
route cost		cost from originator to gateway	
IPv6 prefix (truncated)			

Figure 3.19
Specification of an IPv6 route update message

message header: message length	`12`
message header: message type	`0x63`
update type flags	*8 bits / 1 byte* the update flags are specified by setting one or more of the following hexadecimal codes: `0x01` begin `0x02` commit `0x04` null `0x08` update `0x10` delete `0x80` gateway
prefix length	*8 bits / 1 byte* length of the prefix mask (a number between 0 and 128)
route cost	*16 bits / 2 bytes* cost of the new route being advertised
cost from originator to gateway	*16 bits / 2 bytes* cost of the link between the advertising router and it's gateway
prefix	*32 bits / 4 bytes* destination prefix

Batch processing of update messages

Updates typically happen in batches as a router sends its current routing table to all peers, one update message at a time. The first message of an update batch carries a BEGIN flag and the final message of the batch carries a COMMIT flag. Messages in between the BEGIN and the COMMIT are assumed to belong to the same batch. Any individual route that is also a gateway route carries a GATEWAY flag as well.

There are instances where update messages do not occur as a batch. A router can tell a peer that it has nothing to share by sending a single update carrying the BEGIN + NULLRT + COMMIT flags. Modification to or deletion of an individual entry in the routing table carry BEGIN + UPDATE + COMMIT or BEGIN + DELETE + COMMIT.

To avoid processing out–of–date routes, each time a new BEGIN is received, any previously unCOMMITed batches are discarded in favour of the new arrival. If the batch sequencing is interrupted or lost for any reason, the current batch is likewise discarded.

FRP packet exchange

In common with any other routing protocol, FRP has a set of rules for the exchange of packets between FRP routers to ensure the secure delivery of messages in the correct order. The diagrams show only the packet header and the message headers. The colour coding for the different message types is used consistently throughout this document.

The packet header

The packet header holds the information that is unique to the entire packet — the sender's sequence number (or local sequence number or `lseq`), the recipient's acknowledgement number (or remote sequence number or `rseq`), and a security code created by using the SHA library to create a hash of the entire packet plus the sender's secret. The hash is duplicated by the recipient on receipt of the message. If the two do not match, the packet is discarded. This process is followed every time two FRP routers exchange packets.

A new header and the subsequent hash are produced each time a packet is constructed, although the hash is actually the last element to be generated as it requires the packet contents to be in place before it can be created.

Initial handshake

When a router comes online for the first time, it works through the list of peers provided in the configuration file. By sending a SYN packet to each one, the router determines which peers are 'alive' and lets each one know that it also is now 'alive'.

To start the initial conversation with a new peer, the initiating router creates a packet header setting the sequence number to the randomly generated starting sequence number for this peer and the acknowledgement number to 0, as the peer's current sequence is not yet known. The 0 indicates to the remote peer that this is the start of a new conversation.

This is a SYN packet and is specified as a null packet, so no messages are added to the packet header. Packet creation is now complete so the security hash can be computed and the packet sent to the remote peer.

Figure 3.20
The start of a new conversation

The remote peer takes receipt of the packet, checking both the security hash and that the initiating router is one of its peers. It responds with a null ACK packet, setting the sequence number to the next available sequence number for this router and the acknowledgement number to the sequence number sent in the SYN packet.

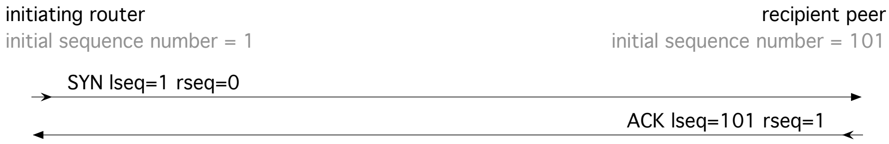


Figure 3.21
Handshake is complete

Batches of messages

The initiating router receives the ACK packet from the peer and replies, incrementing the sequence number and building a new packet setting the acknowledgement number to the one sent by the peer in the ACK packet. This response packet may contain more than one message, as the host router will scan through the information held on this peer creating messages for each of the ‘send message’ flags currently set.

Following the initial handshake, a configuration message is added to the packet header, sending the peer the initiating router’s poll and retry times, address, and the cost of the link between them. A path to gateway message is also added, sending the peer information about the router’s gateway, which, at this point, will be either “I have no gateway” or “I am a gateway”.

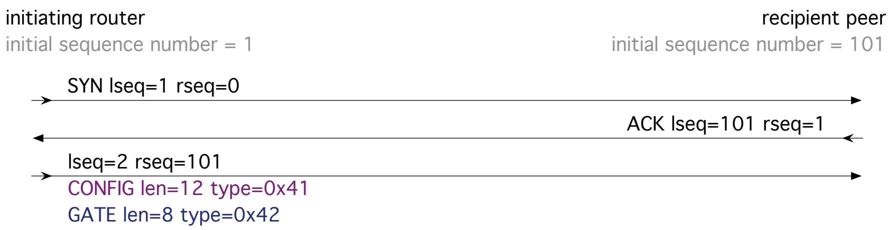


Figure 3.22
Sending more than one message in a packet

The responding peer receives the configuration and gateway messages and replies in kind by adding a configuration message and then a gateway message to the return packet. Incidentally, it is at this point that a router can begin to build a path to gateway if it needs to, using the path to gateway information it has just received.

Triggered updates

In addition to the configuration and gateway messages, the remote peer may have route updates to send the initiating router. Initially these updates are a complete exchange of routing tables. Subsequent information exchanges are triggered by one of the hosts receiving an update from elsewhere and propagating the route information on to all its peers.

The remote peer can add a batch of update messages, with the appropriate flags set, to the packet at this point.

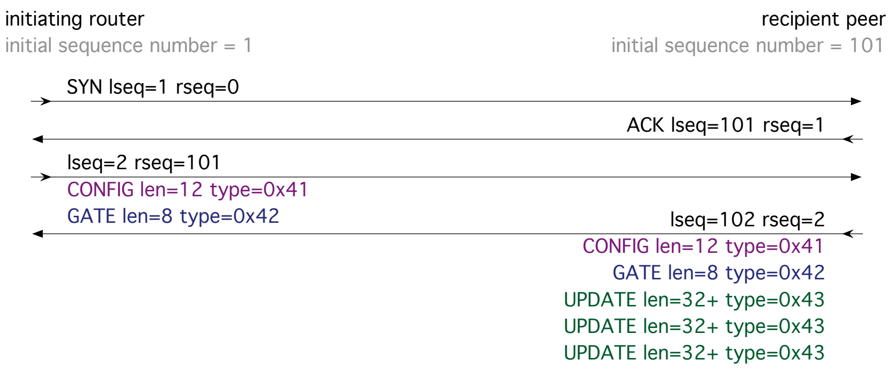


Figure X 3.23
Sending config, gateway and update messages
in response to receiving a config and a gateway message

Closing the conversation

The initiating router receives and processes the messages. If it has nothing to send in return, it closes the conversation by sending a packet header plus a control message with the type field set to ACK, letting the peer know that this exchange is over.

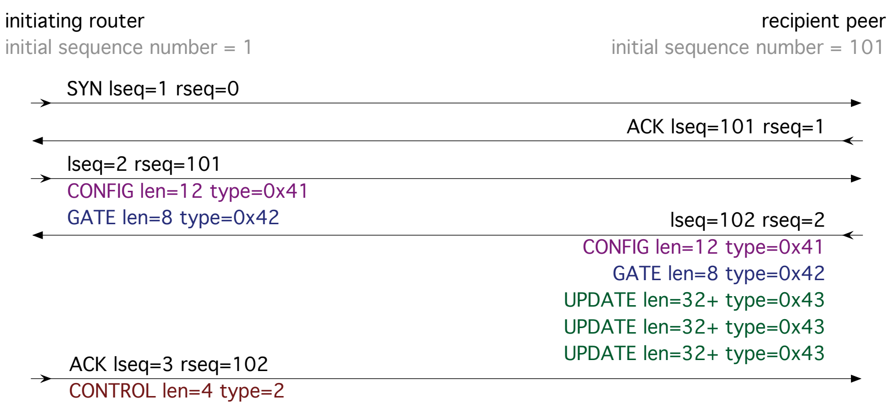


Figure X 3.24
Closing a conversation

Alternatively, if the initiating router also has route information to share, it may send its own batch of route update messages back to the remote peer. The peer is then the one to conclude the conversation by sending a packet header plus a control message with the type field set to ACK, letting the router know that this exchange is over.

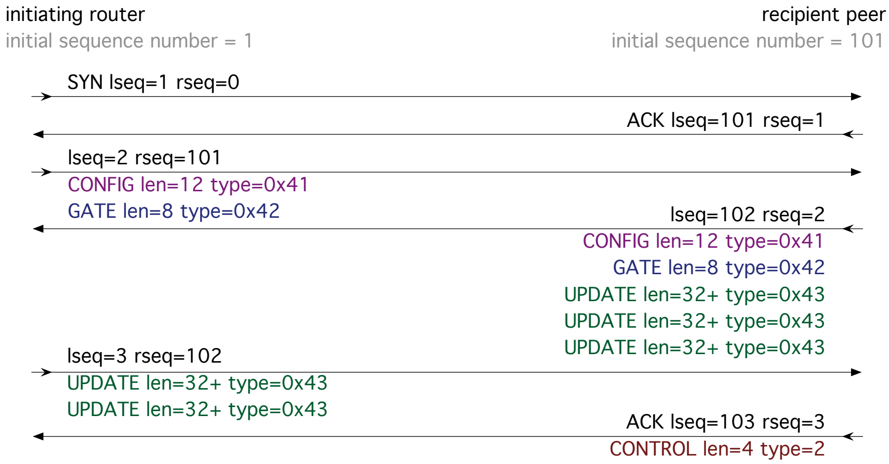


Figure X 3.25
A complete conversation

Starting a subsequent conversation

Next time the router needs to communicate with this peer, it starts the conversation process again by sending a SYN packet with the next number in the sequence used for this peer and the acknowledgement number set to 0, indicating the start of a new exchange. The peer ACKs this SYN, providing it's next sequence number. The initiating router then proceeds to send a packet of messages. The peer may respond with messages of it's own but eventually one or the other will send a final ACK to close the conversation again.

Figure X 3.26
Starting a subsequent conversation

Polling

If a peer has not been heard from for a pre–specified period of time (set as a configuration variable), then a host sends a poll message, essentially saying "Are you still there?". The host still needs to conduct the standard SYN/ACK handshake before sending a control message with the POLL control type (1) set. The correct response is a control message with the ACK control type (2) set.

Figure X 3.27
A poll conversation

Dealing with failure

A common delivery issue in many protocols is the problem of packets arriving out of order, especially when relying on UDP as the transport protocol as UDP has no delivery guarantees. The synchronous nature of the FRP message exchange provides it with the robustness it needs by insisting on a TCP style requirement that a host pause to receive an acknowledgement from the peer before sending the next packet. The synchronicity rises from FRP sending multiple messages in one packet rather than each individual message on its own.

A potential failure occurs when a router disappears and so stops replying to messages. This particular failure cannot be recovered from unless the peer reappears sometime in the future. All that can be done in the interim is to make sure the peer really has disappeared, rather than just becoming slow to respond, and then note the fact that it has gone. If a host does not receive a response to any sent packet within a pre–specified (configurable) period of time, the packet is resent and then resent again until the retry time runs out. At this point the peer is declared 'dead'.

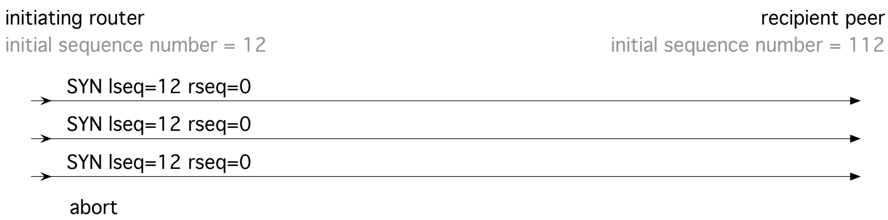


Figure X 3.28
The peer has 'died'

When the peer reappears, an initial handshake exchange updates the information held in the host's list of peers and conversations continue as usual.

Other common problems in packet delivery are detected as sequence number mismatches. In a similar situation to the one above for example, if the peer was merely having network issues causing it to slow down, it may get a message from the initiating router more than once. If the message is a SYN packet, each new one will be treated as a new conversation request, so the previous conversation request is dropped. Hopefully an ACK will get through before the host declares the peer 'dead' but if it doesn't an initial handshake will take place next time the peer has a communication for the host.

Alternatively, the host's repeated message could occur in the middle of a conversation. In this case, the first message received by the peer is replied to and any subsequent ones dropped as having the incorrect '`rseq`'. This is because the peer is expecting an incremented sequence number but the host is still ACKing the original until the peer's reply arrives — at which point a normal conversation continues.

Another possible cause of failure is a corrupt packet, either by damage to the packet in transit or by more malignant means. In this case the problem is detected when the receiving router attempts to validate the packet. As the initial security hash performed by the sending host utilises both its secret and the contents of the packet, any deliberate attempt to modify the packet will only succeed if the third party knows the secret. Otherwise, the recomputation of the hash by the recipient fails and the packet is ignored and discarded.

4. Designing the Quagga FRP daemon

This chapter covers the architecture of the *Quagga* system, the initial process of designing and building the *Quagga* FRP daemon, the architecture and structure of the daemon and the program flows necessary for the correct application of the FRP algorithm and associated operational requirements.

The design problem is to, within the confines of the *Quagga* software routing suite, create a fringe routing protocol daemon that:

- can bootstrap itself, — ie: get up and running using a configuration file and can inject any routes in that configuration into the kernel;
- can be told a set of peers via the configuration file or via a terminal, can send messages to those peers, and can receive messages from those peers;
- can extract information from the received peer messages, can make the correct decisions about building a routing table using the data in those messages, and can distribute changes in the routing table back to the peers;
- can use the routing table to correctly route traffic.

Quagga Architecture

Quagga is not a router in the physical sense of the word, in that it is not a dedicated piece of hardware, optimised to perform one specific task as quickly and efficiently as possible. It is instead, a modular, open source software suite for Unix and Linux platforms. In place of the specialist hardware, the package allows a standard Unix OS desktop to operate as a router using Quagga to facilitate the routing functions.

The Quagga command set and the syntax it uses is very similar to that used by Cisco for their routers and so is familiar to most users, although Quagga does not have the same extent of functionality provided by Cisco routers [http://sourceforge.net/apps/mediawiki/quagga/].

At Quagga's core is the Zebra daemon providing an abstraction layer using a client/server model to interface between the individual protocol daemons and the Unix kernel [www.quagga.net/about.php]. Each protocol daemon is a standalone process, maintaining its own state and routing table consistent with its protocol's algorithm. Zebra is responsible for handling the injection of routes from each individual daemon into the kernel routing table and for the redistribution of routes between the individual protocol daemons. The Zebra daemon also manages other shared functions that rely on cooperation with the kernel, such as using interfaces and sockets.

Each Quagga daemon is run as a separate and independent process on the host operating system. The Zebra process must be running before any of the protocol processes will run correctly. Individual protocol daemons may be taken up and down without affecting any other Quagga process.

Figure 4.1 shows the way Quagga separates the two major functions of a router, that of routing traffic from that of creating and maintaining a table of destinations and the directly connected nexthops. The top part of the diagram shows Quagga enabling the Unix system at the forwarding plane level where incoming traffic is accepted, the destination address consulted, and the nexthop determined. The underlying Unix kernel's routing table is used as the forwarding information base (FIB) and is consulted to determine which interface is used to send outgoing traffic on its way.

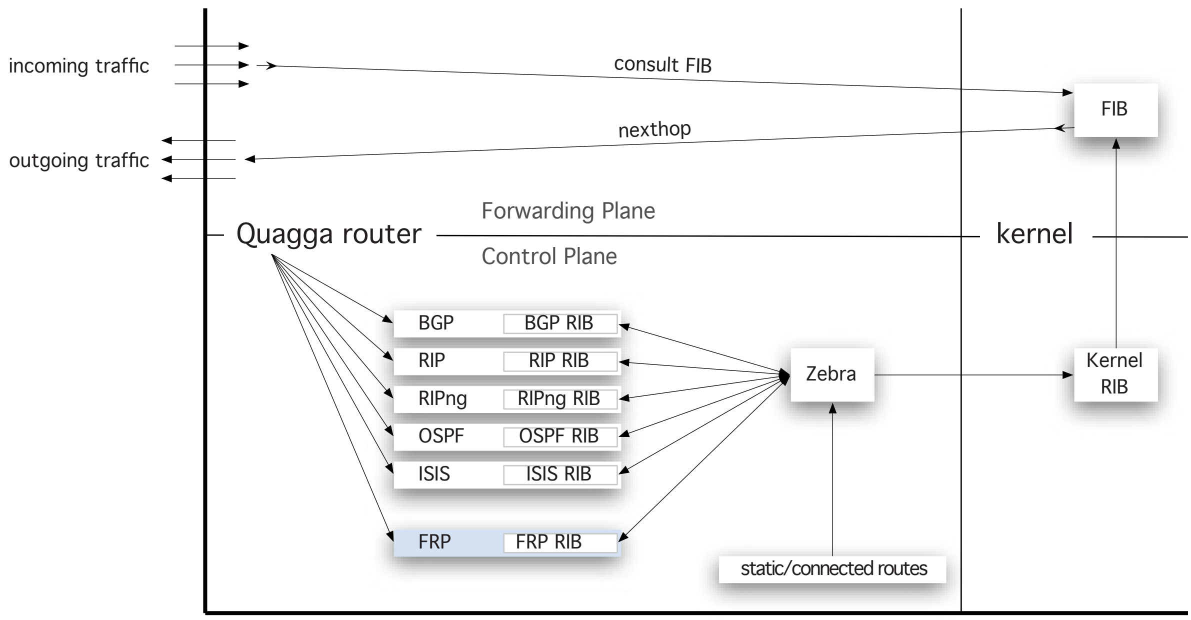


Figure 4.1
The architecture of a Quagga software router

The control plane part of the diagram shows the modular approach of *Quagga*. The individual protocol daemons, each with their own individual routing table, are clients of the Zebra server daemon that facilitates access to the Unix kernel. Protocol daemons provide their chosen 'best' routes to the kernel and to each other via the Zebra daemon. Zebra also allows user defined static and connected routes to be defined, duplicating some of the functions that can also be performed directly via the Unix OS.

Communicating with Zebra

Communication between Zebra and the protocol daemons is provided by *Quagga*'s Zserv API running over a TCP or Unix stream to the client daemons [www.quagga.net/about.php]. Knowledge of the message protocol used for this communication is not required in order to create a new routing daemon to add to the suite.

Zserv supplies zclient functions to the client daemons, allowing them to make calls to API functions that provide access to standard kernel mechanisms. Each new protocol daemon must also implement the callback functions necessary to allow Zebra to communicate with that daemon, as Zebra makes these calls to the daemon assuming that they have been implemented. These API and callback functions handle updating the router id, detecting, adding and deleting interfaces, taking interfaces up and down, adding and deleting addresses to interfaces, discovering addresses attached to interfaces, and adding and deleting routes in the kernel's routing table [60].

Configuration and commands

Like any other router, *Quagga* daemons cater to the custom needs of the network they are running on and consequently need to be configured. This can happen in one of two different ways. A minimal amount of information is required for loading at start up and this is stored in a simple, text based configuration file. Additional data may be added either to the configuration file or entered by an administrator while the daemon is running. The current running configuration may be written out to the configuration file at any point.

Re–configuring the daemon while running is facilitated by the built in virtual teletype terminal (VTY), which provides a command line interface (CLI) for issuing daemon commands. The Zebra command set provides the standard settings for the routing system — setting static and connected routes for example. A basic set is provided for each new daemon and these are fully expandable and customisable to fit the needs of the protocol. The VTY commands are loaded during daemon initialisation.

Quagga has an elegant solution for handling these configuration commands. A macro has been defined which makes adding new commands reasonably straightforward by providing a standardised framework. The format in which a command is executed is identical whether it is issued by a user or comes from the configuration file, so both methods use the same piece of code. Commands can be created anywhere in the code base and so are typically defined in the same file as the feature they relate to.

Threads and events

Quagga is a multi–threaded, event driven system. On initialisation, a daemon spawns a master thread within an infinite `while` loop which responds to events. Each daemon has its own list of events, typically the arrival of a packet, a change to the routing table, and the triggering of a poll timer but there is scope for other events to be included.

Within the main daemon engine, the initialisation sequence triggers the first instance of each event. The event thread is created and the appropriate function attached. The thread then sleeps until that event is triggered at which point the thread wakes and the code is executed. The first action within the function is to create a new event thread, attach the function to it, and set to wait for the next trigger.

Library support

The Quagga suite supplies a good set of library files to provide support for protocol daemon development [www.quagga.net/about.php]. These include zclient.h/c, which provides the Zserv API mentioned above and zebra.h with the global defines, macros, external variables and prototypes made available by the Zebra daemon.

Other library files provide support for:

- the management of memory, signal handling, threads, logs, and privileges;
- creating user commands, extracting them from the configuration file, and executing them in the VTY
- working with networks, interfaces, and sockets;
- all manner of routing related functions like route filtering and distribution, and working with prefixes, routemaps and tables;
- the standard program requirements of buffers, hash tables, linked lists, strings, vectors and the network.

Initial process

The first step towards building a FRP daemon was to get to grips with the *Quagga* code and see how a protocol daemon hooked in to the Zebra daemon and the associated libraries. RIPng is the smallest of the existing *Quagga* protocol daemons and the name also supplies a good, unique search string. So RIPng was duplicated and turned in to an instant 'FRP' daemon simply by running a global find/replace of '*ripng*' with '*frp*' within the duplicated directory, then recompiling and working through the errors generated.

Once the new 'FRP' daemon compiled in isolation, the next step was to get the Zebra daemon to recognise it. This became an extensive hunt through the Zebra and library code looking for every instance of RIPng, duplicating the necessary code and updating it for the new daemon — which at this point mainly meant changing RIPng to FRP. Later on, it required analysing what was happening at that point and deciding what modifications were required for the real FRP daemon. It also became apparent that looking at more than one protocol was necessary, so RIP and BGP searches were conducted as well. Once again, this method eventually came down to constantly recompiling and working through the errors generated. The overall technique yielded a good indication of where the Zebra daemon, the library code, and the `make` and `configure` files need to be modified, adjusted and added to in order to support a new protocol daemon.

The other major unknown was how a *Quagga* daemon behaved at the basic level before any routing protocol specific code was added. The initial idea was to take the RIPng version of FRP and remove all the RIPng specific code, leaving a basic daemon shell for the FRP protocol to be built on top of. However, the fact that RIPng is an IPv6 only protocol started to cause problems. This left a choice between RIP, which as it is IPv4 only, was also not ideal but easier to work with than RIPng, and BGP, which was considerably more complex. RIP was the obvious choice and a new 'FRP/RIP' daemon was created and compiled. The technique ultimately failed however, as it became too difficult to separate the RIP protocol specific code from the generic daemon code.

Going through the above process was necessary but had left the code base in an irreparable mess. A clean start was required. This time the modifications to the Zebra daemon and the libraries were systematically worked through (see Appendix C) before a new daemon was built from the ground up using the *Zebra for dummies* [60] document as a guide and referring to the RIP, RIPng and BGP code as necessary. The daemon was created in two stages, the first being the creation of a shell and the second being the addition of the FRP protocol.

In the creation of the shell, the RIP, RIPng and BGP code base became an extremely useful resource and in many cases, code for basic functionality was lifted directly from one or another of them. The RIP/RIPng combination was invaluable as a guide to the differences between IPv4 and IPv6 whereas BGP showed how to combine IPv4 and 6 in one daemon.

When reading the FRP daemon code included in this document, note that as *Quagga* only uses the `/*comment*/` style of comments, `/*comment*/` denotes code taken directly from an existing daemon and `//comment` indicates new code.

Code structure

The Quagga protocol daemons mostly adhere to a common code file structure, although there is some diversity between them. Inside the main Quagga directory, a lib directory holds shared code and a zebra directory stores the implementation of the Zebra daemon. Each individual Quagga protocol daemon has its own directory called *protocol*. The initial code file is called *protocol*_main.c that calls *protocol*d.c, and it is this file that contains the core daemon code. Both files use the same header, *protocol*d.h, which holds all the core datastructures and declarations. Other files are created as necessary using a *protocol*_ prefix. Interaction with Zebra happens inside *protocol*_zebra.c and other common suffixes appear in multiple daemons — _debug, _interface, _peer, _route and _routemap for example. The FRP daemon was designed to fit in to this structure.

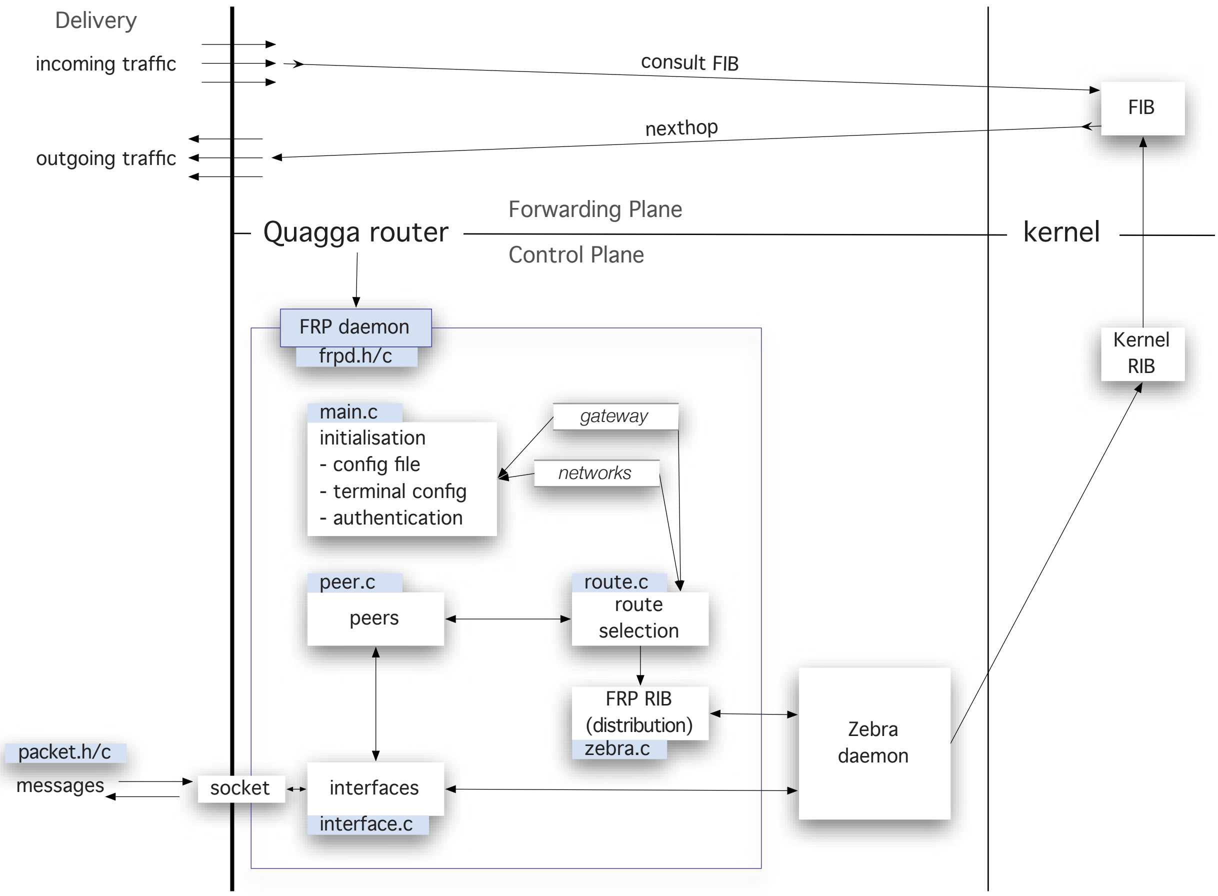


Figure 4.2
The FRP daemon architecture
(note that the file name frp_ prefixes have been dropped)

frp_main.c

The *protocol*_main.c files are not only common to all Quagga daemons but the code is very similar in each case. It is here that the set up and initialisation of the system occurs. The main program:

- ‘includes’ the frpd.h file that sets up many of the data structures and global variables
- sets the daemon’s privileges
- initialises debug logging
- extracts and executes the command line arguments used when the daemon was booted
- creates and sets up the master thread
- calls the system level initialisation functions — zebra privileges, signal handlers, commands, VTY, memory
- calls the FRP and Zebra (zclient) daemon initialisation functions
- installs and sorts the VTY commands
- reads the config file and executes the commands
- changes to running in daemon mode
- creates VTY socket and starts the socket listeners
- creates the pid file
- starts up the main program loop, an infinite while loop in which the master thread continuously fetches and executes the next thread

frp_zebra.c

The primary function of the frp_zebra.c file is to initialise zclient and to set up the callback functions for the Zserv API. This file is present in all the Quagga protocol daemons. The callback functions implemented in FRP are the following:

```
int (*interface_add) (int, struct zclient *, uint16_t);
int (*interface_delete) (int, struct zclient *, uint16_t);
int (*interface_up) (int, struct zclient *, uint16_t);
int (*interface_down) (int, struct zclient *, uint16_t);
int (*interface_address_add) (int, struct zclient *, uint16_t);
int (*interface_address_delete) (int, struct zclient *, uint16_t);
int (*ipv4_route_add) (int, struct zclient *, uint16_t);
int (*ipv4_route_delete) (int, struct zclient *, uint16_t);
```

which need to be mapped to the functions that actually implement them in the appropriate files. Two are implemented here in the function frp_zebra_read_ipv4.

```
zclient->ipv4_route_add = frp_zebra_read_ipv4;
zclient->ipv4_route_delete = frp_zebra_read_ipv4;
```

Two more callbacks need to be added to complete the daemon:

```
int (*ipv6_route_add) (int, struct zclient *, uint16_t);
int (*ipv6_route_delete) (int, struct zclient *, uint16_t);
```

frp_interface.c

Much of the interface code is common to all Quagga daemons and handles anything to do with network interfaces. The interface initialisation function is defined here, as is the VTY enable FRP network command. The rest of the functions define the many requirements of using interfaces including bringing them up and down, creating, enabling and checking them, attaching addresses, networks and peers to them, and looking up information about them.

The interface callback functions are implemented and linked to the Zebra daemon here:

```
zclient->interface_add = frp_interface_add;
zclient->interface_delete = frp_interface_delete;
zclient->interface_up = frp_interface_up;
zclient->interface_down = frp_interface_down;
zclient->interface_address_add = frp_interface_address_add;
zclient->interface_address_delete = frp_interface_address_delete;
```

frp_debug.h and frp_debug.c

The Zebra daemon and the Quagga libraries handle a wide range of debugging options that can be hooked into as necessary. Output to screen, terminal or log file is handled by embedding zlog_debug statements into the daemon code. Individual protocol daemons can also specify their own custom debug statements to be called and displayed via the zlog_debug system. FRPs small amount of debugging code resides here and consists predominantly of VTY commands used to turn various debug levels on and off.

frpd.h

In common with other Quagga prototype daemons, the main repository of FRPs external and global includes, #defines, variables, data structures, and function prototypes. One of the key structures is struct frp which holds information about the FRP router including its routing table and its gateway status. Another is struct frp_peer which stores data on a single peer, each one being held in a linked list called frp_peers. There is only one struct frp and only one list frp_peers. Both are specified as prototypes here.

Also defined here is struct frp_route which stores a single route given by a peer, a collection of which makes up the stored copy of that peer's forwarding table, and struct frp_info which holds meta data about the router's routing table and is used to handle access to it via the systems set up by Quagga.

frpd.c

This is the main daemon file containing much of the code specific to running the daemon. frpd.c handles the daemon's internal setup and configuration including creating the struct frp instance, creating and setting up the socket for FRP to run on, and adding the FRP specific commands to the VTY interface. This file also manages re–writing the configuration file on demand, once gain via the VTY interface.

The bulk of the code stored here however, controls the mechanics of running the fringe routing protocol. FRP's implementation of Quagga's triggered event driven system is defined here, as are the functions that handle each individual event.

frp_packet.h

The repository of all information relating to FRP messages and packets. This file defines all the different sizes, types and flags required, and creates all the data structures needed to build a FRP packet. Data structures are defined for a packet header, a message header, and for each of the different message types.

frp_packet.c

Drawing on the data defined in frp_packet.h, frp_packet.c defines the specific message and packet handling functions. These are called on to make up individual messages dependent on the message type provided and to build FRP packets out of one or more messages and a packet header. The frp_send_packet function, which is mostly made up of code taken from other protocol daemons, also resides here.

frp_peer.c

Functions for the creation, initialisation, handling and support of peers, including the associated VTY commands. Where overlaps occur, code has been lifted from other protocol daemons. Of particular interest are the two security functions, `dohash` and `checksecure`, which are taken directly from the original Knossos implementation of FRP.

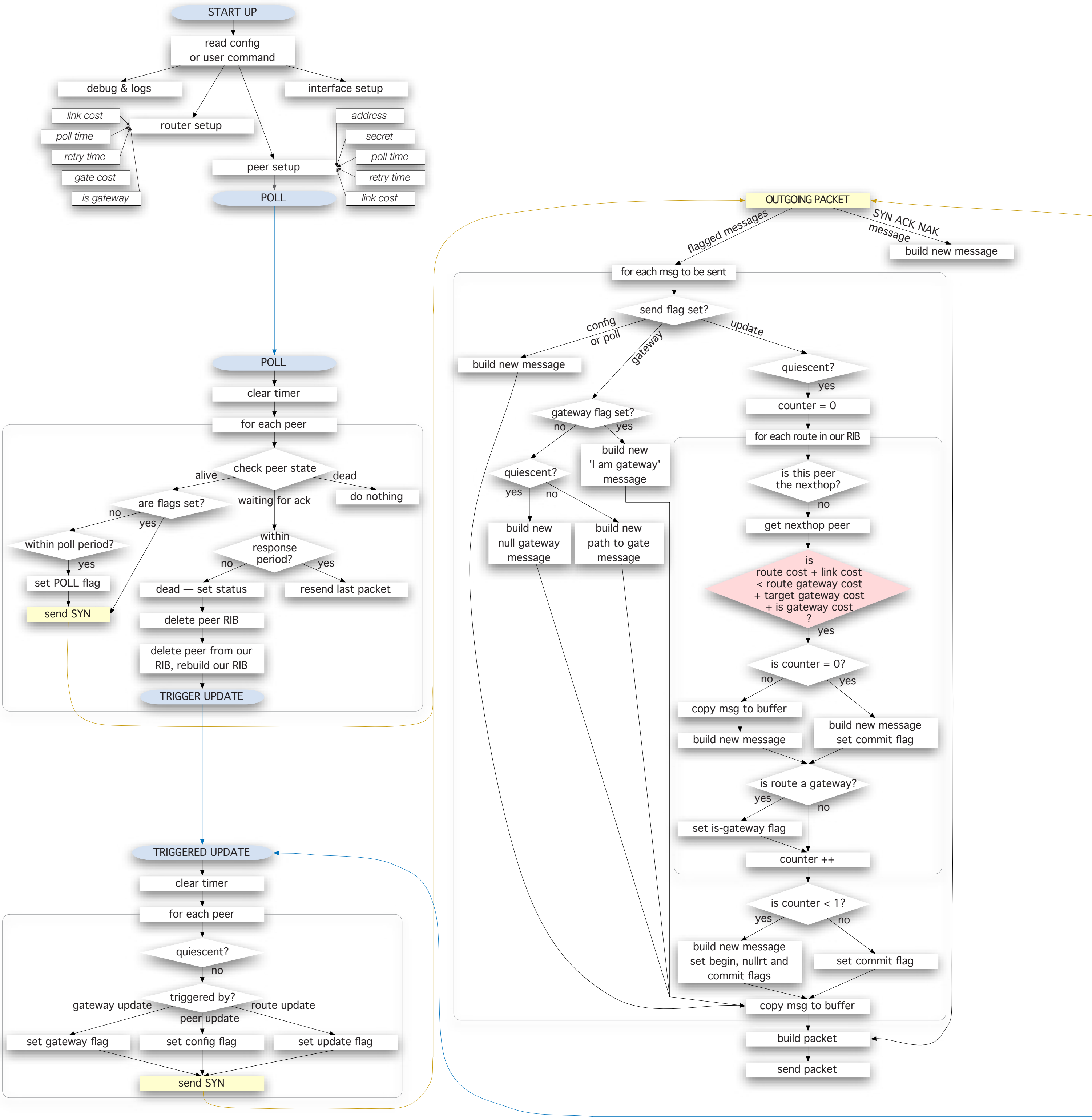
frp_route.c

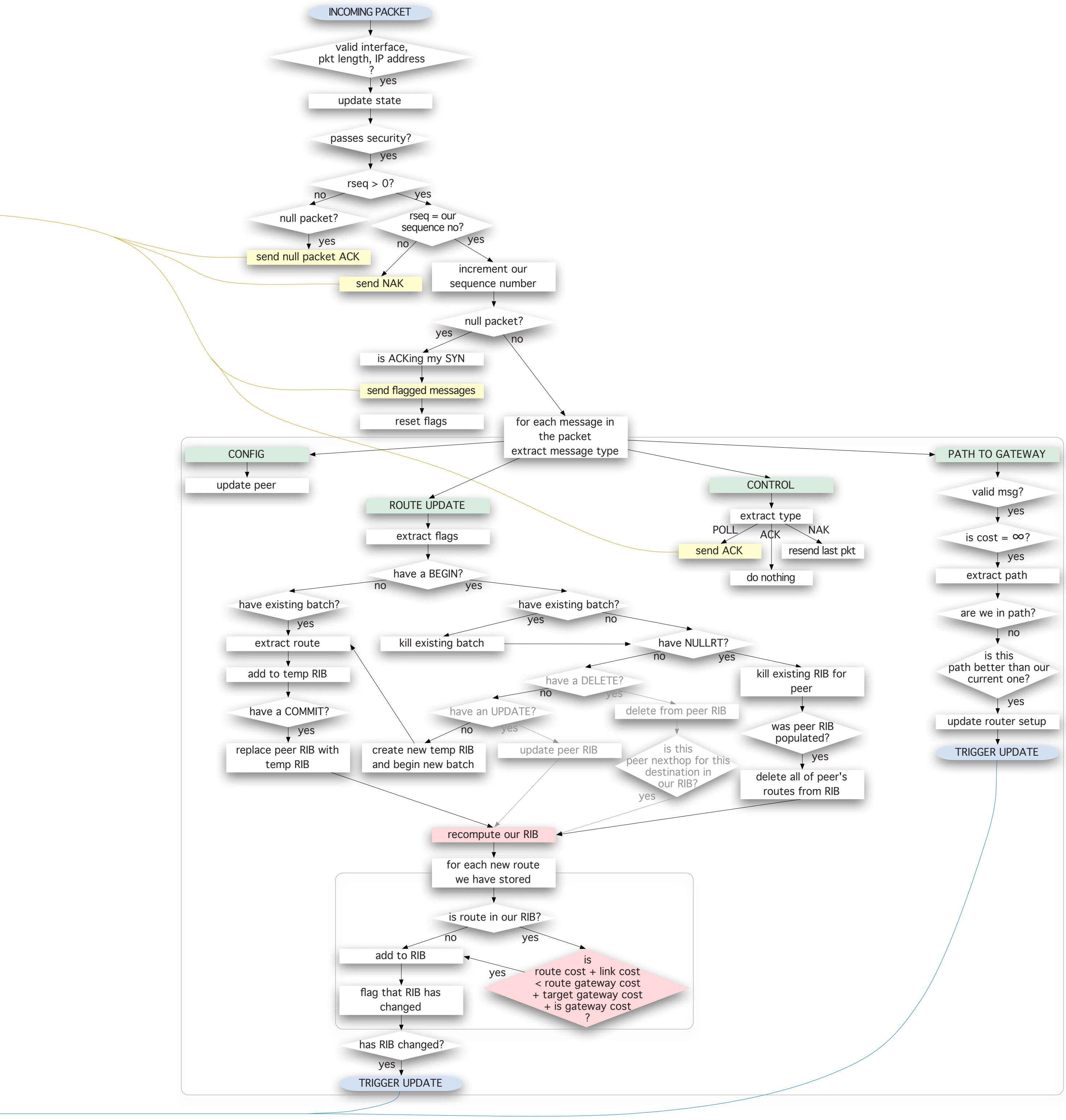
Quagga supplies many of the mechanisms required to handle the FRP routing table. The FRP daemon only needs to set up the necessary data structures and make calls to the relevant library functions. frp_route.c holds the functions that address the specific FRP routing mechanisms and apply the FRP decision algorithm.

Program flow

Once the empty daemon shell was working correctly, the fringe routing protocol itself could be built into it, necessitating a complete change of direction in the development phase. The Quagga system is a framework providing an API for basic or common router functionality — the mechanics of building the routing table for example. Any FRP specific functionality — deciding what actually goes into the routing table for instance — needed to be written from scratch. The design for this functionality is shown over the page. It shows the program flows necessary for the correct application of the FRP algorithm and associated operational requirements. The implementation of this FRP daemon design is described in the following section.

Figure 4.3
FRP daemon program flows





INCOMING PACKET
valid interface, pkt length, IP address ?
yes
update state
passes security?
yes
rseq > 0?
no
yes
null packet?
yes
send null packet ACK
rseq = our sequence no?
no
yes
send NAK
increment our sequence number
null packet?
yes
no
is ACKing my SYN
send flagged messages
reset flags
for each message in the packet extract message type
CONFIG
update peer
ROUTE UPDATE
extract flags
have a BEGIN?
no
yes
have existing batch?
yes
extract route
add to temp RIB
have a COMMIT?
yes
replace peer RIB with temp RIB
have existing batch?
yes
no
kill existing batch
have NULLRT?
no
yes
have a DELETE?
no
yes
have an UPDATE?
no
yes
create new temp RIB and begin new batch
update peer RIB
delete from peer RIB
is this peer nexthop for this destination in our RIB?
yes
kill existing RIB for peer
was peer RIB populated?
yes
delete all of peer's routes from RIB
CONTROL
extract type
POLL
ACK
NAK
send ACK
resend last pkt
do nothing
PATH TO GATEWAY
valid msg?
yes
is cost = ∞?
yes
extract path
are we in path?
no
is this path better than our current one?
yes
update router setup
TRIGGER UPDATE
recompute our RIB
for each new route we have stored
is route in our RIB?
no
yes
add to RIB
yes
is route cost + link cost < route gateway cost + target gateway cost + is gateway cost ?
flag that RIB has changed
has RIB changed?
yes
TRIGGER UPDATE

5. Implementing and testing the Quagga FRP daemon

This chapter describes in detail the implementation and testing of the fringe routing protocol as a *Quagga* routing daemon. It covers the daemon start up, peers, events, polling and triggered updates. It describes how an incoming packet can trigger the sending of the different types of FRP messages — control, configuration, gateway, and route updates — and describes how route updates force a recomputation of the routing table. Finally, the process of building outgoing messages and packets is explains. The IPv4 version of the daemon is used as the example throughout the section.

Daemon start up

On start up, the FRP daemon, like all the other *Quagga* protocol daemons, is either launched from a startup script or from the Unix command line using the standard Unix launch command plus any of the additional arguments built in to the FRP implementation. For example:

```
sudo ./frpd -d -f /opt/local/etc/quagga/frpd.conf
```

Here, `-d` is specifying that the daemon should be launched in daemon mode and `-f` is providing the path to the configuration file. The start up arguments are implemented in `frp_main.c` and allow customisation of various settings including the VTY port and address, and the user and group names.

```
struct option longopts[] =
{  { “daemon”,       no_argument,        NULL,  ‘d’},
   { “config_file”,  required_argument,  NULL,  ‘f’},
   { “pid_file”,     required_argument,  NULL,  ‘i’},
   { “dryrun”,       no_argument,        NULL,  ‘C’},
   { “help”,         no_argument,        NULL,  ‘h’},
   { “vty_addr”,     required_argument,  NULL,  ‘A’},
   { “vty_port”,     required_argument,  NULL,  ‘P’},
   { “retain”,       no_argument,        NULL,  ‘r’},
   { “user”,         required_argument,  NULL,  ‘u’},
   { “group”,        required_argument,  NULL,  ‘g’},
   { “version”,      no_argument,        NULL,  ‘v’},
   { 0 }
};
```

The header file `frpd.h` contains most of the daemon's external function prototypes and global variables. It also specifies a number of defines to hold various initialisation values required to set up the daemon as a FRP router.

```
/* FRP version number. */
#define FRP_VERSION          1
/* Default config file name */
#define FRP_DEFAULT_CONFIG   "frpd.conf"
/* FRP ports */
#define FRP_PORT_DEFAULT     343
#define FRP_VTY_PORT         2609
// frp router defaults
#define FRP_DEFAULT_COST     1
#define FRP_DEFAULT_POLL     5
#define FRP_DEFAULT_RETRY    1
#define FRP_INFINITY         0xffff
// set to determine how many polls to wait before declaring a peer
'dead'
#define FRP_PEER_DEAD        5
```

The version number is for compatibility with other *Quagga* daemons but is not really used in FRP at present. The name of the configuration file follows existing *Quagga* convention and is stored in the default directory set by *Quagga*. This is different on each platform and is configurable in *Quagga* itself.

The FRP port and the VTY port are necessary settings. `343` is the port number chosen by Don and used in his implementation of FRP. *Quagga* is currently using ports `2600` to `2608` for Zebra and other protocol demons so port `2609` was chosen for FRP. It needs to be noted that neither of these port choices are official IANA assigned numbers.

There are a number of parameters required to set up a FRP router and these are stored in the data structure `frp` defined in `frpd.h`.

```
struct frp
{  int                             version;
   /* frp output buffer */
   struct stream*                  obuf;
   /* frp socket */
   int                             sock;
   // secret
   const char*                     secret;
   // cost of the link (16 bits)
   u_short                         cost;
   // poll time (16 bits)
   u_short                         poll;
   // retry time (16 bits)
   u_short                         retry;
   /* frp routing information base (linked list) */
   struct route_table*             rib;
   /* frp only static routing information (linked list) */
   struct route_table*             routes;
```

```
  /* frp neighbors (linked list) */
  struct route_table*              neighbors;
  // gateway types
#define FRP_GATEWAY_ALWAYS         0
#define FRP_GATEWAY_YES            1
#define FRP_GATEWAY_NO             2
  // is this peer a gateway?
  enum flag                        is_gateway_flag;
  /* frp gateway path (linked list) */
  struct list*                     gateway_path;
  // which peer is the current next hop to the current gateway?
  // - use this to trigger update if this peer changes its path to
gateway
  struct frp_peer*                 gateway_nexthop;
  // number of hops to gateway
  int                              gateway_cost;
  /* frp threads */
  struct thread*                   t_read;
  struct thread*                   t_poll;
  struct thread*                   t_update;
  struct thread*                   t_update_interval;
  int                              update_trigger;
  /* timer values. */
  unsigned long                    update_time;
  unsigned long                    timeout_time;
  unsigned long                    garbage_time;
};
```

This data structure stores the basic information the router needs to interact with peers; the secret to be exchanged; the poll interval for contacting peers to check they are still there and the retry interval for contacting peers that have not responded for a period of time. It stores gateway information; if this router is a gateway, the current path to the currently designated gateway (if there is one), the nexthop peer in that path and the cost to reach it. The threads and the timers are used to handle events. Note that `struct route_table* routes` is required so that Zebra can handle static routes and that `struct route_table* neighbors` has been left in at present because it is not clear whether Zebra also requires it to be here, despite the fact that the FRP daemon is handling peers in a different manner.

The `main` function in `frp_main.c` initialises the daemon and sets many of it's defaults, setting up the basic router configuration by reading and executing the commands written in the configuration files. The basic, non–FRP router configuration requirements are set via the Zebra daemon.

For example:

```
! Zebra configuration
hostname sapphire
password zebra
enable password zebra
log file zebra.log
!
interface en0
  ip address 10.0.1.20/24
```

At the barest minimum, the FRP configuration file must contain a host name for the router and a password for accesses that host. Zebra typically encrypts passwords but the ones shown here are in clear text. A destination for logging output (typically stdout) is likewise usually specified. To set up the FRP router, specify the network range(s) it can route traffic to, state whether it is it a gateway (default is no) and set the gateway cost to `1` if it is (default is infinity, or `0xffff` if it isn't). State what secret this router will it share with its peers. If the `linkcost` (1), `poll` (60sec) and `retry` (60sec) times are different to the defaults, set them here as well. Finally, set up any known peers in the format:

```
neighbor address secret theirsecret
```

For example:

```
! FRPd configuration
hostname sapphire
password zebra
log file frp.log
log stdout
!
router frp
  network 10.0.1.0/24
  network en0
  gateway yes
  gatecost 0
  secret sapphire
  cost 2
  poll 30
  retry 30
  neighbor 10.0.1.50 secret artemis
  neighbor 10.0.1.60 secret emerald
```

Note that the American spelling of ‘neighbor’ has been used to stay consistent with *Quagga* usage.

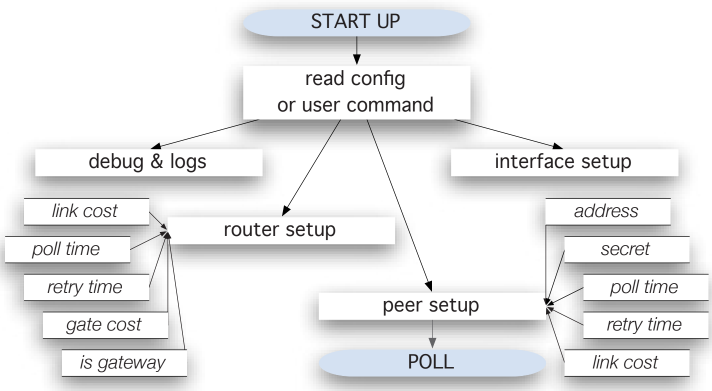


Figure 5.1
Start up flow

The `main` function also calls the functions that initialise the FRP `zclient` and set up the Zebra callback functions, as well as installing all the FRP specific commands in the VTY. Debugging is initialised via a call to a function in the `frp_debug.c` file, as is interface initialisation via `frp_interface.c`. In both these cases, the code is essentially the same as all the other *Quagga* protocol daemons, suitably modified for FRP. Peer initialisation, beyond that handled by the configuration file, is simply setting up data structures, including the `struct frp_peer` that holds the list of peers, and VTY commands at this point in the process. Additional information about peers is gathered immediately after start up is complete by triggering a POLL event that works through the peer list contacting each one.

The following commands were modified or added to the daemon menu system (highlighted in red):

```
frpd(config)#
  banner     Set banner string
  debug      Debugging functions (see also 'undebug')
  enable     Modify enable password parameters
  end        End current mode and change to enable mode.
  exit       Exit current mode and down to previous mode
  help       Description of the interactive help system
  hostname   Set system's network name
  line       Configure a terminal line
  list       Print command list
  log        Logging control
  no         Negate a command or set its defaults
  password   Assign the terminal connection password
  quit       Exit current mode and down to previous mode
  router     Enable a routing process
  service    Set up miscellaneous service
  show       Show running system information
  write      Write running configuration to memory, network, or terminal
frpd(config)# router frp
frpd(config-router)#
  cost       Cost of link
  end        End current mode and change to enable mode.
  exit       Exit current mode and down to previous mode
  gateway    Is this router a FRP gateway? (yes|no)
  help       Description of the interactive help system
  list       Print command list
  neighbor   Specify a neighbor router
  network    Enable routing on an IP network
  no         Negate a command or set its defaults
  poll       Poll / keepalive frequency
  quit       Exit current mode and down to previous mode
  retry      Timeout to failure after acked packet
  secret     Specify secret
  show       Show running system information
  write      Write running configuration to memory, network, or terminal
frpd(config-router)#
```

Peers

Peers (or neighbours) are other FRP routers directly connected to the host router. Therefore peers are known not only by their network address but also by the interface they are connected on. Due to the security of the shared secrets, peers are only ever set up by manual means — there is no automated polling required.

Two pieces of information about a peer are required: the address (which must be matched to an interface) and the secret, which is to be shared for the security hashes during packet exchange. A peer must be specified in either the start up configuration or via the command line, as the secret cannot be entered in any other way. Any additional information is sent by the peer during the first communication exchange and then stored.

A list of all peers known to a host is stored in the linked list

```
struct list* frp_peers = NULL;
```

which is created in `frp_peer.c` and initialised

```
frp_peers = list_new ();
```

in the function `frp_peer_init` in the same file. A prototype

```
extern struct list* frp_peers;
```

is listed in `frpd.h` to ensure there is only one list in existence.

`frp_peers` holds a list of peer records, one for each peer known. These records are data structures containing all the information held on the peer.

```
struct frp_peer
{ /* peer address */
  struct in_addr        address;
  const char*           secret;
  u_short               cost;
  u_short               poll;
  u_short               retry;
  // our current sequence number with this peer
  u_int32_t             lseq;
  // this peer's current sequence number
  u_int32_t             rseq;
  // current path to gateway for this peer
  struct list*          gateway_path;
  // cost of gateway, ie: number of hops in path to gateway
  int                   gateway_cost;
  // temporary storage for new incoming routes
  struct list*          rib;
  struct list*          temp_rib;
  // latest packet sent to this peer
  u_char*               packet_latest;
  u_int8_t              packet_latest_length;
  u_int32_t             packet_latest_lseq;
```

```
        // latest timer timestamps
        time_t                  time_latest_packet;
        time_t                  time_last_heard;
        time_t                  time_sent_config;
        /* timeout thread */
        struct thread*          t_timeout;
        // flags
        enum flag               flag_alive;
        enum flag               flag_send_syn;
        enum flag               flag_send_poll;
        enum flag               flag_send_config;
        enum flag               flag_send_gateway;
        enum flag               flag_send_update;
        enum flag               flag_awaiting_ack;
    };
```

Many of the variable names make it obvious what information is being stored. Of particular interest is the use of `lseq` and `rseq`. These are the terms used in the initial FRP specification and stand for 'local sequence' and 'remote sequence'. Local always equates to the router whose perspective is current, the host or 'me' or 'us'; and remote equates to the peers of this router, or 'them'. Thus, `lseq` is 'our' sequence and `rseq` is 'their' sequence. This perspective shifts as the focus shifts to another router or host. Each host maintains a separate local sequence, or `lseq`, with each individual peer.

The linked list `rib` is the routing table as supplied by the peer. `temp_rib` is the temporary storage used when changes to that routing table are received from the peer. The new routing table is built piece by piece in `temp_rib` and then copied into `rib` when confirmed as complete and correct.

The latest packet sent to a peer is stored so that it can be easily resent if a 'no acknowledgement' (NAK) is received indicating that the packet has not arrived at the peer, or if the peer does not acknowledge (ACK) the packet within a specified period of time.

The timers are required to implement polling and keepalives and the timeout thread is part of *Quagga*'s thread management system.

The flags are an enumerated boolean

```
enum flag
{  OFF,
   ON,
};
```

which is turn `ON` to indicate which message types are ready to be sent the next time a packet is put together, except for `flag_alive` which is turned `OFF` to indicate that a peer has not been heard from for a certain period of time and subsequently pronounced 'dead'.

The FRP daemon in action

On start up, a FRP host works through its list of peers passing the necessary configuration information. The following *FRPsniffer* trace shows a standard configuration exchange between two *Quagga* FRP daemons, 10.0.1.20 and 10.0.1.50. Both peers are quiescent so no route information is exchanged.

20 starts the sequence with the initial handshake by sending a SYN packet header with no message (a null packet), telling 50 what 20's current sequence number is and indicating that is a new conversation by setting the recipient's sequence number to 0. Note the security hash which 20 has created using its secret.

```
FRP PACKET HEADER
     source address: 10.0.1.20 (a.0.1.14)
destination address: 10.0.1.50 (a.0.1.32)
      security hash: ??;??
    sender's seq no: 1 (0x1)
 recipient's ack no: 0 (0x0)
--SYN Packet

Waiting for incoming FRP packet
```

50 continues the initial handshake by sending a null ACK packet header saying "acking your 1, my next sequence number is 2". Note that for 50 to reach the point of sending this response, the initial packet security hash must has been successfully reversed using 50's stored copy of 20's secret. This new packet is encrypted with 50's secret.

```
FRP PACKET HEADER
     source address: 10.0.1.50 (a.0.1.32)
destination address: 10.0.1.20 (a.0.1.14)
      security hash: ??'L?q
    sender's seq no: 2 (0x2)
 recipient's ack no: 1 (0x1)
       Null Message: 0x0
--ACK Packet


Waiting for incoming FRP packet
```

The initial handshake is now complete so 20 can send the configuration message that triggered this conversation. First comes the packet header — note the incremented sequence number; "acking your 2, my next sequence number is 2". Then comes the config message (type `0x41`) providing the cost, poll and rety values, plus 20's id which in this case is its address.

```
FRP PACKET HEADER
     source address: 10.0.1.20 (a.0.1.14)
destination address: 10.0.1.50 (a.0.1.32)
      security hash: d??????N
    sender's seq no: 2 (0x2)
 recipient's ack no: 2 (0x2)

IPV4 Config message: 0x41
               cost: 3
               poll: 60
              retry: 60
          router-id: 10.0.1.20

Waiting for incoming FRP packet
```

50 receives the message and discovers that their 'send config' flag is set for 20. The packet is constructed — header plus config message containing data — and sent.

```
FRP PACKET HEADER
     source address: 10.0.1.50 (a.0.1.32)
destination address: 10.0.1.20 (a.0.1.14)
      security hash: q>??r???
    sender's seq no: 3 (0x3)
 recipient's ack no: 2 (0x2)

IPV4 Config message: 0x41
               cost: 10
               poll: 60
              retry: 60
          router-id: 10.0.1.50

Waiting for incoming FRP packet
```

20's response contains a packet header and a control message of type 'ack' indicating that, as far as 20 is concerned, this conversation is over.

```
FRP PACKET HEADER
      source address: 10.0.1.20 (a.0.1.14)
 destination address: 10.0.1.50 (a.0.1.32)
       security hash: ??!?H?
     sender's seq no: 3 (0x3)
  recipient's ack no: 3 (0x3)

Control message: 0x1
--Control ACK, param = 0
```

Events

FRP's event handler, function `frp_event` in `frpd.c`, recognises three different events: a poll, an update triggered by a change to the routing table and an incoming packet. These are defined in `frpd.h` as an enumerated list.

```
enum frp_event
{  FRP_EVENT_INCOMING,
   FRP_EVENT_UPDATE,
   FRP_EVENT_POLL,
};
```

Outgoing packets are not handled as an `frp_event` because they are created as required in response to one of the three listed events.

As in the other *Quagga* protocol daemons, a call to each event is triggered during initialisation and an event thread is created and packaged up with the appropriate function code and data. Subsequent event calls create a new thread for the next event of that type to use. This process starts in `frp_event` where, to use `FRP_EVENT_INCOMING` as an example, a new read thread is created and given all the necessary data required to run independently — the thread that spawned it, the code it will run and the current socket.

```
void frp_event (enum frp_event event, int sock)
{  switch (event)
   {  case FRP_EVENT_INCOMING:
         // create a new read thread and tell it to run frp_incoming_
                                                       packet()
         frp->t_read = thread_add_read (master, frp_incoming_packet,
                                                       NULL, sock);
         break;
      case FRP_EVENT_UPDATE:
         ...
      case FRP_EVENT_POLL:
         ...
      default:
         ...
   }
}
```

Consequently, `frp_incoming_packet` is required to be an independent piece of code, the only parameter taken being the thread. The first thing these thread called functions must do is create a new thread which immediately blocks and sits waiting for the next event of the correct type to trigger, at which point the process starts again. The original thread meanwhile executes the code in the function passed to it, thus completing the current event.

```
int frp_incoming_packet (struct thread* t)
{  variables etc ...

   /* fetch socket then register myself */
   sock = THREAD_FD (t);
   frp->t_read = NULL;
   /* add myself to the next event */
   frp_event (FRP_EVENT_INCOMING, sock);

   // read the packet from the socket
   ...
}
```

Polling

The FRP daemon keeps a single poll timer that activates on a regular basis triggering a POLL event. Each time this happens, the list of peers is traversed and each one individually checked to see if a POLL message needs to be sent. The criteria to be met are that the peer is alive, it is not waiting for other messages to be received or sent, and it has been silent for more than the individual poll period for that peer.

The original FRP specification states

Keep traffic down to keepalives (5 sec?) if no updates.

and

Poll / keepalive frequency in tenths of seconds. Minimise received value with local configuration.

The current is 60 seconds as this is the most convenient frequency for debugging purposes. Quagga uses seconds so this FRP daemon also uses seconds. At present, the implementation only uses the host router's poll period, this needs to be changed to compare the two potentially different values and choose the lowest one as per the specification. The modification requires the router to store a little more state on each individual peer than happens at the moment. Both the router and the peer poll settings are customisable.

Polls are triggered within the FRP event handler. If there is already a POLL event active, it is cancelled and a new POLL thread is created.

```
case FRP_EVENT_POLL:
  if (frp->t_poll)
  {  thread_cancel (frp->t_poll);
     frp->t_poll = NULL;
  }
  frp->t_poll = thread_add_timer (master, frp_poll_peers, NULL,
                                  (unsigned long)frp->poll);
  break;
```

When a POLL event is issued, the function `frp_poll_peers` in `frpd.c` is executed. First a new POLL thread is created and set to wait for the next POLL to occur. Then the poll timer is cleared and reset. The list of peers in `frp_peers` is iterated through and the state of each peer is checked. This information is held in two flags, `flag_alive` and `flag_awaiting_ack`. Flags are boolean types that can be either ON or OFF.

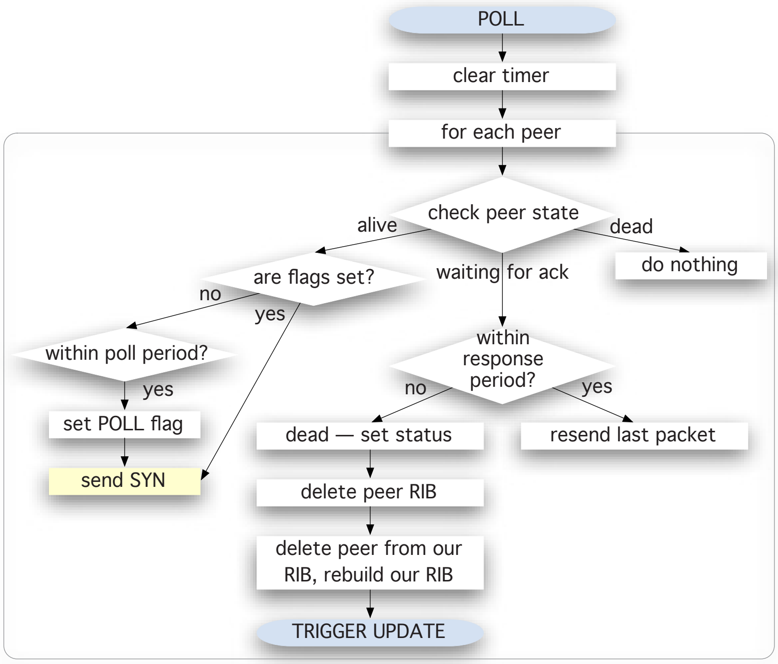


Figure 5.2
Polling flow

If `flag_alive` is OFF, then the peer has not responded for the specified number of retries and has been declared dead. So do nothing.

If `flag_alive` is ON and `flag_awaiting_ack` is OFF, then the peer has simply not had any information to share within the last poll period and this router's poll timer has triggered before the peer's has. The first thing to do is determine if any outgoing messages are waiting to be sent by checking the `flag_send_xxx` flags. If none of these are currently set, check that the poll period for this peer has expired and, if it has, turn on `flag_send_poll`. Finally, send a SYN packet to start the outgoing packet sequence. The outgoing packet will be made up of messages corresponding to any of the flags that were set.

If `flag_alive` is ON and `flag_awaiting_ack` is ON, check to see if the retry timer is still within the retry time. If it is, resend the last packet — which will be the one that the outstanding ACK is for. To facilitate this, the last sent packet is stored in `packet_latest`. Send it exactly as is with the same sequence number as previously.

If the retry time has over–run the specified retry period, then declare the peer dead and clean up. Leave the peer record in the peers list (it may return) but set `flag_alive` to OFF. Delete the peer's routing table — if the peer does return, it will send a new set of routes at that point. Work through the router's RIB and delete every route that uses the dead peer as a nexthop, then re–compute the table. Finally, trigger an UPDATE event to indicate that the routing table has changed.

The FRP daemon in action

The following *FRPsniffer* trace shows a standard poll exchange between two *Quagga* FRP daemons, 10.0.1.20 and 10.0.1.50.

50 starts the sequence with the initial handshake by sending a SYN packet.

```
FRP PACKET HEADER
     source address: 10.0.1.50 (a.0.1.32)
destination address: 10.0.1.20 (a.0.1.14)
      security hash: I??? {yk
    sender's seq no: 4 (0x4)
 recipient's ack no: 0 (0x0)
--SYN Packet

Waiting for incoming FRP packet
```

20 responds with a null packet ACK.

```
FRP PACKET HEADER
     source address: 10.0.1.20 (a.0.1.14)
destination address: 10.0.1.50 (a.0.1.32)
      security hash: \???J??x
    sender's seq no: 4 (0x4)
 recipient's ack no: 4 (0x4)
       Null Message: 0x0
--ACK Packet

Waiting for incoming FRP packet
```

The initial handshake is now complete so 50 can send the poll message that triggered this conversation. First comes the packet header, followed by the control message of type 'poll'.

```
FRP PACKET HEADER
     source address: 10.0.1.50 (a.0.1.32)
destination address: 10.0.1.20 (a.0.1.14)
      security hash: q?G??N??
    sender's seq no: 5 (0x5)
 recipient's ack no: 4 (0x4)

Control message: 0x1
--Poll, param = 0
```

The response from 20 contains a packet header and a control message of type ‘ack’ indicating that, as far as 20 is concerned, this conversation is over.

```
Waiting for incoming FRP packet


FRP PACKET HEADER
     source address: 10.0.1.20 (a.0.1.14)
destination address: 10.0.1.50 (a.0.1.32)
      security hash: ?0?)??
    sender's seq no: 5 (0x5)
 recipient's ack no: 5 (0x5)

Control message: 0x1
--Control ACK, param = 0
```

Some times when a host polls a peer, that peer has disappeared for some reason. The configuration information provided to each host at start up specifies how long that host will wait before re–polling and how many polls will be sent before declaring the peer ‘dead’. The following trace shows 10.0.1.50 polling 10.0.1.20 after 20 has been shut down.

50’s poll timer triggers and so sends a SYN to 20. A copy of the packet is placed in the latest packet storage for peer 20. 50 waits.

```
FRP PACKET HEADER
     source address: 10.0.1.50 (a.0.1.32)
destination address: 10.0.1.20 (a.0.1.14)
      security hash: &4[1???
    sender's seq no: 24 (0x18)
 recipient's ack no: 0 (0x0)
--SYN Packet

Waiting for incoming FRP packet
Waiting for incoming FRP packet
Waiting for incoming FRP packet
Waiting for incoming FRP packet
Waiting for incoming FRP packet
Waiting for incoming FRP packet
Waiting for incoming FRP packet
```

No reply is received from 20 within the retry time so 50 retrieves and sends the same packet again. 50 waits

```
FRP PACKET HEADER
     source address: 10.0.1.50 (a.0.1.32)
destination address: 10.0.1.20 (a.0.1.14)
      security hash: &4[1???
    sender's seq no: 24 (0x18)
 recipient's ack no: 0 (0x0)
--SYN Packet

Waiting for incoming FRP packet
Waiting for incoming FRP packet
Waiting for incoming FRP packet
Waiting for incoming FRP packet
Waiting for incoming FRP packet
Waiting for incoming FRP packet
Waiting for incoming FRP packet
```

No reply is received from 20 within the retry time so 50 retrieves and sends the same packet a third time. 50 waits.

In this example, the number of retries was set to 3, so when no reply is received from 20 within the retry time, 50 declares 20 to be dead.

Time passes and 20 is brought back up again.

20 starts the usual configuration handshake with a SYN packet.

50 ACKs.

```
FRP PACKET HEADER
     source address: 10.0.1.50 (a.0.1.32)
destination address: 10.0.1.20 (a.0.1.14)
      security hash: &4[1???
    sender’s seq no: 24 (0x18)
 recipient’s ack no: 0 (0x0)
--SYN Packet

Waiting for incoming FRP packet
Waiting for incoming FRP packet
Waiting for incoming FRP packet
Waiting for incoming FRP packet
Waiting for incoming FRP packet
Waiting for incoming FRP packet
Waiting for incoming FRP packet
Waiting for incoming FRP packet
Waiting for incoming FRP packet
Waiting for incoming FRP packet
Waiting for incoming FRP packet
Waiting for incoming FRP packet
Waiting for incoming FRP packet
Waiting for incoming FRP packet
Waiting for incoming FRP packet
Waiting for incoming FRP packet
Waiting for incoming FRP packet
Waiting for incoming FRP packet
Waiting for incoming FRP packet
Waiting for incoming FRP packet
Waiting for incoming FRP packet
20 Restarted
Waiting for incoming FRP packet
Waiting for incoming FRP packet

FRP PACKET HEADER
     source address: 10.0.1.20 (a.0.1.14)
destination address: 10.0.1.50 (a.0.1.32)
      security hash: ??;??
    sender’s seq no: 1 (0x1)
 recipient’s ack no: 0 (0x0)
--SYN Packet

Waiting for incoming FRP packet

FRP PACKET HEADER
     source address: 10.0.1.50 (a.0.1.32)
destination address: 10.0.1.20 (a.0.1.14)
      security hash: ?NT:Q???
    sender’s seq no: 25 (0x19)
 recipient’s ack no: 1 (0x1)
       Null Message: 0x0
--ACK Packet

Waiting for incoming FRP packet
```

20 sends a configuration message.

```
FRP PACKET HEADER
     source address: 10.0.1.20 (a.0.1.14)
destination address: 10.0.1.50 (a.0.1.32)
      security hash: ?J??c?
    sender's seq no: 2 (0x2)
 recipient's ack no: 25 (0x19)

IPV4 Config message: 0x41
               cost: 3
               poll: 60
              retry: 60
          router-id: 10.0.1.20

Waiting for incoming FRP packet
```

50 responds in kind with its own configuration message

```
FRP MESSAGE
     source address: 10.0.1.50 (a.0.1.32)
destination address: 10.0.1.20 (a.0.1.14)
      security hash: S8O?f
    sender's seq no: 26 (0x1a)
 recipient's ack no: 2 (0x2)

IPV4 Config message: 0x41
               cost: 10
               poll: 60
              retry: 60
          router-id: 10.0.1.50

Waiting for incoming FRP packet
```

and 20 closes the conversation with a control ACK.

```
FRP PACKET HEADER
     source address: 10.0.1.20 (a.0.1.14)
destination address: 10.0.1.50 (a.0.1.32)
      security hash: $???/?
    sender's seq no: 3 (0x3)
 recipient's ack no: 26 (0x1a)

Control message: 0x1
--Control ACK, param = 0
```

Triggered updates

An UPDATE event is triggered within the FRP event handler to indicate that a change has been made to the hosts routing table, path to gateway, or configuration and this new information needs to be propagated out to Zebra and to the host's peers. *Quagga* uses an update timer to try to batch updates to a certain extent — *"after a triggered update is sent, a timer should be set for a random interval between 1 and 5 seconds. If other changes that would trigger updates occur before the timer expires, a single update is triggered when the timer expires"* [35] and this has been retained in the FRP daemon.

```
case FRP_EVENT_UPDATE:
  if (frp->t_update_interval)
    frp->update_trigger = 1;
  else if (! frp->t_update)
    frp->t_update = thread_add_event (master, frp_update_peers,
                                                         NULL, 0);
  break;
```

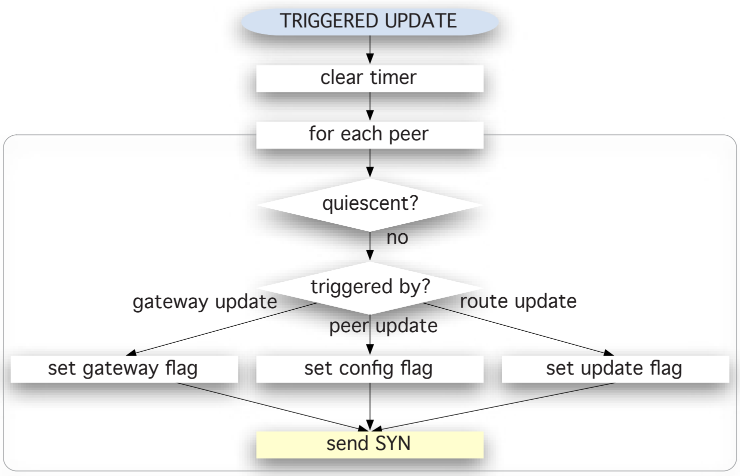


Figure 5.3
Triggered updates flow

When a UPDATE event is issued, the function `frp_update_peers` in `frpd.c` is executed. First a new UPDATE thread is created and set to wait for the next UPDATE to occur. Then the update timer is cleared and reset. The list of peers in `frp_peers` is iterated through, checking to see if the peer is quiescent. If it is, then no path to gateway currently exists for that peer and consequently, no update is needed. Otherwise, the correct flag is set for the type of update triggered, Quagga waits it's random period, then a SYN packet is created and sent.

Incoming packets

Dealing with an incoming packet is the most complex part of the FRP demon, mainly because of the many decisions to be made at each step. It is within this sequence that the different types of message are dealt with and the correct responding message sent in reply.

FRP packets

Note that the terms ‘packet’ and ‘message’ have different meanings in relation to FRP communications. A message is a single announcement of a set of data in a rigid, pre–specified format. A packet is the unit of transfer of data across the network and consists of 0 or more messages plus a packet header.

Quagga uses a `union` to store the contents of an incoming packet in a buffer. When combined with pointers to mark how much of the data has been extracted, this provides the flexibility to use the one buffer type for each incoming packet and to decide what underlying structure to give those contents based on what is extracted.

```
/* buffer to store frp data */
union frp_pkt_buf
{  char buf[FRP_PKT_MAXSIZE];
   struct frp_pkt_hdr              frp_pkt_hdr;
   struct frp_msg_hdr              frp_msg_hdr;
   struct frp_msg_control          frp_msg_control;
   struct frp_msg_ipv4config       frp_msg_ip4config;
   struct frp_msg_ipv4gateway      frp_msg_ip4gateway;
   struct frp_msg_ipv4update       frp_msg_ip4update;
};
```

Initially, the contents are treated as a `frp_pkt_hdr`. If dealing with a null packet, the process ends here. Otherwise the first message header is extracted with provides the length type of that message, allowing it to be extracted using the correct underlying structure once again. This process is repeated until the buffer is empty.

Incoming packet events

The `frp_incoming_packet` function in `frpd.c` is triggered by a packet arriving via the socket set up by the daemon during initialisation. The function code is divided in to three sequential parts, the mechanics of receiving a packet, extracting data and validity and security checking, and determining what type of messages have arrived and where they fit in the current conversation. `frp_incoming_packet` calls on an appropriate separate function (`frp_incoming_`*xxx*`_msg`) to handle each of the different types of incoming message. The diagram below concentrates on the second and third of these three parts.

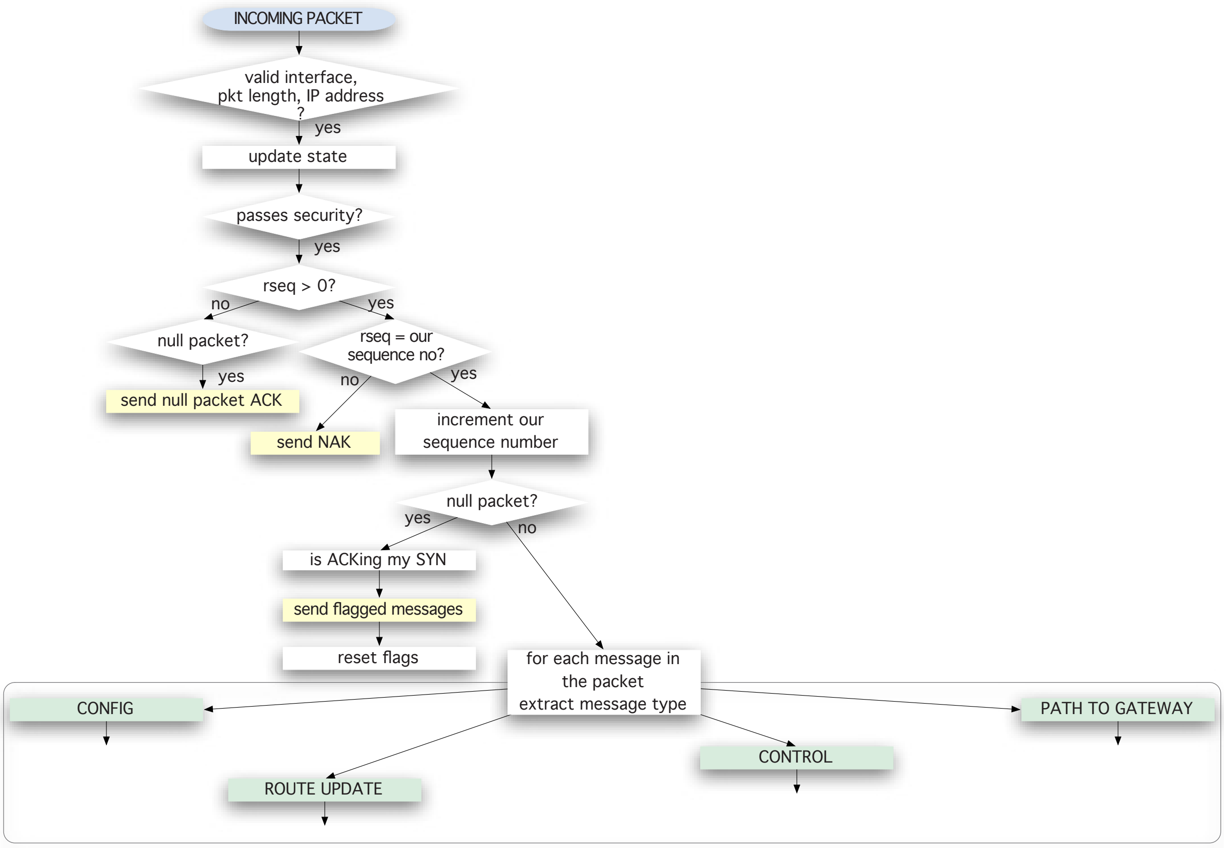


Figure 5.4
Incoming packet flow

When an `INCOMING PACKET` event is triggered by the arrival of a packet, a new `INCOMING PACKET` thread is created and set to wait for the next `INCOMING PACKET` event to occur. The daemon initialises a buffer of the correct size and reads in the contents of the packet as a continuous stream. It then checks that

- ➢ the packet comes from a known interface with a legitimate address
- ➢ FRP is running on this interface and the address belongs to a known FRP peer
- ➢ the packet length fits within the minimum and maximum size range pre–defined in `frp_packet.h`

The buffer is discarded if any of the checks fail.

Using the pre–defined sizes, types and flags specified in the file `frp_packet.h`, the daemon can now decode the single sequence of data into the correctly sized pieces of information contained in each individual message.

First the packet header

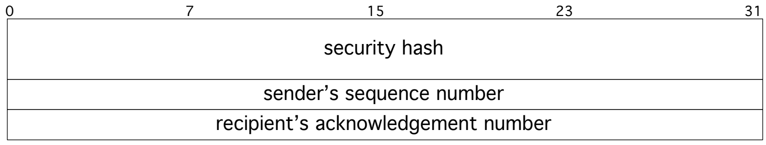


Figure 5.5
Packet header

```
#define FRP_PKT_HDRSIZE      16            // (128 bits)
#define FRP_PKT_MINSIZE      16            // (128 bits)
#define FRP_PKT_MAXSIZE      1400          // as specified by Don

struct frp_pkt_hdr
{  u_int8_t          hash[8];
   u_int32_t         sendSeq;
   u_int32_t         recipAck;
};
```

is extracted and the peer address confirmed. The peer `flag_alive` is turned ON and the `time_last_heard` flag time stamped. The function `checksecure`, which has been lifted directly from the original implementation of FRP, is run against the security hash using extracted peer address, the host router's local address and the secret stored in `frp_peers` for this particular peer. `checksecure` must succeed for processing of the packet to continue.

The peer's sequence number (the sender's sequence number in this instance) is checked to see whether this is the start of a new conversation or whether this packet is the next instalment in an on–going one. If it is equal to 0 and the entire packet consists only of the packet header with no messages attached (a null packet), then the peer is initiating a new conversation and the outgoing packet is a null packet ACK. Otherwise, check the peer's acknowledgement of the host's current sequence number (the recipient's acknowledgement number in this instance) against the actual sequence number stored by the host for that peer to see if they match. If they do not, then there has been a problem in the exchange of messages — a lost packet perhaps — and the host indicates this to the peer by sending a control NAK packet, essentially asking the peer to start the conversation again.

It has now been established that this is part of an on–going exchange, that the initial handshake is complete, and that the packet and the sequence numbers are valid. If the packet is a null packet, then the peer is ACKing a SYN sent by this host to start a new conversation. As communication has been established, the out–going packet will contain all flagged messages for that peer.

The only remaining possibility is that the packet contains one or messages in an on–going conversation, so the final section of the `incoming_packet` function contains a `while` loop that iterates to the end of the buffer holding the packet data extracting individual messages.

Within the `while`, the message header is extracted

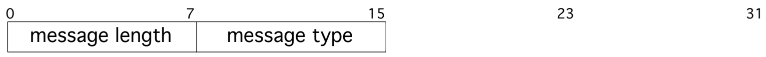


Figure 5.6
Message header

```
struct frp_msg_hdr
{  u_int8_t          length;
   u_int8_t          type;
};

// frp message types
#define FRP_MSG_CONTROL             0x01
#define FRP_MSG_IPV4CONFIG          0x41
#define FRP_MSG_IPV4GATEWAY         0x42
#define FRP_MSG_IPV4UPDATE          0x43
```

and the length checked, allowing the full message to be pulled from the buffer and stored. Pointers keep track of the start of the buffer and the current position within it. The type is then used to consult a `switch` statement which ensures the message is passed to the correct message handling function.

Control messages

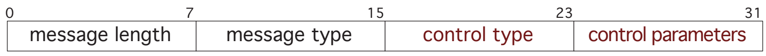


Figure 5.7
Control message

```
struct frp_msg_control
{  struct frp_msg_hdr       msg_hdr;
   u_int8_t                 type;
   u_int8_t                 param;
};

// frp control types
#define FRP_CTRL_POLL  1
#define FRP_CTRL_ACK   2
#define FRP_CTRL_NAK   3
```

Control messages are used to indicate that a has reached it's natural end (type 2, ACK), that there has been a problem in the exchange of messages so the conversation is being prematurely terminated and should begin again (type 3, NAK), and to check that a peer who has not been heard from for a period of time is still alive (type 1, POLL).

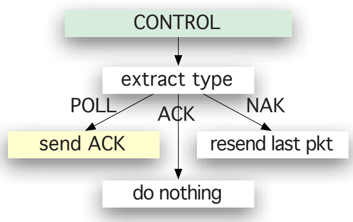


Figure 5.8
Control flow

A control ACK is different to a null ACK. The null ACK occurs when a router is responding to an initial SYN with an empty packet (header only) accepting the communication and establishing sequence numbers.

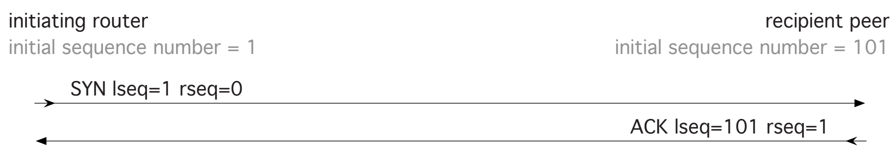


Figure 5.9
Null ACK

The control ACK follows a successful exchange of message(s) specifying that the conversation is at an end and no further communication is expected.

Figure 5.10
Control ACK

Consequently, the correct response to a control ACK is to do nothing except clear the `awaiting_ack` flag for that peer.

A control NAK is used when the sequence number the peer acknowledges is not the same as the one the host knows was the latest one sent. The most likely cause of this is lost or delayed messages. In response to a control NAK, a host retrieves the last message sent to that peer, strips the packet header from it. A new header is generated with a new sequence number and the security hash is re–generated before the packet is sent again.

A POLL is received because the sender wishes to know that the recipient is still alive so all that is required in response is a control ACK.

The FRP daemon in action

The standard poll exchange illustrates the use of control messages and the difference between null packet ACKs and control ACKs.

```
FRP PACKET HEADER
     source address: 10.0.1.50 (a.0.1.32)
destination address: 10.0.1.20 (a.0.1.14)
      security hash: I??? {yk
    sender’s seq no: 4 (0x4)
 recipient’s ack no: 0 (0x0)
--SYN Packet

Waiting for incoming FRP packet
```

The second packet in the trace below shows 10.0.1.20 responding to a SYN with a null packet ACK.

```
FRP PACKET HEADER
     source address: 10.0.1.20 (a.0.1.14)
destination address: 10.0.1.50 (a.0.1.32)
      security hash: \???J??x
    sender’s seq no: 4 (0x4)
 recipient’s ack no: 4 (0x4)
       Null Message: 0x0
--ACK Packet

Waiting for incoming FRP packet
```

The third packet shows 10.0.1.50 using a control message to send a poll

```
FRP PACKET HEADER
     source address: 10.0.1.50 (a.0.1.32)
destination address: 10.0.1.20 (a.0.1.14)
      security hash: q?G??N??
    sender's seq no: 5 (0x5)
 recipient's ack no: 4 (0x4)

Control message: 0x1
--Poll, param = 0

Waiting for incoming FRP packet
```

and in the fourth 20 responds with a control message containing an ACK.

```
FRP PACKET HEADER
     source address: 10.0.1.20 (a.0.1.14)
destination address: 10.0.1.50 (a.0.1.32)
      security hash: ?0?)??
    sender's seq no: 5 (0x5)
 recipient's ack no: 5 (0x5)

Control message: 0x1
--Control ACK, param = 0
```

Configuration messages

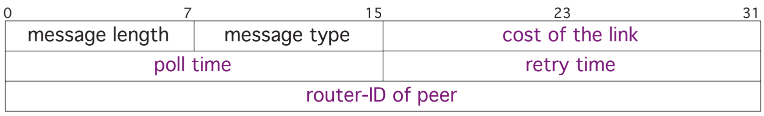


Figure 5.11
Configuration message

```
struct frp_msg_ipv4config
{  struct frp_msg_hdr       msg_hdr;
   u_short                  cost;
   u_short                  poll;
   u_short                  retry;
   struct in_addr           id;
};
```

The easiest of the message types: copy the new cost, poll, retry and id values into the correct places in `frp_peers`.

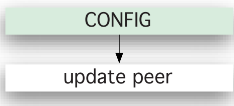


Figure 5.12
Configuration flow

Path to gateway messages

Gateway path messages are used to exchange gateway routes among peers. The decision making tree is relatively straightforward but the messages have a unique complication in that it is unknown how many addresses will make up the path. Consequently the message length must be used to iterate through the addresses, storing each one in a list. The only other piece of information that needs extracting and storing is the cost of the path from the peer to the gateway. If this is infinity (`0xffff`), then the peer has no gateway and is quiescent.

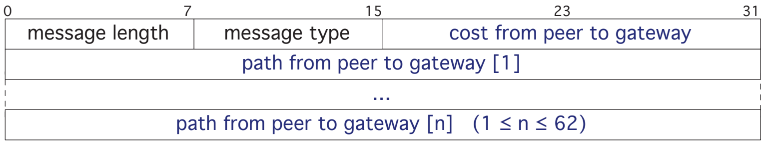


Figure 5.13
Path to gateway message

```
struct frp_msg_ipv4gateway
{  struct frp_msg_hdr       msg_hdr;
   u_short                  cost;
   struct in_addr           path[62];
};
```

First the message is checked to ensure that length falls within the maximum (252 bytes) and minimum (8 bytes) message size, and that the length of the path section is a multiple of 4 bytes. If it fails either of these tests, the peer's gateway cost is set to infinity (`0xffff`) and the message is discarded.

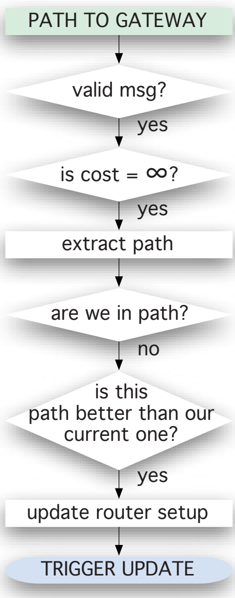


Figure 5.14
Path to gateway flow

Next the gateway cost is examined. If it is infinity (`0xffff`), then the peer record is updated and the message is discarded. Otherwise, the path is extracted stored in a linked list, which in turn is stored in `frp_peers`.

Finally, check to see if the host's local address is in the path. If it is not, add the cost of the link between the peer and the host is to the gateway cost just provided by the peer. This is matched against the cost of the current gateway for the host. If the new path is lower, and therefore 'better', update the router's data so that the `gateway_cost` is this peer's new gateway cost plus the link cost, `gateway_nexthop` is this peer, and `gateway_path` is this peer's new path. Iterate through the list of peers and set the `send_gateway` flag to ON and trigger an UPDATE event to send out new path to gateway messages to all peers.

The FRP daemon in action

This trace shows the conversation between 10.0.1.20 and 10.0.1.50 once 20 is told that it is now a gateway.

Following the standard SYN/ACK exchange,

```
FRP PACKET
     source address: 10.0.1.20 (a.0.1.14)
destination address: 10.0.1.50 (a.0.1.32)
       security hash: ??%W?J
    sender's seq no: 10 (0xa)
 recipient's ack no: 0 (0x0)
--SYN Packet

Waiting for incoming FRP message


FRP PACKET
     source address: 10.0.1.50 (a.0.1.32)
destination address: 10.0.1.20 (a.0.1.14)
      security hash:
    sender's seq no: 10 (0xa)
 recipient's ack no: 10 (0xa)
       Null Message: 0x0
--ACK Packet

Waiting for incoming FRP message
```

20 builds a path to gateway message, setting the cost to 0 as 20 is the gateway and the path length to 1 as 20's address is the only address in the path.

```
FRP PACKET
     source address: 10.0.1.20 (a.0.1.14)
destination address: 10.0.1.50 (a.0.1.32)
      security hash: !S<ØY?
    sender's seq no: 11 (0xb)
 recipient's ack no: 10 (0xa)

IPV4 path to gateway message: 0x42
                cost: 0
  # of nodes in path: 1
             path[0]: 10.0.1.20

Waiting for incoming FRP message
```

50 ACKs receipt.

```
FRP PACKET
     source address: 10.0.1.50 (a.0.1.32)
destination address: 10.0.1.20 (a.0.1.14)
      security hash: ?MD?ŒÜ?
    sender's seq no: 11 (0xb)
 recipient's ack no: 11 (0xb)

Control Message: 0x1
--Control ACK, param = 0
```

Route update messages

Route updates are the most intricate of the messages as they potentially trigger changes to the routing table and thus bring the FRP decision algorithm into play. They also have the added complexity of the message flag system that handles batching of multiple route updates over multiple messages, and potentially multiple packets.

0–7	7–15	15–23	23–31
message length	message type	update type flag	prefix length
route cost		cost from originator to gateway	
IP prefix			

Figure 5.15
Route update message

```
struct frp_msg_ipv4update
{  struct frp_msg_hdr       msg_hdr;
   u_int8_t                 flags;
   u_int8_t                 length;
   u_short                  routecost;
   u_short                  gatecost;
   struct in_addr           prefix;
};

// frp update flags
#define FRP_FLAG_BEGIN      0x01
#define FRP_FLAG_COMMIT     0x02
#define FRP_FLAG_NULLRT     0x04
#define FRP_FLAG_UPDATE     0x08
#define FRP_FLAG_DELETE     0x10
#define FRP_FLAG_GATEWAY    0x80
```

The important piece of information is the update type flag, which specifies where the current route update lies in the potential chain of messages. The six flags, defined in `frp_packet.h`, are stored as single pre–specified bits in a byte of memory and a message may have more than one flag set. A bitwise `AND` is used to extract each of the flagged bits.

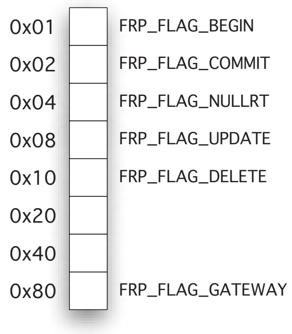


Figure 5.16
Route update message flags

Each FRP router shares all the routes it knows about that pass the FRP algorithm test as a batch of update messages. There should not be a great many of them as FRP explicitly tries '*to keep routing table sizes to a minimum*' [57]. The first message in the batch has the BEGIN flag set and the final message has the COMMIT flag set. It is assumed that all the routes between the BEGIN and the COMMIT belong to that batch. Any route within the batch may be flagged as a GATEWAY route.

If flags are not encountered in the correct sequence, it is assumed that a mistake has occurred and the entire batch is aborted. So if a second BEGIN appears before the first is closed by a COMMIT, the entire chain of routes arriving after the first BEGIN but before the second is decreed out–of–date. The arrival of the second BEGIN starts a new batch replacing the first one, as the peer has obviously updated it's routing table again in the time it has taken for the messages to be delivered.

There are three alternatives to sending a batch of FRP routes. The first is to indicate that no routes are present in the message by using the NULLRT flag. The specification says "*If there are no routes in the update, send route message with length=1, flags=BEGIN + COMMIT + NULLRT*", so when NULLRT is used, BEGIN and COMMIT flags must also be present. The "*length=1*" refers to the prefix length (as the message length must be 12) but the specification also says, "*if NULLRT is specified, fields beyond the flags field may be omitted*". So the FRP Quagga implementation reads in the entire message but if the NULLRT flag is set, then it does not use any data beyond the 3rd byte. For compatibility, it sets the prefix length to 1 when creating a message but makes no use of it when processing.

The second and third alternatives allow a single route to be deleted or updated using the DELETE or UPDATE flag in combination with a BEGIN and a COMMIT in the same single message. The original specification is unclear about the use of BEGIN and COMMIT flags with DELETEs and UPDATEs and as the original version of FRP did not implement these functions, the code does not provide addition information. Therefore, an independent decision had to be made when adding these two options to the Quagga daemon. The specification does state "*Update: single route add/change (abort batch update)*" and "*Delete: delete specified route (abort batch update)*", and also uses BEGIN and COMMIT with the NULLRT flag. Consequently, it was decided that these two should include a BEGIN as this makes processing route updates easier; all three of the alternatives can be treated as a new batch of one. Although a COMMIT is not actually needed from a processing point of view, it seems cleaner to match the batch updates. So in the FRP Quagga daemon, BEGINs with DELETEs and UPDATEs are required and COMMITs are optional.

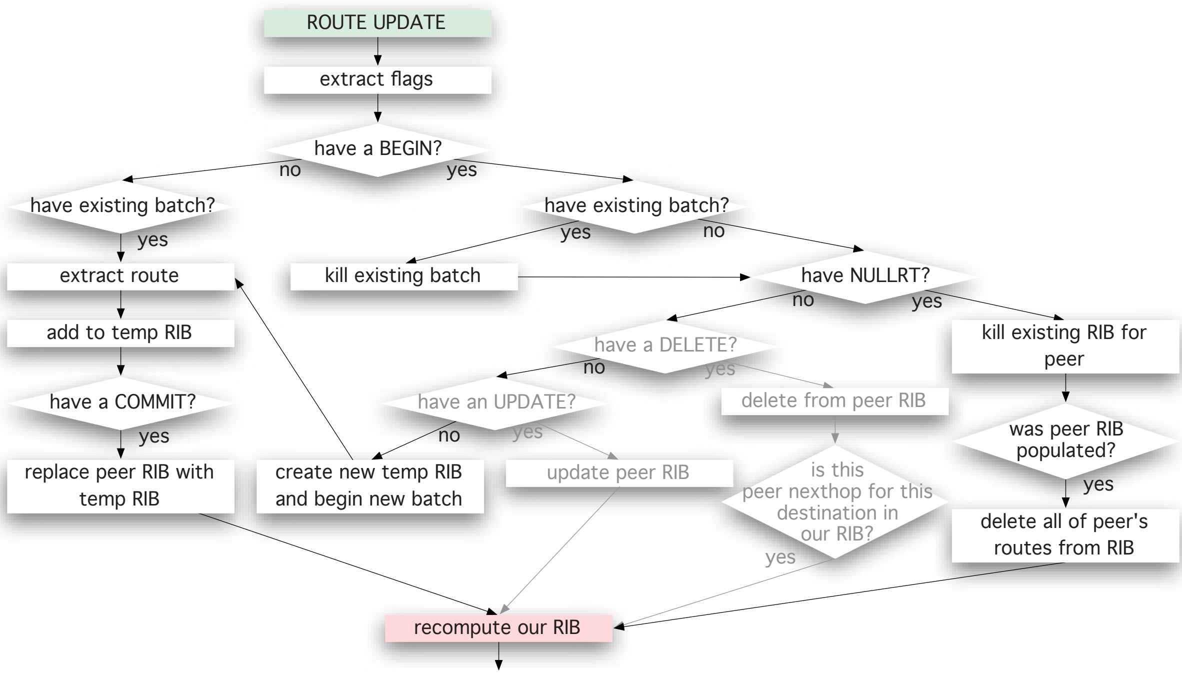


Figure 5.17
Route update flow
(note that DELETE and UPDATE not fully implemented in FRP Quagga daemon)

The diagram above shows the route update message decision tree. Having extracted the flags belonging to the message, the first aspect to check is the presence of a BEGIN and, no matter what the result, the next is to check for the presence of an existing batch. When distributing a batched route update, a peer sends it's entire table of all acceptable FRP routes. In the FRP Quagga daemon, the recipient creates a temporary RIB (`temp_rib`) for that peer when a new batch is begun and builds the new routing table for the peer in it. When a COMMIT is received, the currently stored routing table for that peer is replaced with the new one. So the presence of an existing batch can be determined by seeing if a temporary RIB has been set up for the peer.

If both a BEGIN and a temporary RIB are present, a now out–of–date batch already exists so this is aborted and a new one started. The next steps involve checking for the NULLRT, DELETE and UPDATE flags. Only one of the three can be present, so the process ends once one has been found and dealt with. If none of the three exist, then a new multi–message batch begins.

The presence of a BEGIN and a NULLRT means that the peer is sending an empty update message — that is, it has no FRP routes to share — so it is necessary to delete the RIB currently stored for that peer. Before this can happen, it is necessary to work through the host's routing table checking for routes using the peer as a next hop and deleting them. This in turn triggers the recomputation of the hosts routing table to see if other viable routes are available to replace the deleted ones.

If both a BEGIN and a DELETE appear, the host needs to first remove the route from the RIB stored for the peer and then check the host RIB to see if the route used the current peer as the nexthop for that route. If it does, delete the route from the host RIB and recompute in case another peer has provided an alternative.

In the case of receiving a BEGIN and an UPDATE, modify the affected route appropriately in the stored peer RIB, then recompute the host's RIB.

If there is no BEGIN, then a current batch should already exist and this message is an addition to the chain. Extract the route, add it to the temporary RIB for the peer, and check for a COMMIT. If one is not present, start the loop again with the next message, otherwise replace the old stored peer RIB with the new temporary RIB and recompute the host RIB.

The FRP daemon in action

The first trace below shows 10.0.1.20 initiating a null route exchange.

```
FRP PACKET HEADER
     source address: 10.0.1.20 (a.0.1.14)
destination address: 10.0.1.50 (a.0.1.32)
      security hash: ??%W?J
    sender's seq no: 10 (0xa)
 recipient's ack no: 0 (0x0)
--SYN Packet

Waiting for incoming FRP packet


FRP PACKET HEADER
     source address: 10.0.1.50 (a.0.1.32)
destination address: 10.0.1.20 (a.0.1.14)
      security hash:
    sender's seq no: 10 (0xa)
 recipient's ack no: 10 (0xa)
       Null Message: 0x0
--ACK Packet

Waiting for incoming FRP packet
```

The third packet of the exchange shows the single route update message with the BEGIN, NULLRT and COMMIT flags turned on and everything else set to 0.

```
FRP PACKET HEADER
     source address: 10.0.1.20 (a.0.1.14)
destination address: 10.0.1.50 (a.0.1.32)
      security hash: ?(:W?
    sender's seq no: 11 (0xb)
 recipient's ack no: 10 (0xa)

IPV4 Route Update Message: 0x43
                flags: 1792
                 FRP_FLAG_BEGIN flag: 256
                FRP_FLAG_NULLRT flag: 1024
                FRP_FLAG_DELETE flag: 0
                FRP_FLAG_COMMIT flag: 512
                FRP_FLAG_UPDATE flag: 0
               FRP_FLAG_GATEWAY flag: 0
               length: 0
            routecost: 0
             gatecost: 0
                prefix: 0.0.0.0

Waiting for incoming FRP packet


FRP PACKET HEADER
     source address: 10.0.1.50 (a.0.1.32)
destination address: 10.0.1.20 (a.0.1.14)
      security hash: ?MD??
    sender's seq no: 11 (0xb)
 recipient's ack no: 11 (0xb)

Control message: 0x1
--Control ACK, param = 0
```

In this second example, 10.0.1.20 is a gateway and 10.0.1.50 is not. Both hosts have the 'backbone' network, 10.0.1.0/24, set as static routes. Just before the trace begins, the VTY terminal is used to tell 20 to start advertising the network 10.0.20.0/24 via its interface address 10.0.20.1.

```
FRP PACKET HEADER
     source address: 10.0.1.20 (a.0.1.14)
destination address: 10.0.1.50 (a.0.1.32)
      security hash: x44M?
    sender's seq no: 8 (0x8)
 recipient's ack no: 0 (0x0)
--SYN Packet

Waiting for incoming FRP packet

FRP PACKET HEADER
     source address: 10.0.1.50 (a.0.1.32)
destination address: 10.0.1.20 (a.0.1.14)
      security hash: ?K?h??
    sender's seq no: 9 (0x9)
 recipient's ack no: 8 (0x8)
       Null Message: 0x0
--ACK Packet

Waiting for incoming FRP packet
```

Following the expected SYN and ACK, 20 builds a series of routes messages to hold the route it is now advertising to the network 10.0.20.0/24. In this case the route is only two hops long.

```
FRP PACKET HEADER
     source address: 10.0.1.20 (a.0.1.14)
destination address: 10.0.1.50 (a.0.1.32)
      security hash: ??P??]?U
    sender's seq no: 9 (0x9)
 recipient's ack no: 9 (0x9)

IPV4 Route Update Message: 0x43
                flags: 256
                 FRP_FLAG_BEGIN flag: YES
                FRP_FLAG_NULLRT flag: NO
                FRP_FLAG_DELETE flag: NO
                FRP_FLAG_COMMIT flag: NO
                FRP_FLAG_UPDATE flag: NO
               FRP_FLAG_GATEWAY flag: NO
               length: 24
            routecost: 0
             gatecost: 0
               prefix: 10.0.1.0

IPV4 Route Update Message: 0x43
                flags: 512
                 FRP_FLAG_BEGIN flag: NO
                FRP_FLAG_NULLRT flag: NO
                FRP_FLAG_DELETE flag: NO
                FRP_FLAG_COMMIT flag: YES
                FRP_FLAG_UPDATE flag: NO
               FRP_FLAG_GATEWAY flag: NO
               length: 24
            routecost: 0
             gatecost: 0
               prefix: 10.0.20.0

Waiting for incoming FRP packet

FRP PACKET HEADER
     source address: 10.0.1.50 (a.0.1.32)
destination address: 10.0.1.20 (a.0.1.14)
      security hash: ??_????Z
    sender's seq no: 10 (0xa)
 recipient's ack no: 9 (0x9)

Control message: 0x1
--Control ACK, param = 0
```

The following trace shows the FRP daemon routing table after the route update message has been received and processed on 50. the `F` prefix (here highlighted in red) denotes a route passed to Zebra from the FRP daemon. The first is the default route. The second is the statically set backbone route, which is inactive because the directly connected version is favoured. The third is a route that 50 has received form 20. Note that 50 has the network 10.0.50.0/24 set but as it is not a gateway, it is quiescent and therefore does not advertise the route.

```
frp50router# sh ip ro
Codes: K - kernel route, C - connected, S - static, R - RIP, F - FRP, O - OSPF,
       I - ISIS, B - BGP, > - selected route, * - FIB route

F   0.0.0.0/0 [184/3] via 10.0.1.20, en0, 00:00:39
K>* 0.0.0.0/0 via 10.0.1.1, en0
F   10.0.1.0/24 [184/3] via 10.0.1.20 inactive, 00:02:17
S   10.0.1.0/24 [1/0] is directly connected, en0
C * 10.0.1.0/24 is directly connected, en1
C>* 10.0.1.0/24 is directly connected, en0
F>* 10.0.20.0/24 [184/3] via 10.0.1.20, en0, 00:10:20
S   10.0.50.0/24 [1/0] is directly connected, en1
C>* 10.0.50.0/24 is directly connected, en1
C>* 127.0.0.0/8 is directly connected, lo0
frp50router#
```

The corresponding kernel routing table on 50. The route from 20 has been received from *Quagga* and inserted (highlighted in red).

```
Artemis:~ deb$ netstat -nrf inet
Routing tables

Internet:
Destination        Gateway            Flags    Refs      Use  Netif Expire
default            10.0.1.1           UGSc        2        0    en0
10.0.1/24          link#4             UCS         4        0    en0
10.0.1.1           0:3:93:e3:fc:92    UHLW        6      191    en0   1130
10.0.1.20          0:25:4b:ca:7b:b2   UHLW        3    13223    en0   1147
10.0.1.50          127.0.0.1          UHS         0        0    lo0
10.0.1.254         34:15:9e:18:d0:28  UHLW        0      571    en0    104
10.0.1.255         ff:ff:ff:ff:ff:ff  UHLWb       0        3    en0
10.0.20/24         10.0.1.20          UG1c        0        0    en0
10.0.50/24         link#5             UC          1        0    en1
10.0.50.1          0:1f:5b:c6:a4:da   UHLW        0        0    lo0
127                127.0.0.1          UCS         0        0    lo0
127.0.0.1          127.0.0.1          UH          0        0    lo0
169.254            link#4             UCS         0        0    en0
Artemis:~ deb$
```

Re-computing the RIB

Once the processing of the packet containing the route updates is complete, the `frp_recompute_rib` function is called. This steps through the routes stored for a peer, which will be the ones newly deposited in what was the `temp_rib` and is now the stored RIB for that peer. Each stored route is checked against the host's RIB. If the host does not currently have a route to that destination, the route is added with the current peer as the nexthop. If the route is already in the host's RIB, the FRP algorithm is applied and if the new route is 'better' than the old one, an exchange is made. In each case, the RIB is flagged as having changed so that once the loop through the peer's routes is complete an update can be triggered so that all the newly acquired routes can be passed on to other peers.

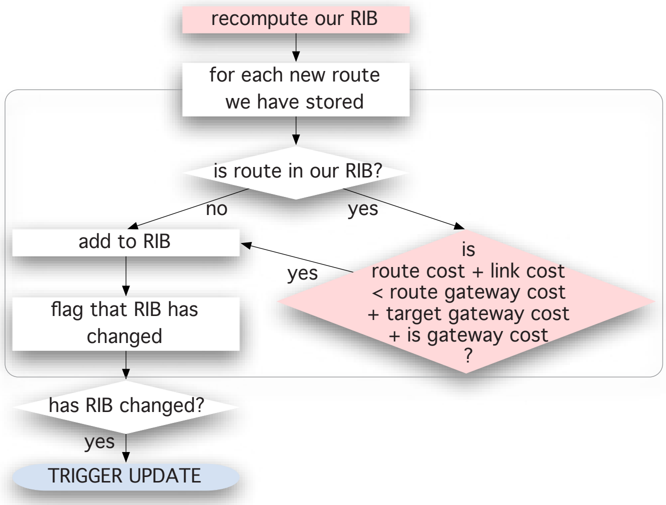


Figure 5.18
Recomputing the routing table flow

The FRP algorithm is used when the 'new' route has the same destination as a route already in the host's RIB. A comparison is made between the existing and the new route to determine which one belongs in the table of 'best' routes.

In the case of receiving a BEGIN and an UPDATE, modify the affected route appropriately in the stored peer RIB and check to see if the current peer is the nexthop for the route in the host's RIB. If it is, check to see if the new route is 'better' than the old one, changing it in the host RIB if it is — no other peer can have a better route or it would already be in there — and forcing a recompute if it is not in case another peer has a better one. If the current peer is not the nexthop, nothing more needs to be done.

Outgoing packets

There is no one outgoing packet function but rather a series of individual ones, each handling one specific message or packet creation task. These functions are all gathered together in `frp_packet.c` and draw heavily on the packet and message structures defined in `frp_packet.h`. When building a new packet, a buffer is created and the various parts of the packet's contents are 'memcopied' in to the correct place as they are created. Pointers are used to keep track of the beginning and end of the buffer, as well as the current insertion point.

The exact specification for each of the different message types can be found chapter 3.

Building a message

Each individual message within a packet begins with a message header. These are 2 bytes in length and contain the length and the type of the message. The function `make_frp_msg_header` uses a `switch` to create a message header with the correct values in each of the two fields.

The FRP control message is 4 bytes in length including the message header. The only useful data carried by the message is the control type — `POLL`, `ACK` or `NAK` — as the parameters field is not currently used. So the `make_frp_msg_control` function creates a control message header, sets the type appropriately, and sets the parameters to 0.

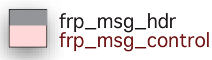


Figure 5.19
Control message buffer

`make_frp_msg_ipv4_config` sets up a 12 byte message to carry the configuration settings of a router. It creates a configuration message header, then extracts the cost, poll and retry settings from the current router configuration to fill out these fields of a configuration message. The final field is filled in with the host's address of the interface that the peer communicates with the host on.

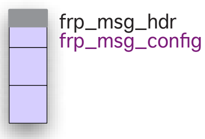


Figure 5.20
Configuration message buffer

When it comes to gateway messages, things get a little more complex because a 'path to gateway' length varies. The `make_frp_msg_ipv4_gateway` function has two strands governed by the `is_gateway_flag`. If the flag is set, the host router is a gateway so the path to gateway contains one address — that of the router's gateway interface — and the cost equals 0. Otherwise, a check is made to see if the host is quiescent. If it is, the cost is set to infinity (`0xffff`) and the path to 0. At 8 bytes, these are the smallest gateway messages.

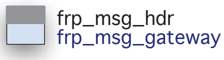


Figure 5.21
Path to gateway message with no path buffer

If the flag is not set and the host is not quiescent, cost is set to the current gateway cost and the entire path to gateway is appended with a further 4 bytes added to accommodate each additional address.

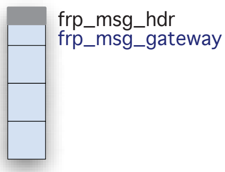


Figure 5.22
Path to gateway message with a path buffer

The `make_frp_msg_ipv4_update` function is more complex still. Called whenever the host's routing table is modified, this works through the RIB putting together a routing update for each peer, excluding routes that have that peer as the nexthop. Each route update message handles a single destination so a separate message is created for each entry in the table.

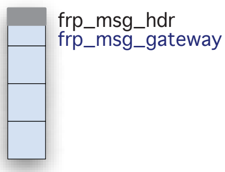


Figure 5.23
Single route update message buffer

All the messages required to send the RIB are sent as a batch using flags to provide the information necessary to interpret the entire table.

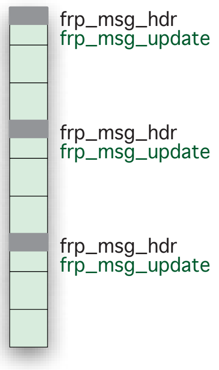


Figure 5.24
Batch of route update messages buffer

The function starts by setting a counter to 0 to indicate that there are no messages in the batch and then enters a loop that steps through each route stored in the host RIB. The nexthop for the route is checked to make sure that the route does not originate form the current peer before the peer record for the nexthop is retrieved. This provides the necessary information to make a test of the FRP decision algorithm between the route from the RIB and the route to the same destination in the peer record. If the new route is 'better' then a message containing the route is created to send to the peer as an update.

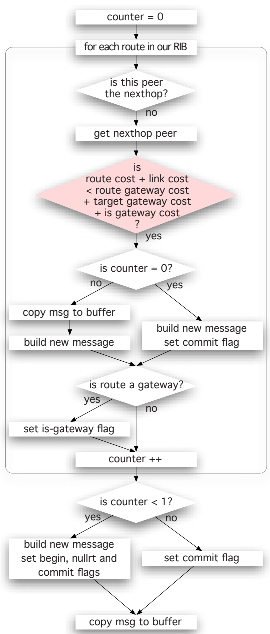


Figure 5.25
Building a batch of route update messages flow

If the counter equals `0`, then this is the first route in a new batch so a new message is created and the BEGIN update flag is set. If the counter is greater than `0`, then there is a previous message to be 'memcopied' to the buffer before creating a new message for the current route. The BEGIN flag is obviously not used in this case.

The next check is to see if the current route is flagged by the host as a gateway route and, if so, the GATEWAY update flag is set. The message length, route cost, gateway cost, and destination prefix are then filled in appropriately and the counter is incremented. The loop ends when all routes in the RIB have been considered.

The final part of the function tidies up the batch ready to send. The counter is again checked. If it is greater than `0`, then there are legitimate messages in the batch, the last one of which needs have the COMMIT update flag set before being copied to the buffer.

If the counter is still equal to `0` however, no routes are being passed to the peer so a null route packet needs to be sent. A new message is created, the message length is added appropriately, and the BEGIN, NULLRT and COMMIT update flags are set. The message is copied to the buffer and is ready to be turned into a completed packet.

Building a packet

A packet is one or more messages, each one complete with its message header, chained together with a packet header attached to the beginning. Although it is the first data in the packet, the 16 byte packet header is actually the last part of the packet to be created. The function `make_frp_pkt_hdr` fills out the sender's sequence number field by adding 1 to the last sequence number used for that peer and then replacing the stored one with the new one. The recipient's acknowledgement number is either set to the last received sequence number stored for the peer or to `0` depending on context.

It is the security hash that requires this part of the packet to be created last as the hash uses the entire packet as a parameter. The actual hash is performed in the function `dohash` in `frp_peer.c`. This function is one of only two pieces of code lifted directly from the original FRP implementation. This is because unless the hash is performed in exactly the same way in every implementation, the reverse hash when the packet is received invariably fails. As the code is essentially a series of library calls, it seemed entirely reasonable just to use the original `dohash` and `checksecure` functions.

First a hash is formed using the contents of the packet, minus the 16 bytes of the header. This is then rehashed with the host's address, rehashed again with the peer's address, and finally again with the host's secret. The first 64 bits of the final hash form the security hash for the packet.

```
dohash (u_char* buf, int len, const char* secret, IPADDR sa, IPADDR da)
   {  static u_int8_t hash[SHA_DIGEST_LENGTH];
      SHA_CTX shctx;
      SHA_Init(&shctx);
      SHA_Update(&shctx, buf, len);
      SHA_Update(&shctx, (u_char*)&sa, sizeof(IPADDR));
      SHA_Update(&shctx, (u_char*)&da, sizeof(IPADDR));
      SHA_Update(&shctx, secret, strlen(secret));
      SHA_Final(hash, &shctx);
      return hash;
   }
```

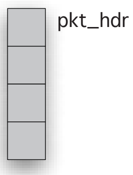


Figure 5.26
Packet header buffer

SYN packets and the initial response ACK packets are just packet headers with no additional messages attached and so are straightforward to create. The function `build_syn_pt` finds the address of the interface the peer is attached to, creates a SYN packet (ie: a packet header), opens a socket to use and calls `frp_send_packet` to send the packet to the peer. If the packet is successfully sent, a complete copy is stored in the `packet_latest` variable in the host's peer record, along with the packet length and the sequence number used — this enables the host to resend the packet should a NAK be received. The awaiting ACK flag is also set for this peer.

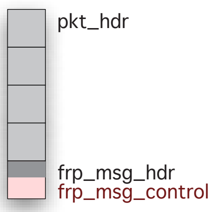


Figure 5.27
Packet header plus control message buffer

Control messages (POLL, ACK and NAK) consist of a packet header and a single control message with the correct flag set. In this implementation, there are currently two functions — `build_ack_pkt` and `build_nak_pkt` — that are virtually identical. It would be neater and more efficient to have a single `build_control_pkt` that handled all three in one function. These two functions create a buffer to build the packet in and then call `make_frp_msg_control`, setting the type field appropriately, before calling `make_frp_pkt_hdr` to create the packet header. The packet is sent, and as in the SYN packet, a copy is stored in the peer record and the awaiting ACK flag is set.

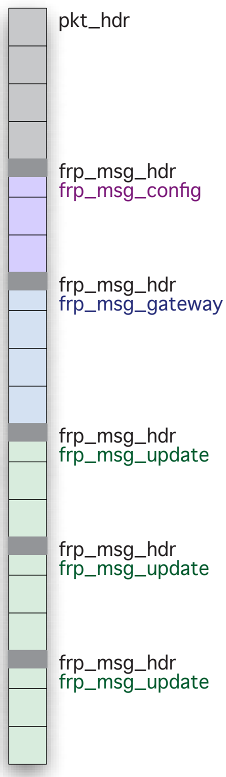


Figure 5.28
Multiple messages in one packet buffer

SYN, ACK and control messages are sent with a single message per packet. All the other messages types can be assembled into batches and sent with multiple messages carried by a single packet. The complete flow for an outgoing packet is shown below. It can be seen that most of the complexity is inside the loop sending flagged messages triggered by an update to a router configuration, a gateway route or a routing table.

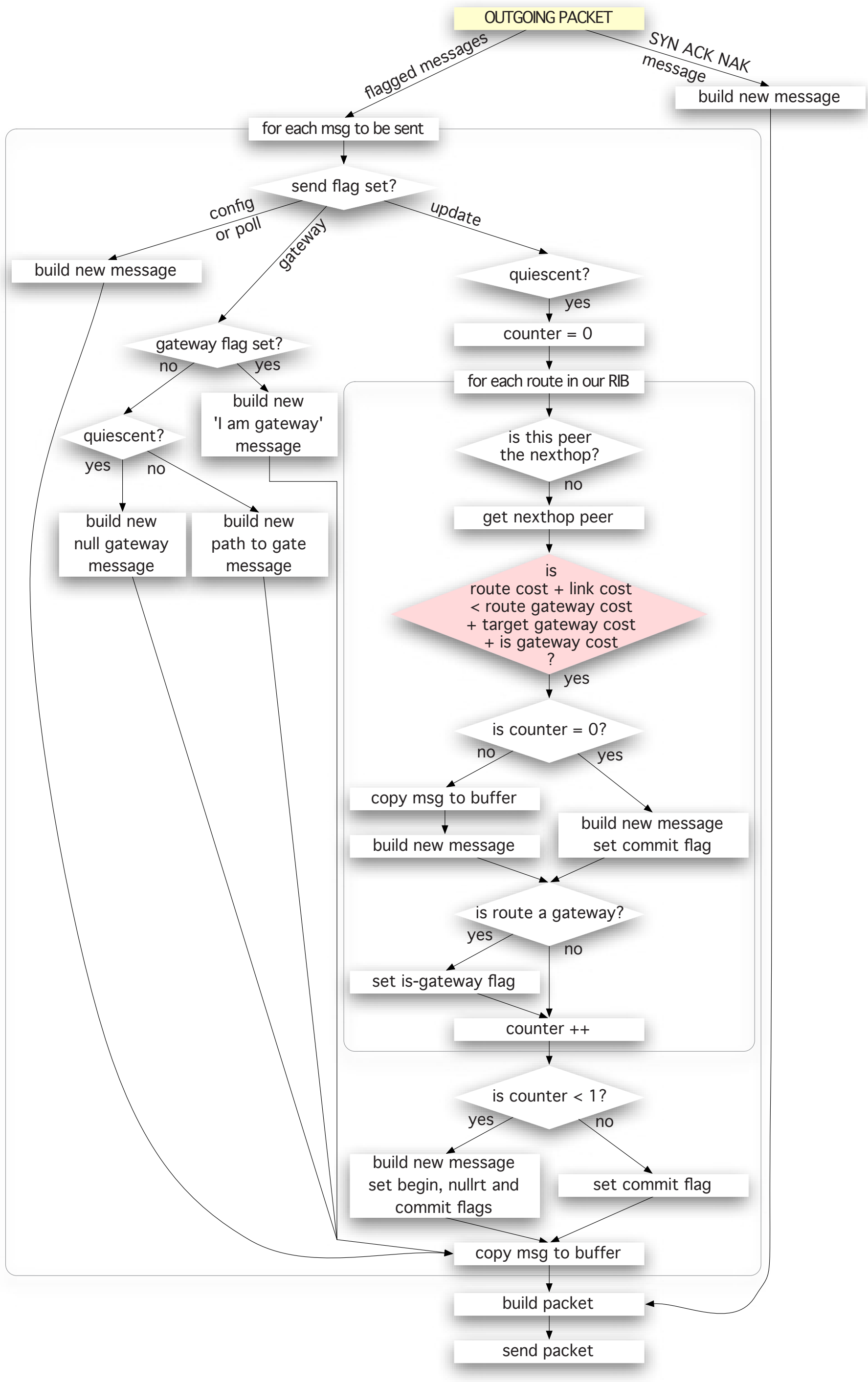


Figure 5.29
Outgoing packet flow

The `build_batch_pkt` function handles creating packets that potentially carry more than one message — specifically, all the messages currently flagged as needing to be sent to a single peer. The first step is to create the buffer that will store the data while the packet is being built. Next a local flag is created and turned `OFF`, and each of the peer flags — `flag_send_poll`, `flag_send_config`, `flag_send_gateway`, `flag_send_update` — are checked to see if any have been activated. For each of these that are set, a message of the appropriate type is created and sent. A check is made to ensure that the new message, when added to the packet buffer, will not cause the packet to exceed the specified maximum size of 1400 bytes.

The poll flag, the configuration flag and the gateway flag are all handled in the same way. In each case, the code segment calls the appropriate message function, the message and message header is created and 'memcopied' to the buffer, the packet length is adjusted and the local flag is switched `ON`. The update flag is only checked if the host is not quiescent (ie: it has a gateway) as update messages are not sent when the host has no gateway.

> *If there is no path to a gateway, the routing updates enter into a quiescent state, where link state and path information is still exchanged, but no changes are made to the routing table, and no routing updates are issued.* [57]

The `make_frp_msg_ipv4_update` function must be passed a calculation of how much packet space is left as it is potentially creating a batch of more than one update message. It must also return the amount of space it has actually used to allow the packet length to be accurately tallied. Upon completion, the update segment turns `ON` the local flag.

Once `build_batch_pkt` has worked through the flags and created messages as necessary for each one set, the next step is the check the local flag. If this is `ON`, messages have indeed been created so a packet header is created and added to the front of the buffer, and the packet is sent. The peer record is updated by turning all the send flags `OFF`, storing a copy of the latest sent packet plus its length and sequence number, and turning on the awaiting ACK flag. If the local flag is `OFF`, no messages have been created so `build_batch_pkt` checks the awaiting ACK flag and if it is `ON`, creates and sends a control ACK message to indicate that the current conversation is over.

The final function in `frp_packet.c` is `frp_send_packet`. This function is essentially the relevant code copied and pasted from the RIP and RIPng daemons. At present, this FRP implementation does minimal checking for error conditions. The next version will need to augment this function. The send routine essentially sets up the necessary addressing, socket and port issues, opens the socket, sends the packet to the peer, and closes the socket, passing the success or failure of the send back to the calling function.

Testing

Testing the *Quagga* FRP daemon did not happen as a single stage in the development but as a continuous process throughout the project. This testing was to ensure the correctness of the *Quagga* implementation — to make sure that the daemon worked as specified. Testing the design of the protocol itself to see if it resolves the problems it sets out to solve was beyond the scope of this project.

The first major round of testing occurred at the point where the shell demon was complete. It needed to successfully accomplish a number of tasks before the FRP additions where added in. The testing regime included loading the daemon from the configuration file, making sure the menu system in the VTY interface worked properly and that new commands could not only be added but also executed correctly. These are all things that could be tested directly.

It was vital that the shell daemon talked to the Zebra daemon via the zserv client and that routing information was passed through to Zebra and on to the kernel. This was tested by setting static routes (which also tested the VTY commands system) and then checking the Zebra and kernel routing tables to see that the routes appeared (see examples earlier in this section on page 92).

A useful testing and debugging tool is the `frp.log` file. *Quagga* provides the basic setup and support for these logs but it is possible to extend on those basics. The `frp_debug` files handle these extensions, adding in additional, FRP specific `#ifdef` statements that appear interlaced with all the other *Quagga* debug and status output in the log and in the terminal console. The bonus of tapping in to this system is that the various levels of debugging support can be live switched on and off via the VTY system without taking the daemon down.

Implementation of the fringe routing protocol into the shell daemon created a whole new range of issues and problems to be tested. The problem is that conversations between routers are silent and invisible. A major hurdle was the exchange of packets once the security was in place. The first iteration produced a very secure protocol that wouldn't even talk to itself. Running the *Quagga* implementation against the Knossos implementation was the only way to ensure that this stage was operating correctly.

Wireshark was invaluable for watching traffic move between routers and for drilling down into the packet to see the raw data. *FRPsniffer* was even more useful as it showed the interpreted data in an instantly readable form. Once again, keeping an eye on the routing tables via the standard Unix tools was necessary to ensure routes were passed through correctly. Setting FRPs administrative distance ridiculously high was another useful testing technique for ensuring route changes could be traced through the process.

One area where testing was not required was in the realm of the router, as opposed to the realm of FRP. It was simply assumed that *Quagga* was successfully handling these functions.

Finally, general protocol testing was carried out and the results of this are shown at the end of each of the earlier sections of this chapter where they illustrate that the implementation of that section does in fact work.

6. Conclusions and future work

This project set out to achieve two distinct goals: to understand and specify the new fringe routing protocol; and to create an independent, standalone implementation of the protocol in the *Quagga software routing suite*. Achieving the first goal required the development of a detailed knowledge of routing and routing protocols. The second generated a new challenge, that of becoming familiar with all aspects of the *Quagga* environment.

The first outcome has been achieved in chapter 3 in the expansion of the initial concepts and notes provided by Don Stokes into a detailed description of the fringe routing protocol. This has addressed a number of ambiguities and assumptions of knowledge, both implicit and explicit, in the supplied material. A future goal is to expand this description further into a formal specification to be submitted to the Internet Engineering Task Force (IETF) as an Internet Draft.

The second goal is realised as described in chapters 4 and 5. Achieving this required extending the *Quagga* suite to add FRP to the set of supported routing protocols. The new *Quagga* FRP daemon is able to bootstrap itself using a supplied configuration file and communicate with other FRP peers. It can apply the FRP routing algorithms to correctly process routing information and inject routes into the kernel. A future goal is to extend this daemon to support IPv6.

The work on this project was presented, in conjunction with Don Stokes of Knossos Networks, at the NZ Network Operators Group (NZNOG) conference in Wellington in February 2011 where it was well received by the professional networking community. Slides from the presentation are available in Appendix B.

Bibliography

[1] Anon, 2008, *Quagga: the easy tutorial*, viewed 18 March 2011 < http://openmaniak.com/quagga.php>.

[2] Ascigil, O., Yuan, S., Griffioen, J. and Ken Calvert, K., 2008, "Deconstructing the Network Layer", *Proceedings of the 17th International Conference on Computer Communications and Networks, 2008*, ICCCN 2008, St. Thomas, US Virgin Islands, pp. 1-6.

[3] van Beijnum, P. I. and van Beijnum, I. V., 2002, *BGP*, O'Reilly & Associates, Inc., Sebastopol, Canada.

[4] van Beijnum, I., 2006, "IPv6 Internals", *The Internet Protocol Journal*, vol. 9, no. 3, viewed 18 March 2011, <http://www.cisco.com/web/about/ac123/ac147/archived_issues/ipj_9-3/ipv6_internals.html>

[5] van Beijnum, I., 2007, *Everything you need to know about IPv6*, viewed 18 March 2011 <http://arstechnica.com/hardware/news/2007/03/IPv6.ars>.

[6] van Beijnum, I., 2010, *There is no Plan B: why the IPv4-to-IPv6 transition will be ugly*, viewed 18 March 2011 <http://arstechnica.com/business/news/2010/09/there-is-no-plan-b-why-the-ipv4-to-ipv6-transition-will-be-ugly.ars/>.

[7] Bhargavan, K., Obradovic, D. and Gunter, C.A., 2002, "Formal Verification of Standards for Distance Vector Routing Protocols", *Journal of the ACM*., vol. 49, no. 4, pp. 538-576.

[8] Bradner, S., "The Internet Standards Process -- Revision 3", BCP 9, RFC 2026, Harvard University, October 1996.

[9] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, Harvard University, March 1997.

[10] Bush, R. and D. Meyer, "Some Internet Architectural Guidelines and Philosophy", RFC 3439, The Internet Society, December 2002.

[11] Caesar, M. and Rexford, J., 2005, "BGP Routing Policies in ISP Networks", *IEEE Network*, vol. 19, no. 6, pp. 5-11.

[12] Carpenter, B., Ed., "Architectural Principles of the Internet", RFC 1958, IAB, June 1996.

[13] Cisco, 2009, *Border Gateway Protocol*, viewed 18 March 2011 < http://docwiki.cisco.com/wiki/Border_Gateway_Protocol>.

[14] Cisco, n.d., *Basic Router Configuration* viewed 18 March 2011 < http://www.cisco.com/en/US/docs/routers/access/800/850/software/configuration/guide/routconf.pdf>.

[15] Cisco, n.d., *Cisco IOS IP Routing Protocols Configuration Guide*. Viewed 18 March 2011 < http://www.cisco.com/en/US/docs/ios/12_3/featlist/ip_vcg.html>

[16] Cisco, n.d., *Route Selection in Cisco Routers*, viewed 18 March 2011 < http://www.cisco.com/en/US/tech/tk365/technologies_tech_note09186a0080094823.shtml>.

[17] Cisco, n.d., *IP Addressing and Subnetting for New Users*, viewed 18 March 2011 < http://www.cisco.com/en/US/tech/tk365/technologies_tech_note09186a00800a67f5.shtml>.

[18] Cisco, n.d., *Routing Basics*, viewed 18 March 2011 < http://docwiki.cisco.com/wiki/Routing_Basics>.

[19] Cisco, n.d., *What Is Administrative Distance?*, viewed 18 March 2011 < http://www.cisco.com/en/US/tech/tk365/technologies_tech_note09186a0080094195.shtml>.

[20] Deering, S. and Hinden, R., "Internet Protocol, Version 6 (IPv6)", RFC 2460, Cisco/Nokia, December 1998.

[21] Feldmann, A. and Rexford, J. 2001, "IP Network Configuration for Intradomain Traffic Engineering", *IEEE Network*, vol. 15, no. 5, pp. 46-57.

[22] Fuller, V. and T. Li, "Classless Inter-domain Routing (CIDR): The Internet Address Assignment and Aggregation Plan", BCP 122, RFC 4632, Cisco Systems/Tropos Networks, August 2006.

[23] Griffin, T.G. and Sobrinho, J.L., 2005, "Metarouting", *Proceedings of the 2005 conference on Applications, technologies, architectures, and protocols for computer communications*, SIGCOMM '05, Philadelphia, Pennsylvannia, USA, pp. 1-12.

[24] Hagen, S., 2006, *IPv6 essentials,* 2nd ed., O'Reilly Media, Sebastopol, Canada.

[25] Hall, B. J., 2009, *Beej's guide to network programming: using Internet sockets* (version 3.0.14). [Online]. Available: http://beej.us/guide/bgnet/

[26] He, L., 2005, "A Verified Distance Vector Routing Protocol for Protection of Internet Infrastructure", *Lecture Notes in Computer Science*, vol. 3421, pp. 463-470.

[27] He, J., Rexford, J. and Chiang, M., 2007, “Don’t Optimize Existing Protocols, Design Optimizable Protocols”, *ACM SIGCOMM Computer Communication Review*, vol. 37, no.3, pp. 53-58.

[28] Hedrick, C., “Routing Information Protocol”, RFC 1058, Rutgers University, June 1988.

[29] Hinden, R., “Internet Engineering Task Force Internet Routing Protocol Standardization Criteria”, RFC 1264, BBN, October 1991.

[30] Hinden, R. and Deering, S., “IP Version 6 Addressing Architecture”, Nokia/Cisco, February 2006.

[31] Huitema, C., 1995, *Routing in the Internet,* Prentice-Hall, Inc., Harlow, United Kingdom.

[32] Huitema, C., Postel, J. and Crocker, S., “Not All RFCs are Standards”, RFC 1796, INRIA/ISI/CyberCash, April 1995

[33] Hulse, R., 2005, “WIX: a distributed internet exchange”, *Linux Journal*, vol. 135, no. July, pp. 2.

[34] Hunt, C., 2002, *TCP/IP network administration*, O’Reilly Media, Inc., Sebastopol, Canada.

[35] Ishiguro, K. et al., 2005, *Quagga: a routing software package for TCP/IP networks*, viewed 18 March 2011 <http://wenku.baidu.com/view/fae88f74f46527d3240ce003.html>.

[36] Kurose, J. F. and Ross, K., 2002, *Computer Networking: a Top-Down Approach Featuring the Internet*, 2nd. Ed, Addison-Wesley Longman Publishing Co., Inc., Harlow, United Kingdom.

[37] Le, F., Xie, G.G., Pei, D., Wang, J. and Zhang, H., 2008, “Shedding Light on the Glue Logic of the Internet Routing Architecture”, *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 4, pp. 39-50.

[38] Le, F., Xie, G.G. and Zhang, H., 2007, “Understanding route redistribution”, *proceedings of the IEEE International Conference on Network Protocols*, Beijing, China, pp. 81-92.

[39] Le, F., Xie, G.G. and Zhang, H., 2010, “Understanding route aggregation”, Carnegie Mellon University Computer Science Report CS-10-106, Carnegie Mellon University, Pittsburgh, Pennsylvania, USA.

[40] Little, M., “Goals and functional requirements for inter-autonomous system routing”, RFC 1126, SAIC, October 1989.

[41] Malhotra, R., 2002, *IP routing*, O’Reilly Media, Inc., Sebastopol, Canada.

[42] Malkin, G., “RIPng Protocol Applicability Statement”, RFC 2081, Xylogics, January 1997.

[43] Malkin, G., “RIP Version 2”, RFC 2453, Bay Networks, November 1998.

[44] Malkin, G. and Minnear, R., “RIPng for IPv6”, RFC 2080, Xylogics/Ipsilon Networks, January 1997.

[45] Medhi, D. and Ramasamy, K., 2007, *Network Routing: Algorithms, Protocols, and Architectures*, Morgan Kaufmann Publishers Inc., Burlington, Massachusetts, USA.

[46] Moy, J. T., 1998, *OSPF: Anatomy of an Internet Routing Protocol*, Addison-Wesley Longman Publishing Co., Inc., Harlow, United Kingdom.

[47] Nucci, A., Bhattacharyya, S., Taft, N. and Diot, C., 2007, “IGP Link Weight Assignment for Operational Tier-1 Backbones”, IEEE/ACM Transactions on Networking, vol. 15, no. 4, pp. 789-802.

[48] Parkhurst, W., 2004, *Routing First-Step*, Cisco Press, Indianapolis, Indiana, USA.

[49] Partridge, C., Snoeren, A.C., Strayer, W.T., Schwartz, B., Condell, M. and Castiñeyra, I., 2000, “FIRE: Flexible Intra-AS Routing Environment”, *ACM SIGCOMM Computer Communication Review*, vol. 30, no. 4, pp. 191-203.

[50] Perlman, R., 2000, *Interconnections: Bridges, Routers, Switches, and Internetworking Protocols*, 2nd Ed., Addison-Wesley Longman Publishing Co., Inc., Harlow, United Kingdom.

[51] Pilot, n.d., *Zebra hacking notes*, viewed 18 March 2011 <http://quagga.net/faq/zebra-hacking-guide.txt>.

[52] Postel, J. and J. Reynolds, “Instructions to RFC Authors”, RFC 2223, ISI, October 1997.

[53] Raghavan, B., Verkaik, P. and Snoeren, A.C., 2009, "Secure and Policy-Compliant Source Routing", *IEEE/ACM Transactions on Networking*, vol. 17, no. 3, pp. 764-777.

[54] Rekhter, Y., Li, T. and Hares, S., "A Border Gateway Protocol 4 (BGP-4)", The Internet Society, January 2006.

[55] Stallings, W., 1993, *Data and Computer Communications*, 4th Ed., Prentice Hall, Harlow, United Kingdom.

[56] Stevens, W. R. *et al*, *Unix network programming: the sockets networking API*, 3rd ed. Boston, MA: Addison–Wesley, 2004.

[57] Stokes, D., 2010, *Notes on FRP*, personal communication.

[58] Stokes, D., 2010, *FRP router implementation code*, personal communication.

[59] Tanenbaum, A. S., 1985, *Computer Networks*, Prentice Hall, Harlow, United Kingdom.

[60] Uriarte, Y., 2001, *Zebra for dummies, aka Zebra hacking how to*, viewed 18 March 2011 < http://www.quagga.net/zhh.html>.

[61] Walrand, J. and Parekh, S., 2010, "Communication networks: a concise introduction", *Synthesis Lectures on Communication Networks*, vol. 3, no. 1, pp. 1-192.

[60] Wang, Y. and Wang, Z., 1999, "Explicit Routing Algorithms for Internet Traffic Engineering", *proceedings of the eighth international conference on Commuter Communications and Networks*, Boston, Massachusetts, USA, pp. 582-588.

[61] White, R., 2005, "Caveats in Testing Routing Protocol Convergence", *The Internet Protocol Journal*, vol. 8, no. 4, pp. 20-27.

Appendix A

Initial Fringe Routing Protocol specification from Don Stokes

Goals

- Unicast, configured neighbor relationship
- Secure, must work on public access networks
- Intended to be subordinate to IGP; routes gatewayed at one location only
- Nodes forward routes toward their designated gateway, Nodes forward routes away from the gateway if link cost < sum of gateway cost of originated route + gateway cost of local node (i.e. it's closer to route direct than via the gateways)
- Try to keep routing table sizes to a minimum
- Keep traffic down to keepalives (5 sec?) if no updates
- Only update kernel routing table or transmit routing information if there is a path to a gateway. Activity is quiescent until gateway route is established.

Route forwarding

Forward route from local-node to remote-node if:

route-cost + link-cost < route-gw-cost + target-gw-cost + is-gw-route

where:

route-cost is the accumulated sum of the costs of intervening links;

link-cost is the cost of the link to remote-node;

route-gw-cost is the cost from the route's originating node to its gateway, as expressed in the routing update;

target-gw-cost is the cost from the remote-node to its gateway, as expressed in its link advertisements;

is-gw-route is 1 if the route is being passed toward the gateway, 0 otherwise. The is-gw flag is only set to 1 on a route at the originator, and set to 0 any time the route is passed to a peer rather than a gateway.

This algorithm means that routes are only passed across the network if they are "better" than going up to the backbone and back -- in many cases this will be preferable to using backup paths.

Gateway paths

Each host expresses to all its peers the path to its gateway. This allows a host to exclude paths that include itself from being considered as potential gateway routes, and avoids counting to infinity.

Quiescence

If there is no path to a gateway, the routing updates enter into a quiescent state, where link state and path information is still exchanged, but no changes are made to the routing table, and no routing updates are issued. Without this, routes quickly count to inifity when the last gateway path disappears.

Split Horizon

A route should not be advertised back up the link it was learned from.

Messages

Message header

hash (64): SHA1(secret, lseq, rseq, message ...)

lseq (32): Local (sent) sequence number

rseq (32): Receive (acknowledge) sequence number

Each packet is one message

Messages sent as UDP packets < 1400 data bytes long.

mlen: 8 bits length of message in longwords including type/len

mtype: 8 bits. Note: mtype is in form <proto><type>; IPv4-specific messages are 0x4n, IPv6 messages are 0x6n.

Message type 41: IPv4 configuration

linkcost(16): link cost. Maximise received value with local configuratio.

polltime(16): Poll/keepalive frequency in tenths of seconds. Minimise received value with local configurtion.

failtime(15): Timeout to failure after last acked packet, in tenths of seconds. Minimise received value with local configuration.

routerid(32): Unique ID (usually IP address) of peer router, used to avoid loops.

Message type 42: Path change (path to gateway),

gwcost(16) Distance to gateway. 0 = is gateway, 0xffff = gateway unknown,

path(32 x n): Path to gateway, list of 32 bit node IDs, used to prevent counting to infinity. Router will not learn a path that contains itself. Path length defined by message length, limited to 62 entries.

Message type 0x43: Add/change/delete IPv4 route

flags(8): Flags, see below

prefixlen(8) prefix length (0-32)

rcost(16) route cost (distance vector)

rgwcost(16) distance of original route from gateway

prefix(32) IPv4 or IPv6 address

Message type 61: IPv6 configuration

linkcost(16): link cost

gw(128): Gateway address for upstream routes. Note, may be overridden by local configuration.

Message type 0x63: Add/change/delete IPv6 route

flags(8): see below

prefixlen(8) prefix length (0-32 for IPv4, 0-128 for IPv6)

rcost(16) route cost (distance vector)

rgwcost(16) distance of original route from gateway

prefix(32-128) IPv6 address, truncated to 32 bit boundary following prefix length.

Routing flags

BEGIN (0x01): This is the first route of a batch update. Begin the update transaction, abort any already in progress.

COMMIT (0x02): Last route of batch update, commit the update.

NULLRT (0x04): Only interpret flags; no route exists in message. Note, if NULLRT is specified, fields beyond the flags field may be omitted.

UPDATE (0x08): Single route add/change. (Abort batch update.) *

DELETE (0x10): Delete specified route. (Abort batch update.) *

GATEWAY (0x80): Route is a gateway route

* UPDATE and DELETE are not implemented in frpd.

Batch updates

Batch processing is handles as follows:

If a route message is received with the BEGIN flag set, any existing batch update is flushed, and a new one started. The route should be added to the new batch.

If a route message is received with the COMMIT flag set, the route should be added to the batch, and the whole batch processed as a complete update.

If a route message with all flags (except the GATEWAY flag) clear is received, the route should be added to the inbound batch.

If any other type of message is received, the batch should be flushed and the message processed as if the batch never started.

If there are no routes in the update, send route message with length=1, flags=BEGIN + COMMIT + NULLRT.

Initial handshake procedure

Assume isn & rsn, isn = initiator's initial sequence number, rsn = responder's initial sequence number.

Initiator sends null packet, lseq = isn, rseq = 0.

Responder replies with null packet, lseq = rsn, rseq = isn.

Initiator sends IP config packet plus path len (both messages must be present), lseq = isn+1, rseq=rsn.

Responder replies in kind, lseq = rsn+1, rseq = lsn+1. Responder may send routing information in the same packet.

Intiator replies, lseq = lsn+2, rseq = rsn+1, optionally with updated path and with routing information.

Responder acks, lseq = rsn + 1, rseq = lsn + 2.

Sequence numbers

Given state variables, locseq (Local sequence number), ackseq (acknowledged local sequence number) & remseq (remote sequence number):

If a packet is received with rseq = 0: Discard packet if not null. Set remseq = lsec. Send reply lsec = locseq, rseq = remseq.

All other packets, check rseq = locseq. If not, discard. Update ackseq.

If lseq != remseq, set remseq = lseq, process packet, reply lseq = locseq, rseq = remseq. (locseq may be incremented by update if data is sent that needs to be acknowledged.)

Sent packets should increment locseq prior to constructing packet. If locseq wraps to 0, set locseq to 1.

Periodically resend last packet if ackseq != locseq.

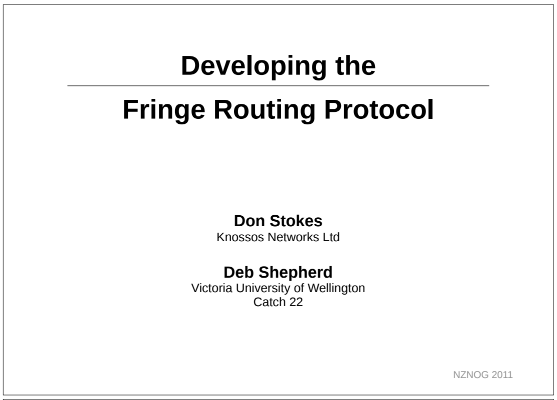
Example, n1 -> n2

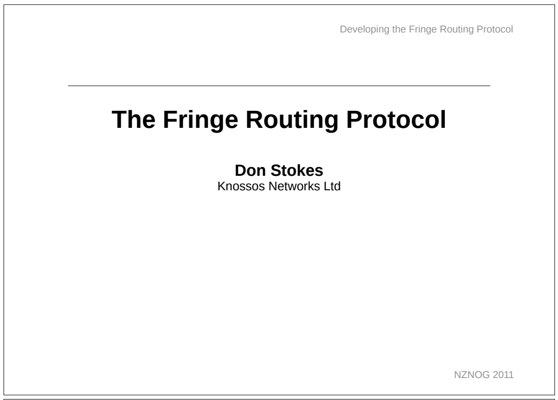
n1 ip=10.0.0.1: lseq = 100
n2 ip=10.0.0.2: lseq = 200

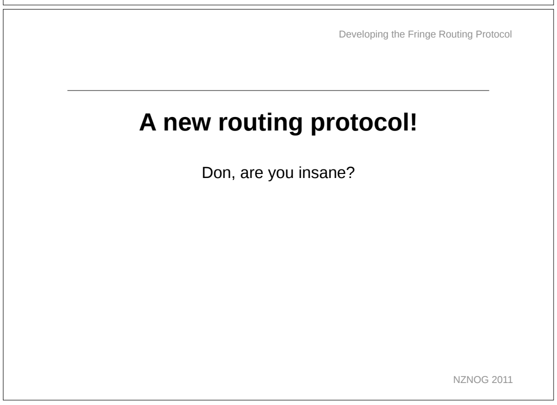
10.0.0.1 -> 10.0.0.2 lseq=100 rseq=0	n2 xseq=101
10.0.0.2 -> 10.0.0.1 rseq=100 lseq=200	
10.0.0.1 -> 10.0.0.2 lseq=101 rseq=200 config, path	vaidate xseq
10.0.0.2 -> 10.0.0.1 rseq=101 lseq=201 config, path, routes	
10.0.0.1 -> 10.0.0.2 lseq=102 rseq=201 routes	
10.0.0.2 -> 10.0.0.1 rseq=102 lseq=201	
10.0.0.1 -> 10.0.0.2 lseq=103 rseq=201 poll	
10.0.0.2 -> 10.0.0.1 rseq=103 lseq=202 response	
10.0.0.1 -> 10.0.0.2 lsec=103 rseq=202	
10.0.0.1 -> 10.0.0.2 lseq=103 rseq=202	Repeat packet

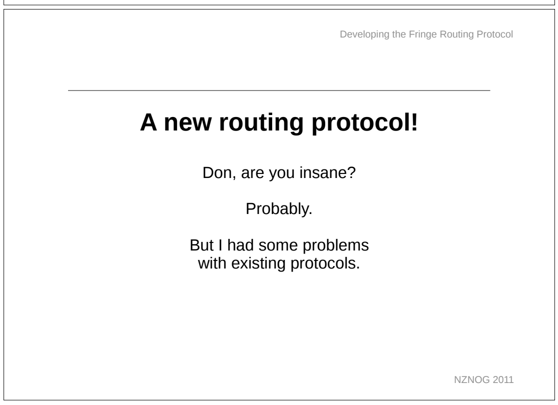
Appendix B

Slides from the FRP presentation in conjunction with Don Stokes of Knossos Networks, at the NZ Network Operators Group (NZNOG) conference in Wellington in February 2011.









Developing the Fringe Routing Protocol

FRP design goals

Subservient to primary IGP;

Routing table minimisation;

Unicast traffic;

Asymmetric route avoidance;

Security;

IPv6 support in single session (v4 or v6).

NZNOG 2011

Developing the Fringe Routing Protocol

Backbone Path Computation

Each router finds the shortest path to the backbone. This path forms the default route.

Paths can only be formed via routers that are a gateway or have a path to a gateway, thus:

- routing updates not processed unless a gateway path is established;
- paths propagate outwards from the backbone to the leaves.

Each router announces its gateway path to all its peers. This provides loop detection.

Routes announced toward the backbone retain the gateway flag (if set). Announcements to other peers clear the flag.

Only routes with the gateway flag set are announced to the backbone.

NZNOG 2011

Developing the Fringe Routing Protocol

Routing Metrics and Propagation

Each route has two metrics:

- The path cost of the route from the origin (c)
- The path cost to the designated gateway (gc), learned at routes origin.
- Plus a flag to indicate if the route has only been announced toward the backbone.

Routes are only propagated to peers if the cost of getting to the remote peer is less than that of going via the backbone, i.e:

c(route) + c(link) < gc(route) + gc(remote_peer)

NZNOG 2011

Developing the Fringe Routing Protocol

Protocol (1)

Unicast UDP, Multiple messages per packet. TTL set to 1.

64bit checksum, based on agreed secure hash and shared secret (currently SHA1).

Sequence numbers prevent replay attacks and detect restarts.

Separate message types for IPv4 and IPv6 actions, common link control message.

Regular Poll/ACK keeps link alive.

NZNOG 2011

Developing the Fringe Routing Protocol

Protocol (2)

Packet header: sum(64), loc-seq(32), rem-seq(32).

Message header: length(8), type(8). Note length is count of 32 bit words; max message is 1024 octets.

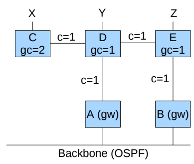
Message types:

- Session control (1), Subtypes: Poll, ACK, NACK
- Configuration (0x41, 0x61): cost, poll & fail times, router-ID.
- Gateway path (0x42, 0x62): Gateway cost, path
- Route announcement (0x43, 0x63): flags, prefix length, cost, gateway cost.
  - Flags: Begin, commit, null, update, delete, is gateway

NZNOG 2011

Developing the Fringe Routing Protocol

Example Network (1)



A announces X (c=2) & Y (c=1)

B announces Z (c=1)

D sees:

- X (via C, c=1, gw=1 gc=2)
- Y (local, gw=1, gc=1)
- Z (via E, c=1, gw=0, gc=1)

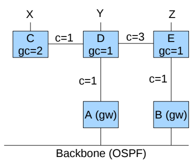
E sees:

- X (via D, c=2, gc=0 gc=2)
- Y (via D, c=1, gw=0, gc=1)
- Z (local, gw=1, gc=1)

NZNOG 2011

Developing the Fringe Routing Protocol

Example Network (2)



Note cost of D-E link now set to 3

A announces X (c=2) & Y (c=1)

B announces Z (c=1)

D sees:

- X (via C, c=1, gw=1 gc=2)
- Y (local, gw=1, gc=1)

Routes for X & Y not announced from D to E:

c(X) + c(D-E) > gc(X) + gc(E)

1 + 3 > 2 + 1

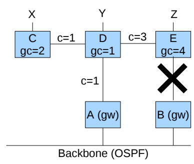
I.e. backbone path is lower cost.

Ditto announcements from E to D.

NZNOG 2011

Developing the Fringe Routing Protocol

Example Network (3)



E now chooses D as its gateway

A announces all routes via D

D sees:

- X (via C, c=1, gw=1 gc=2)
- Y (local, gw=1, gc=1)
- Z (via E, c=3, gw=1, gc=4)

E announces routes to D.

NZNOG 2011

Developing the Fringe Routing Protocol

Implementations

Knossos implementation:
- FreeBSD based (also run on NetBSD)
  - Porting would require rewrite of routing table interface.
- IPv4 only.
- Used in anger (i.e. paying customers) on Knossos FreeBSD routing platform.

Deb Shepherd's implementation:
- Portable, Quagga process;
- IPv6 capable (Deb note: eventually).

NZNOG 2011

Developing the Fringe Routing Protocol

FRP as a Quagga daemon: edited highlights

Deb Shepherd
Victoria University of Wellington
Catch 22

NZNOG 2011

Developing the Fringe Routing Protocol

What is Quagga?

- software routing suite
- open source
- Unix / Linux platforms
- fork of earlier GNU Zebra project

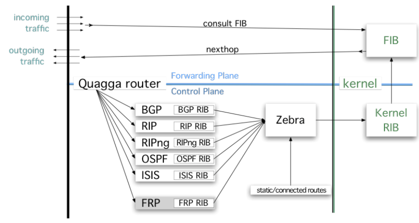
The quagga (*Equus quagga quagga*) is an extinct subspecies of the plains zebra — Wikipedia

Code and documentation (of sorts)
- http://www.quagga.net/
- Manual (of user commands, a selection)
- *Zebra for Dummies* aka *Zebra How-To*
  - http://www.quagga.net/zhh.html
- *Zebra hacking notes*
  - http://quagga.net/faq/zebra-hacking-guide.txt

NZNOG 2011

Developing the Fringe Routing Protocol

How Quagga works



NZNOG 2011

Developing the Fringe Routing Protocol

Developing in Quagga

User view
- 'vty' terminal front end with Cisco style commands

Developer view
- modular (mostly)
- core Zebra daemon
  - library
  - call back functions
- independent protocol daemons
  - `bgpd`, `isisd`, `ospfd`, `ospf6d`, `ripd`, `ripngd` & `frpd`
- various makefiles, install & config scripts, and platform specific additions

NZNOG 2011

Developing the Fringe Routing Protocol

FRP in Quagga — getting started

- FRP is distance vector, as are RIP and BGP
- FRP needs to be both IPv4 and IPv6

Ripped the RIP, RIPng and BGP daemons apart
- using RIP to look at IPv4 and RIPng to look at IPv6
- and BGP to look at an integrated Ipv4/6 daemon

Took `ripd` and used find/replace to turn it into `frpd`
- then started to pull out all the RIP specific code
- the intention was to develop a daemon shell

...
- then threw everything away & wrote shell from scratch

NZNOG 2011

Developing the Fringe Routing Protocol

FRP in Quagga — the daemon

`frp_main.c`
- set up and initialisation

`frp_zebra.c` / `frp_interface.c` / `frp_debug.c`
- hooks into Zebra including zclient & call back functions

`frpd.h`
- FRP daemon includes, defines, external variables, macros, prototypes

`frpd.c`
- main FRP daemon code
- config changes, sockets, events, incoming packets, polls

NZNOG 2011

Developing the Fringe Routing Protocol

FRP in Quagga — the daemon

`frp_packet.h`
- packet and message includes, defines, external variables, macros, prototypes

`frp_packet.c`
- FRP daemon code specific to packet handling

`frp_peer.c`
- FRP daemon code specific to peers and secrets

`frp_route.c`
- FRP daemon code specific to routing
- the FRP decision algorithm, the FRP RIB, injection of routes into the kernel via Zebra

NZNOG 2011

Developing the Fringe Routing Protocol

FRP in Quagga — events & threads

```
case FRP_EVENT_INCOMING:
   frp->t_read = thread_add_read (master,
            frp_incoming_packet, NULL, sock);
case FRP_EVENT_UPDATE:
   if (frp->t_update_interval)
      frp->update_trigger = 1;
   else if (! frp->t_update)
      frp->t_update = thread_add_event (master,
                  frp_update_peers, NULL, 0);
case FRP_EVENT_POLL:
   if (frp->t_poll)
   {  thread_cancel (frp->t_poll);
      frp->t_poll=NULL; }
   frp->t_poll = thread_add_timer (master,
            frp_poll_peers, NULL, frp->poll);
```

NZNOG 2011

Developing the Fringe Routing Protocol

FRP in Quagga — events & threads

```
int frp_incoming_packet (struct thread* t)
{  variables etc ...

   /* fetch socket then register myself */
   sock = THREAD_FD (t);
   frp->t_read = NULL;
   /* add myself to tne next event */
   frp_event (FRP_EVENT_INCOMING, sock);

   // read the packet from the socket
   ...
}
```

NZNOG 2011

Developing the Fringe Routing Protocol

FRP in Quagga — Don’s code

Code re–use?
- why yes! — two whole functions worth in fact

- Quagga is multi threaded, Don’s code is not
- so no real scope for re–use

Secrets
- FRP peers exchange secrets
- the diplomatic etiquette involved is exacting
- so I lifted Don’s security hash creation and checking functions and used them as black boxes

NZNOG 2011

Developing the Fringe Routing Protocol

FRP in Quagga — configuration

Live configuration of Quagga
- commands are executed via the ‘vty’ terminal
- obviously these need to be built in to the daemon

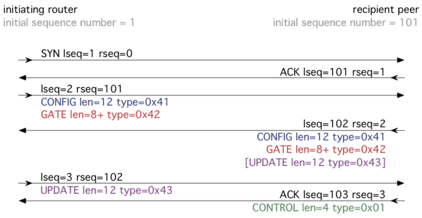
Config file
- loaded on startup (`main.c`)
- calls & executes the same functions as the ‘vty’ terminal

```
router frp
 network 10.0.1.0/24
 secret sapphire
 cost 3
 poll 60
 gateway yes
 neighbor 10.0.1.50 secret artemis
```

NZNOG 2011

Developing the Fringe Routing Protocol

FRP quirk — batched messages



NZNOG 2011

Developing the Fringe Routing Protocol

Questions ...

NZNOG 2011

Appendix C

Changes made to the Quagga code to hook in a new routing daemon

Quagga code changes

quagga/

config.h (actually auto generated so will need more work)

'PATH_FRPD_PID' undeclared	647	add `#define PATH_FRPD_PID "/var/run/frpd.pid"`
'FRP_VTYSH_PATH' undeclared	647	add `#define FRP_VTYSH_PATH "/var/run/frpd.vty"`

quagga/lib/

command.h		
'FRP_NODE' undeclared	63	add `FRP_NODE,` to `enum node_type`
RIP search	275	Is FRP needed here?
command.c		
RIPNG_NODE search	2386	add `case FRP_NODE:` to `switch (vty->node)`
RIPNG_NODE search	2445	add `case FRP_NODE:` to `switch (vty->node)`
distribute.c		
RIP search	762	seems to be rip/ripng specific
if_rmap.c		
RIP search	329	seems to be rip/ripng specific
log.h		
'ZLOG_FRP' undeclared	46	add `ZLOG_FRP,` to `enum`
log.c		
RIP search	42	add `"FRP",` to `char *zlog_proto_names[]`
RIP search	298	seems to be ripng specific
note in zebra.h [449]	826	add `DESC_ENTRY (ZEBRA_ROUTE_FRP, "frp", 'F' ),` to `struct zebra_desc_table`
memory.c		
RIP search	485	add `DEFUN`
RIP search	498	add 3 `install_element`'s

memtypes.h

'MTYPE_FRP_ROUTE' undeclared	7	`/* Auto-generated from memtypes.c by gawk. Do not edit! */` `memtypes.h` produced from `memtypes.c` by `memtypes.awk`

memtypes.c

note in memtypes.h	247	add `memory_list memory_list_frp[]`
note in memtypes.h	265	add `{ memory_list_frp, "FRP" },` to `struct mlist mlists` only added `structure`, `route info`, `interface` and `peer` to begin with check this all works next time compiled

route_types.h

RIP search		`/* Auto-generated from route_types.txt by gawk. Do not edit! */` `route_types.h` produced from `route_types.txt` by `route_types.awk`

route_types.txt

RIP search	43	add `ZEBRA_ROUTE_FRP, frp, frpd, 'F', 1, 1, "FRP"`
RIP search	65	add `ZEBRA_ROUTE_FRP, "Fringe Routing Protocol (FRP)"`

routemap.h

'RMAP_FRP' undeclared	42	add `RMAP_FRP,` to `enum`

vty.c

RIPNG_NODE search	690	add `case FRP_NODE:` to `switch (vty->node)`
RIPNG_NODE search	1101	add `case FRP_NODE:` to `switch (vty->node)`

zebra.h

'ZEBRA_ROUTE_FRP' undeclared	432	add `#define ZEBRA_ROUTE_FRP 11` change `#define ZEBRA_ROUTE_MAX 12`
'ZEBRA_FRP_DISTANCE_DEFAULT' undeclared	519	add `#define ZEBRA_EBGP_DISTANCE_DEFAULT 10` (used 10 because smaller than all other protocols - want FRP to have priority for now)

quagga/vtysh/

vtysh.c

RIPNG_NODE search	715	add `static struct cmd_node frp_node = { FRP_NODE, "%s(config-router)# ", };`
RIPNG_NODE search	1032	add new DEFUNSH currently based on RIPng but as other protocols have difference styles, may need more work potential for different v4/v6 setups in FRP
RIPNG_NODE search	1107	add `case FRP_NODE:` to `switch (vty->node)`
RIPNG_NODE search	2264+	variety of `install_element` stuff (probably needs more work)

vtysh_config.c

RIPNG_NODE search	160 (207)	add `else if (strncmp (line, "router frp", strlen ("router frp")) == 0) config = config_get (FRP_NODE, line);` to `switch (c)`

quagga/zebra/

redistribute.c

RIP search	248	add `case ZEBRA_ROUTE_FRP:` to `switch (type)`
RIP search	279	add `case ZEBRA_ROUTE_FRP:` to `switch (type)`

rib.h

RIP search	86	add FRP to other interior protocols `* sub-queue 2: RIP, RIPng, OSPF, OSPF6, IS-IS, FRP`

zebra_rib.c

RIP search	59	add FRP admin distance `{ZEBRA_ROUTE_FRP, 10},` using 10 because smaller than all other protocols - want FRP to have priority for now
RIP search	1222	add `[ZEBRA_ROUTE_FRP] = 2,` to `u_char meta_queue_map`

zebra_snmp.c

RIP search	231	not doing snmp therefore use default - no change

zebra_vty.c

RIP search	556	add `|| rib->type == ZEBRA_ROUTE_FRP` to `ONE_WEEK_SECOND`
RIP search	778	add `|| rib->type == ZEBRA_ROUTE_FRP` to `if`
RIP search	811	add `R - RIP,` to `SHOW_ROUTE_V4_HEADER`
RIP search	931	add `"Fringe Routing Protocol (FRP)\n"` to `DEFUN`
RIP search	954	add `else if (strncmp (argv[0], "f", 1) == 0) type = ZEBRA_ROUTE_FRP;`

Appendix D

Code

The *Quagga* FRP daemon implementation

The FRPsniffer implementation

The original implementation of FRP by Don Stokes

Appendix D

The Quagga FRP daemon implementation code

```
// ------------------------------------------------------------
// INCLUDES
// ------------------------------------------------------------
#include "frpd.h"



// ------------------------------------------------------------
// GLOBAL VARIABLES
// ------------------------------------------------------------

/* frpd command line options */      // DEB COMMENT: see no reason to change these
struct option longopts[] =
{ { "daemon",      no_argument,        NULL,  'd'},
  { "config_file", required_argument,  NULL,  'f'},
  { "pid_file",    required_argument,  NULL,  'i'},
  { "dryrun",      no_argument,        NULL,  'C'},
  { "help",        no_argument,        NULL,  'h'},
  { "vty_addr",    required_argument,  NULL,  'A'},
  { "vty_port",    required_argument,  NULL,  'P'},
  { "retain",      no_argument,        NULL,  'r'},
  { "user",        required_argument,  NULL,  'u'},
  { "group",       required_argument,  NULL,  'g'},
  { "version",     no_argument,        NULL,  'v'},
  { 0 }
};
/* frpd configuration file name and directory */
char config_default[] = SYSCONFDIR FRP_DEFAULT_CONFIG;     // DEB COMMENT: FRP_DEFAULT_CONFIG frpd.h as
'frpd.conf'
char *config_file;
/* frpd process ID saved for use by init system */
const char *pid_file;
/* frpd VTY bind address */
char *vty_addr;
/* frpd VTY connection port */
int vty_port;
/* frpd route retain mode flag */
int retain_mode;

/* frpd privileges */
zebra_capabilities_t _caps_p [] =
{ ZCAP_NET_RAW,                        // DEB COMMENT: privis.h [34]
  ZCAP_BIND                            // DEB COMMENT: privis.h [31]
};
// DEB COMMENT: zebra_privs_t privs.h [57]
struct zebra_privs_t frpd_privs =
{
#if defined(QUAGGA_USER)
  .user = QUAGGA_USER,                 // DEB COMMENT: config.h [649] as "quagga"
#endif
#if defined QUAGGA_GROUP
  .group = QUAGGA_GROUP,               // DEB COMMENT: config.h [643] as "quagga"
#endif
#ifdef VTY_GROUP
  .vty_group = VTY_GROUP,              // DEB COMMENT: config.h [733] (is this actually used?)
#endif
  .caps_p = _caps_p,                   // DEB FOLLOW UP: ???
  .cap_num_p = 2,                      // DEB FOLLOW UP: ???
  .cap_num_i = 0                       // DEB FOLLOW UP: ???
};

```



```
/* master thread */
struct thread_master *master;




// ------------------------------------------------------------
// FUNCTIONS
// ------------------------------------------------------------

/* frpd help information display */
static void
usage (char *progname, int status)
{ if (status != 0)
    fprintf (stderr, "Try `%s --help' for more information.\n", progname);
  else
    {
     printf ("Usage : %s [OPTION...]\n\
Daemon which manages FRP.\n\n\
-d, --daemon       Runs in daemon mode\n\
-f, --config_file  Set configuration file name\n\
-i, --pid_file     Set process identifier file name\n\
-A, --vty_addr     Set vty's bind address\n\
-P, --vty_port     Set vty's port number\n\
-r, --retain       When program terminates, retain added route by frpd.\n\
-u, --user         User to run as\n\
-g, --group        Group to run as\n\
-v, --version      Print program version\n\
-C, --dryrun       Check configuration for validity and exit\n\
-h, --help         Display this help and exit\n\
\n\
Report bugs to %s\n", progname, ZEBRA_BUG_ADDRESS);    // DEB COMMENT: version.h [31]
    }
  exit (status);
}

/* frpd signal hang up and re-initialise  handler */
static void
sighup (void)
{ zlog_info ("SIGHUP received");     // DEB COMMENT: log.h [124]
  frp_clean ();                      // DEB COMMENT: frpd.c
  frp_reset ();                      // DEB COMMENT: frpd.c
  /* reload config file */
  vty_read_config (config_file, config_default); // DEB COMMENT: vty.c [2333]
  /* create VTY socket */
  vty_serv_sock (vty_addr, vty_port, FRP_VTYSH_PATH);     // DEB COMMENT: vty.c [2141]
  // DEB COMMENT: FRP_VTYSH_PATH config.h [662] as '/var/run/frpd.vty'
}
/* frpd signal interrupt handler */
static void
sigint (void)
{ zlog_notice ("Terminating on signal");

  if (! retain_mode)
    frp_clean ();

  exit (0);
}
/* frpd SIGUSR1 handler */
static void
sigusr1 (void)
```

```
{   zlog_rotate (NULL);
}
/*  frpd signal handler */
struct quagga_signal_t frp_signals[] =
{   {   .signal = SIGHUP,
        .handler = &sighup,
    },
    {   .signal = SIGUSR1,
        .handler = &sigusr1,
    },
    {   .signal = SIGINT,
        .handler = &sigint,
    },
    {   .signal = SIGTERM,
        .handler = &sigint,
    },
};



// ------------------------------------------------------------
// FRPD MAIN
// ------------------------------------------------------------

int
main (int argc, char **argv)
{
    #ifdef DEB_DEBUG
        fprintf (stderr, "DEB DEBUG: entering frp_main.c - main\n");
    #endif //DEB_DEBUG

    int daemon_mode = 0;
    /*  configuration file name and directory */
    config_file = NULL;
    /*  process ID saved for use by init system */
    pid_file = PATH_FRPD_PID;       // DEB COMMENT: config.h [633] as '/var/run/frpd.pid'
    int dryrun = 0;
    /*  FRP VTY bind address */
    vty_addr = NULL;
    /*  FRP VTY connection port */
    vty_port = FRP_VTY_PORT;        // DEB COMMENT: frpd.h as '2609'
    /*  Route retain mode flag. */
    retain_mode = 0;

    char *p;
    char *progname;
    struct thread thread;

    /*  set umask before anything for security */
    umask (0027);                   // DEB COMMENT: like chmod 640

    /*  get program name */
    progname = ((p = strrchr (argv[0], '/')) ? ++p : argv[0]);

    /*  logging init */
    zlog_default = openzlog(progname, ZLOG_FRP, LOG_CONS|LOG_NDELAY|LOG_PID, LOG_DAEMON);

    /*  command line option parse */
    while (1)
    {
        int opt;
```

frp_main.c
Printed: Friday, 18 March 2011 10:38:11 AM


```
        opt = getopt_long (argc, argv, "df:i:hA:P:u:g:vC",    longopts, 0);
        if (opt == EOF)
            break;
        switch (opt)
        {
        case 0:
            break;
        case 'd':
            daemon_mode = 1;
            break;
        case 'f':
            config_file = optarg;
            break;
        case 'A':
            vty_addr = optarg;
            break;
        case 'i':
            pid_file = optarg;
            break;
        case 'P':
            /*  deal with atoi() returning 0 on failure, and frpd not listening on frpd port... */
            if (strcmp(optarg, "0") == 0)
            {   vty_port = 0;
                break;
            }
            vty_port = atoi (optarg);
            if (vty_port <= 0 || vty_port > 0xffff)
                vty_port = FRP_VTY_PORT;
                break;
        case 'r':
            retain_mode = 1;
            break;
        case 'u':
            frpd_privs.user = optarg;
            break;
        case 'g':
            frpd_privs.group = optarg;
            break;
        case 'v':
            print_version (progname);
            exit (0);
            break;
        case 'C':
            dryrun = 1;
            break;
        case 'h':
            usage (progname, 0);
            break;
        default:
            usage (progname, 1);
            break;
        }
    }

    /*  set up master thread / threads management */
    master = thread_master_create ();

    /*  library initialization */
    zprivs_init (&frpd_privs);
    signal_init (master, Q_SIGC(frp_signals), frp_signals);
    cmd_init (1);
```

```
    vty_init  (master);
    memory_init  ();

    /*  FRPd initialization  */
    frp_init  ();                          //  DEB COMMENT:  frpd.c,  where  most of the set up happens
    frp_zclient_init   ();                 //  DEB COMMENT:  frp_zebra.c

    /*  sort installed  commands */
    sort_node ();

    /*  get configuration  file  */
    vty_read_config  (config_file,  config_default);

    /*  start  execution only  if  not in dry-run  mode */
    if(dryrun)
       return(0);

    /*  change to the daemon program  */
    if  (daemon_mode &&  daemon (0, 0) < 0)
    {   zlog_err("FRPd  daemon failed:  %s",  strerror(errno));
       exit  (1);
    }

    /*  create  VTY socket,  start  tcp and unix  socket listeners  */
    vty_serv_sock  (vty_addr,  vty_port,  FRP_VTYSH_PATH);

    /*  create  process  id file  */
    pid_output (pid_file);

    /*  print  banner */
    zlog_notice ("FRPd %s starting:  vty@%d",  QUAGGA_VERSION, vty_port);

    /*  fetch next active  thread --  main program  loop */
    while  (thread_fetch (master,  &thread))
       thread_call  (&thread);

    /*  not reached */
    return  0;
}
```

```c
#ifndef _QUAGGA_FRPD_H
#define _QUAGGA_FRPD_H


#define DEB_DEBUG
//#define DEB_DEBUG_D
//#define DEB_DEBUG_IF
//#define DEB_DEBUG_PEER
#define DEB_DEBUG_PKT


//-------------------------------------------------------------
/* INCLUDES */
//-------------------------------------------------------------
#include <sys/types.h>
#include <stdlib.h>
#include <stdio.h>
#include <stdint.h>
#include <stddef.h>
#include <netinet/in.h>

#include "zebra.h"
#include "command.h"
#include "getopt.h"
#include "if.h"
#include "log.h"
#include "memory.h"
#include "privs.h"
#include "sigevent.h"
#include "thread.h"
#include "version.h"
#include "vty.h"
#include "prefix.h"
#include "zclient.h"
#include "table.h"
#include "sockopt.h"
#include "sockunion.h"
#include "stream.h"

#include "frp_packet.h"



//-------------------------------------------------------------
/* DEFINES */
//-------------------------------------------------------------

/* FRP version number. */
#define FRP_VERSION             1

/* Default config file name */
#define FRP_DEFAULT_CONFIG      "frpd.conf"     // DEB COMMENT: same as all the other protocols

/* FRP ports */
#define FRP_PORT_DEFAULT        343             // DEB COMMENT: port used by Don
#define FRP_VTY_PORT            2609            // DEB COMMENT: next available unused port in Quagga
                                                // DEB COMMENT: check on quagga-dev?
// frp router defaults
#define FRP_DEFAULT_COST        1
#define FRP_DEFAULT_POLL        5
#define FRP_DEFAULT_RETRY       1
```



```c

#define FRP_METRIC

#define FRP_INFINITY            0xffff
/* frp route types. */
#define FRP_ROUTE_RTE           0
#define FRP_ROUTE_STATIC        1

/* frp read/write types */
#define FRP_READ_REQUEST        1
#define FRP_READ_RESPONSE       2
#define FRP_WRITE_UPDATE        3
#define FRP_WRITE_KEEPALIVE     4

// need a macro here that generates a random number - using 1 for debugging purposes
#define NEW_SEQ_NO              0

// set to determine how many polls to wait before declaring a peer 'dead'
#define FRP_PEER_DEAD           5

// to support code lifted from Don's implementation
#define IPADDR                  u_int32_t
#define FRP_HASHSIZE            8

// random number generation
#define RANDOM_SEED() srandom(time(NULL))
#define RANDOM_INT(__MIN__, __MAX__) ((__MIN__) + random() % ((__MAX__+1) - (__MIN__)))

//-------------------------------------------------------------
/* EXTERNAL VARIABLES */
//-------------------------------------------------------------

/* frp event. */
enum frp_event
{   FRP_EVENT_INCOMING,
    FRP_EVENT_UPDATE,
    FRP_EVENT_POLL,
};

// flags
enum flag
{   OFF,
    ON,
};

/* frp structure. */
struct frp
{   /* frp version */
    int             version;
    /* frp output buffer */
    struct stream*  obuf;
    /* frp socket */
    int sock;
    /* frp threads */
    struct thread*  t_read;
    struct thread*  t_poll;
    struct thread*  t_update;
    struct thread*  t_update_interval;
    int             update_trigger;
    /* timer values. */
    unsigned long   update_time;
```

```
    unsigned long          timeout_time;
    unsigned long          garbage_time;
    /* frp routing information base (linked list) */
    struct route_table*  rib;
    /* frp only static routing information (linked list) */
    struct route_table*  routes;
    /* frp neighbors (linked list) */
    struct route_table*  neighbors;
    /* frp gateway path (linked list) */
    struct list*  gateway_path;
    // is this peer a gateway?
    enum flag              is_gateway_flag;
    // which peer is the current next hop to the current gateway?
    // - use this to trigger an update if this peer changes it's path to gateway
    struct frp_peer*       gateway_nexthop;
    // number of hops to gateway
    int                    gateway_cost;
    // gateway types
#define FRP_GATEWAY_ALWAYS  0
#define FRP_GATEWAY_YES     1
#define FRP_GATEWAY_NO      2
    // secret
    const char*            secret;
    // cost of the link (16 bits)
    u_short                cost;
    // poll time (16 bits)
    u_short                poll;
    // retry time (16 bits)
    u_short                retry;
};

/* FRP specific interface configuration */
struct frp_interface
{   /* frp is enabled on this interface */
    int enable_network;
    int enable_interface;
    /* frp is running on this interface */
    int running;
    /* for filter type slot */
#define FRP_FILTER_IN      0
#define FRP_FILTER_OUT     1
#define FRP_FILTER_MAX     2
    /* access-list */
    struct access_list* list[FRP_FILTER_MAX];
    /* prefix-list */
    struct prefix_list* prefix[FRP_FILTER_MAX];
    /* route-map */
    struct route_map* routemap[FRP_FILTER_MAX];
    /* wake up thread */
    struct thread* t_wakeup;
    /* interface statistics */
    int recv_badpackets;
    int recv_badroutes;
    int sent_updates;
    /* passive interface */
    int passive;
};

// frp peer
struct frp_peer
{   struct frp_peer*  next;
```



```
    struct frp_peer*  prev;
    /* peer address */
    struct in_addr      address;
    const char*         secret;
    u_short             cost;
    u_short             poll;
    u_short             retry;
    // our current sequence number with this peer
    u_int32_t           lseq;
    // this peer's current sequence number
    u_int32_t           rseq;
    // current path to gateway for this peer
    struct list*        gateway_path;
    // cost of gateway, ie: number of hops in path to gateway
    int                 gateway_cost;
    // temporary storage for new incoming routes
    struct list*        rib;
    struct list*        temp_rib;
    // latest packet sent to this peer
    u_char*             packet_latest;
    u_int8_t            packet_latest_length;
    u_int32_t           packet_latest_lseq;
    // latest timer timestamps
    time_t              time_latest_packet;
    time_t              time_last_heard;
    time_t              time_sent_config;
    /* timeout thread */
    struct thread*      t_timeout;
    // flags
    enum flag           flag_alive;
    enum flag           flag_send_syn;
    enum flag           flag_send_poll;
    enum flag           flag_send_config;
    enum flag           flag_send_gateway;
    enum flag           flag_send_update;
    enum flag           flag_awaiting_ack;
    /* statistics */
    int                 recv_badpackets;
    int                 recv_badroutes;
};

// store route updates from a peer
struct frp_rte
{   u_int8_t             flags;          // update type flags (8 bits)
    u_int8_t             length;         // prefix length (8 bits)
    u_short              routecost;      // route cost (16 bits)
    u_short              gatecost;       // cost from originator to gateway (16 bits)
    struct prefix_ipv4   prefix;         // IP prefix (32 bits)
};

/* frp route information */
struct frp_info
{   struct in_addr          nexthop;
    u_int32_t               cost;
    enum flag               is_gateway_flag;
    /* this route's type */
    int type;
    /* sub type */
    int sub_type;
    // back pointer
    struct route_node*      rte_node;
```

```c
    /* which interface does this route come from */
    unsigned int ifindex;
};

/* buffer to store frp data */
union frp_pkt_buf
{  char buf[FRP_PKT_MAXSIZE];
    struct frp_pkt_hdr              frp_pkt_hdr;
    struct frp_msg_hdr              frp_msg_hdr;
    struct frp_msg_control          frp_msg_control;
    struct frp_msg_ipv4config       frp_msg_ip4config;
    struct frp_msg_ipv4gateway      frp_msg_ip4gateway;
    struct frp_msg_ipv4update       frp_msg_ip4update;
    struct frp_msg_ipv6config       frp_msg_ip6config;
    struct frp_msg_ipv6gateway      frp_msg_ip6gateway;
    struct frp_msg_ipv6update       frp_msg_ip6update;
};

extern struct zebra_privs_t frpd_privs;




//------------------------------------------------------------
// PROTOTYPES
//------------------------------------------------------------
/* there is only one frp strucutre */
extern struct frp* frp;
// and only one linked list of frp peers
extern struct list* frp_peers;

/* master thread */
extern struct thread_master* master;

extern void frp_init (void);
extern void frp_clean (void);
extern void frp_reset (void);
extern void frp_event (enum frp_event, int);

// frp_route.c
extern void frp_recompute_rib (struct frp_peer* peer);
extern void frp_delete_peer_from_rib (struct frp_peer* peer);

// frp_packet.c
extern int frp_send_packet (u_char *  buf, int size, struct sockaddr_in *to);
extern struct frp_pkt_hdr make_frp_pkt_hdr (struct in_addr local, struct frp_peer* peer, int flag, int len, u_char*
buf);
extern struct frp_msg_hdr make_frp_msg_hdr (int type);
extern struct frp_msg_control make_frp_msg_control (int type);
extern struct frp_msg_ipv4config make_frp_msg_ipv4config (struct in_addr peer);
extern int make_frp_msg_ipv4gateway (struct in_addr peer, struct frp_msg_ipv4gateway* msg);
extern int make_frp_msg_ipv4update (struct frp_peer* peer, int available_length, u_char* buf);
extern int build_syn_pkt (struct frp_peer* peer);
extern int build_ack_pkt (struct frp_peer* peer, struct sockaddr_in* from, struct in_addr local);
extern int build_nak_pkt (struct frp_peer* peer, struct sockaddr_in* from, struct in_addr local);
extern int build_batch_pkt (struct frp_peer* peer, struct sockaddr_in* from, struct in_addr local);

// frp_peer.c
extern void frp_peer_init (void);
extern int frp_neighbor_add (struct vty* vty, const char* ip_str, const char* secret);
extern struct frp_peer* frp_peer_lookup (struct in_addr *addr);
```



```c
extern u_int8_t* dohash(u_char* buf, int len, const char* secret, IPADDR sa, IPADDR da);
extern int checksecure(u_char* pkt, int len, struct frp_peer* peer, struct in_addr local);
extern void secure(struct frp_peer* peer, struct in_addr local, struct frp_pkt_hdr* pkt, int len);
extern struct prefix* find_local_address_for_peer(struct in_addr dest);

// frp_interface.c
extern struct route_table *frp_enable_network;
extern void frp_interface_init (void);
extern void frp_config_write_network (struct vty* vty);
extern int frp_neighbor_lookup (struct sockaddr_in*);
extern struct in_addr frp_get_interface_address (struct in_addr peer);

// frp_zebra.c
extern void frp_zclient_init (void);
extern void frp_zebra_ipv4_add (struct prefix_ipv4 *p, struct in_addr *nexthop, u_int32_t metric, u_char distance
extern void frp_zebra_ipv4_delete (struct prefix_ipv4 *p, struct in_addr *nexthop, u_int32_t metric);
/* zebra client API callback functions */
extern int frp_interface_add (int command, struct zclient *zclient, zebra_size_t length);
extern int frp_interface_delete (int command, struct zclient *zclient, zebra_size_t length);
extern int frp_interface_up (int command, struct zclient *zclient, zebra_size_t length);
extern int frp_interface_down (int command, struct zclient *zclient, zebra_size_t length);
extern int frp_interface_address_add (int command, struct zclient *zclient, zebra_size_t length);
extern int frp_interface_address_delete (int command, struct zclient *zclient, zebra_size_t length);



#endif /* _QUAGGA_FRPD_H */
```

```c
// ------------------------------------------------------------
// INCLUDES
// ------------------------------------------------------------
#include "frpd.h"
#include "frp_debug.h"



// ------------------------------------------------------------
// GLOBAL VARIABLES
// ------------------------------------------------------------

/* frp node structure */
static struct cmd_node frp_node =
{ FRP_NODE, "%s(config-router)#  ", 1 };

/* frp structure */
struct frp* frp = NULL;



// ------------------------------------------------------------
// PROTOTYPES
// ------------------------------------------------------------
static int frp_config_write (struct vty* vty);
static int frp_create (void);
static int frp_create_socket (struct sockaddr_in* from);
static int frp_incoming_packet (struct thread* t);
static int16_t frp_incoming_control_msg (struct frp_msg_control* msg, struct frp_peer* peer, struct sockaddr_in*
from, struct in_addr local);
static int16_t frp_incoming_config_msg (struct frp_msg_ipv4config* msg, struct frp_peer* peer, struct sockaddr_in
from, struct in_addr local);
static int16_t frp_incoming_gateway_msg (struct frp_msg_ipv4gateway* msg, struct frp_peer* peer, struct
sockaddr_in* from, struct in_addr local);
static int16_t frp_incoming_update_msg (struct frp_msg_ipv4update* msg, struct frp_peer* peer, struct sockaddr_i
from, struct in_addr local);

static int frp_update_peers (struct thread *t);
static int frp_update_interval (struct thread *t);
static int frp_poll_peers (struct thread *t);



// ------------------------------------------------------------
// FUNCTIONS
// ------------------------------------------------------------

/* create new frp instance and set it to global variable */
int
frp_create (void)
{
  #ifdef DEB_DEBUG
    fprintf (stderr, "DEB DEBUG: entering frpd.c - frp_create\n");
  #endif //DEB_DEBUG
  frp = XCALLOC (MTYPE_FRP, sizeof (struct frp));
  /* set initial values */
  frp->version = FRP_VERSION;
  frp->cost = FRP_DEFAULT_COST;
  frp->poll = FRP_DEFAULT_POLL;
  frp->retry = FRP_DEFAULT_RETRY;
  frp->is_gateway_flag = OFF;
```



```c
  frp->gateway_cost = FRP_INFINITY;
  /* initialize frp routing table */                    // DEB COMMENT: route_table_init table.h [56]
  frp->rib = route_table_init ();
  frp->routes = route_table_init ();
  frp->neighbors = route_table_init ();
  /* make output stream */
  frp->obuf = stream_new (FRP_PKT_MAXSIZE);
  /* make socket */
  frp->sock = frp_create_socket (NULL);
  if (frp->sock < 0)
  {
    #ifdef DEB_DEBUG
      fprintf (stderr, "DEB DEBUG: -- frp_create, failed to create socket\n");
    #endif //DEB_DEBUG
    return frp->sock;
  }
  #ifdef DEB_DEBUG
    fprintf (stderr, "DEB DEBUG: -- frp_create, created socket: %d\n", frp->sock);
  #endif //DEB_DEBUG
  /* create read and timer thread */
  frp_event (FRP_EVENT_INCOMING, frp->sock);
  frp_event (FRP_EVENT_UPDATE, 1);
  frp_event (FRP_EVENT_POLL, 1);
  #ifdef DEB_DEBUG
    fprintf (stderr, "DEB DEBUG: -- frp_create, finished events\n");
  #endif //DEB_DEBUG
  return 0;
}

/* create socket for frp protocol */
int
frp_create_socket (struct sockaddr_in* from)
{ int ret;
  int sock;
  struct sockaddr_in addr;

  #ifdef DEB_DEBUG
    fprintf (stderr, "DEB DEBUG: entering frpd.c - frp_create_socket\n");
  #endif //DEB_DEBUG

  // set address to '0'
  memset (&addr, 0, sizeof (struct sockaddr_in));
  // if address from elsewhere, set to that address
  if (!from)
  { addr.sin_family = AF_INET;
    addr.sin_addr.s_addr = INADDR_ANY;
    #ifdef HAVE_STRUCT_SOCKADDR_IN_SIN_LEN
      addr.sin_len = sizeof (struct sockaddr_in);
    #endif /* HAVE_STRUCT_SOCKADDR_IN_SIN_LEN */
  } else
  { memcpy(&addr, from, sizeof(addr));
  }

  /* sending port must always be the frp port */
  addr.sin_port = htons (FRP_PORT_DEFAULT);

  /* make datagram socket */
  sock = socket (AF_INET, SOCK_DGRAM, 0);
  if (sock < 0)
  { zlog_err("Cannot create UDP socket: %s", safe_strerror(errno));
    exit (1);
```

```
    }
    // set the socket options
    sockopt_reuseaddr (sock);                    // DEB COMMENT: sockunion.c [455]
    sockopt_reuseport (sock);                    // DEB COMMENT: sockunion.c [472]
    setsockopt_so_recvbuf (sock, FRP_PKT_MAXSIZE);   // DEB COMMENT: sockopt.c [27]

    #ifdef DEB_DEBUG
        fprintf (stderr, "DEB DEBUG: -- frp_create_socket, after make socket, port: %d\n", (int) ntohs(addr.sin_port)
    #endif //DEB_DEBUG

    // bind address to socket
    if (frpd_privs.change (ZPRIVS_RAISE))
    {   zlog_err ("frp_create_socket: could not raise privs");
    }
    if ( (ret = bind (sock, (struct sockaddr *) & addr, sizeof (addr))) < 0)
    {   int save_errno = errno;
        if (frpd_privs.change (ZPRIVS_LOWER))
        {   zlog_err ("frp_create_socket: could not lower privs");
        }
        zlog_err("%s: Can't bind socket %d to %s port %d: %s", __func__, sock, inet_ntoa(addr.sin_addr), (int) ntohs(addr.sin_port), safe_strerror(save_errno));
        close (sock);
        #ifdef DEB_DEBUG
            fprintf (stderr, "DEB DEBUG: -- frp_create_socket, after bind %d\n", ret);
        #endif //DEB_DEBUG
        return ret;
    }
    if (frpd_privs.change (ZPRIVS_LOWER))
    {   zlog_err ("frp_create_socket: could not lower privs");
    }

    return sock;
}

/* frp configuration write function */
// DEB COMMENT: called when FRP_NODE is installed
// provides the FRP part of the 'write' command from 'router frp' to the end of the config
// needs adding to each time a command set is added - use rip/ripng as templates
// commands are written out in same form as entered from command line
int
frp_config_write (struct vty* vty)
{   int write = 0;
    struct route_node* rn;
    #ifdef DEB_DEBUG
        zlog_debug ("DEB DEBUG: entering frpd.c - frp_config_write");
    #endif //DEB_DEBUG
    if (frp)                                  // DEB COMMENT: ie: only in frpd(config-router)# mode
    {   /* router frp */
        vty_out (vty, "router frp%s", VTY_NEWLINE);
        /* network */
        frp_config_write_network (vty);
        // router setup
        if (frp->is_gateway_flag == ON)
        {   vty_out (vty, " gateway yes %s", VTY_NEWLINE);

        }
        vty_out (vty, " secret %s %s", frp->secret, VTY_NEWLINE);
        if (frp->cost != FRP_DEFAULT_COST)
        {   vty_out (vty, " cost %d %s", frp->cost, VTY_NEWLINE);
        }
        if (frp->poll != FRP_DEFAULT_POLL)
```



```
        {   vty_out (vty, " poll %d %s", frp->poll, VTY_NEWLINE);

        }
        if (frp->retry != FRP_DEFAULT_RETRY)
        {   vty_out (vty, " retry %d %s", frp->retry, VTY_NEWLINE);

        }
        /* neighbours */
        for (rn = route_top (frp->neighbors); rn; rn = route_next (rn))
        {   if (rn->info)
            {   struct prefix* rn_p = (struct prefix*)&rn->p;
                struct in_addr address;
                int addr = inet_pton (AF_INET, inet_ntoa (rn_p->u.prefix4), &address);
                #ifdef DEB_DEBUG
                    zlog_debug ("DEB DEBUG: -- frp_config_write - address=%s", inet_ntoa (address));
                #endif //DEB_DEBUG
                struct frp_peer* peer = frp_peer_lookup (&address);
                #ifdef DEB_DEBUG
                    zlog_debug ("DEB DEBUG: -- frp_config_write - peer->addr=%s, peer->secret=%s", inet_ntoa (peer->address), peer->secret);
                #endif //DEB_DEBUG
                if (peer == NULL)
                {   vty_out (vty, " neighbor %s peer is null %s", inet_ntoa (rn_p->u.prefix4), VTY_NEWLINE);
                }
                else
                {   vty_out (vty, " neighbor %s secret %s %s", inet_ntoa (rn_p->u.prefix4), peer->secret, VTY_NEWLINE)
                }
            }
        }
        write++;
    }
    return write;
}

void
frp_clean ()
{
    #ifdef DEB_DEBUG
        fprintf (stderr, "DEB DEBUG: entering frpd.c - frp_clean\n");
    #endif //DEB_DEBUG
    return;
}

void
frp_reset ()
{
    #ifdef DEB_DEBUG
        fprintf (stderr, "DEB DEBUG: entering frpd.c - frp_reset\n");
    #endif //DEB_DEBUG
    return;
}



// _____events____________________________________________________
void
frp_event (enum frp_event event, int sock)
{
    #ifdef DEB_DEBUG
        zlog_debug ("DEB DEBUG: entering frpd.c - frp_event");
    #endif //DEB_DEBUG
```

```
    switch (event)
    {
      case FRP_EVENT_INCOMING:
        #ifdef DEB_DEBUG
          zlog_debug ("DEB DEBUG:  -- frp_event, case: FRP_EVENT_INCOMING");
        #endif //DEB_DEBUG
        // create a new read thread and tell it to run frp_incoming_packet()
        frp->t_read = thread_add_read (master, frp_incoming_packet, NULL, sock);
        break;
      case FRP_EVENT_UPDATE:
        #ifdef DEB_DEBUG
          zlog_debug ("DEB DEBUG:  -- frp_event, case: FRP_EVENT_UPDATE");
        #endif //DEB_DEBUG
        if (frp->t_update_interval)
          frp->update_trigger = 1;
        else if (! frp->t_update)
          frp->t_update = thread_add_event (master, frp_update_peers, NULL, 0);
        break;
      case FRP_EVENT_POLL:
        #ifdef DEB_DEBUG
          zlog_debug ("DEB DEBUG:  -- frp_event, case: FRP_EVENT_POLL");
        #endif //DEB_DEBUG
        if (frp->t_poll)
        { thread_cancel (frp->t_poll);
          frp->t_poll = NULL;
        }
        frp->t_poll = thread_add_timer (master, frp_poll_peers, NULL, (unsigned long)frp->poll);
        break;
      default:
        #ifdef DEB_DEBUG
          fprintf (stderr, "DEB DEBUG:  -- frp_event, case: default\n");
        #endif //DEB_DEBUG
        // complain and die
        zlog_info ("Unknown FRP event %d received", event);
        break;
    }
}



// _____FRP_EVENT_INCOMING_____________________________________________________
int
frp_incoming_packet (struct thread* t)
{
    #ifdef DEB_DEBUG
      zlog_debug ("DEB DEBUG: entering frpd.c - frp_incoming_packet");
    #endif //DEB_DEBUG
    int             sock;
    socklen_t       fromlen;
    int             pkt_length;
    struct sockaddr_in      from;
    struct in_addr          local;
    struct interface*       ifp;
    struct connected*       ifc;
    struct frp_interface*   ri;
    // read buffer
    union frp_pkt_buf       in_pkt_buf;
//    u_char*                   current_pos;
    //peer packet is from
    struct frp_peer*        peer;
    int                     secure;
```



```
    // outgoing packet storage
    struct frp_pkt_hdr*     pkt_hdr;
//    struct frp_msg_hdr*     msg_hdr;
    struct frp_pkt_hdr      ack_pkt;
    int                     sent;

    // ---1:  SET UP STUFF-------------------
    /* fetch socket then register myself */
    sock = THREAD_FD (t);
    frp->t_read = NULL;
    /* add myself to tne next event */
    frp_event (FRP_EVENT_INCOMING, sock);
    /* frpd manages only IPv4 */
    memset (&from, 0, sizeof (struct sockaddr_in));
    fromlen = sizeof (struct sockaddr_in);
    // read the packet from the socket
    // DEB COMMENT: rip/ng have a recvmsg/recv_packet  function that does all sorts of addotional checking - COME E
TO THIS
    pkt_length = recvfrom (sock, (char *)&in_pkt_buf.buf, FRP_PKT_MAXSIZE, 0, (struct sockaddr *) &from, &froml
    if (pkt_length < 0)
    { zlog_info ("recvfrom failed: %s", safe_strerror  (errno));
      return pkt_length;
    }
    /* which interface is this packet from */
    ifp = if_lookup_address (from.sin_addr);
    /* frp packet received */
    if (IS_FRP_DEBUG_EVENT)
      zlog_debug ("RECV packet from %s port %d on %s", inet_ntoa (from.sin_addr), ntohs (from.sin_port), ifp ?
ifp->name : "unknown");
    /* if this packet comes from unknown interface, ignore it */
    if (ifp == NULL)
    { zlog_info ("frp_incoming_packet: cannot find interface for packet from %s port %d", inet_ntoa(from.sin_addr),
(from.sin_port));
      return -1;
    }
    ifc = connected_lookup_address (ifp, from.sin_addr);
    if (ifc == NULL)
    { zlog_info ("frp_incoming_packet: cannot find connected address for packet from %s " "port %d on interface %s"
inet_ntoa(from.sin_addr), ntohs (from.sin_port), ifp->name);
      return -1;
    }
    /* packet length check */
    if (pkt_length < FRP_PKT_MINSIZE)
    { zlog_warn ("packet size %d is smaller than minimum size %d", pkt_length, FRP_PKT_MINSIZE);
      return pkt_length;
    }
    if (pkt_length > FRP_PKT_MAXSIZE)
    { zlog_warn ("packet size %d is larger than max size %d", pkt_length, FRP_PKT_MAXSIZE);
      return pkt_length;
    }
    /* is frp running or is this frp neighbor */
    ri = ifp->info;
    if (! ri->running && ! frp_neighbor_lookup (&from))
    { if (IS_FRP_DEBUG_EVENT)
      { zlog_debug ("FRP is not enabled on interface %s.", ifp->name);
      }
      return -1;
    }

    // ---2:  EXTRACT PACKET HEADER AND CHECK HAVE A VALID PACKET --------------------
    /* get the packet header from the buffer */
```

```c
    pkt_hdr = &in_pkt_buf.frp_pkt_hdr;
    // which peer is the packet from?
    peer = frp_peer_lookup (&from.sin_addr);
    // is it from an IP address we are allowed to talk to?
    //    - need to set up access lists for this - come back to
    // indicate this peer is still alive
    peer->flag_alive = ON;
    // timestamp this communication
    peer->time_last_heard = time (NULL);
    // does the security hash match?
    local = frp_get_interface_address (peer->address);
    // set secure to fail
    secure = 0;
    if (peer->secret != NULL)
    {  // check security hash: secret = peer's, sender = peer, dest = us
       secure = checksecure((u_char *)&in_pkt_buf.buf, pkt_length, peer, local);
    } else  // no secret so set secure to succeed
    {  secure = 1;
    }
    if (!secure)
    {  zlog_info ("secure check failed");
       return -1;
    }
    #ifdef DEB_DEBUG_D
       zlog_debug ("DEB DEBUG:  -- frp_incoming_packet - passed security check, SYN=%d (0x%x), ACK=%d (0x%x)"
ntohl(pkt_hdr->sendSeq), ntohl(pkt_hdr->sendSeq), ntohl(pkt_hdr->recipAck), ntohl(pkt_hdr->recipAck));
    #endif //DEB_DEBUG

    // ---3:  DETERMINE IF NEW OR ON-GOING CONVERSATION--------------------
    // store the peer's sequence number
    peer->rseq = ntohl(pkt_hdr->sendSeq);
    if (pkt_hdr->recipAck == 0)
    // have a SYN packet - peer has initiated a new conversation
    {
       #ifdef DEB_DEBUG
          zlog_debug ("DEB DEBUG:  -- frp_incoming_packet - have a SYN packet, SYN=%d (0x%x), ACK=%d (0x%x)",
ntohl(pkt_hdr->sendSeq), ntohl(pkt_hdr->sendSeq), ntohl(pkt_hdr->recipAck), ntohl(pkt_hdr->recipAck));
       #endif //DEB_DEBUG
       // is the packet the correct length?
       if (pkt_length!= FRP_PKT_HDRSIZE)
       {  zlog_info ("packet size check failed");
          return -1;
       }
       // create an ACK packet to send to new peer
       ack_pkt = make_frp_pkt_hdr (local, peer, FRP_ACK, FRP_PKT_HDRSIZE, NULL);
       // send ACK packet to peer
       sent = frp_send_packet ((u_char*)&ack_pkt, FRP_PKT_HDRSIZE, &from);
       if (sent)
       {  memcpy (peer->packet_latest, (u_char*)&ack_pkt, FRP_PKT_HDRSIZE);
          peer->packet_latest_length = FRP_PKT_HDRSIZE;
          peer->packet_latest_lseq = ntohl(pkt_hdr->sendSeq);
          peer->flag_awaiting_ack = ON;
          #ifdef DEB_DEBUG_D
             zlog_debug ("DEB DEBUG:  -- frp_incoming_packet - packet_latest_lseq=%d", peer->packet_latest_lseq);
          #endif //DEB_DEBUG
       } else
       {  zlog_debug ("packet number %d not sent", ack_pkt.sendSeq);
       }
    } else if (pkt_hdr->recipAck > 0)
    // have an ACK packet - ongoing conversiion
    {
```



```c
       #ifdef DEB_DEBUG
          zlog_debug ("DEB DEBUG:  -- frp_incoming_packet - have an ACK packet, SYN=%d (0x%x), ACK=%d (0x%x)"
ntohl(pkt_hdr->sendSeq), ntohl(pkt_hdr->sendSeq), ntohl(pkt_hdr->recipAck), ntohl(pkt_hdr->recipAck));
       #endif //DEB_DEBUG
       // does the packet rseq match our previous lseq?
       if (ntohl(pkt_hdr->recipAck) == peer->lseq)
       {  // is the packet just a packet header or does it contain a payload?
          if (pkt_length == FRP_PKT_HDRSIZE)    // is acking my syn so send flagged message(s) - config, gateway, up
          {
             #ifdef DEB_DEBUG
                zlog_debug ("DEB DEBUG:  -- frp_incoming_packet - is acking my syn so send flagged message(s), about
build_batch_pkt");
             #endif //DEB_DEBUG
             build_batch_pkt (peer, &from, local);
          } else  // for each message in the packet
          {  char*                 start_pos;
             char*                 current_pos;
             struct frp_msg_hdr*   current_msg;
             int16_t               msg_length;

             start_pos = (char*)&in_pkt_buf.buf;
             current_pos = start_pos + FRP_PKT_HDRSIZE;
             current_msg = NULL;
             msg_length = 0;
             current_msg = (struct frp_msg_hdr*)current_pos;

             // while the in buffer still has data in it
             int nak_response_sent = 0;
             while ((current_msg != NULL) && (current_pos < (start_pos + pkt_length)))
             {  // extract message length and message type and process message
//                msg_hdr = (struct frp_msg_hdr*)((char*)&in_pkt_buf.buf  + FRP_PKT_HDRSIZE);
                #ifdef DEB_DEBUG
                   zlog_debug ("DEB DEBUG:  -- frp_incoming_packet - length=%d, type=0x%x", current_msg->length,
current_msg->type);
                #endif //DEB_DEBUG
                switch (current_msg->type)
                {  case FRP_MSG_CONTROL:
                      #ifdef DEB_DEBUG
                         zlog_debug ("DEB DEBUG:  -- frp_incoming_packet - SWITCH control");
                      #endif //DEB_DEBUG
                      current_msg = (struct frp_msg_control*)current_pos;
                      msg_length = frp_incoming_control_msg ((struct frp_msg_control*)current_msg, peer, &from, lc
                      if (msg_length == -1)
                      {  nak_response_sent = 1;
                         current_pos += FRP_MSG_CONTROL_SIZE;
                      } else
                      {  current_pos += msg_length;
                      }
                      current_msg = (struct frp_msg_hdr*)current_pos;
                      break;
                   case FRP_MSG_IPV4CONFIG:
                      #ifdef DEB_DEBUG
                         zlog_debug ("DEB DEBUG:  -- frp_incoming_packet - SWITCH config");
                      #endif //DEB_DEBUG
                      current_msg = (struct frp_msg_ipv4config*)current_pos;
                      msg_length = frp_incoming_config_msg ((struct frp_msg_ipv4config*)current_msg, peer, &from,
                      current_pos += msg_length;
                      current_msg = (struct frp_msg_hdr*)current_pos;
                      break;
                   case FRP_MSG_IPV4GATEWAY:
                      #ifdef DEB_DEBUG
```

```
                        zlog_debug ("DEB DEBUG:  -- frp_incoming_packet - SWITCH gateway");
                    #endif //DEB_DEBUG
                    current_msg = (struct frp_msg_ipv4gateway*)current_pos;
                    msg_length = frp_incoming_gateway_msg ((struct frp_msg_ipv4gateway*)current_msg,  peer, &fr
local);
                    current_pos += msg_length;
                    current_msg = (struct frp_msg_hdr*)current_pos;
                    break;
                case FRP_MSG_IPV4UPDATE:
                    #ifdef DEB_DEBUG
                        zlog_debug ("DEB DEBUG:  -- frp_incoming_packet - SWITCH update");
                    #endif //DEB_DEBUG
                    current_msg = (struct frp_msg_ipv4update*)current_pos;
                    msg_length = frp_incoming_update_msg ((struct frp_msg_ipv4update*)current_msg,  peer, &from,
                    current_pos += msg_length;
                    current_msg = (struct frp_msg_hdr*)current_pos;
                    break;
                case FRP_MSG_IPV6CONFIG:
                    //call function
                    break;
                case FRP_MSG_IPV6GATEWAY:
                    //call function
                    break;
                case FRP_MSG_IPV6UPDATE:
                    //call function
                    break;
                default:
                    #ifdef DEB_DEBUG
                        zlog_debug ("DEB DEBUG:  -- frp_incoming_packet - SWITCH default");
                    #endif //DEB_DEBUG
                    break;
            }
        }
        // send packet, unless have responded to a NAK
        if (!nak_response_sent)
        {
            #ifdef DEB_DEBUG
                zlog_debug ("DEB DEBUG:  -- frp_incoming_packet - send packet, unless have responded to a NAK, ab
call build_batch_pkt");
            #endif //DEB_DEBUG
            build_batch_pkt (peer, &from, local);
        }
      }
    } else  // packet rseq does not match our previous lseq
    {
      // SEND NAK
      sent = build_nak_pkt (peer, &from, local);
      #ifdef DEB_DEBUG
         zlog_debug ("DEB DEBUG:  -- 4 - pkt_hdr->recipAck != peer->lseq - STOP");
      #endif //DEB_DEBUG
    }
  } else  // have an invalid packet
  {  zlog_info ("invalid incoming packet");
     return -1;
  }
  return pkt_length;
}


int16_t
frp_incoming_control_msg (struct frp_msg_control* msg, struct frp_peer* peer, struct sockaddr_in* from, struct
```



```
in_addr local)
{  int   sent;
   #ifdef DEB_DEBUG
      zlog_debug ("DEB DEBUG: entering frpd.c - frp_incoming_control_msg");
   #endif //DEB_DEBUG
   switch (msg->type)
   {  case FRP_CTRL_POLL:
         #ifdef DEB_DEBUG
            zlog_debug ("DEB DEBUG:  -- frp_incoming_control_msg - FRP_CTRL_POLL");
         #endif //DEB_DEBUG
         // send sn ACK
//          sent = build_ack_pkt (peer, from, local);
         break;
      case FRP_CTRL_ACK:
         #ifdef DEB_DEBUG
            zlog_debug ("DEB DEBUG:  -- frp_incoming_control_msg - FRP_CTRL_ACK");
         #endif //DEB_DEBUG
         // end of conversation - do nothing execpt update peer record
         peer->flag_awaiting_ack = OFF;
         #ifdef DEB_DEBUG_D
            zlog_debug ("DEB DEBUG:  -- frp_incoming_control_msg - packet_latest_lseq=%d", peer->packet_latest_ls
         #endif //DEB_DEBUG
         break;
      case FRP_CTRL_NAK:
         #ifdef DEB_DEBUG
            zlog_debug ("DEB DEBUG:  -- frp_incoming_control_msg - FRP_CTRL_NAK");
         #endif //DEB_DEBUG
         // update last packet header with new seqs & hash
         struct frp_pkt_hdr new_pkt_hdr = make_frp_pkt_hdr (local, peer, FRP_ACK, peer->packet_latest_length,
(u_char*)peer->packet_latest);
         memcpy((u_char*)peer->packet_latest,  &new_pkt_hdr, FRP_PKT_HDRSIZE);
         // resend last packet
         sent = frp_send_packet ((u_char*)&peer->packet_latest,  peer->packet_latest_length, from);
         if (!sent)
         {  zlog_debug ("packet not sent");
         }
         return -1;
         break;
   }
   return msg->msg_hdr.length;
}

int16_t
frp_incoming_config_msg (struct frp_msg_ipv4config* msg, struct frp_peer* peer, struct sockaddr_in* from, struc
in_addr local)
{
   #ifdef DEB_DEBUG
      zlog_debug ("DEB DEBUG: entering frpd.c - frp_incoming_config_msg");
   #endif //DEB_DEBUG
   // extract config info and update peer record
   peer->cost = ntohs(msg->cost);
   peer->poll = ntohs(msg->poll);
   peer->retry = ntohs(msg->retry);
   #ifdef DEB_DEBUG
      zlog_debug ("DEB DEBUG:  -- frp_incoming_control_msg - msg cost=%d, poll=%d, retry=%d",  ntohs(msg->cost
ntohs(msg->poll), ntohs(msg->retry));
      zlog_debug ("DEB DEBUG:  -- frp_incoming_control_msg - peer cost=%d, poll=%d, retry=%d",  peer->cost,
peer->poll, peer->retry);
   #endif //DEB_DEBUG
   return msg->msg_hdr.length;
}
```

```

int16_t
frp_incoming_gateway_msg (struct frp_msg_ipv4gateway* msg, struct frp_peer* peer, struct sockaddr_in* from,
struct in_addr local)
{
    #ifdef DEB_DEBUG
        zlog_debug ("DEB DEBUG: entering frpd.c - frp_incoming_gateway_msg");
    #endif //DEB_DEBUG
    // extract config info and update peer record
    int msg_length = msg->msg_hdr.length;
    // (msg_length - msg_hdr - cost) / length of single address in path
    int path_length = (msg_length - 32) / 32;
    int remainder = (msg_length - 32) % 32;
    struct in_addr* current_pos = (struct in_addr*)((char*)msg + 32);

    if ((path_length == 0) || (remainder != 0))
    { //not a valid path
        peer->gateway_cost = FRP_INFINITY;
        #ifdef DEB_DEBUG
            zlog_debug ("DEB DEBUG: -- frp_incoming_control_msg - invalid path length - msg cost=%d", ntohs(msg->cc
        #endif //DEB_DEBUG
        return msg->msg_hdr.length;
    } else
    { // create new list
        struct list* gate_path = list_new ();
        // step through addresses and store in list
        for (int i = 1; i <= path_length; i++)
        { // collect and format address
            struct in_addr* current_addr = XCALLOC (MTYPE_FRP_PEER, sizeof (struct in_addr));
            memcpy (current_addr, current_pos, sizeof (struct in_addr));
            if (current_addr->s_addr == local.s_addr)
            { // if this is our address, don't want this path because is using us as a gateway.
                peer->gateway_cost = FRP_INFINITY;
                // get rid of list and current_addr
                list_free (gate_path);
                XFREE (MTYPE_FRP_PEER, current_addr);
                #ifdef DEB_DEBUG
                    zlog_debug ("DEB DEBUG: -- frp_incoming_control_msg - we are in list - msg cost=%d", ntohs(msg->c
                #endif //DEB_DEBUG
                return msg->msg_hdr.length;
            } else
            { // add address to list
                listnode_add (gate_path, current_addr);
                #ifdef DEB_DEBUG
                    zlog_debug ("DEB DEBUG: -- frp_incoming_control_msg - path address=%d", current_addr->s_addr);
                #endif //DEB_DEBUG
            }
            current_pos++;
        }
        // pass list to peer record
        peer->gateway_path = gate_path;
        peer->gateway_cost = ntohs(msg->cost);
        // check if this path is better than our current gateway path
        if ((peer->gateway_cost + peer->cost) < frp->gateway_cost)
        { frp->gateway_cost = peer->gateway_cost + peer->cost;
            frp->gateway_nexthop = peer;
            frp->gateway_path = peer->gateway_path;
            // flag each peer to send out a new gateway message
            struct listnode *node, *nnode;
            for (ALL_LIST_ELEMENTS (frp_peers, node, nnode, peer))
            { peer->flag_send_gateway = ON;
```



```
            }
        }
    }
    #ifdef DEB_DEBUG
        zlog_debug ("DEB DEBUG: -- frp_incoming_control_msg - msg cost=%d", ntohs(msg->cost));
        zlog_debug ("DEB DEBUG: -- frp_incoming_control_msg - peer cost=%d", peer->cost);
    #endif //DEB_DEBUG
    return msg->msg_hdr.length;
}

int16_t
frp_incoming_update_msg (struct frp_msg_ipv4update* msg, struct frp_peer* peer, struct sockaddr_in* from, stru
in_addr local)
{
    #ifdef DEB_DEBUG
        zlog_debug ("DEB DEBUG: entering frpd.c - frp_incoming_update_msg");
    #endif //DEB_DEBUG

    struct frp_rte*       new_route;
    new_route = XCALLOC (MTYPE_FRP_PEER, sizeof(struct frp_rte));
    struct frp_rte*     existing_route;

    // do we have a BEGIN flag? (using bitwise &)
    if (msg->flags & FRP_FLAG_BEGIN)
    { // have an existing batch?
        if (peer->temp_rib != NULL)
        { // kill existing temp rib
            list_delete (peer->temp_rib);
            peer->temp_rib = NULL;
        }
        // have a NULLRT?
        if (msg->flags & FRP_FLAG_NULLRT)
        { int peer_rib_size = peer->rib->count;
            // clear peer rib
            list_delete_all_node (peer->rib);
            // re-compute our routing table
            if (peer_rib_size > 0)
            { frp_recompute_rib (peer);
            }
            return msg->msg_hdr.length;
        }
        // extract route information
        new_route->flags = msg->flags;
        new_route->length = msg->length;
        new_route->routecost = ntohs(msg->routecost);
        new_route->gatecost = ntohs(msg->gatecost);
        // prefix requires a little more work
        struct prefix_ipv4 address;
        memset (&address, 0, sizeof (address));
        address.family = AF_INET;
        address.prefix = msg->prefix;
        address.prefixlen = msg->length; //or the length provided
        apply_mask_ipv4(&address);
        new_route->prefix = address;

        // have a DELETE?
        if (msg->flags & FRP_FLAG_DELETE)
        { // work thru peer rib looking for destination match to delete
            struct listnode* node;
            struct listnode* delete_node = NULL;
            for (ALL_LIST_ELEMENTS_RO (peer->rib, node, existing_route))
```

```
         // compare the two prefixes  to see if they are  the same host and the same network
         {  if (prefix_same  (&new_route->prefix,   &existing_route->prefix))
            {  delete_node = node;
               break;
            }
         }
         if (delete_node != NULL)
         {  list_delete_node (peer->rib,  delete_node);
         }
         return  msg->msg_hdr.length;
      }

      // have an UPDATE?
      if (msg->flags  & FRP_FLAG_UPDATE)
      {  // work thru peer rib looking for destination match to update
         struct listnode*  node;
         struct listnode*  update_node = NULL;
         for (ALL_LIST_ELEMENTS_RO  (peer->rib,  node, existing_route))
         // compare the two prefixes  to see if they are  the same host and the same network
         {  if (prefix_same  (&new_route->prefix,   &existing_route->prefix))
            {  update_node = node;
               break;
            }
         }
         if (update_node != NULL)
         {  list_delete_node (peer->rib,  update_node);
            listnode_add (peer->rib,  new_route);
         }
         return  msg->msg_hdr.length;
      }

      // create new temp rib
      peer->temp_rib  = list_new  ();

   } else       // no BEGIN flag
   {
      // have an existing  batch?
      if (peer->temp_rib  == NULL)
      {  zlog_info ("invalid  route update packet");
         return  msg->msg_hdr.length;
      }

   }

   // add info to temp rib
   listnode_add (peer->temp_rib,  new_route);

   // have a COMMIT?
   if (msg->flags  & FRP_FLAG_COMMIT)
   {  // replace peer rib with temp rib
      list_delete  (peer->rib);
      peer->rib  = peer->temp_rib;
      peer->temp_rib  = NULL;
      frp_recompute_rib  (peer);
   }

   return  msg->msg_hdr.length;
}
```



```


// _____FRP_EVENT_UPDATE____________________________________________

/*  Execute an event update */
static int
frp_update_peers (struct thread *t)
{
   #ifdef DEB_DEBUG
      zlog_debug ("DEB DEBUG: entering frpd.c - frp_update_peers");
   #endif //DEB_DEBUG
   int interval;
   /*  clear thred pointer */
   frp->t_update = NULL;
   /*  cancel interval  timer */
   if (frp->t_update_interval)
   {  thread_cancel (frp->t_update_interval);
      frp->t_update_interval  = NULL;
   }
   frp->update_trigger  = 0;
   /*  logging triggered  update */
   if (IS_FRP_DEBUG_EVENT)
   {  zlog_debug ("update triggered");
   }

   // for each peer, unless quiescent
   if ((frp->gateway_cost  == FRP_INFINITY)  || (frp->gateway_cost  == 0))
   {
      struct frp_peer*  peer;
      struct listnode *node, *nnode;
      for (ALL_LIST_ELEMENTS (frp_peers,  node, nnode, peer))
      {
         // set update flag
         peer->flag_send_update = ON;

         // if flag_awaiting_ack  is off
         if (peer->flag_awaiting_ack  == OFF)
         {
            // send SYN packet to peer
            int sent = build_syn_pkt (peer);
            if (sent)
            {  zlog_debug ("sent SYN packet to %s",  inet_ntoa (peer->address));
            }
         }
      }
   }

   /*  After  a triggered  update is sent, a timer  should be set for a
   random interval  between 1 and 5 seconds.  If other changes that
   would trigger  updates occur before the timer  expires,  a single
   update is triggered  when the timer  expires  */
   interval  = (random () % 5) + 1;
   frp->t_update_interval  = thread_add_timer  (master,  frp_update_interval,   NULL, interval);
   return  0;
}

/*  Event update interval  timer  */
static int
frp_update_interval  (struct thread *t)
{  int frp_update_peers (struct thread *);
   frp->t_update_interval  = NULL;
```

```
    if (frp->update_trigger)
    {  frp->update_trigger  = 0;
       frp_update_peers (t);
    }
    return  0;
}


// _____FRP_EVENT_POLL__________________________________________

/*  frp's  periodical  timer  */
static  int
frp_poll_peers  (struct thread *t)
{  /*  clear  timer  pointer  */
    frp->t_poll  = NULL;
    if  (IS_FRP_DEBUG_EVENT)
    {  zlog_debug ("poll  timer  fired");
    }
    // check for  peers  with  update flags  set
    // check for  new peers  and send a SYN packet
    // check for  peers  with  update flags  set and send a batch message
    // check for  peers  that haven't responded and resend last  message or declare  dead
    int sent;
    struct frp_peer*  peer;
    struct listnode *node,  *nnode;
    for (ALL_LIST_ELEMENTS  (frp_peers,  node, nnode, peer))
    {  //  if  peer hasn't responded for  X polls,  declare  dead and move on - set flags  so handle resurrection  correctly
       if ((peer->flag_alive  == OFF) || (peer->flag_awaiting_ack  > FRP_PEER_DEAD))
       {  peer->flag_alive  = OFF;
          peer->flag_awaiting_ack  = OFF;
          peer->flag_send_poll  = OFF;
          peer->flag_send_syn  = OFF;
          peer->flag_send_config  = ON;
          peer->flag_send_update = ON;
          peer->flag_awaiting_ack  = ON;
          #ifdef  DEB_DEBUG
             zlog_debug ("DEB DEBUG:  --  frp_poll_peers  -  %s  is  dead", inet_ntoa (peer->address));
          #endif //DEB_DEBUG
       } else   // ie: not dead
       {  if (peer->flag_awaiting_ack  == 1)
          {  peer->flag_awaiting_ack  ++;
          } else if (peer->flag_awaiting_ack  > 1)
          {  //  if a peer hasn't acked for  more than a poll cycle,  resend last  message
             struct  sockaddr_in  socket;
             socket.sin_family  = AF_INET;
             socket.sin_port  = htons(FRP_PORT_DEFAULT);
             socket.sin_addr  = peer-> address;
             sent = frp_send_packet (peer->packet_latest,  peer->packet_latest_length,  &socket);
             if (sent)
             {  peer->flag_awaiting_ack  ++;
             } else
             {  zlog_debug ("packet not sent");
             }
             #ifdef  DEB_DEBUG
                zlog_debug ("DEB DEBUG:  --  frp_poll_peers  -  awaiting  ACK %d from  %s",  peer->packet_latest_lseq,  ir
(peer->address));
             #endif //DEB_DEBUG
          } else if ((peer->flag_send_syn)  || (peer->flag_send_config)  || (peer->flag_send_gateway)  ||
(peer->flag_send_update))
          {  //  if flags  are set,  send a batch packet
             sent = build_syn_pkt  (peer);
```



```
             if (sent)
             {  peer->flag_send_syn  = OFF;
                zlog_debug ("sent SYN packet to %s",  inet_ntoa (peer->address));
             }
          } else
          {  //  have we heard from  the peer within  a poll period?  if  not, poll
             time_t now = time (NULL);
             double test = difftime  (now,  peer->time_last_heard);
             if (test > frp->poll)
             {  peer->flag_send_poll  = ON;
                sent = build_syn_pkt  (peer);
                if  (sent)
                {  peer->flag_send_syn  = OFF;
                   zlog_debug ("sent SYN packet to %s",  inet_ntoa (peer->address));
                }
             }
          }
       }
    }
    /*  polls may be suppressed  if  a regular  update is due by the time the polls would be sent */
    if (frp->t_update_interval)
    {  thread_cancel  (frp->t_update_interval);
       frp->t_update_interval  = NULL;
    }
    frp->update_trigger  = 0;
    /*  register  myself  */
    frp_event  (FRP_EVENT_POLL, 0);
    return  0;
}








// -----------------------------------------------------------------
// ROUTER COMMAND DEFUN
// -----------------------------------------------------------------

// Create  the FRP router  config command -  frpd(config)#
DEFUN (router_frp,
    router_frp_cmd,
    "router  frp",
    "Enable a routing process\n"
    "Fringe  Routing Protocol  (FRP)\n")
{
    int ret;
    /*  if frp  is  not enabled before */
    if (! frp)
    {
       #ifdef  DEB_DEBUG
          fprintf  (stderr,  "DEB DEBUG: entering frpd.c - router_frp\n");
       #endif //DEB_DEBUG
       ret = frp_create  ();
       if (ret < 0)
       {  zlog_info ("Can't create FRP");
          return  CMD_WARNING;
       }
    }
```

```
   vty->node = FRP_NODE;
   vty->index = frp;
   return CMD_SUCCESS;
}
DEFUN (no_router_frp,
      no_router_frp_cmd,
      "no router frp",
      NO_STR
      "Enable a routing process\n"
      "Routing Information Protocol (FRP)\n")
{
   if (frp)
      frp_clean ();
   return CMD_SUCCESS;
}

// is this router a gateway>
DEFUN (frp_gateway,
      frp_gateway_cmd,
      "gateway (yes|no)",
      "Is this router a FRP gateway? (yes|no)\n"
      "\n"
      "\n")
{
   const char* test1 = "yes";
   const char* test2 = argv[0];
   int result = strcmp(test1, test2);
   if (result == 0 || (*argv[0] == 'y'))
   {  frp->is_gateway_flag = ON;
      frp->gateway_cost = 0;
   } else
   {  frp->is_gateway_flag = OFF;
      frp->gateway_cost = FRP_INFINITY;
   }
   // flag each peer to send out a new gateway message
   struct frp_peer* peer;
   struct listnode *node, *nnode;
   for (ALL_LIST_ELEMENTS (frp_peers, node, nnode, peer))
   {  frp_recompute_rib (peer);
      peer->flag_send_gateway = ON;
      peer->flag_send_update = ON;
   }
   #ifdef DEB_DEBUG
      vty_out (vty, "DEB DEBUG: -- frp_gateway_cmd - result=%d, frp->is_gateway_flag=%d %s", result,
frp->is_gateway_flag, VTY_NEWLINE);
   #endif //DEB_DEBUG
   return CMD_SUCCESS;
}

// set our secret
// - this is only set in the conf file, not on the command line
// - so the command is not installed
DEFUN (frp_secret,
      frp_secret_cmd,
      "secret WORD",
      "\n"
      "\n")
{
   if (!strcmp(argv[0], ""))
   {  vty_out (vty, "Please specify address and secret A.B.C.D secret WORD%s", VTY_NEWLINE);
      return CMD_WARNING;
```



```
   }
   char* secret = (char*)argv[0];
   frp->secret = (char*) XCALLOC (MTYPE_FRP, strlen(secret) + 1);
   memcpy (frp->secret, secret, strlen(secret));
   #ifdef DEB_DEBUG
      vty_out (vty, "DEB DEBUG: -- frp_secret_cmd - frp->secret = %s %s", frp->secret, VTY_NEWLINE);
   #endif //DEB_DEBUG
   return CMD_SUCCESS;
}

// set the cost of the link
DEFUN (frp_cost,
      frp_cost_cmd,
      "cost INT",
      "Cost of link\n"
      "Set cost of link as an integer\n")
{
   // set cost
   frp->cost = atoi(argv[0]);
   // flag each peer to send out a new config message
   struct frp_peer* peer;
   struct listnode *node, *nnode;
   for (ALL_LIST_ELEMENTS (frp_peers, node, nnode, peer))
   {  peer->flag_send_config = ON;
   }
   #ifdef DEB_DEBUG
      vty_out (vty, "DEB DEBUG: -- frp_cost_cmd - frp->cost = %d %s", frp->cost, VTY_NEWLINE);
   #endif //DEB_DEBUG
   return CMD_SUCCESS;
}

// set the poll time
DEFUN (frp_poll,
      frp_poll_cmd,
      "poll INT",
      "Poll / keepalive frequency\n"
      "Set the poll time in seconds as an integer\n")
{
   // set poll
   frp->poll = atoi(argv[0]);
   // flag each peer to send out a new config message
   struct frp_peer* peer;
   struct listnode *node, *nnode;
   for (ALL_LIST_ELEMENTS (frp_peers, node, nnode, peer))
   {  peer->flag_send_poll = ON;
   }
   #ifdef DEB_DEBUG
      vty_out (vty, "DEB DEBUG: -- frp_poll_cmd - frp->poll = %d %s", frp->poll, VTY_NEWLINE);
   #endif //DEB_DEBUG
   return CMD_SUCCESS;
}

// set the retry time
DEFUN (frp_retry,
      frp_retry_cmd,
      "retry INT",
      "Timeout to failure after acked packet\n"
      "Set the retry time in seconds as an integer\n")
{
   // set retry
   frp->retry = atoi(argv[0]);
```

```
    // flag each peer to send out a new config message
    struct frp_peer* peer;
    struct listnode *node, *nnode;
    for (ALL_LIST_ELEMENTS (frp_peers, node, nnode, peer))
    { peer->flag_send_config = ON;
    }
    #ifdef DEB_DEBUG
        vty_out (vty, "DEB DEBUG: -- frp_retry_cmd - frp->retry = %d %s", frp->retry, VTY_NEWLINE);
    #endif //DEB_DEBUG
    return CMD_SUCCESS;
}




// ------------------------------------------------------------
// INIT FUNCTION
// ------------------------------------------------------------
// initialise frp structure and set commands
void
frp_init (void)
{
    #ifdef DEB_DEBUG
        fprintf (stderr, "DEB DEBUG: entering frpd.c - frp_init\n");
    #endif //DEB_DEBUG
    /* Install FRP_NODE */
    // DEB COMMENT: seems to be the initial install point, used in all other daemons (although not neccessarily in the
daemon.c)
    install_node (&frp_node, frp_config_write);                    // DEB COMMENT: install_node command.h [33

    // DEB COMMENT: seems to be the place to install frp specific commands)
    install_default (FRP_NODE);                                    // DEB COMMENT: install_default command.h [3
                                                                   // DEB COMMENT: FRP_NODE needs to be define

    // create the frp router config command - frpd(config)#
    install_element (CONFIG_NODE, &router_frp_cmd);
    install_element (CONFIG_NODE, &no_router_frp_cmd);

    // install the frp router commands
    install_element (FRP_NODE, &frp_secret_cmd);
    install_element (FRP_NODE, &frp_cost_cmd);
    install_element (FRP_NODE, &frp_poll_cmd);
    install_element (FRP_NODE, &frp_retry_cmd);
    install_element (FRP_NODE, &frp_gateway_cmd);

    /* initialise frp debugging commands and functions */
    frp_debug_init ();

    /* initialise frp interface related commands and functions */
    frp_interface_init ();

    /* initialise frp neighbour related commands and functions */
    frp_peer_init ();

    #ifdef DEB_DEBUG
        fprintf (stderr, "DEB DEBUG: -- leaving frp_init\n");
    #endif //DEB_DEBUG

    return;
}

```

```c
// ------------------------------------------------------------
// INCLUDES
// ------------------------------------------------------------
#include "frpd.h"
#include "frp_debug.h"
#include "linklist.h"

// to support code lifted from Don's implementation
#include <openssl/sha.h>

#define IPADDR u_int32_t




// ------------------------------------------------------------
// GLOBAL VARIABLES
// ------------------------------------------------------------

/* linked list of frp peers */
struct list* frp_peers = NULL;

/* static prototypes */
static int frp_neighbor_delete (struct prefix_ipv4 *p);

static void frp_peer_free (struct frp_peer *peer);
static int frp_frp_peers_cmp (struct frp_peer *p1, struct frp_peer *p2);
static struct frp_peer* frp_peer_add (struct in_addr* addr, const char* secret);
static struct frp_peer* frp_peer_new (void);


/* prototypes */
struct frp_peer* frp_peer_lookup (struct in_addr *addr);
struct frp_peer* frp_peer_lookup_next (struct in_addr *addr);



// ------------------------------------------------------------
// FUNCTIONS
// ------------------------------------------------------------


// -----SUPPORT------------------------------------------------------

struct frp_peer*
frp_peer_new (void)
{ return XCALLOC (MTYPE_FRP_PEER, sizeof (struct frp_peer));
}

void
frp_peer_free (struct frp_peer *peer)
{ XFREE (MTYPE_FRP_PEER, peer);
}

struct frp_peer*
frp_peer_lookup (struct in_addr *addr)
{ struct frp_peer *peer;
  struct listnode *node, *nnode;
  for (ALL_LIST_ELEMENTS (frp_peers, node, nnode, peer))
  { if (IPV4_ADDR_SAME (&peer->address, addr))
      return peer;
```



```c
  }
  return NULL;
}

struct frp_peer*
frp_peer_lookup_next (struct in_addr *addr)
{ struct frp_peer *peer;
  struct listnode *node, *nnode;
  for (ALL_LIST_ELEMENTS (frp_peers, node, nnode, peer))
  { if (htonl (peer->address.s_addr) > htonl (addr->s_addr))
      return peer;
  }
  return NULL;
}

static struct frp_peer*
frp_peer_add (struct in_addr* addr, const char* secret)
{ struct frp_peer *peer;
  peer = frp_peer_lookup (addr);
  if (peer)
  { peer->secret = secret;
  } else
  { peer = frp_peer_new ();
    peer->address = *addr;
    peer->secret = (char*) XCALLOC (MTYPE_FRP_PEER, strlen(secret) + 1);
    memcpy (peer->secret, secret, strlen(secret));
    peer->packet_latest = (u_char*) XCALLOC (MTYPE_FRP_PEER, FRP_PKT_MAXSIZE);
    listnode_add_sort (frp_peers, peer);
    peer->rib = list_new ();
    peer->temp_rib = NULL;
  }
  return peer;
}

int
frp_frp_peers_cmp (struct frp_peer *p1, struct frp_peer *p2)
{ return htonl (p1->address.s_addr) > htonl (p2->address.s_addr);
}

struct prefix*
find_local_address_for_peer(struct in_addr dest)
{ struct interface* ifp;
  struct listnode* node;
  if (iflist == NULL)
  {
    #ifdef DEB_DEBUG
      zlog_debug ("DEB DEBUG: -- find_local_address_for_peer - iflist=null");
    #endif //DEB_DEBUG
  }
  /* Check each interface. */
  for (ALL_LIST_ELEMENTS_RO (iflist, node, ifp))
  { struct connected* conn = connected_lookup_address (ifp, dest);
    if (conn != NULL)
    { return conn->address;
    } else
    {
      #ifdef DEB_DEBUG
        zlog_debug ("DEB DEBUG: -- find_local_address_for_peer - conn=null");
      #endif //DEB_DEBUG
    }
  }
```

```c
    return NULL;
}

// -----SECURITY--------------------------------------------------------
// LIFTED STRAIGHT FROM DON'S CODE

/*
 Security stuff
 Hash calculation: SHA1 over packet, IP addresses & secret
 Note that only the first 64 bits (FRP_HASHZIE) of the hash are used.
 */
//#include <sha.h>
u_int8_t*
dohash(u_char* buf, int len, const char* secret, IPADDR sa, IPADDR da)
{  static u_int8_t hash[SHA_DIGEST_LENGTH];
   SHA_CTX shctx;
   SHA_Init(&shctx);
   SHA_Update(&shctx, buf, len);
   SHA_Update(&shctx, (u_char*)&sa, sizeof(IPADDR));
   SHA_Update(&shctx, (u_char*)&da, sizeof(IPADDR));
   SHA_Update(&shctx, secret, strlen(secret));
   SHA_Final(hash, &shctx);
   #ifdef DEB_DEBUG_PEER
      zlog_debug ("DEB DEBUG:  -- dohash - hash: %d%d%d%d%d%d%d%d%d%d%d%d%d%d%d%d%d%d%d%d",
         hash[0],hash[1],hash[2],hash[3],hash[4],hash[5],hash[6],hash[7],hash[8],hash[9],
         hash[10],hash[11],hash[12],hash[13],hash[14],hash[15],hash[16],hash[17],hash[18],hash[19]);
   #endif //DEB_DEBUG
   return hash;
}

/*
 Check that hash matches packet hash.
 Do not call if peer->secret is null.
 */
int
checksecure(u_char* pkt, int len, struct frp_peer* peer, struct in_addr local)
{  u_int8_t *hash;
   hash = dohash(pkt + FRP_HASHSIZE, len - FRP_HASHSIZE, peer->secret, peer->address.s_addr, local.s_addr);
   #ifdef DEB_DEBUG_PEER
      zlog_debug ("DEB DEBUG:  -- checksecure - packet len: %d", len);
      zlog_debug ("DEB DEBUG:  -- checksecure - checking from: %s secret: %s", inet_ntoa (peer->address),
peer->secret);
      zlog_debug ("DEB DEBUG:  -- checksecure - checking to: %s secret: %s", inet_ntoa (local), peer->secret);
   #endif //DEB_DEBUG
   if(!memcmp(hash, pkt, FRP_HASHSIZE))
   {  return 1;
   }
   zlog_debug("packet security failure");
   return 0;
}

/*
 Compute packet hash
 Or zero hash field if secret is null
 */
 // NOT ACTUALLY USED
void
secure(struct frp_peer* peer, struct in_addr local, struct frp_pkt_hdr* pkt, int len)
{  u_int8_t *hash;
   if(!peer->secret)
   {  memset(pkt->hash, 0, FRP_HASHSIZE);
```



```c
      return;
   }
   hash = dohash(&pkt->hash[FRP_HASHSIZE], len-FRP_HASHSIZE, peer->secret, local.s_addr, peer->address.s_addr);
   memcpy(pkt->hash, hash, FRP_HASHSIZE);
}

// -----NEIGHBOURS-------------------------------------------------------
/* add new frp neighbor to struct route_table *neighbors */
int
frp_neighbor_add (struct vty *vty, const char* ip_str, const char* secret)
{
   #ifdef DEB_DEBUG
      vty_out (vty, "DEB DEBUG: entering frp_peer.c - frp_neighbor_add %s %s %s", ip_str, secret, VTY_NEWLINE);
      zlog_debug ("DEB DEBUG: entering frp_peer.c - frp_neighbor_add %s secret %s", ip_str, secret);
   #endif //DEB_DEBUG

   /* convert incoming peer address to a struct prefix */
   struct prefix_ipv4    p;
   int ret = str2prefix_ipv4  (ip_str, &p);                    // DEB COMMENT: prefix.c [217] (also v6 & combinec
   // check have a valid address
   if (ret <= 0)
   {
   #ifdef DEB_DEBUG
      zlog_debug ("DEB DEBUG:  -- frp_neighbor_add - invalid address");
   #endif //DEB_DEBUG
      vty_out (vty, "Please specify address as A.B.C.D%s", VTY_NEWLINE);
      return CMD_WARNING;
   }
   // check have a secret
   if (!strcmp(secret, ""))
   {
      #ifdef DEB_DEBUG
         zlog_debug ("DEB DEBUG:  -- frp_neighbor_add - invalid secret");
      #endif //DEB_DEBUG
      vty_out (vty, "Please specify address and secret as A.B.C.D secret WORD%s", VTY_NEWLINE);
      return CMD_WARNING;
   }

   // store peer in struct frp.neighbors
   struct route_node* node;
   node = route_node_get (frp->neighbors, (struct prefix *) &p);              // DEB COMMENT: table.c [272]
   if (node->info)     // if peer already exists
   {
      #ifdef DEB_DEBUG
         zlog_debug ("DEB DEBUG:  -- frp_neighbor_add - peer already exists");
      #endif //DEB_DEBUG
      route_unlock_node (node);
      return -1;
   }
   node->info = frp->neighbors;

   // convert the struct prefix to an in_addr
   struct prefix*  node_p;
   struct in_addr  address;
   node_p = (struct prefix*)&node->p;
   #ifdef DEB_DEBUG
      zlog_debug ("DEB DEBUG:  -- frp_neighbor_add - node_p=%s", inet_ntoa (node_p->u.prefix4));
   #endif //DEB_DEBUG
   int addr_test = inet_pton (AF_INET, inet_ntoa (node_p->u.prefix4), &address);

   // store address and secret of peer in list frp_peers
```

```c
  struct frp_peer*  peer;
  if (addr_test)
  {  peer = frp_peer_add (&address, secret);
     #ifdef DEB_DEBUG
        zlog_debug ("DEB DEBUG:  -- frp_neighbor_add, %s, %s added to frp_peers", inet_ntoa (peer->address),
peer->secret);
     #endif //DEB_DEBUG
  } else
  {
     #ifdef DEB_DEBUG
        zlog_debug ("DEB DEBUG:  -- frp_neighbor_add, add to frp_peers failed");
     #endif //DEB_DEBUG
     route_unlock_node (node);
     return -1;
  }

  // set peer defaults
  //peer->lseq  = NEW_SEQ_NO;
  peer->lseq  = RANDOM_INT(0,INT_MAX);
  peer->rseq  = 0;
  peer->gateway_path  = NULL;
  peer->gateway_cost  = FRP_INFINITY;
  if ((frp->gateway_cost  == FRP_INFINITY)  || (frp->gateway_cost  == 0))
  {  peer->flag_send_gateway  = OFF;
     peer->flag_send_update = OFF;
  } else
  {  peer->flag_send_gateway  = ON;
     peer->flag_send_update = ON;
  }
  peer->flag_alive  = ON;
  peer->flag_send_syn  = ON;
  peer->flag_send_config  = ON;
//    route_unlock_node (node);
  return 0;
}

/*  delete a frp neighbor from struct route_table *neighbors */
int
frp_neighbor_delete (struct prefix_ipv4  *p)
{  struct route_node *node;
   /*  lock for look up */
   node = route_node_lookup (frp->neighbors,  (struct prefix  *) p);
   if (! node)
   {  return -1;
   }
   node->info = NULL;
   /*  unlock lookup lock */
   route_unlock_node (node);
   /*  unlock real neighbor information lockup */
   route_unlock_node (node);
   return 0;
}







// ------------------------------------------------------------
// ROUTER COMMAND DEFUNs
// ------------------------------------------------------------
```



```c

DEFUN (frp_neighbor,
       frp_neighbor_cmd,
       "neighbor A.B.C.D secret WORD",
       "Specify  a neighbor router\n"
       "Neighbor address\n"
       "Neighbor secret\n"
       "the secret\n")
{
   int ret = frp_neighbor_add (vty, argv[0], argv[1]);
   if (ret == 0)
   {
      #ifdef DEB_DEBUG
         vty_out (vty, "DEB DEBUG:  -- successfully called frp_neighbor_add%s", VTY_NEWLINE);
      #endif //DEB_DEBUG
      return CMD_SUCCESS;
   } else if (ret == -1)
   {
      #ifdef DEB_DEBUG
         vty_out (vty, "DEB DEBUG:  -- frp_neighbor_add - address already exists%s",  VTY_NEWLINE);
      #endif //DEB_DEBUG
      return ret;
   }
   return ret;
}

DEFUN (no_frp_neighbor,
       no_frp_neighbor_cmd,
       "no neighbor A.B.C.D",
       NO_STR
       "Specify  a neighbor router\n"
       "Neighbor address\n")
{  int ret;
   struct prefix_ipv4  p;
   ret = str2prefix_ipv4  (argv[0], &p);
   if (ret <= 0)
   {  vty_out (vty, "Please specify address by A.B.C.D%s", VTY_NEWLINE);
      return CMD_WARNING;
   }
   frp_neighbor_delete (&p);
   return CMD_SUCCESS;
}




// ------------------------------------------------------------
// INIT FUNCTION
// ------------------------------------------------------------
void
frp_peer_init (void)
{
   #ifdef DEB_DEBUG
      fprintf (stderr, "DEB DEBUG: entering frp_peer.c - frp_peer_init\n");
   #endif //DEB_DEBUG

   //initialise  the random number generation
   RANDOM_SEED();

   // initialise  peer secret structure
   frp_peers  = list_new  ();
   frp_peers->cmp  = (int (*)(void  *,  void *))  frp_frp_peers_cmp;
```

```c

  // create the FRP neighbor command
  install_element (FRP_NODE, &frp_neighbor_cmd);
  install_element (FRP_NODE, &no_frp_neighbor_cmd);

  #ifdef DEB_DEBUG
     fprintf (stderr, "DEB DEBUG:  -- leaving frp_peer_init\n");
  #endif //DEB_DEBUG

  return;
}

```

```c
#define FRP_PKT_HDRSIZE                 16      // (128 bits)
#define FRP_PKT_MINSIZE                 16      // (128 bits)
#define FRP_PKT_MAXSIZE                 1400    // as specified by Don
#define FRP_MSG_CONTROL_SIZE            4       // (32 bits)
#define FRP_MSG_IPV4CONFIG_SIZE         12      // (96 bits)
#define FRP_MSG_IPV4GATEWAY_MINSIZE     8       // (64 bits)
#define FRP_MSG_IPV4UPDATE_SIZE         12      // (96 bits)
#define FRP_MSG_IPV6CONFIG_SIZE         24      // (192 bits)
#define FRP_MSG_IPV6GATEWAY_MINSIZE     20      // (160 bits)
#define FRP_MSG_IPV6UPDATE_SIZE         24      // (192 bits)

// frp message types
#define FRP_MSG_CONTROL             0x01
#define FRP_MSG_IPV4CONFIG          0x41
#define FRP_MSG_IPV4GATEWAY         0x42
#define FRP_MSG_IPV4UPDATE          0x43
#define FRP_MSG_IPV6CONFIG          0x61
#define FRP_MSG_IPV6GATEWAY         0x62
#define FRP_MSG_IPV6UPDATE          0x63

// frp control types
#define FRP_CTRL_POLL               1
#define FRP_CTRL_ACK                2
#define FRP_CTRL_NAK                3

// frp update flags
#define FRP_FLAG_BEGIN              0x01
#define FRP_FLAG_COMMIT             0x02
#define FRP_FLAG_NULLRT             0x04
#define FRP_FLAG_UPDATE             0x08
#define FRP_FLAG_DELETE             0x10
#define FRP_FLAG_GATEWAY            0x80


// frp packet header (128 bits)
struct frp_pkt_hdr
{   u_int8_t    hash[8];                    // security hash (64 bits)
    u_int32_t   sendSeq;                    // sender's sequence number (32 bits)
    u_int32_t   recipAck;                   // recipient's acknowledgement number (32 bits)
};

// frp header (16 bits)
struct frp_msg_hdr
{   u_int8_t    length;                     // message length (8 bits)
    u_int8_t    type;                       // message type (8 bits)
};

// frp control (32 bits)
struct frp_msg_control
{   struct frp_msg_hdr  msg_hdr;            // message header (16 bits)
    u_int8_t            type;               // control type (8 bits)
    u_int8_t            param;              // control parameters (8 bits)
};

// frp IPv4 configuration (96 bits)
struct frp_msg_ipv4config
{   struct frp_msg_hdr  msg_hdr;            // message header (16 bits)
    u_short             cost;               // cost of the link (16 bits)
    u_short             poll;               // poll time (16 bits)
    u_short             retry;              // retry time (16 bits)
    struct in_addr      id;                 // router-ID of peer (32 bits)
};

// frp IPv4 path to gateway (64-2016 bits)
struct frp_msg_ipv4gateway
{   struct frp_msg_hdr  msg_hdr;            // message header (16 bits)
    u_short             cost;               // cost from peer to gateway (16 bits)
    struct in_addr      path[62];           // path from peer to gateway (32xn bits, 1 <= n <= 62)
};

// frp IPv4 route update (96 bits)
struct frp_msg_ipv4update
{   struct frp_msg_hdr  msg_hdr;            // message header (16 bits)
    u_int8_t            flags;              // update type flags (8 bits)
    u_int8_t            length;             // prefix length (8 bits)
    u_short             routecost;          // route cost (16 bits)
    u_short             gatecost;           // cost from originator to gateway (16 bits)
    struct in_addr      prefix;             // IP prefix (32 bits)
};

// frp IPv6 configuration (192 bits)
struct frp_msg_ipv6config
{   struct frp_msg_hdr  msg_hdr;            // message header (16 bits)
    u_short             cost;               // cost of the link (16 bits)
    u_short             poll;               // poll time (16 bits)
    u_short             retry;              // retry time (16 bits)
    struct in6_addr     id;                 // router-ID of peer (128 bits)
};

// frp IPv6 path to gateway (160-7968 bits)
struct frp_msg_ipv6gateway
{   struct frp_msg_hdr  msg_hdr;            // message header (16 bits)
    u_short             cost;               // cost from peer to gateway (16 bits)
    struct in6_addr     path[1];            // path from peer to gateway (32xn bits, 1 <= n <= 62)
};

// frp IPv6 route update (192 bits)
struct frp_msg_ipv6update
{   struct frp_msg_hdr  msg_hdr;            // message header (16 bits)
    u_int8_t            flags;              // update type flags (8 bits)
    u_int8_t            length;             // prefix length (8 bits)
    u_short             routecost;          // route cost (16 bits)
    u_short             gatecost;           // cost from originator to gateway (16 bits)
    struct in6_addr     prefix;             // IP prefix (128 bits)
};



// -------------------------------------------------------------------
// GLOBAL VARIABLES AND DEFINES
// -------------------------------------------------------------------

struct frp_pkt_hdr      pkt_hdr;
struct frp_pkt_hdr*     pkt_hdr_ptr;

/* frp event. */
enum frp_makepkthdr_flag
{   FRP_ACK,
    FRP_SYN,
};
```

```c
// ------------------------------------------------------------
// INCLUDES
// ------------------------------------------------------------
#include "frpd.h"
#include "frp_debug.h"



// ------------------------------------------------------------
// FUNCTIONS
// ------------------------------------------------------------

/* frp packet send to destination address, on interface denoted by
* by connected argument. NULL to argument denotes destination should be
* should be frp multicast group
*/
// DEB COMMENT: no multicast in frp
int
//frp_send_packet (u_char* buf, int size, struct sockaddr_in *to, struct connected *ifc)
frp_send_packet (u_char* buf, int size, struct sockaddr_in *to)
{ int ret, send_sock;
  struct sockaddr_in sin;
  send_sock = frp->sock;
  sin.sin_port = htons (FRP_PORT_DEFAULT);
  sin.sin_addr.s_addr = htonl (0);
//  assert (ifc != NULL);
  if (IS_FRP_DEBUG_PACKET)
  {
#define ADDRESS_SIZE 20
    char dst[ADDRESS_SIZE];
    dst[ADDRESS_SIZE - 1] = '\0';
    if (to)
    { strncpy (dst, inet_ntoa(to->sin_addr), ADDRESS_SIZE - 1);
    }
#undef ADDRESS_SIZE
//    zlog_debug("frp_send_packet %s > %s (%s)", inet_ntoa(ifc->address->u.prefix4), dst, ifc->ifp->name);
    zlog_debug("frp_send_packet %s", dst);
  }
  /* Make destination address. */
  memset (&sin, 0, sizeof (struct sockaddr_in));
  sin.sin_family = AF_INET;
#ifdef HAVE_STRUCT_SOCKADDR_IN_SIN_LEN
  sin.sin_len = sizeof (struct sockaddr_in);
#endif /* HAVE_STRUCT_SOCKADDR_IN_SIN_LEN */
  /* When destination is specified, use it's port and address. */
  if (to)
  { sin.sin_port = to->sin_port;
    #ifdef DEB_DEBUG
      zlog_debug ("DEB DEBUG: -- frp_send_packet, port = %d", ntohs (sin.sin_port));
    #endif //DEB_DEBUG
    sin.sin_addr = to->sin_addr;
    send_sock = frp->sock;
  }
  ret = sendto (send_sock, buf, size, 0, (struct sockaddr *)&sin,
  sizeof (struct sockaddr_in));
  if (IS_FRP_DEBUG_EVENT)
    zlog_debug ("SEND to %s.%d", inet_ntoa(sin.sin_addr), ntohs (sin.sin_port));
  if (ret < 0)
    zlog_warn ("can't send packet : %s", safe_strerror (errno));
  if (!to)
  close(send_sock);
```



```c
  return ret;
}

struct frp_pkt_hdr
make_frp_pkt_hdr (struct in_addr local, struct frp_peer* peer, int flag, int len, u_char* buf)
{ struct frp_pkt_hdr  pkt_hdr;
  u_int8_t*           hash;
  u_int32_t           seq_no;
  u_int32_t           ack_no;
  #ifdef DEB_DEBUG
    zlog_debug ("DEB DEBUG: entering frp_packet.c - make_frp_pkt_hdr - flag=%d", flag);
  #endif //DEB_DEBUG
  // create and fill SYN and ACK fields - store current lseq for this peer
  #ifdef DEB_DEBUG_PEER
    zlog_debug ("DEB DEBUG: -- make_frp_pkt_hdr - peer->lseq=%u, peer->rseq=%u,", peer->lseq, peer->rseq);
  #endif //DEB_DEBUG
  seq_no = peer->lseq + 1;
  pkt_hdr.sendSeq = htonl(seq_no);
  peer->lseq = seq_no;
  #ifdef DEB_DEBUG_PEER
    zlog_debug ("DEB DEBUG: -- C - peer->lseq=%d, peer->rseq=%d,", peer->lseq, peer->rseq);
    zlog_debug ("DEB DEBUG: -- C - sendSeq=%d", ntohl(pkt_hdr.sendSeq));
  #endif //DEB_DEBUG
  if (flag) // ie: starting a new conversation
  { // create packet header fields - security hash: secret = ours, sender = us, dest = peer
    ack_no = 0;
    pkt_hdr.recipAck = htonl(ack_no);
    hash = dohash((u_char*)&pkt_hdr + FRP_HASHSIZE, FRP_PKT_HDRSIZE - FRP_HASHSIZE, frp->secret, local.s_a
peer->address.s_addr);
//      hash = dohash((u_char*)&pkt_hdr + FRP_HASHSIZE, len, frp->secret, local.s_addr, peer->address.s_addr);
    memcpy(pkt_hdr.hash, hash, FRP_HASHSIZE);
    #ifdef DEB_DEBUG_PEER
      zlog_debug ("DEB DEBUG: -- make_frp_pkt_hdr - packet len: %d", len);
      zlog_debug ("DEB DEBUG: -- make_frp_pkt_hdr - packet length: %d", FRP_PKT_HDRSIZE - FRP_HASHSIZE);
      zlog_debug ("DEB DEBUG: -- B - peer->lseq=%d, peer->rseq=%d,", peer->lseq, peer->rseq);
      zlog_debug ("DEB DEBUG: -- B - sendSeq=%d, recipAck=%d", ntohl(pkt_hdr.sendSeq), ntohl(pkt_hdr.recipAcl
    #endif //DEB_DEBUG
    #ifdef DEB_DEBUG
      zlog_debug ("DEB DEBUG: -- make_frp_pkt_hdr - SYN from: %s secret: %s", inet_ntoa (local), frp->secret);
      zlog_debug ("DEB DEBUG: -- make_frp_pkt_hdr - SYN to: %s secret: %s", inet_ntoa (peer->address),
frp->secret);
      zlog_debug ("DEB DEBUG: -- make_frp_pkt_hdr - SYN sendSeq=%d, recipAck=%d", ntohl(pkt_hdr.sendSeq),
ntohl(pkt_hdr.recipAck));
    #endif //DEB_DEBUG
  } else        // ie: continuing a conversation
  { // create packet header fields - security hash: secret = ours, sender = us, dest = peer
    ack_no = peer->rseq;
    pkt_hdr.recipAck = htonl(ack_no);
    if (buf != NULL)
    { memcpy(buf, &pkt_hdr, FRP_PKT_HDRSIZE);
      hash = dohash(buf + FRP_HASHSIZE, len - FRP_HASHSIZE, frp->secret, local.s_addr, peer->address.s_addr);
    } else
    { hash = dohash((u_char*)&pkt_hdr + FRP_HASHSIZE, FRP_PKT_HDRSIZE - FRP_HASHSIZE, frp->secret,
local.s_addr, peer->address.s_addr);
    }
    memcpy(pkt_hdr.hash, hash, FRP_HASHSIZE);
    #ifdef DEB_DEBUG
      zlog_debug ("DEB DEBUG: -- make_frp_pkt_hdr - ACK from: %s secret: %s", inet_ntoa (local), frp->secret)
      zlog_debug ("DEB DEBUG: -- make_frp_pkt_hdr - ACK to: %s secret: %s", inet_ntoa (peer->address),
frp->secret);
      zlog_debug ("DEB DEBUG: -- make_frp_pkt_hdr - ACK sendSeq=%d, recipAck=%d", ntohl(pkt_hdr.sendSeq),
```

```
ntohl(pkt_hdr.recipAck));
      #endif //DEB_DEBUG
      #ifdef DEB_DEBUG_PEER
         zlog_debug ("DEB DEBUG:  -- D - peer->lseq=%d,  peer->rseq=%d,",  peer->lseq,  peer->rseq);
         zlog_debug ("DEB DEBUG:  -- D - sendSeq=%d, recipAck=%d",  ntohl(pkt_hdr.sendSeq), ntohl(pkt_hdr.recipAcl
      #endif //DEB_DEBUG
   }
   return pkt_hdr;
}

struct frp_msg_hdr
make_frp_msg_hdr (int type)
{  struct frp_msg_hdr  msg_hdr;
   memset (&msg_hdr,  0, sizeof (msg_hdr));
   // length field
   switch (type)
   {  case FRP_MSG_CONTROL:
         msg_hdr.length = FRP_MSG_CONTROL_SIZE;
         break;
      case FRP_MSG_IPV4CONFIG:
         msg_hdr.length = FRP_MSG_IPV4CONFIG_SIZE;
         break;
      case FRP_MSG_IPV4GATEWAY:
         msg_hdr.length = FRP_MSG_IPV4GATEWAY_MINSIZE;
         break;
      case FRP_MSG_IPV4UPDATE:
         msg_hdr.length = FRP_MSG_IPV4UPDATE_SIZE;
         break;
      default:
         break;
   }
   // type field
   msg_hdr.type = type;
   #ifdef DEB_DEBUG
      zlog_debug ("DEB DEBUG:  -- make_frp_msg_hdr - type=0x%x",  msg_hdr.type);
   #endif //DEB_DEBUG
   return msg_hdr;
}


struct frp_msg_control
make_frp_msg_control (int type)
{  struct frp_msg_control    msg;
   msg.msg_hdr = make_frp_msg_hdr (FRP_MSG_CONTROL);
   msg.type = type;
   msg.param = 0;
   return msg;
}

struct frp_msg_ipv4config
make_frp_msg_ipv4config (struct in_addr peer)
{  struct frp_msg_ipv4config   msg;
   msg.msg_hdr = make_frp_msg_hdr (FRP_MSG_IPV4CONFIG);
   msg.cost = htons(frp->cost);
   msg.poll = htons(frp->poll);
   msg.retry  = htons(frp->retry);
   msg.id = frp_get_interface_address  (peer);
   #ifdef DEB_DEBUG_PEER
      zlog_debug ("DEB DEBUG:  -- make_frp_msg_ipv4config - cost=%d, poll=%d, retry=%d,  id=%s", ntohs(msg.co
ntohs(msg.poll), ntohs(msg.retry),  inet_ntoa (peer));
   #endif //DEB_DEBUG
```



```
   return msg;
}

int
make_frp_msg_ipv4gateway  (struct in_addr peer, struct frp_msg_ipv4gateway*  msg)
{  msg->msg_hdr = make_frp_msg_hdr (FRP_MSG_IPV4GATEWAY);
   switch (frp->is_gateway_flag)
   {  // we are the gateway
      case FRP_GATEWAY_ALWAYS:
      case FRP_GATEWAY_YES:
         msg->cost = 0;
         msg->path[0] = frp_get_interface_address  (peer);
         return FRP_MSG_IPV4GATEWAY_MINSIZE;
         break;
      // we are not the gateway
      case FRP_GATEWAY_NO:
         if(frp->gateway_cost  == 0)     // have no gateway
         {  msg->cost = htons(FRP_INFINITY);
            msg->path[0].s_addr = 0;
            return FRP_MSG_IPV4GATEWAY_MINSIZE;
         } else   // someone else is the gateway
         {  msg->cost = htons(frp->gateway_cost);
            struct listnode *node;
            struct in_addr *path_step_addr;
            msg->path[0] = frp_get_interface_address  (peer);
            int i = 1;
            for (ALL_LIST_ELEMENTS_RO (frp->gateway_path,  node, path_step_addr))
            {  msg->path[i] = *path_step_addr;
               i++;
            }
            return FRP_MSG_IPV4GATEWAY_MINSIZE  + ((i - 1) * sizeof (struct in_addr));
         }
         break;
      default:
         break;
   }
   return FRP_MSG_IPV4GATEWAY_MINSIZE;
}

int
make_frp_msg_ipv4update (struct frp_peer*  peer, int available_length,  u_char*  buf)
{
   int counter = 0;
   int length = 0;
   struct route_node* rn;
   struct frp_info*  route_info;
   struct frp_peer*  nexthop;
   struct frp_msg_ipv4update  msg;
   msg.flags = 0;
   msg.length = 0;
   msg.routecost = 0;
   msg.gatecost = 0;
   struct in_addr nullPrefix;
   nullPrefix.s_addr   = 0;
   msg.prefix  = nullPrefix;

   // for each route in our RIB
   for (rn = route_top (frp->rib);  rn;  rn = route_next (rn))
   {
      if (rn->info)
      {
```

```
        route_info = (struct frp_info*)(rn->info);
        // if this peer is not the nexthop
        if (peer->address.s_addr != route_info->nexthop.s_addr)
        {
            // get nexthop peer
            nexthop = frp_peer_lookup (&route_info->nexthop);
            if (nexthop != NULL)
            { // FRP decision algorithm
                // if route cost + link cost < route gateway cost + target gateway cost + is gateway route
                if ((route_info->cost + nexthop->cost) < (nexthop->gateway_cost + peer->gateway_cost +
route_info->is_gateway_flag))
                {
                    if (counter == 0) // if counter = 0
                    {
                        // make new route message with BEGIN flag (& GATE flag if appropriate)
                        msg.msg_hdr = make_frp_msg_hdr (FRP_MSG_IPV4UPDATE);
                        if (route_info->is_gateway_flag)
                        { msg.flags = FRP_FLAG_BEGIN + FRP_FLAG_GATEWAY;
                        } else
                        { msg.flags = FRP_FLAG_BEGIN;
                        }
                        msg.length = htons(rn->p.prefixlen);
//                          msg.length = 24;
                        msg.routecost = htons(route_info->cost + nexthop->cost);
                        msg.gatecost = htons(nexthop->gateway_cost);
                        msg.prefix = rn->p.u.prefix4;
                        // increment counter
                        counter++;
                    } else // counter != 0
                    {
                        // memcopy message to buffer
                        if ((length + FRP_MSG_IPV4UPDATE_SIZE) <= available_length)
                        { memcpy(buf + length, &msg, FRP_MSG_IPV4UPDATE_SIZE);
                            length += FRP_MSG_IPV4UPDATE_SIZE;
                            // make new route message (with & GATE flag if appropriate)
                            msg.msg_hdr = make_frp_msg_hdr (FRP_MSG_IPV4UPDATE);
                            if (route_info->is_gateway_flag)
                            { msg.flags = FRP_FLAG_GATEWAY;
                            } else
                            { msg.flags = 0;
                            }
                            msg.length = htons(rn->p.prefixlen);
                            msg.routecost = htons(route_info->cost + nexthop->cost);
                            msg.gatecost = htons(nexthop->gateway_cost);
                            msg.prefix = rn->p.u.prefix4;
                            // increment counter
                            counter++;
                        }
                    }
                }
            } else // nexthop is null
            {
                // complain
            }
        }
    }
  }
  // add in static routes (enabled networks)
  for (rn = route_top (frp_enable_network); rn; rn = route_next (rn))
  { if (rn->info)
    {
```



```
        #ifdef DEB_DEBUG
          zlog_debug ("DEB DEBUG: -- make_frp_msg_ipv4update %s/%d", inet_ntoa (rn->p.u.prefix4), rn->p.prefi
        #endif //DEB_DEBUG
        if (counter == 0) // if counter = 0
        {
          // make new route message with BEGIN flag (& GATE flag if appropriate)
          msg.msg_hdr = make_frp_msg_hdr (FRP_MSG_IPV4UPDATE);
          msg.flags = FRP_FLAG_BEGIN;
          msg.length = htons(rn->p.prefixlen);
          msg.routecost = 0;
          msg.gatecost = htons(frp->gateway_cost);
          msg.prefix = rn->p.u.prefix4;
          // increment counter
          counter++;
        } else // counter != 0
        {
          // memcopy message to buffer
          if ((length + FRP_MSG_IPV4UPDATE_SIZE) <= available_length)
          { memcpy(buf + length, &msg, FRP_MSG_IPV4UPDATE_SIZE);
            length += FRP_MSG_IPV4UPDATE_SIZE;
            // make new route message (with & GATE flag if appropriate)
            msg.msg_hdr = make_frp_msg_hdr (FRP_MSG_IPV4UPDATE);
            msg.flags = 0;
            msg.length = htons(rn->p.prefixlen);
            msg.routecost = 0;
            msg.gatecost = htons(frp->gateway_cost);
            msg.prefix = rn->p.u.prefix4;
            // increment counter
            counter++;
          }
        }
      }
    }



  if (counter > 0) //sending routes to this peer
  {
    // set COMMIT flag, memcopy message to buffer
    msg.flags += FRP_FLAG_COMMIT;
  } else // sending nullrt to this peer
  {
    // make new route message with BEGIN & COMMIT & NULLRT flags
    #ifdef DEB_DEBUG
      zlog_debug ("DEB DEBUG: -- make_frp_msg_ipv4update, sending nullrt to this peer");
    #endif //DEB_DEBUG
    msg.msg_hdr = make_frp_msg_hdr (FRP_MSG_IPV4UPDATE);
    msg.flags = FRP_FLAG_BEGIN + FRP_FLAG_COMMIT + FRP_FLAG_NULLRT;
  }
  // memcopy final message to buffer
  if ((length + FRP_MSG_IPV4UPDATE_SIZE) <= available_length)
  { memcpy(buf + length, &msg, FRP_MSG_IPV4UPDATE_SIZE);
    length += FRP_MSG_IPV4UPDATE_SIZE;
  }
  return length;
}


// build and send a SYN packet to a peer
int
build_syn_pkt (struct frp_peer* peer)
```

```c
{ // find local address of interface peer is attached to
  struct prefix*  local_p;
  struct in_addr  local_a;
  int             sent;
  local_p = find_local_address_for_peer(peer->address);
  if (local_p != NULL)
  { int local_p_test = inet_pton (AF_INET, inet_ntoa (local_p->u.prefix4),  &local_a);
  } else
  {
    #ifdef DEB_DEBUG
      zlog_debug ("DEB DEBUG:  -- build_syn_pkt, failed to get local interface address");
    #endif //DEB_DEBUG
    return 0;
  }
  // create a SYN packet to send
  struct frp_pkt_hdr  syn_pkt;
  syn_pkt = make_frp_pkt_hdr (local_a, peer, FRP_SYN, FRP_PKT_HDRSIZE, NULL);
  // create socket
  struct sockaddr_in socket;
  socket.sin_family  = AF_INET;
  socket.sin_port = htons(FRP_PORT_DEFAULT);
  socket.sin_addr = peer-> address;
  #ifdef DEB_DEBUG
    zlog_debug ("DEB DEBUG:  -- build_syn_pkt - SYN=%u (0x%x), ACK=%u (0x%x), addr=%s, port=%u",
ntohl(syn_pkt.sendSeq), ntohl(syn_pkt.sendSeq), syn_pkt.recipAck, syn_pkt.recipAck, inet_ntoa(socket.sin_addr),
ntohs(socket.sin_port));
  #endif //DEB_DEBUG
  // send SYN packet to peer
  sent = frp_send_packet ((u_char*)&syn_pkt,  FRP_PKT_HDRSIZE,  &socket);
  if (sent)
  { memcpy (peer->packet_latest, (u_char*)&syn_pkt,  FRP_PKT_HDRSIZE);
    peer->packet_latest_length = FRP_PKT_HDRSIZE;
    peer->packet_latest_lseq  = ntohl(syn_pkt.sendSeq);
    peer->flag_awaiting_ack  = ON;
    #ifdef DEB_DEBUG_PKT
      zlog_debug ("DEB DEBUG:  -- build_syn_pkt - packet_latest_lseq=%d", peer->packet_latest_lseq);
    #endif //DEB_DEBUG
  } else
  { zlog_debug ("packet number %d not sent", syn_pkt.sendSeq);
  }
  return sent;
}

// build and send an ACK packet to a peer
int
build_ack_pkt (struct frp_peer*  peer, struct sockaddr_in*  from, struct in_addr local)
{ u_char*                 out_pkt_buf;
  int                     out_pkt_len;
  int                     sent;
  struct frp_pkt_hdr           out_pkt_hdr;
  struct frp_msg_control       out_pkt;
  // create the control message
  out_pkt_len = FRP_PKT_HDRSIZE  + FRP_MSG_CONTROL_SIZE;
  out_pkt_buf = XCALLOC (MTYPE_FRP,  out_pkt_len);
  out_pkt = make_frp_msg_control  (FRP_CTRL_ACK);
  memcpy(out_pkt_buf + FRP_PKT_HDRSIZE,  &out_pkt, FRP_MSG_CONTROL_SIZE);
  // create the packet header - has to be last because of the hash
  out_pkt_hdr = make_frp_pkt_hdr (local, peer, FRP_ACK, out_pkt_len, out_pkt_buf);
  memcpy(out_pkt_buf, &out_pkt_hdr, FRP_PKT_HDRSIZE);
  #ifdef DEB_DEBUG_PEER
    zlog_debug ("DEB DEBUG:  -- 7 - out_pkt_hdr size =%d", sizeof(out_pkt_hdr));
```



```c
    zlog_debug ("DEB DEBUG:  -- 7 - out_pkt_hdr sendSeq=%d", ntohl(out_pkt_hdr.sendSeq));
    zlog_debug ("DEB DEBUG:  -- 7 - out_pkt_hdr recipAck=%d", ntohl(out_pkt_hdr.recipAck));
    zlog_debug ("DEB DEBUG:  -- 7 - out_pkt_hdr length=%d, type=0x%x,  type=%d, param=%d",
out_pkt.msg_hdr.length, out_pkt.msg_hdr.type, out_pkt.type, out_pkt.param);
  #endif //DEB_DEBUG
  // send control ack packet
  sent = frp_send_packet (out_pkt_buf, out_pkt_len, from);
  if (sent)
  { memcpy (peer->packet_latest, (u_char*)out_pkt_buf, out_pkt_len);
    peer->packet_latest_length = out_pkt_len;
    peer->packet_latest_lseq  = ntohl(out_pkt_hdr.sendSeq);
    peer->flag_awaiting_ack  = ON;
    #ifdef DEB_DEBUG_PKT
      zlog_debug ("DEB DEBUG:  -- build_ack_pkt - packet_latest_lseq=%d", peer->packet_latest_lseq);
    #endif //DEB_DEBUG
  } else
  { zlog_debug ("packet number %d not sent", out_pkt_hdr.sendSeq);
  }
  return sent;
}

// build and send a NAK packet to a peer
int
build_nak_pkt (struct frp_peer*  peer, struct sockaddr_in*  from, struct in_addr local)
{ u_char*                 out_pkt_buf;
  int                     out_pkt_len;
  int                     sent;
  struct frp_pkt_hdr           out_pkt_hdr;
  struct frp_msg_control       out_pkt;
  // create the control message
  out_pkt_len = FRP_PKT_HDRSIZE  + FRP_MSG_CONTROL_SIZE;
  out_pkt_buf = XCALLOC (MTYPE_FRP,  out_pkt_len);
  out_pkt = make_frp_msg_control  (FRP_CTRL_NAK);
  memcpy(out_pkt_buf + FRP_PKT_HDRSIZE,  &out_pkt, FRP_MSG_CONTROL_SIZE);
  // create the packet header - has to be last because of the hash
  out_pkt_hdr = make_frp_pkt_hdr (local, peer, FRP_ACK, out_pkt_len, out_pkt_buf);
  memcpy(out_pkt_buf, &out_pkt_hdr, FRP_PKT_HDRSIZE);
  // send control nak packet
  sent = frp_send_packet (out_pkt_buf, out_pkt_len, from);
  if (sent)
  { memcpy (peer->packet_latest, (u_char*)out_pkt_buf, out_pkt_len);
    peer->packet_latest_length = out_pkt_len;
    peer->packet_latest_lseq  = ntohl(out_pkt_hdr.sendSeq);
    peer->flag_awaiting_ack  = ON;
    #ifdef DEB_DEBUG_PKT
      zlog_debug ("DEB DEBUG:  -- build_nak_pkt - packet_latest_lseq=%d", peer->packet_latest_lseq);
    #endif //DEB_DEBUG
  } else
  { zlog_debug ("packet number %d not sent", out_pkt_hdr.sendSeq);
  }
  return sent;
}

// build and send a batch packet to a peer
int
build_batch_pkt (struct frp_peer*  peer, struct sockaddr_in*  from, struct in_addr local)
{
  #ifdef DEB_DEBUG
    zlog_debug ("DEB DEBUG: entering frp_packet.c - build_batch_pkt");
  #endif //DEB_DEBUG
  // create an ACK packet to send to peer
```

```
    u_char*                 out_pkt_buf;
    int                     out_pkt_len;
    enum flag               localflag;
    int                     sent;
    struct frp_pkt_hdr              out_pkt_hdr;
    struct frp_msg_control          out_pkt_ctrl;
    struct frp_msg_ipv4config       out_pkt_conf;
    struct frp_msg_ipv4gateway      out_pkt_gate;
    // create buffer for outgoing packet
    localflag = OFF;
    out_pkt_buf = XCALLOC (MTYPE_FRP, FRP_PKT_MAXSIZE);
    out_pkt_len = FRP_PKT_HDRSIZE;
    // if required, create a control packet, send a poll and add to outgoing packet
    if ((peer->flag_send_poll) && ((out_pkt_len + FRP_MSG_CONTROL_SIZE) < FRP_PKT_MAXSIZE))
    {  out_pkt_ctrl = make_frp_msg_control (FRP_CTRL_POLL);
       memcpy(out_pkt_buf + out_pkt_len, &out_pkt_ctrl, FRP_MSG_CONTROL_SIZE);
       out_pkt_len += FRP_MSG_CONTROL_SIZE;
       localflag = ON;
    }
    // if required, create a config packet and add to outgoing packet
    if ((peer->flag_send_config) && ((out_pkt_len + FRP_MSG_IPV4CONFIG_SIZE) < FRP_PKT_MAXSIZE))
    {  out_pkt_conf = make_frp_msg_ipv4config (peer->address);
       memcpy(out_pkt_buf + out_pkt_len, &out_pkt_conf, FRP_MSG_IPV4CONFIG_SIZE);
       out_pkt_len += FRP_MSG_IPV4CONFIG_SIZE;
       localflag = ON;
    }
    // if required, create a path to gateway packet and add to outgoing packet
    if (peer->flag_send_gateway)
    {  struct frp_msg_ipv4gateway    out_pkt_gate;
       int   out_pkt_gate_length;
       out_pkt_gate_length = make_frp_msg_ipv4gateway (peer->address, &out_pkt_gate);
       if ((out_pkt_len + out_pkt_gate_length) < FRP_PKT_MAXSIZE)
       {  memcpy(out_pkt_buf + out_pkt_len, &out_pkt_gate, out_pkt_gate_length);
          out_pkt_len += out_pkt_gate_length;
          localflag = ON;
       }
    }
    // if not quiescent
    if (frp->is_gateway_flag == ON)
    {  // if required, create a route update packet and add to outgoing packet
       if ((peer->flag_send_update) && ((out_pkt_len + FRP_MSG_IPV4UPDATE_SIZE) < FRP_PKT_MAXSIZE))
       {  int       available_length = 0;
          int       used_length = 0;
          // calculate available space left
          available_length = FRP_PKT_MAXSIZE - out_pkt_len;
          // return how much was added
          used_length = make_frp_msg_ipv4update (peer, available_length, out_pkt_buf + out_pkt_len);
          out_pkt_len += used_length;
          localflag = ON;
       }
    }

    if (localflag == ON)
    {  // create the packet header - has to be last because of the hash
       out_pkt_hdr = make_frp_pkt_hdr (local, peer, FRP_ACK, out_pkt_len, out_pkt_buf);
       #ifdef DEB_DEBUG_PEER
          zlog_debug ("DEB DEBUG:  -- 7 - out_pkt_hdr.sendSeq=%d", ntohl(out_pkt_hdr.sendSeq));
          zlog_debug ("DEB DEBUG:  -- 7 - out_pkt_hdr.recipAck=%d", ntohl(out_pkt_hdr.recipAck));
          zlog_debug ("DEB DEBUG:  -- 7 - out_pkt_hdr size =%d", sizeof(out_pkt_hdr));
       #endif //DEB_DEBUG
       memcpy(out_pkt_buf, &out_pkt_hdr, FRP_PKT_HDRSIZE);
```



```
       // send packet
       #ifdef DEB_DEBUG_PEER
          zlog_debug ("DEB DEBUG:  -- 6 - out_pkt_len=%d", out_pkt_len);
          struct frp_pkt_hdr* test_hdr = (struct frp_pkt_hdr*)out_pkt_buf;
          zlog_debug ("DEB DEBUG:  -- 6 - out_pkt_buf.sendSeq=%d", ntohl(test_hdr->sendSeq));
          zlog_debug ("DEB DEBUG:  -- 6 - out_pkt_buf.recipAck=%d", ntohl(test_hdr->recipAck));
       #endif //DEB_DEBUG
       sent = frp_send_packet (out_pkt_buf, out_pkt_len, from);
       // reset peer flags
       if (sent)
       {  peer->flag_send_poll = OFF;
          peer->flag_send_config = OFF;
          peer->flag_send_gateway = OFF;
          peer->flag_send_update = OFF;
          memcpy (peer->packet_latest, (u_char*)out_pkt_buf, out_pkt_len);
          peer->packet_latest_length = out_pkt_len;
          peer->packet_latest_lseq = ntohl(out_pkt_hdr.sendSeq);
          peer->flag_awaiting_ack = ON;
          #ifdef DEB_DEBUG_PKT
             zlog_debug ("DEB DEBUG:  -- build_batch_pkt - packet_latest_lseq=%d", peer->packet_latest_lseq);
          #endif //DEB_DEBUG
       } else
       {  zlog_debug ("packet number %d not sent", out_pkt_hdr.sendSeq);
       }
    } else   //if no messages created because localflag == OFF
    {  // if peer awaiting ACK, send ACK packet to finish conversation
       if (peer->flag_awaiting_ack == ON)
       {
          #ifdef DEB_DEBUG_PKT
             zlog_debug ("DEB DEBUG:  -- build_batch_pkt - IF peer->flag_awaiting_ack == ON");
          #endif //DEB_DEBUG
          sent = build_ack_pkt (peer, from, local);
          peer->flag_awaiting_ack = OFF;
       }
    }
    return sent;
}
```

```
// ---------------------------------------------------------------
// INCLUDES
// ---------------------------------------------------------------
#include "frpd.h"



// ---------------------------------------------------------------
// FUNCTIONS
// ---------------------------------------------------------------

// works thru a list of routes given to us by a peer
// checks to see if each route is in our RIB
// where appropriate, replaces old routes with 'better' new routes
// deletes old routes from and passes new routes to Zebra
void
frp_recompute_rib (struct frp_peer* peer)
{
  #ifdef DEB_DEBUG
    zlog_debug ("DEB DEBUG: entering route.c - frp_recompute_rib");
  #endif //DEB_DEBUG
  int flag = 0;
  // step throu each new route in peer rib
  struct listnode* node;
  struct frp_rte* rte;
  struct interface* peer_if = if_lookup_address (peer->address);
  for (ALL_LIST_ELEMENTS_RO (peer->rib, node, rte))
  {
    struct route_node* rib_rte_ptr;
    rib_rte_ptr = route_node_get(frp->rib, (struct prefix*)&rte->prefix);
    // is the route in our rib?
    if (rib_rte_ptr->info != NULL)
    {
      // calculate existing route cost
      struct frp_info* rte_info = (struct frp_info*)rib_rte_ptr->info;
      u_int32_t old_cost = rte_info->cost;
      // calculate new route cost
      u_int32_t new_cost = rte->routecost + peer->cost;
      // is it a 'better' route using the frp algorithm?
      if (new_cost < old_cost)
      {
        // delete old from zebra
        frp_zebra_ipv4_delete ((struct prefix_ipv4*)&rte->prefix, &rte_info->nexthop, old_cost);
        // new route info for rib
        struct frp_info new_route_info;
        new_route_info.nexthop = peer->address;
        new_route_info.cost = new_cost;
        new_route_info.rte_node = rib_rte_ptr;
        new_route_info.type = ZEBRA_ROUTE_FRP;
        new_route_info.sub_type = FRP_ROUTE_RTE;
        new_route_info.ifindex = peer_if->ifindex;
        // replace old route with new in our rib
        memcpy (rte_info, &new_route_info, sizeof(struct frp_info));
        // add new route to zebra
        frp_zebra_ipv4_add ((struct prefix_ipv4*)&rte->prefix, &peer->address, new_cost, ZEBRA_FRP_DISTANCE_DEFAULT);
        // set update flag
        flag = 1;
      }
      // no else because ignore route
    } else   // not in our rib
```



```
    {
      // calculate new route cost
      u_int32_t new_cost = rte->routecost + peer->cost;
      // new route info for rib
      struct frp_info new_route_info;
      new_route_info.nexthop = peer->address;
      new_route_info.cost = new_cost;
      new_route_info.rte_node = rib_rte_ptr;
      new_route_info.type = ZEBRA_ROUTE_FRP;
      new_route_info.sub_type = FRP_ROUTE_RTE;
      new_route_info.ifindex = peer_if->ifindex;
      // add to our rib
      struct frp_info* rte_info = XCALLOC (MTYPE_FRP, sizeof (struct frp_info));
      memcpy (rte_info, &new_route_info, sizeof(struct frp_info));
      rib_rte_ptr->info = rte_info;
      // add to zebra rib
      frp_zebra_ipv4_add ((struct prefix_ipv4*)&rte->prefix, &peer->address, new_cost, ZEBRA_FRP_DISTANCE_DEFAULT);
      // set update flag
      flag = 1;
    }
    //route_unlock_node (rib_rte_ptr);
  }
  if (flag)
  {
//      frp_event (FRP_EVENT_UPDATE, 1);
  }
}


// deletes all routes from a particular peer - in our RIB and in Zebra
// checks each other peer for the best route to replace a deleted one
// passes new route to Zebra
void
frp_delete_peer_from_rib (struct frp_peer* deleted_peer)
{
  #ifdef DEB_DEBUG
    zlog_debug ("DEB DEBUG: entering route.c - frp_delete_peer_from_rib");
  #endif //DEB_DEBUG
   struct route_node* rib_rte_ptr;
   // for each entry in our rib
  for (rib_rte_ptr = route_top (frp->rib); rib_rte_ptr; rib_rte_ptr = route_next (rib_rte_ptr))
  {
    struct frp_info* rte_info = (struct frp_info*)rib_rte_ptr->info;
    // nexthop matches the deleted peer address
    if ((rte_info != NULL) && (rte_info->nexthop.s_addr == deleted_peer->address.s_addr))
    {
      int flag = 0;
      struct prefix_ipv4 deleted_dest;
      memset (&deleted_dest, 0, sizeof (struct prefix_ipv4));
      deleted_dest.family = AF_INET;
      deleted_dest.prefix = rib_rte_ptr->p.u.prefix4;
      deleted_dest.prefixlen = IPV4_MAX_BITLEN;
      // delete from zebra
      frp_zebra_ipv4_delete (&deleted_dest, &rte_info->nexthop, rte_info->cost);
      // delete from rib (actually just sets the node info in the list to null)
      XFREE (MTYPE_FRP, rte_info);
      rib_rte_ptr->info = NULL;

      struct listnode* node1;
      struct frp_peer* list_peer;
```

```c
    struct route_node*      new_route_node = NULL;
    // for each peer
    for (ALL_LIST_ELEMENTS_RO (frp_peers, node1, list_peer))
    {
      // this is not the deleted peer
      if (list_peer->address.s_addr  != deleted_peer->address.s_addr)
      {
        struct listnode*  node2;
        struct frp_rte*  rte;
        //for  each route in the peer rib
        for (ALL_LIST_ELEMENTS_RO (list_peer->rib,  node2, rte))
        {
        // this route is to the deleted destination
        // compare the two prefixes  to see if they are the same host and the same network
          if (prefix_same  (&rte->prefix,  &deleted_dest))
          {
            struct route_node*      new_rte_ptr;
            new_rte_ptr  = route_node_get(frp->rib,  (struct prefix*)&rte->prefix);
            new_route_node = new_rte_ptr;
            // this route is not in the rib
            if (new_rte_ptr->info  == NULL)
            {
              // add route to rib
              struct frp_info  new_route_info;
              struct interface*  peer_if  = if_lookup_address  (list_peer->address);
              new_route_info.nexthop  = list_peer->address;
              new_route_info.cost  = rte->routecost;
              new_route_info.is_gateway_flag  = (rte->flags  & FRP_FLAG_GATEWAY);
              new_route_info.rte_node  = new_rte_ptr;
              new_route_info.type  = ZEBRA_ROUTE_FRP;
              new_route_info.sub_type  = FRP_ROUTE_RTE;
              new_route_info.ifindex  = peer_if->ifindex;
              struct frp_info*  rte_info  = XCALLOC (MTYPE_FRP,  sizeof (struct frp_info));
              memcpy (rte_info,  &new_route_info,  sizeof(struct  frp_info));
              new_rte_ptr->info  = rte_info;
              flag = 1;
            } else      // is already  in the rib
            {
              // calculate  existing  route cost
              struct frp_info*  rte_info  = (struct  frp_info*)new_rte_ptr->info;
              u_int32_t old_cost = rte_info->cost;
              // calculate  new route cost
              u_int32_t new_cost = rte->routecost  + list_peer->cost;
              // this route is better than the existing  route
              if (new_cost < old_cost)
              {
                // replace  existing  route with  this route
                struct frp_info  new_route_info;
                struct interface*  peer_if  = if_lookup_address  (list_peer->address);
                new_route_info.nexthop  = list_peer->address;
                new_route_info.cost  = rte->routecost;
                new_route_info.is_gateway_flag  = (rte->flags  & FRP_FLAG_GATEWAY);
                new_route_info.rte_node  = new_rte_ptr;
                new_route_info.type  = ZEBRA_ROUTE_FRP;
                new_route_info.sub_type  = FRP_ROUTE_RTE;
                new_route_info.ifindex  = peer_if->ifindex;
                memcpy (rte_info,  &new_route_info,  sizeof(struct  frp_info));
                flag = 1;
              }
            }
            route_unlock_node (new_rte_ptr);
```



```c
          }
        }
      }
    }
    if ((flag)&&(new_route_node  != NULL))
    {  struct frp_info*  rte_info  = (struct frp_info*)new_route_node->info;
      // add new route to zebra
      frp_zebra_ipv4_add  (&deleted_dest,  &rte_info->nexthop,  rte_info->cost,  ZEBRA_FRP_DISTANCE_DEFAULT
      flag = 0;
    }
  }
 }
}
```

```c
// ------------------------------------------------------------
// INCLUDES
// ------------------------------------------------------------
#include "frpd.h"



// ------------------------------------------------------------
// GLOBAL VARIABLES
// ------------------------------------------------------------
/* create an instance of the zebra client */
struct zclient *zclient = NULL;                    // DEB COMMENT: zclient.h [35]

/* static prototypes */
static int frp_zebra_read_ipv4 (int command, struct zclient *zclient, zebra_size_t length);



// ------------------------------------------------------------
// ZEBRA CALLBACK FUNCTIONS
// ------------------------------------------------------------
// DEB COMMENT: need to supply functions to implement each of the zebra client API callback functions
//  -- taken from zclient.h [70]
// DEB COMMENT: prototypes defined in frpd.h
//   int (*router_id_update) (int, struct zclient *, uint16_t);          // not needed
   int (*interface_add) (int, struct zclient *, uint16_t);             // defined in frp_interface.c [44]
   int (*interface_delete) (int, struct zclient *, uint16_t);          // defined in frp_interface.c [67]
   int (*interface_up) (int, struct zclient *, uint16_t);              // defined in frp_interface.c [91]
   int (*interface_down) (int, struct zclient *, uint16_t);            // defined in frp_interface.c [115]
   int (*interface_address_add) (int, struct zclient *, uint16_t);     // defined in frp_interface.c [136]
   int (*interface_address_delete) (int, struct zclient *, uint16_t);  // defined in frp_interface.c [163]
   int (*ipv4_route_add) (int, struct zclient *, uint16_t);            // defined in frp_zebra.c [39]
   int (*ipv4_route_delete) (int, struct zclient *, uint16_t);         // defined in frp_zebra.c [39]
//   int (*ipv6_route_add) (int, struct zclient *, uint16_t);            // defined in frp_zebra.c
//   int (*ipv6_route_delete) (int, struct zclient *, uint16_t);         // defined in frp_zebra.c


/* zebra route add and delete */
int
frp_zebra_read_ipv4 (int command, struct zclient *zclient, zebra_size_t length)
{  struct stream *s;
   struct zapi_ipv4 api;
   unsigned long ifindex;
   struct in_addr nexthop;
   struct prefix_ipv4 p;
   #ifdef DEB_DEBUG
      zlog_debug ("DEB DEBUG: entering frp_interface.c - frp_zebra_read_ipv4");
   #endif //DEB_DEBUG
   s = zclient->ibuf;
   ifindex = 0;
   nexthop.s_addr = 0;
   /* type, flags, message */
   api.type = stream_getc (s);
   api.flags = stream_getc (s);
   api.message = stream_getc (s);
   /* IPv4 prefix */
   memset (&p, 0, sizeof (struct prefix_ipv4));
   p.family = AF_INET;
   p.prefixlen = stream_getc (s);
   stream_get (&p.prefix, s, PSIZE (p.prefixlen));
   /* nexthop, ifindex, distance, metric */
```



```c
   if (CHECK_FLAG (api.message, ZAPI_MESSAGE_NEXTHOP))
   {  api.nexthop_num = stream_getc (s);
      nexthop.s_addr = stream_get_ipv4 (s);
   }
   if (CHECK_FLAG (api.message, ZAPI_MESSAGE_IFINDEX))
   {  api.ifindex_num = stream_getc (s);
      ifindex = stream_getl (s);
   }
   if (CHECK_FLAG (api.message, ZAPI_MESSAGE_DISTANCE))
   {  api.distance = stream_getc (s);
   } else
   {  api.distance = 255;
   }
   if (CHECK_FLAG (api.message, ZAPI_MESSAGE_METRIC))
   {  api.metric = stream_getl (s);
   } else
   {  api.metric = 0;
   }
   return 0;
}



// ------------------------------------------------------------
// FUNCTIONS
// ------------------------------------------------------------

/* frpd to zebra command interface. */
void
frp_zebra_ipv4_add (struct prefix_ipv4 *p, struct in_addr *nexthop, u_int32_t metric, u_char distance)
{  struct zapi_ipv4 api;
   if (zclient->redist[ZEBRA_ROUTE_FRP])
   {  api.type = ZEBRA_ROUTE_FRP;
      api.flags = 0;
      api.message = 0;
      SET_FLAG (api.message, ZAPI_MESSAGE_NEXTHOP);
      api.nexthop_num = 1;
      api.nexthop = &nexthop;
      api.ifindex_num = 0;
      SET_FLAG (api.message, ZAPI_MESSAGE_METRIC);
      api.metric = metric;
      if (distance && distance != ZEBRA_FRP_DISTANCE_DEFAULT)
      {  SET_FLAG (api.message, ZAPI_MESSAGE_DISTANCE);
         api.distance = distance;
      }
      zapi_ipv4_route (ZEBRA_IPV4_ROUTE_ADD, zclient, p, &api);
      // DEB COMMENT: frp is not doing SNMP
//       frp_global_route_changes++;
   }
}

void
frp_zebra_ipv4_delete (struct prefix_ipv4 *p, struct in_addr *nexthop, u_int32_t metric)
{  struct zapi_ipv4 api;
   if (zclient->redist[ZEBRA_ROUTE_FRP])
   {  api.type = ZEBRA_ROUTE_FRP;
      api.flags = 0;
      api.message = 0;
      SET_FLAG (api.message, ZAPI_MESSAGE_NEXTHOP);
      api.nexthop_num = 1;
      api.nexthop = &nexthop;
```

```
        api.ifindex_num = 0;
        SET_FLAG (api.message, ZAPI_MESSAGE_METRIC);
        api.metric = metric;
        zapi_ipv4_route (ZEBRA_IPV4_ROUTE_DELETE, zclient, p, &api);
    }
}


// ------------------------------------------------------------
// INIT FUNCTION
// ------------------------------------------------------------
// initialise the zebra client structure and it's commands
// DEB COMMENT: zclient etc zclient.h

void
frp_zclient_init (void)
{
    #ifdef DEB_DEBUG
        fprintf (stderr, "DEB DEBUG: entering frp_zebra.c - frp_zclient_init\n");
    #endif //DEB_DEBUG

    /* allocate and initialise zebra structure */
    zclient = zclient_new ();
    zclient_init (zclient, ZEBRA_ROUTE_FRP);                    // DEB COMMENT: ZEBRA_ROUTE_FRP zebra.h [4

    /* set up callback functions */
    zclient->interface_add = frp_interface_add;
    zclient->interface_delete = frp_interface_delete;
    zclient->interface_up = frp_interface_up;
    zclient->interface_down = frp_interface_down;
    zclient->interface_address_add = frp_interface_address_add;
    zclient->interface_address_delete = frp_interface_address_delete;
    zclient->ipv4_route_add = frp_zebra_read_ipv4;
    zclient->ipv4_route_delete = frp_zebra_read_ipv4;
//    zclient->ipv6_route_add = frp_ipv6_route_add;
//    zclient->ipv6_route_delete = frp_ipv6_route_delete;

    #ifdef DEB_DEBUG
        fprintf (stderr, "DEB DEBUG: -- leaving frp_zclient_init\n");
    #endif //DEB_DEBUG
}
```

```
// ------------------------------------------------------------
// INCLUDES
// ------------------------------------------------------------
#include "frpd.h"
#include "frp_debug.h"



// ------------------------------------------------------------
// GLOBAL VARIABLES
// ------------------------------------------------------------

/* frp enabled interface table */
struct route_table *frp_enable_network;

/* frp enabled network vector */
vector frp_enable_interface;

/* static prototypes */
static void frp_apply_address_del (struct connected *ifc);
static void frp_connect_set (struct interface *ifp, int set);
static void frp_enable_apply (struct interface *);
static void frp_enable_apply_all (void);
static int frp_enable_if_add (const char *ifname);
static int frp_enable_if_lookup (const char *ifname);
static int frp_enable_network_add (struct prefix *p);
static int frp_enable_network_lookup_if (struct interface *ifp);
static int frp_if_down(struct interface *ifp);
static int frp_if_ipv4_address_check (struct interface *ifp);
static struct frp_interface * frp_interface_new (void);
static int frp_interface_delete_hook (struct interface *ifp);
static int frp_interface_new_hook (struct interface *ifp);
static int frp_interface_wakeup (struct thread *t);




// ------------------------------------------------------------
// ZEBRA CALLBACK FUNCTIONS
// ------------------------------------------------------------
// DEB COMMENT: need to supply a function to implement each of the zebra client API callback functions
// -- taken from zclient.h [70]
// DEB COMMENT: prototypes defined in frpd.h

int
frp_interface_add (int command, struct zclient *zclient, zebra_size_t length)
{ struct interface *ifp;
  #ifdef DEB_DEBUG_IF
    zlog_debug ("DEB DEBUG: entering frp_interface.c - frp_interface_add");
  #endif //DEB_DEBUG
  ifp = zebra_interface_add_read (zclient->ibuf);
  if (IS_FRP_DEBUG_ZEBRA)
  { zlog_debug ("interface add %s index %d flags %#llx metric %d mtu %d", ifp->name, ifp->ifindex, (unsigned long
long) ifp->flags, ifp->metric, ifp->mtu);
  }
  /* check if this interface is frp enabled or not */
  frp_enable_apply (ifp);
  return 0;
}

int
```



```
frp_interface_delete (int command, struct zclient *zclient, zebra_size_t length)
{ struct interface *ifp;
  struct stream *s;
  #ifdef DEB_DEBUG_IF
    zlog_debug ("DEB DEBUG: entering frp_interface.c - frp_interface_delete");
  #endif //DEB_DEBUG
  s = zclient->ibuf;
  /* zebra_interface_state_read() updates interface structure in iflist */
  ifp = zebra_interface_state_read(s);
  if (ifp == NULL)
  { return 0;
  }
  if (if_is_up (ifp))
  { frp_if_down(ifp);
  }
  zlog_info("interface delete %s index %d flags %#llx metric %d mtu %d", ifp->name, ifp->ifindex, (unsigned long lc
ifp->flags, ifp->metric, ifp->mtu);
  /* To support pseudo interface do not free interface structure. */
  /* if_delete(ifp); */
  ifp->ifindex = IFINDEX_INTERNAL;
  return 0;
}

int
frp_interface_up (int command, struct zclient *zclient, zebra_size_t length)
{ struct interface *ifp;
  #ifdef DEB_DEBUG_IF
    zlog_debug ("DEB DEBUG: entering frp_interface.c - frp_interface_up");
  #endif //DEB_DEBUG
  /* zebra_interface_state_read () updates interface structure in iflist */
  ifp = zebra_interface_state_read (zclient->ibuf);
  if (ifp == NULL)
  { return 0;
  }
  if (IS_FRP_DEBUG_ZEBRA)
  { zlog_debug ("interface %s index %d flags %#llx metric %d mtu %d is up", ifp->name, ifp->ifindex, (unsigned lor
long) ifp->flags, ifp->metric, ifp->mtu);
  }
  /* check if this interface is frp enabled or not*/
  frp_enable_apply (ifp);
  return 0;
}

int
frp_interface_down (int command, struct zclient *zclient, zebra_size_t length)
{ struct interface *ifp;
  struct stream *s;
  #ifdef DEB_DEBUG_IF
    zlog_debug ("DEB DEBUG: entering frp_interface.c - frp_interface_up");
  #endif //DEB_DEBUG
  s = zclient->ibuf;
  /* zebra_interface_state_read() updates interface structure in iflist. */
  ifp = zebra_interface_state_read(s);
  if (ifp == NULL)
  { return 0;
  }
  frp_if_down(ifp);
  if (IS_FRP_DEBUG_ZEBRA)
  { zlog_debug ("interface %s index %d flags %llx metric %d mtu %d is down", ifp->name, ifp->ifindex, (unsigned l
long)ifp->flags, ifp->metric, ifp->mtu);
  }
```

```
  return 0;
}

int
frp_interface_address_add (int command, struct zclient *zclient, zebra_size_t length)
{ struct connected *ifc;
  struct prefix *p;
  #ifdef DEB_DEBUG_IF
    zlog_debug ("DEB DEBUG: entering frp_interface.c - frp_interface_address_add");
  #endif //DEB_DEBUG
  ifc = zebra_interface_address_read (ZEBRA_INTERFACE_ADDRESS_ADD, zclient->ibuf);
  if (ifc == NULL)
  { return 0;
  }
  p = ifc->address;
  if (p->family == AF_INET)
  { if (IS_FRP_DEBUG_ZEBRA)
    { zlog_debug ("connected address %s/%d is added", inet_ntoa (p->u.prefix4), p->prefixlen);
    }
    frp_enable_apply(ifc->ifp);
  }
  return 0;
}

int
frp_interface_address_delete (int command, struct zclient *zclient, zebra_size_t length)
{ struct connected *ifc;
  struct prefix *p;
  #ifdef DEB_DEBUG_IF
    zlog_debug ("DEB DEBUG: entering frp_interface.c - frp_interface_address_delete");
  #endif //DEB_DEBUG
  ifc = zebra_interface_address_read (ZEBRA_INTERFACE_ADDRESS_DELETE, zclient->ibuf);
  if (ifc)
  { p = ifc->address;
    if (p->family == AF_INET)
    { if (IS_FRP_DEBUG_ZEBRA)
      { zlog_debug ("connected address %s/%d is deleted", inet_ntoa (p->u.prefix4), p->prefixlen);
      }
      /* Chech wether this prefix needs to be removed */
      frp_apply_address_del(ifc);
    }
    connected_free (ifc);
  }
  return 0;
}



// ------------------------------------------------------------
// FUNCTIONS
// ------------------------------------------------------------

/* Allocate new frp's interface configuration. */
struct frp_interface*
frp_interface_new (void)
{ struct frp_interface *ri;
  ri = XCALLOC (MTYPE_FRP_INTERFACE, sizeof (struct frp_interface));
  return ri;
}

/* add frp enable network */
```



```
int
frp_enable_network_add (struct prefix *p)
{
  #ifdef DEB_DEBUG_IF
    zlog_debug ("DEB DEBUG: entering frp_interface.c - frp_enable_network_add");
  #endif //DEB_DEBUG
  struct route_node *node;
  node = route_node_get (frp_enable_network, p);
  #ifdef DEB_DEBUG_IF
    zlog_debug ("DEB DEBUG: -- frp_enable_network_add, after node =");
  #endif //DEB_DEBUG
  if (node->info)
  {
    #ifdef DEB_DEBUG_IF
      zlog_debug ("DEB DEBUG: -- frp_enable_network_add, IF");
    #endif //DEB_DEBUG
    route_unlock_node (node);
    return -1;
  } else
  {
    #ifdef DEB_DEBUG_IF
      zlog_debug ("DEB DEBUG: -- frp_enable_network_add, ELSE");
    #endif //DEB_DEBUG
    node->info = (char *) "enabled";

    struct frp_peer* peer;
    struct listnode *node, *nnode;
    for (ALL_LIST_ELEMENTS (frp_peers, node, nnode, peer))
    {
      frp_recompute_rib (peer);
    }
  }
  #ifdef DEB_DEBUG_IF
    zlog_debug ("DEB DEBUG: -- frp_enable_network_add, before frp_enable_apply_all");
  #endif //DEB_DEBUG
  frp_enable_apply_all();
  #ifdef DEB_DEBUG_IF
    zlog_debug ("DEB DEBUG: -- frp_enable_network_add, after frp_enable_apply_all");
  #endif //DEB_DEBUG
//  route_unlock_node (node);
  #ifdef DEB_DEBUG_IF
    zlog_debug ("DEB DEBUG: -- leaving frp_enable_network_add");
  #endif //DEB_DEBUG
  return 1;
}

/* add interface to frp_enable_if. */
int
frp_enable_if_add (const char *ifname)
{ int ret;
  ret = frp_enable_if_lookup (ifname);
  if (ret >= 0)
  { return -1;
  }
  vector_set (frp_enable_interface, strdup (ifname));
  frp_enable_apply_all();
  return 1;
}

/* apply network configuration to all interface */
void
```

```
frp_enable_apply_all ()
{  struct interface *ifp;
   struct listnode *node, *nnode;
   /* Check each interface. */
   for (ALL_LIST_ELEMENTS (iflist, node, nnode, ifp))
   {  frp_enable_apply (ifp);
   }
}

/* update interface status */
void
frp_enable_apply (struct interface *ifp)
{  int ret;
   struct frp_interface *ri = NULL;
   #ifdef DEB_DEBUG_IF
      zlog_debug ("DEB DEBUG: entering frp_interface.c - frp_enable_apply");
   #endif //DEB_DEBUG
   /* check interface */
   if (! if_is_operative (ifp))
   {  return;
   }
   ri = ifp->info;
   #ifdef DEB_DEBUG_IF
      if (ri == NULL)
         zlog_debug ("DEB DEBUG: -- frp_enable_apply - IF ifp->info = null");
      else
         zlog_debug ("DEB DEBUG: -- frp_enable_apply - ELSE ifp->info = %p", ifp->info);
   #endif //DEB_DEBUG
   /* check network configuration */
   ret = frp_enable_network_lookup_if (ifp);
   #ifdef DEB_DEBUG_IF
      zlog_debug ("DEB DEBUG: -- frp_enable_apply - ret = %d", ret);
   #endif //DEB_DEBUG
   /* if the interface is matched */
   if (ret > 0)
   {  ri->enable_network = 1;
   } else
   {  ri->enable_network = 0;
   }
   #ifdef DEB_DEBUG_IF
      zlog_debug ("DEB DEBUG: -- frp_enable_apply - ri->enable_network = %d", ri->enable_network);
   #endif //DEB_DEBUG
   /* check interface name configuration */
   ret = frp_enable_if_lookup (ifp->name);
   #ifdef DEB_DEBUG_IF
      zlog_debug ("DEB DEBUG: -- frp_enable_apply - ret = %d", ret);
   #endif //DEB_DEBUG
   if (ret >= 0)
   {  ri->enable_interface = 1;
   } else
   {  ri->enable_interface = 0;
   }
   #ifdef DEB_DEBUG_IF
      zlog_debug ("DEB DEBUG: -- frp_enable_apply - ri->enable_interface = %d", ri->enable_interface);
   #endif //DEB_DEBUG
   /* any interface MUST have an IPv4 address */
   if ( ! frp_if_ipv4_address_check (ifp) )
   {  ri->enable_network = 0;
      ri->enable_interface = 0;
   }
   /* update running status of the interface */
```



```
   if (ri->enable_network || ri->enable_interface)
   {  if (IS_FRP_DEBUG_EVENT)
      {  zlog_debug ("turn on %s", ifp->name);
      }
      /* add interface wake up thread */
      if (! ri->t_wakeup)
      {  ri->t_wakeup = thread_add_timer (master, frp_interface_wakeup, ifp, 1);
         frp_connect_set (ifp, 1);
      }
   } else
   {  if (ri->running)
      {  /* might as well clean up the route table as well frp_if_down sets to 0 ri->running, and displays "turn off %s
         frp_if_down(ifp);
         frp_connect_set (ifp, 0);
      }
   }
}

/* check interface is enabled by ifname statement */
int
frp_enable_if_lookup (const char *ifname)
{  unsigned int i;
   char *str;
   for (i = 0; i < vector_active (frp_enable_interface); i++)
   {  if ((str = vector_slot (frp_enable_interface, i)) != NULL)
      {  if (strcmp (str, ifname) == 0)
         {  return i;
         }
      }
   }
   return -1;
}

/* check interface is enabled by network statement */
/* check wether the interface has at least a connected prefix that is within the frp_enable_network table */
int
frp_enable_network_lookup_if (struct interface *ifp)
{  struct listnode *node, *nnode;
   struct connected *connected;
   struct prefix_ipv4 address;
   #ifdef DEB_DEBUG_IF
      zlog_debug ("DEB DEBUG: entering frp_interface.c - frp_enable_network_lookup_if");
      zlog_debug ("DEB DEBUG: -- frp_enable_network_lookup_if - ifp->connected = %p", ifp->connected);
   #endif //DEB_DEBUG
   if (ifp->connected == NULL)
      return -1;
   for (ALL_LIST_ELEMENTS (ifp->connected, node, nnode, connected))
   {
      struct prefix *p;
      struct route_node *node;
      #ifdef DEB_DEBUG_IF
         if (connected == NULL)
            zlog_debug ("DEB DEBUG: -- frp_enable_network_lookup_if - IF connected = null");
         else
            zlog_debug ("DEB DEBUG: -- frp_enable_network_lookup_if - ELSE connected->address = %s",
inet_ntoa(connected->address->u.prefix4));
      #endif //DEB_DEBUG
      p = connected->address;
      if (p->family == AF_INET)
      {  address.family = AF_INET;
         address.prefix = p->u.prefix4;
```

```c
        address.prefixlen  = IPV4_MAX_BITLEN;
        node = route_node_match (frp_enable_network,  (struct prefix  *)&address);
        if (node)
        {  route_unlock_node (node);
           return  1;
        }
     }
  }
  return -1;
}

/*  is there an address on interface  that can be used */
int
frp_if_ipv4_address_check  (struct interface *ifp)
{  struct listnode *nn;
   struct connected *connected;
   int count = 0;
   if (ifp->connected == NULL)
      return  0;
   for (ALL_LIST_ELEMENTS_RO (ifp->connected, nn, connected))
   {  struct prefix  *p;
      p = connected->address;
      if (p->family  == AF_INET)
      {  count++;
      }
   }
   return  count;
}

/*  join  to multicast  group and send request to the interface  */
int
frp_interface_wakeup  (struct thread *t)
{  struct interface *ifp;
   struct frp_interface  *ri;
   /*  get interface  */
   ifp = THREAD_ARG (t);
   ri  = ifp->info;
   ri->t_wakeup  = NULL;
   /*  set running flag */
   ri->running  = 1;
   return  0;
}

void
frp_connect_set (struct interface *ifp,  int set)
{  struct listnode *node,  *nnode;
   struct connected *connected;
   struct prefix_ipv4  address;
   for (ALL_LIST_ELEMENTS (ifp->connected, node, nnode, connected))
   {  struct prefix  *p;
      p = connected->address;
      if (p->family  != AF_INET)
      {  continue;
      }
      address.family  = AF_INET;
      address.prefix  = p->u.prefix4;
      address.prefixlen  = p->prefixlen;
      apply_mask_ipv4  (&address);
   }
}

```



```c
int
frp_if_down(struct  interface *ifp)
{  struct route_node *rp;
   struct frp_info  *rinfo;
   struct frp_interface  *ri  = NULL;
   if (frp)
   {  for (rp = route_top (frp->rib);   rp;  rp = route_next (rp))
      {  if ((rinfo  = rp->info)  != NULL)
         {  /*  routes got through this interface.  */
            if (rinfo->ifindex  == ifp->ifindex  &&  rinfo->type  == ZEBRA_ROUTE_FRP  &&  rinfo->sub_type  ==
FRP_ROUTE_RTE)
            {  frp_zebra_ipv4_delete  ((struct prefix_ipv4  *) &rp->p,  &rinfo->nexthop,  rinfo->cost);
            } //else
         }
      }
   }
   ri  = ifp->info;
   if (ri->running)
   {  if (IS_FRP_DEBUG_EVENT)
      zlog_debug ("turn  off %s",  ifp->name);
      ri->running  = 0;
    }
   return  0;
}

void
frp_apply_address_del  (struct connected *ifc)  {
  struct prefix_ipv4  address;
  struct prefix  *p;

  if (!frp)
    return;

  if (! if_is_up(ifc->ifp))
    return;

  p = ifc->address;

  memset (&address,  0, sizeof (address));
  address.family  = p->family;
  address.prefix  = p->u.prefix4;
  address.prefixlen  = p->prefixlen;
  apply_mask_ipv4(&address);
}

// find local address of interface  a peer is attached to
struct in_addr
frp_get_interface_address  (struct in_addr peer)
{  struct prefix*   local_p;
   struct in_addr  local_a;
   local_p = find_local_address_for_peer(peer);
   if (local_p != NULL)
   {
      #ifdef DEB_DEBUG_IF
         zlog_debug ("DEB DEBUG:   --  frp_get_interface_address  - got a valid  interface  address");
      #endif //DEB_DEBUG
      int local_p_test = inet_pton (AF_INET,  inet_ntoa (local_p->u.prefix4),   &local_a);
   } else
   {
      #ifdef DEB_DEBUG_IF
         zlog_debug ("DEB DEBUG:   --  frp_get_interface_address  - failed to get local interface address");
```

```c
      #endif //DEB_DEBUG
    }
    return local_a;
}

int
frp_neighbor_lookup (struct sockaddr_in* from)
{   struct prefix_ipv4 p;
    struct route_node *node;
    memset (&p, 0, sizeof (struct prefix_ipv4));
    p.family = AF_INET;
    p.prefix = from->sin_addr;
    p.prefixlen = IPV4_MAX_BITLEN;
    node = route_node_lookup (frp->neighbors, (struct prefix *) &p);
    if (node)
    {  route_unlock_node (node);
       return 1;
    }
    return 0;
}



void
frp_config_write_network (struct vty *vty)
{   unsigned int i;
    char *ifname;
    struct route_node* rn;
    /* network type frp enable interface statement */
    for (rn = route_top (frp_enable_network); rn; rn = route_next (rn))
    {  if (rn->info)
       {  vty_out (vty, "%s%s/%d%s", " network ", inet_ntoa (rn->p.u.prefix4), rn->p.prefixlen, VTY_NEWLINE);
       }
    }
    /* interface name frp enable statement */
    for (i = 0; i < vector_active (frp_enable_interface); i++)
    {  if ((ifname = vector_slot (frp_enable_interface, i)) != NULL)
       {  vty_out (vty, "%s%s%s", " network ", ifname, VTY_NEWLINE);
       }
    }
}




/* called when interface structure allocated */
int
frp_interface_new_hook (struct interface *ifp)
{
    #ifdef DEB_DEBUG_IF
      zlog_debug ("DEB DEBUG: entering frp_interface_new_hook - interface->name = %s, interface->desc = %s, ",
ifp->name, ifp->desc);
    #endif //DEB_DEBUG
    ifp->info = frp_interface_new ();
    return 0;
}

/* called when interface structure deleted */
int
frp_interface_delete_hook (struct interface *ifp)
{   XFREE (MTYPE_FRP_INTERFACE, ifp->info);
```



```c
    ifp->info = NULL;
    return 0;
}



// ------------------------------------------------------------
// ROUTER COMMAND DEFUNs
// ------------------------------------------------------------

/* frp enable a network or interface configuration */
DEFUN (frp_network,
      frp_network_cmd,
      "network (A.B.C.D/M|WORD)",
      "Enable routing on an IP network\n"
      "IP prefix <network>/<length>, e.g., 35.0.0.0/8\n"
      "Interface name\n")
{   int ret;
    struct prefix_ipv4 p;
    #ifdef DEB_DEBUG_IF
      vty_out (vty, "DEB DEBUG: entering frp_interface.c - frp_network_cmd %s", VTY_NEWLINE);
    #endif //DEB_DEBUG
    ret = str2prefix_ipv4 (argv[0], &p);
    if (ret)
    {
      #ifdef DEB_DEBUG_IF
        vty_out (vty, "DEB DEBUG: -- frp_network_cmd - inside IF %s", VTY_NEWLINE);
      #endif //DEB_DEBUG
      ret = frp_enable_network_add ((struct prefix *) &p);
      #ifdef DEB_DEBUG_IF
        vty_out (vty, "DEB DEBUG: -- frp_network_cmd - successfully called frp_enable_network_add %s",
VTY_NEWLINE);
      #endif //DEB_DEBUG
    } else
    {
      #ifdef DEB_DEBUG_IF
        vty_out (vty, "DEB DEBUG: -- frp_network_cmd - inside ELSE %s", VTY_NEWLINE);
      #endif //DEB_DEBUG
      ret = frp_enable_if_add (argv[0]);
      #ifdef DEB_DEBUG_IF
        vty_out (vty, "DEB DEBUG: -- frp_network_cmd - successfully called frp_enable_if_add %s", VTY_NEWLINI
      #endif //DEB_DEBUG
    }
    if (ret < 0)
    {  vty_out (vty, "There is a same network configuration %s%s", argv[0], VTY_NEWLINE);
       return CMD_WARNING;
    } else  //send an update event
    {  frp_event (FRP_EVENT_UPDATE, 1);
    }
    return CMD_SUCCESS;
}



// ------------------------------------------------------------
// INIT FUNCTION
// ------------------------------------------------------------
void
frp_interface_init (void)
{
    #ifdef DEB_DEBUG_IF
```

```
        fprintf (stderr, "DEB DEBUG: entering frp_interface.c - frp_interface_init\n");
    #endif //DEB_DEBUG

    /* default initial size of interface vector */
    if_init();
    if_add_hook (IF_NEW_HOOK, frp_interface_new_hook);
    if_add_hook (IF_DELETE_HOOK, frp_interface_delete_hook);

    /* frp network init. */
    frp_enable_interface = vector_init (1);
    frp_enable_network = route_table_init ();

    // create the FRP network command
    install_element (FRP_NODE, &frp_network_cmd);
//    install_element (FRP_NODE, &no_frp_neighbor_cmd);

    return;
}
```

```
#ifndef _ZEBRA_FRP_DEBUG_H
#define _ZEBRA_FRP_DEBUG_H


// -----------------------------------------------------------
/* INCLUDES */
// -----------------------------------------------------------
#include "frpd.h"


// -----------------------------------------------------------
/* DEFINES */
// -----------------------------------------------------------
/* frp debug zebra flags */
#define FRP_DEBUG_ZEBRA      0x01
/* frp debug event flags */
#define FRP_DEBUG_EVENT      0x01
/* frp debug packet flags */
#define FRP_DEBUG_PACKET     0x01
#define FRP_DEBUG_SEND       0x20
#define FRP_DEBUG_RECV       0x40
#define FRP_DEBUG_DETAIL     0x80


// -----------------------------------------------------------
/* EXTERNAL VARIABLES */
// -----------------------------------------------------------
extern unsigned long frp_debug_zebra;
extern unsigned long frp_debug_event;
extern unsigned long frp_debug_packet;


// -----------------------------------------------------------
/* MACROS */
// -----------------------------------------------------------
#define IS_FRP_DEBUG_ZEBRA  (frp_debug_zebra & FRP_DEBUG_ZEBRA)

#define IS_FRP_DEBUG_EVENT  (frp_debug_event & FRP_DEBUG_EVENT)

#define IS_FRP_DEBUG_PACKET  (frp_debug_packet & FRP_DEBUG_PACKET)
#define IS_FRP_DEBUG_SEND    (frp_debug_packet & FRP_DEBUG_SEND)
#define IS_FRP_DEBUG_RECV    (frp_debug_packet & FRP_DEBUG_RECV)
#define IS_FRP_DEBUG_DETAIL  (frp_debug_packet & FRP_DEBUG_DETAIL)


// -----------------------------------------------------------
// PROTOTYPES
// -----------------------------------------------------------
extern void frp_debug_init (void);
extern void frp_debug_reset (void);


#endif /* _ZEBRA_FRP_DEBUG_H */
```

```
// -------------------------------------------------------------
// INCLUDES
// -------------------------------------------------------------
#include <zebra.h>
#include "command.h"

#include "frpd.h"
#include "frp_debug.h"


// -------------------------------------------------------------
// GLOBAL VARIABLES
// -------------------------------------------------------------
unsigned long frp_debug_zebra = 0;
unsigned long frp_debug_event = 0;
unsigned long frp_debug_packet = 0;

/*  frp debug node */
static struct cmd_node debug_node =
{  DEBUG_NODE,
   "",            /*  Debug node has no interface.  */
   1
};



// -------------------------------------------------------------
// FUNCTIONS
// -------------------------------------------------------------

static int
frp_config_write_debug  (struct vty *vty)
{  int write  = 0;
   if (IS_FRP_DEBUG_ZEBRA)
   {  vty_out (vty,  "debug frp  zebra%s",  VTY_NEWLINE);
      write++;
   }
   if (IS_FRP_DEBUG_EVENT)
   {  vty_out (vty,  "debug frp  events%s",  VTY_NEWLINE);
      write++;
   }
   if (IS_FRP_DEBUG_PACKET)
   {  if (IS_FRP_DEBUG_SEND  &&  IS_FRP_DEBUG_RECV)
      {  vty_out (vty,  "debug frp  packet%s%s",  IS_FRP_DEBUG_DETAIL  ?  " detail"  :  "",  VTY_NEWLINE);
         write++;
      } else
      {  if (IS_FRP_DEBUG_SEND)
         {  vty_out (vty,  "debug frp  packet send%s%s",  IS_FRP_DEBUG_DETAIL  ?  " detail"  :  "",  VTY_NEWLINE);
         } else
         {  vty_out (vty,  "debug frp  packet recv%s%s",   IS_FRP_DEBUG_DETAIL  ?  " detail"  :  "",  VTY_NEWLINE);
         }
         write++;
      }
   }
   return  write;
}

void
frp_debug_reset  (void)
{  frp_debug_zebra = 0;
   frp_debug_event = 0;
```



```
   frp_debug_packet = 0;
}



// -------------------------------------------------------------
// ROUTER COMMAND DEFUNs
// -------------------------------------------------------------

DEFUN (show_debugging_frp,
          show_debugging_frp_cmd,
          "show  debugging frp",
          SHOW_STR
          DEBUG_STR
          FRP_STR)                                          // DEB COMMENT:  these 3 _STR defined in command.h
{   vty_out (vty,  "FRP  debugging status:%s",   VTY_NEWLINE);
    if (IS_FRP_DEBUG_ZEBRA)
    {  vty_out (vty,  "  FRP  zebra  debugging is  on%s",  VTY_NEWLINE);
    }
    if (IS_FRP_DEBUG_EVENT)
    {  vty_out (vty,  "  FRP  event  debugging is  on%s",  VTY_NEWLINE);
    }
    if (IS_FRP_DEBUG_PACKET)
    {  if (IS_FRP_DEBUG_SEND  &&  IS_FRP_DEBUG_RECV)
       {  vty_out (vty,  "  FRP  packet%s  debugging is  on%s",  IS_FRP_DEBUG_DETAIL  ?  " detail"  :  "",  VTY_NEWLINE);
       } else
       {  if (IS_FRP_DEBUG_SEND)
          {  vty_out (vty,  "  FRP  packet send%s  debugging is  on%s",  IS_FRP_DEBUG_DETAIL  ?  " detail"  :  "",  VTY_NEW
          } else
          {  vty_out (vty,  "  FRP  packet receive%s   debugging is  on%s",  IS_FRP_DEBUG_DETAIL  ?  " detail"  :  "",
VTY_NEWLINE);
          }
       }
    }
    return  CMD_SUCCESS;
}

DEFUN (debug_frp_zebra,
          debug_frp_zebra_cmd,
          "debug frp  zebra",
          DEBUG_STR
          FRP_STR
          "FRP  and ZEBRA  communication\n")
{   frp_debug_zebra = FRP_DEBUG_ZEBRA;
    return  CMD_WARNING;
}
DEFUN (no_debug_frp_zebra,
          no_debug_frp_zebra_cmd,
          "no debug frp  zebra",
          NO_STR
          DEBUG_STR
          FRP_STR
          "FRP  and ZEBRA  communication\n")
{   frp_debug_zebra = 0;
    return  CMD_WARNING;
}

DEFUN (debug_frp_events,
          debug_frp_events_cmd,
          "debug frp  events",
          DEBUG_STR
```

```
        FRP_STR
        "FRP events\n")
{
  frp_debug_event = FRP_DEBUG_EVENT;
  return CMD_WARNING;
}
DEFUN (no_debug_frp_events,
        no_debug_frp_events_cmd,
        "no debug frp events",
        NO_STR
        DEBUG_STR
        FRP_STR
        "FRP events\n")
{
  frp_debug_event = 0;
  return CMD_SUCCESS;
}

DEFUN (debug_frp_packet,
        debug_frp_packet_cmd,
        "debug frp packet",
        DEBUG_STR
        FRP_STR
        "FRP packet\n")
{  frp_debug_packet = FRP_DEBUG_PACKET;
   frp_debug_packet |= FRP_DEBUG_SEND;
   frp_debug_packet |= FRP_DEBUG_RECV;
   return CMD_SUCCESS;
}
DEFUN (no_debug_frp_packet,
        no_debug_frp_packet_cmd,
        "no debug frp packet",
        NO_STR
        DEBUG_STR
        FRP_STR
        "FRP packet\n")
{  frp_debug_packet = 0;
   return CMD_SUCCESS;
}

DEFUN (debug_frp_packet_direct,
        debug_frp_packet_direct_cmd,
        "debug frp packet (recv|send)",
        DEBUG_STR
        FRP_STR
        "FRP packet\n"
        "FRP receive packet\n"
        "FRP send packet\n")
{  frp_debug_packet |= FRP_DEBUG_PACKET;
   if (strncmp ("send", argv[0], strlen (argv[0])) == 0)
   {  frp_debug_packet |= FRP_DEBUG_SEND;
   }
   if (strncmp ("recv", argv[0], strlen (argv[0])) == 0)
   {  frp_debug_packet |= FRP_DEBUG_RECV;
   }
   frp_debug_packet &= ~FRP_DEBUG_DETAIL;
   return CMD_SUCCESS;
}
DEFUN (no_debug_frp_packet_direct,
        no_debug_frp_packet_direct_cmd,
        "no debug frp packet (recv|send)",
```



```
        NO_STR
        DEBUG_STR
        FRP_STR
        "FRP packet\n"
        "FRP option set for receive packet\n"
        "FRP option set for send packet\n")
{  if (strncmp ("send", argv[0], strlen (argv[0])) == 0)
   {  if (IS_FRP_DEBUG_RECV)
      {  frp_debug_packet &= ~FRP_DEBUG_SEND;
      } else
      {  frp_debug_packet = 0;
      }
   } else if (strncmp ("recv", argv[0], strlen (argv[0])) == 0)
   {  if (IS_FRP_DEBUG_SEND)
      {  frp_debug_packet &= ~FRP_DEBUG_RECV;
      } else
      {  frp_debug_packet = 0;
      }
   }
   return CMD_SUCCESS;
}

DEFUN (debug_frp_packet_detail,
        debug_frp_packet_detail_cmd,
        "debug frp packet (recv|send) detail",
        DEBUG_STR
        FRP_STR
        "FRP packet\n"
        "FRP receive packet\n"
        "FRP send packet\n"
        "Detailed information display\n")
{  frp_debug_packet |= FRP_DEBUG_PACKET;
   if (strncmp ("send", argv[0], strlen (argv[0])) == 0)
   {  frp_debug_packet |= FRP_DEBUG_SEND;
   }
   if (strncmp ("recv", argv[0], strlen (argv[0])) == 0)
   {  frp_debug_packet |= FRP_DEBUG_RECV;
   }
   frp_debug_packet |= FRP_DEBUG_DETAIL;
   return CMD_SUCCESS;
}

DEFUN (debug_write_peers,
   debug_write_peers_cmd,
   "debug peers",
   "Display current peers\n"
   "Detailed peer information display\n")
{
   #ifdef DEB_DEBUG
      fprintf (stderr, "DEB DEBUG: entering frp_debug.c - debug_write_peers\n");
   #endif //DEB_DEBUG
   struct frp_peer* peer;
   struct listnode *node, *nnode;
   for (ALL_LIST_ELEMENTS (frp_peers, node, nnode, peer))
   {  vty_out (vty, "peer information %s", VTY_NEWLINE);
      vty_out (vty, "   address: %s %s", inet_ntoa (peer->address), VTY_NEWLINE);
      vty_out (vty, "   secret:  %s %s", peer->secret, VTY_NEWLINE);
      vty_out (vty, "   cost:    %d %s", peer->cost, VTY_NEWLINE);
      vty_out (vty, "   poll:    %d %s", peer->poll, VTY_NEWLINE);
      vty_out (vty, "   retry:   %d %s", peer->retry, VTY_NEWLINE);
      vty_out (vty, "   lseq:    %d %s", peer->lseq, VTY_NEWLINE);
```

```
        vty_out (vty, "    rseq:      %d %s", peer->rseq, VTY_NEWLINE);
        vty_out (vty, "    packet_latest_lseq:      %d %s", peer->packet_latest_lseq, VTY_NEWLINE);
        vty_out (vty, "    gw cost:%d %s", peer->gateway_cost, VTY_NEWLINE);
        vty_out (vty, "    gateway path %s", VTY_NEWLINE);
        struct in_addr* gw_path_addr;
        struct listnode *gw_node, *gw_nnode;
        for (ALL_LIST_ELEMENTS (peer->gateway_path, gw_node, gw_nnode, gw_path_addr))
        {  vty_out (vty, "        address:      %s %s", inet_ntoa(*gw_path_addr), VTY_NEWLINE);
        }
        vty_out (vty, "    rib %s", VTY_NEWLINE);
        struct frp_rte* rib_route;
        struct listnode* rib_node;
        for (ALL_LIST_ELEMENTS_RO (peer->rib, rib_node, rib_route))
        {  vty_out (vty, "        route %s", VTY_NEWLINE);
           vty_out (vty, "           length:       %d %s", rib_route->length, VTY_NEWLINE);
           vty_out (vty, "           routecost:  %d %s", rib_route->routecost, VTY_NEWLINE);
           vty_out (vty, "           gatecost:    %d %s", rib_route->routecost, VTY_NEWLINE);
           vty_out (vty, "           prefix:        %s %s", inet_ntoa(rib_route->prefix.prefix), VTY_NEWLINE);
        }

    }
    return CMD_SUCCESS;
}

DEFUN (debug_write_frp_rib,
    debug_write_frp_rib_cmd,
    "debug rib",
    "Display the current rib\n"
    "Detailed rib information display\n")
{
    #ifdef DEB_DEBUG
        fprintf (stderr, "DEB DEBUG: entering frp_debug.c - debug_write_frp_rib\n");
    #endif //DEB_DEBUG
    vty_out (vty, "frp rib %s", VTY_NEWLINE);
    struct route_node*      rib_rte_ptr;
    for (rib_rte_ptr = route_top (frp->rib); rib_rte_ptr; rib_rte_ptr = route_next (rib_rte_ptr))
    {  //only display entries with frp information
        struct frp_info* rte_info = (struct frp_info*)rib_rte_ptr->info;
        if (rte_info != NULL)
        {  vty_out (vty, "    rib entry %s", VTY_NEWLINE);
           vty_out (vty, "        prefix:           %s %s", inet_ntoa(rib_rte_ptr->p.u.prefix4), VTY_NEWLINE);
           vty_out (vty, "        nexthop:        %s %s", inet_ntoa(rte_info->nexthop), VTY_NEWLINE);
           vty_out (vty, "        cost:              %d %s", rte_info->cost, VTY_NEWLINE);
           vty_out (vty, "        is_gateway:    %d %s", rte_info->is_gateway_flag, VTY_NEWLINE);
        }
        route_unlock_node (rib_rte_ptr);
    }
    return CMD_SUCCESS;
}



// ------------------------------------------------------------
// INIT FUNCTION
// ------------------------------------------------------------
void
frp_debug_init (void)
{
#ifdef DEB_DEBUG
    fprintf (stderr, "DEB DEBUG: entering frp_debug.c - frp_debug_init\n");
#endif //DEB_DEBUG
```



```

    frp_debug_zebra = 0;
    frp_debug_event = 0;
    frp_debug_packet = 0;

    install_node (&debug_node, frp_config_write_debug);

    install_element (ENABLE_NODE, &show_debugging_frp_cmd);

    install_element (ENABLE_NODE, &debug_frp_zebra_cmd);
    install_element (CONFIG_NODE, &debug_frp_zebra_cmd);
    install_element (ENABLE_NODE, &no_debug_frp_zebra_cmd);
    install_element (CONFIG_NODE, &no_debug_frp_zebra_cmd);

    install_element (ENABLE_NODE, &debug_frp_events_cmd);
    install_element (CONFIG_NODE, &debug_frp_events_cmd);
    install_element (ENABLE_NODE, &no_debug_frp_events_cmd);
    install_element (CONFIG_NODE, &no_debug_frp_events_cmd);

    install_element (ENABLE_NODE, &debug_frp_packet_cmd);
    install_element (CONFIG_NODE, &debug_frp_packet_cmd);
    install_element (ENABLE_NODE, &no_debug_frp_packet_cmd);
    install_element (CONFIG_NODE, &no_debug_frp_packet_cmd);
    install_element (ENABLE_NODE, &debug_frp_packet_direct_cmd);
    install_element (CONFIG_NODE, &debug_frp_packet_direct_cmd);
    install_element (ENABLE_NODE, &no_debug_frp_packet_direct_cmd);
    install_element (CONFIG_NODE, &no_debug_frp_packet_direct_cmd);
    install_element (ENABLE_NODE, &debug_frp_packet_detail_cmd);
    install_element (CONFIG_NODE, &debug_frp_packet_detail_cmd);

    install_element (CONFIG_NODE, &debug_write_peers_cmd);

    install_element (CONFIG_NODE, &debug_write_frp_rib_cmd);

#ifdef DEB_DEBUG
    fprintf (stderr, "DEB DEBUG:  -- leaving frp_debug_init\n");
#endif //DEB_DEBUG
}
```

Appendix D

The FRPsniffer implementation

```
// ------------------------------------------------------------
// FRPsniffer
// a dedicated packet sniffer for the FRP protocol
// written by Deb Shepherd
// ------------------------------------------------------------


#include <stdio.h>
#include <netinet/in.h>
#include <sys/types.h>
#include <stdlib.h>
#include <stdint.h>
#include <stddef.h>
#include <arpa/inet.h>

// PACKET CAPTURE LIBRARY
#include <pcap.h>
// integer, defines max bytes to be captured
#define SNAPLEN 65535
// when true brings into promiscuous mode
#define PROMISC 1
//read timeout in milliseconds (0 means no time out)
#define TO_MS 10000

// contains definition for 'in_addr' used in ip4 header
#include <netinet/in.h>

// ethernet headers are 14 bytes
#define EN_HEADER_LEN 14
// ethernet addresses are 6 bytes
#define EN_ADDRESS_LEN 6
// Ethernet Header (14 bytes)
struct ethernet
{  u_char enDestination[EN_ADDRESS_LEN];    // destination address (8 bits x 6 = 6 bytes)
   u_char enSource[EN_ADDRESS_LEN];         // source address (8 bits x 6 = 6 bytes)
   u_short enType;                          // IP, ARP, etc (16 bits = 2 bytes)
};

// IPv4 Header (20+ bytes)
struct ip4
{  u_char ip4vhl;                           // version and header length (4+4 bits)
   // extract version from ip4vhl by bit shifting 4 (4 bits)
#define IP4VERSION(ipv)   (((ipv)->ip4vhl) >> 4)
   // extract header length from ip4vhl by masking with 0000 1111 (4 bits)
#define IP4HEADER_LEN(iphl)   (((iphl)->ip4vhl) & 0x0f)
   u_char ip4service;                       // type of service (8 bits)
   u_short ip4length;                     // total datagram length (16 bits)
   u_short ip4id;                           // identifier (16 bits)
   u_short ip4offset;                     // flags and fragmentation offset (3+13 bits)
   // reserved fragment flag mask (1 bit) 1000 0000 0000 0000
#define IP4RESERVED 0x8000
   // don't fragment flag mask (1 bit) 0100 0000 0000 0000
#define IP4DONT 0x4000
   // more fragments flag mask (1 bit) 0010 0000 0000 0000
#define IP4MORE 0x2000
   // mask for fragmenting bits (13 bits) 0001 1111 1111 1111
#define IP4MASK 0x1fff
   u_char ip4ttl;                           // time to live (8 bits)
   u_char ip4protocol;                      // protocol (8 bits)
   u_short ip4checksum;                     // checksum (16 bits)
   struct in_addr ip4source;                // source address (32 bits) (needs include <netinet/in.h>)
```



```
   struct in_addr ip4destination;           // destination address (32 bits) (needs include <netinet/in.h>)
};

// udp headers are 8 bytes
#define UDP_HEADER_LEN 8
// UDP Header (8 bytes)
struct udp
{  u_short udpSource;                       // source port (16 bits)
   u_short udpDestination;                // destination port (16 bits)
   u_short udpLength;                       // message length (16 bits)
   u_short udpChecksum;                     // checksum (16 bits)
};
```

```c
// ------------------------------------------------------------
// FRPsniffer
// a dedicated packet sniffer for the FRP protocol
// written by Deb Shepherd
// ------------------------------------------------------------


#define FRP_PKT_HDRSIZE                 16      // (128 bits)
#define FRP_PKT_MINSIZE                 16      // (128 bits)
#define FRP_PKT_MAXSIZE                 1400    // as specified by Don
#define FRP_MSG_CONTROL_SIZE            4       // (32 bits)
#define FRP_MSG_IPV4CONFIG_SIZE         12      // (96 bits)
#define FRP_MSG_IPV4GATEWAY_MINSIZE     8       // (64 bits)
#define FRP_MSG_IPV4UPDATE_SIZE         12      // (96 bits)
#define FRP_MSG_IPV6CONFIG_SIZE         24      // (192 bits)
#define FRP_MSG_IPV6GATEWAY_MINSIZE     20      // (160 bits)
#define FRP_MSG_IPV6UPDATE_SIZE         24      // (192 bits)

// frp message types
#define FRP_MSG_NULL            0x00
#define FRP_MSG_CONTROL         0x01
#define FRP_MSG_IPV4CONFIG      0x41
#define FRP_MSG_IPV4GATEWAY     0x42
#define FRP_MSG_IPV4UPDATE      0x43
#define FRP_MSG_IPV6CONFIG      0x61
#define FRP_MSG_IPV6GATEWAY     0x62
#define FRP_MSG_IPV6UPDATE      0x63

// frp control types
#define FRP_CTRL_POLL           1
#define FRP_CTRL_ACK            2
#define FRP_CTRL_NAK            3

// frp update flags
#define FRP_FLAG_BEGIN          0x01
#define FRP_FLAG_COMMIT         0x02
#define FRP_FLAG_NULLRT         0x04
#define FRP_FLAG_UPDATE         0x08
#define FRP_FLAG_DELETE         0x10
#define FRP_FLAG_GATEWAY        0x80


// frp packet header (128 bits)
struct frp_pkt_hdr
{   u_int8_t    hash[8];                // security hash (64 bits)
    u_int32_t   sendSeq;                // sender's sequence number (32 bits)
    u_int32_t   recipAck;               // recipient's acknowledgement number (32 bits)
};

// frp header (16 bits)
struct frp_msg_hdr
{   u_int8_t    length;                 // message length (8 bits)
    u_int8_t    type;                   // message type (8 bits)
};

// frp control (32 bits)
struct frp_msg_control
{   struct frp_msg_hdr  msg_hdr;        // message header (16 bits)
    u_int8_t            type;           // control type (8 bits)
    u_int8_t            param;          // control parameters (8 bits)
};
```



```c

// frp IPv4 configuration (96 bits)
struct frp_msg_ipv4config
{   struct frp_msg_hdr  msg_hdr;        // message header (16 bits)
    u_short             cost;           // cost of the link (16 bits)
    u_short             poll;           // poll time (16 bits)
    u_short             retry;          // retry time (16 bits)
    struct in_addr      id;             // router-ID of peer (32 bits)
};

// frp IPv4 path to gateway (64-2016 bits)
struct frp_msg_ipv4gateway
{   struct frp_msg_hdr  msg_hdr;        // message header (16 bits)
    u_short             cost;           // cost from peer to gateway (16 bits)
    struct in_addr      path[62];       // path from peer to gateway (32xn bits, 1 <= n <= 62)
};

// frp IPv4 route update (96 bits)
struct frp_msg_ipv4update
{   struct frp_msg_hdr  msg_hdr;        // message header (16 bits)
    u_int8_t            flags;          // update type flags (8 bits)
    u_int8_t            length;         // prefix length (8 bits)
    u_short             routecost;      // route cost (16 bits)
    u_short             gatecost;       // cost from originator to gateway (16 bits)
    struct in_addr      prefix;         // IP prefix (32 bits)
};

// frp IPv6 configuration (192 bits)
struct frp_msg_ipv6config
{   struct frp_msg_hdr  msg_hdr;        // message header (16 bits)
    u_short             cost;           // cost of the link (16 bits)
    u_short             poll;           // poll time (16 bits)
    u_short             retry;          // retry time (16 bits)
    struct in6_addr     id;             // router-ID of peer (128 bits)
};

// frp IPv6 path to gateway (160-7968 bits)
struct frp_msg_ipv6gateway
{   struct frp_msg_hdr  msg_hdr;        // message header (16 bits)
    u_short             cost;           // cost from peer to gateway (16 bits)
    struct in6_addr     path[1];        // path from peer to gateway (32xn bits, 1 <= n <= 62)
};

// frp IPv6 route update (192 bits)
struct frp_msg_ipv6update
{   struct frp_msg_hdr  msg_hdr;        // message header (16 bits)
    u_int8_t            flags;          // update type flags (8 bits)
    u_int8_t            length;         // prefix length (8 bits)
    u_short             routecost;      // route cost (16 bits)
    u_short             gatecost;       // cost from originator to gateway (16 bits)
    struct in6_addr     prefix;         // IP prefix (128 bits)
};



// ------------------------------------------------------------
// GLOBAL VARIABLES AND DEFINES
// ------------------------------------------------------------

struct frp_pkt_hdr      pkt_hdr;
struct frp_pkt_hdr*     pkt_hdr_ptr;
```

```
/* frp event. */
enum frp_makepkthdr_flag
{  FRP_ACK,
   FRP_SYN,
};
```

```c
// ------------------------------------------------------------
// FRPsniffer
// a dedicated packet sniffer for the FRP protocol
// written by Deb Shepherd
// ------------------------------------------------------------


#include "sniffer.h"
#include "frp_packet.h"

u_short displayControlMessage (const struct frp_msg_control* frpControlMessage)
{ switch (frpControlMessage->type) {
      case FRP_CTRL_POLL:
         printf("--Poll,   param = %d \n", frpControlMessage->param);
         break;
      case FRP_CTRL_ACK:
         printf("--Control  ACK, param = %d \n", frpControlMessage->param);
         break;
      case FRP_CTRL_NAK:
         printf("--Control  NAK, param = %d \n", frpControlMessage->param);
         break;
      default:
         break;
   }
   return frpControlMessage->msg_hdr.length;
}

u_short displayIP4ConfigMessage (const struct frp_msg_ipv4config* frpConfigMessage)
{ printf("              cost: %d\n", ntohs(frpConfigMessage->cost));
   printf("              poll: %d\n", ntohs(frpConfigMessage->poll));
   printf("             retry: %d\n", ntohs(frpConfigMessage->retry));
   printf("         router-id: %s\n", inet_ntoa(frpConfigMessage->id));
   return frpConfigMessage->msg_hdr.length;
}

u_short displayIP4GatewayMessage(const struct frp_msg_ipv4gateway* frpGatewayMessage)
{
   printf("              cost: %d\n", ntohs(frpGatewayMessage->cost));
   u_short numNodes = frpGatewayMessage->msg_hdr.length / 8;
   printf("  # of nodes in path: %d\n", numNodes);
   for (int i = 0; i < numNodes; i++)
   {
      printf("           path[%d]: %s\n", i, inet_ntoa(frpGatewayMessage->path[i]));
   }
   return frpGatewayMessage->msg_hdr.length;
}

u_short displayIP4UpdateMessage(const struct frp_msg_ipv4update* frpUpdateMessage)
{ printf("             flags: %d\n", ntohs(frpUpdateMessage->flags));
   if (ntohs(frpUpdateMessage->flags & FRP_FLAG_BEGIN)){
      printf("                 FRP_FLAG_BEGIN flag: YES\n");
   }
   else {
      printf("                 FRP_FLAG_BEGIN flag: NO\n");
   }
   if (ntohs(frpUpdateMessage->flags & FRP_FLAG_NULLRT)){
      printf("                FRP_FLAG_NULLRT flag: YES\n");
   }
   else {
      printf("                FRP_FLAG_NULLRT flag: NO\n");
   }
```



```c
   if (ntohs(frpUpdateMessage->flags & FRP_FLAG_DELETE)){
      printf("                FRP_FLAG_DELETE flag: YES\n");
   }
   else {
      printf("                FRP_FLAG_DELETE flag: NO\n");
   }
   if (ntohs(frpUpdateMessage->flags & FRP_FLAG_COMMIT)){
      printf("                FRP_FLAG_COMMIT flag: YES\n");
   }
   else {
      printf("                FRP_FLAG_COMMIT flag: NO\n");
   }
   if (ntohs(frpUpdateMessage->flags & FRP_FLAG_UPDATE)){
      printf("                FRP_FLAG_UPDATE flag: YES\n");
   }
   else {
      printf("                FRP_FLAG_UPDATE flag: NO\n");
   }
   if (ntohs(frpUpdateMessage->flags & FRP_FLAG_GATEWAY)){
      printf("               FRP_FLAG_GATEWAY flag: YES\n");
   }
   else {
      printf("               FRP_FLAG_GATEWAY flag: NO\n");
   }
   printf("            length: %d\n", frpUpdateMessage->length);
   printf("          routecost: %d\n", ntohs(frpUpdateMessage->routecost));
   printf("           gatecost: %d\n", ntohs(frpUpdateMessage->gatecost));
   printf("            prefix: %s\n", inet_ntoa(frpUpdateMessage->prefix));
   return frpUpdateMessage->msg_hdr.length;
}

int main()
{ // INTERFACE VARIABLES
   char* interface = "en0\0";                     // a pointer to the interface to sniff - hard code for now ...
   bpf_u_int32 netaddress;                        // interface net address
   bpf_u_int32 netmask;                           // interface netmask
   // PCAP SESSION VARIABLES
   char errorbuffer[PCAP_ERRBUF_SIZE];            // pcap error string
   pcap_t* handle = NULL;                         // a pointer to the pcap session handle
   // PCAP filter variables
   char filterexpression[] = "udp port 343";      // filter expression
   struct bpf_program filter;                     // compiled filter expression
   // packet variables
   struct pcap_pkthdr packetHeader;               // the packet header from pcap
   const u_char* packet;                          // a pointer to the actual packet
   const struct ethernet* ethernetHeader;         // a pointer to the ethernet header
   const struct ip4* ip4Header;                   // a pointer to the IPv4 header
   u_int ip4HeaderSize;                           // actual size of the IPv4 header
   const struct udp* udpHeader;                   // a pointer to the UDP header
   u_int udpHeaderSize;                           // actual size of the UDP header
   const struct frp_pkt_hdr* frpHeader;           // a pointer to the FRP header
   u_int frpHeaderSize;                           // actual size of the FRP header
   //const char* payload;                         // a pointer to the packet payload

   // find out about the interface (pcap_lookupnet)
   printf("Interface: %s\n", interface);
   if (pcap_lookupnet(interface, &netaddress, &netmask, errorbuffer) == -1)
   { fprintf(stderr, "Can't get netmask for interface %s\n", interface);
      netaddress = 0;
      netmask = 0;
   }
```

```c

    // set up a session handle and open session (pcap_open_live)
    handle = pcap_open_live(interface, SNAPLEN, PROMISC, TO_MS, errorbuffer);
    if (handle == NULL)
    {  fprintf(stderr, "Couldn't open interface %s: %s\n", interface, errorbuffer);
       return(1);
    }

    // compile and apply 'udp port 343' filter
    if (pcap_compile(handle, &filter, filterexpression, 0, netmask) == -1)
    {  fprintf(stderr, "Couldn't parse filter %s: %s\n", filterexpression, pcap_geterr(handle));
       return(2);
    }
    if (pcap_setfilter(handle, &filter) == -1)
    {  fprintf(stderr, "Couldn't install filter %s: %s\n", filterexpression, pcap_geterr(handle));
       return(2);
    }

    while (1)
    {  printf("Waiting for incoming FRP packet\n");
       // collect a single packet from the interface (pcap_next)
       packet = pcap_next(handle, &packetHeader);
       if (packet == NULL)
       {  //printf("No packet collected");
       } else
       {
          printf("\n");
          printf("\nFRP PACKET HEADER\n");
//        printf("Packet timestamp: [%d]\n", packetHeader.ts);
//        printf("Packet available bytes: [%d]\n", packetHeader.caplen);
//        printf("Packet length: [%d]\n", packetHeader.len);

          // read packet headers
          // ethernet
          ethernetHeader = (struct ethernet*)(packet);               // read from point 0 as an ethernet header
//        printf("          ETHERNET\n");
          u_short byte0 = ethernetHeader->enDestination[0];
          u_short byte1 = ethernetHeader->enDestination[1];
          u_short byte2 = ethernetHeader->enDestination[2];
          u_short byte3 = ethernetHeader->enDestination[3];
          u_short byte4 = ethernetHeader->enDestination[4];
          u_short byte5 = ethernetHeader->enDestination[5];
//        printf("destination address: %x:%x:%x:%x:%x:%x\n",    byte0, byte1, byte2, byte3, byte4, byte5);
          byte0 = ethernetHeader->enSource[0];
          byte1 = ethernetHeader->enSource[1];
          byte2 = ethernetHeader->enSource[2];
          byte3 = ethernetHeader->enSource[3];
          byte4 = ethernetHeader->enSource[4];
          byte5 = ethernetHeader->enSource[5];
//        printf("    source address: %x:%x:%x:%x:%x:%x\n",    byte0, byte1, byte2, byte3, byte4, byte5);
//        printf("     ethernet type: %x\n", ntohs(ethernetHeader->enType));
          // IPv4
          ip4Header = (struct ip4*)(packet + EN_HEADER_LEN);     // read from point 14 as an IPv4 header
          ip4HeaderSize = IP4HEADER_LEN(ip4Header)*4;            // calculate actual IPv4 header length
//        printf("            IPv4\n");
          if (ip4HeaderSize < 20)                          // check IPv4 header is at least the min 20 bytes
          {
//           printf("   * Invalid IP header length: %u (%u) bytes\n", ip4HeaderSize, IP4HEADER_LEN(ip4Header));
             return (2);
          } else
          {
```



```c
//           printf("           version: %u (0x%x)\n", IP4VERSION(ip4Header), IP4VERSION(ip4Header));
//           printf("     header length: %u bytes (0x%x)\n", ip4HeaderSize, ip4HeaderSize);
//           printf("   type of service: 0x%x\n", ip4Header->ip4service);
//           printf("   datagram length: %u bytes (0x%x)\n", ntohs(ip4Header->ip4length), ntohs(ip4Header->ip4len
//           printf("        identifier: %u (0x%x)\n", ntohs(ip4Header->ip4id), ntohs(ip4Header->ip4id));
             byte0 = (ip4Header->ip4offset & IP4RESERVED);
             byte1 = (ip4Header->ip4offset & IP4DONT);
             byte2 = (ip4Header->ip4offset & IP4MORE);
             byte3 = (ip4Header->ip4offset & IP4MASK);
//           printf("             flags: reserved(%x) don't(%x) more(%x) offset(%x)\n", byte0, byte1, byte2, byte3)
//           printf("       time to live: %u (0x%x)\n", ip4Header->ip4ttl, ip4Header->ip4ttl);
//           printf("          protocol: %u (0x%x)\n", ip4Header->ip4protocol, ip4Header->ip4protocol);
//           printf("          checksum: %u (0x%x)\n", ntohs(ip4Header->ip4checksum), ntohs(ip4Header->ip4checksu
             byte0 = (ip4Header->ip4source.s_addr & 0xFF);
             byte1 = (ip4Header->ip4source.s_addr & 0xFF00) >> 8;
             byte2 = (ip4Header->ip4source.s_addr & 0xFF0000) >> 16;
             byte3 = (ip4Header->ip4source.s_addr & 0xFF000000) >> 24;
             printf("    source address: %u.%u.%u.%u (%x.%x.%x.%x)\n",   byte0, byte1, byte2, byte3, byte0, byte1
byte2, byte3);
             byte0 = (ip4Header->ip4destination.s_addr & 0xFF);
             byte1 = (ip4Header->ip4destination.s_addr & 0xFF00) >> 8;
             byte2 = (ip4Header->ip4destination.s_addr & 0xFF0000) >> 16;
             byte3 = (ip4Header->ip4destination.s_addr & 0xFF000000) >> 24;
             printf("destination address: %u.%u.%u.%u (%x.%x.%x.%x)\n",   byte0, byte1, byte2, byte3, byte0, byte
byte2, byte3);
          }
          // UDP
          udpHeader = (struct udp*)(packet + EN_HEADER_LEN + ip4HeaderSize);    // read from point 14+20+? as a U
header
          udpHeaderSize = UDP_HEADER_LEN;                                       // UDP header length
//        printf("            UDP\n");
//        printf("       source port: %u (0x%x)\n", ntohs(udpHeader->udpSource), ntohs(udpHeader->udpSource));
//        printf("  destination port: %u (0x%x)\n", ntohs(udpHeader->udpDestination),
ntohs(udpHeader->udpDestination));
//        printf("     header length: %u bytes (0x%x)\n", ntohs(udpHeader->udpLength), ntohs(udpHeader->udpLeng
//        printf("          checksum: %u (0x%x)\n", ntohs(udpHeader->udpChecksum), ntohs(udpHeader->udpChecksu


          // FRP
          frpHeader = (struct frp_pkt_hdr*)(packet + EN_HEADER_LEN + ip4HeaderSize + udpHeaderSize);    // read f
point 14+20+?+8 as a FRP header
          frpHeaderSize = FRP_PKT_HDRSIZE;                                      // FRP header length
//        printf("            FRP\n");

          //Display the FRP packet header
//        printf("\nFRP MESSAGE\n");

          printf("     security hash: %s\n", frpHeader->hash);
          printf("   sender's seq no: %d (0x%x)\n", ntohl(frpHeader->sendSeq), ntohl(frpHeader->sendSeq));
          printf(" recipient's ack no: %d (0x%x)\n", ntohl(frpHeader->recipAck), ntohl(frpHeader->recipAck));
          if (ntohl(frpHeader->recipAck) == 0)
          {  printf("--SYN Packet\n\n");
             continue;
          }
          //if (ntohl(frpHeader->recipAck) != 0)
          //{  printf("    message length: %u (0x%x)\n", frpHeader->msgLength, frpHeader->msgLength);
          //   printf("      message type: 0x%x\n", frpHeader->msgType);
          //}
          u_long currentPos = EN_HEADER_LEN + ip4HeaderSize + udpHeaderSize + frpHeaderSize;
          struct frp_msg_hdr* messageHeader = (struct frp_msg_hdr*)(packet + currentPos);

```

```c
        //Work through the packet extracting and displaying the messages until we get to the end of the packet
        while ((messageHeader != 0)&&(currentPos < packetHeader.len))
        {  switch (messageHeader->type) {
              case FRP_MSG_CONTROL:
                 printf("\nControl message: 0x%x\n", messageHeader->type);
                 currentPos += displayControlMessage((struct frp_msg_control*)messageHeader);
                 break;
              case FRP_MSG_IPV4CONFIG:
                 printf("\nIPV4 Config message: 0x%x\n", messageHeader->type);
                 currentPos += displayIP4ConfigMessage((struct frp_msg_ipv4config*)messageHeader);
                 break;
              case FRP_MSG_IPV4GATEWAY:
                 printf("\nIPV4 path to gateway message: 0x%x\n", messageHeader->type);
                 currentPos += displayIP4GatewayMessage((struct frp_msg_ipv4gateway*)messageHeader);
                 break;
              case FRP_MSG_IPV4UPDATE:
                 printf("\nIPV4 route update message: 0x%x\n", messageHeader->type);
                 currentPos += displayIP4UpdateMessage((struct frp_msg_ipv4update*)messageHeader);
                 break;
              case FRP_MSG_IPV6CONFIG:
                 printf("      IPV6 Config Message: 0x%x\n", messageHeader->type);
                 printf("        Not currently implemented\n");
                 //Not currently implemented
                 break;
              case FRP_MSG_IPV6GATEWAY:
                 printf("      IPV6 Path To Gateway Message: 0x%x\n", messageHeader->type);
                 printf("        Not currently implemented\n");
                 //Not currently implemented
                 break;
              case FRP_MSG_IPV6UPDATE:
                 printf("      IPV6 Route Update Message: 0x%x\n", messageHeader->type);
                 printf("        Not currently implemented\n");
                 //Not currently implemented
                 break;
              case FRP_MSG_NULL:
                 printf("      Null Message: 0x%x\n", messageHeader->type);
                 if ((ntohl(frpHeader->recipAck) != 0)&&(ntohl(frpHeader->sendSeq) != 0))
                 {  printf("--ACK Packet\n\n");
                 }
                 messageHeader = 0;
                 continue;
                 break;
              default:
                 printf("      INVALID MESSAGE TYPE: 0x%x\n", messageHeader->type);
                 printf("      CORRUPT MESSAGE STOPPING PARSE\n");
                 messageHeader = 0;
                 continue;
                 break;
           }
           printf("\n");
           //Advance the message pointer
           messageHeader = (struct frp_msg_hdr*)(packet + currentPos);

           // the remainder is actual data
           //payload = (char *)(packet + EN_HEADER_LEN + ip4HeaderSize + udpHeaderSize + frpHeaderSize);
//            printf("\nEN_HEADER_LEN: %u, ip4HeaderSize: %u, udpHeaderSize: %u, frpHeaderSize: %u", EN_HEAD
ip4HeaderSize, udpHeaderSize, frpHeaderSize);
           //       payload = (char *)(packet + EN_HEADER_LEN + ip4HeaderSize + tcpHeaderSize);
//            printf("\nPAYLOAD: %s\n\n", payload);
        }
     }
```



```c
   }

   // close session
   pcap_close(handle);
   return(0);
}
```

Appendix D

The orignal implementation of FRP by Don Stokes

```c
/*
 Packet header
 */
#define FRP_HASHSIZE 8
struct frp_hdr {
    u_int8_t hash[FRP_HASHSIZE]; /* Security hash of packet/secret */
    u_int32_t lseq;     /* Local sequence number */
    u_int32_t rseq;         /* Remote sequence number */
};


/*
 Message header
 */
struct frp_mhdr {
    u_int8_t len;   /* Length of message, in 32 bit words */
    u_int8_t type;      /* Message type */
};

/*
 Control message
 */
#define FRP_CTRL 0x01
struct frp_ctrl {
    struct frp_mhdr mh;
    u_int8_t ctrl;
    u_int8_t param;
};
#define FRP_CTRL_POLL   1
#define FRP_CTRL_ACK    2
#define FRP_CTRL_NAK    3

/*
 IPv4 protocol messages
 */
#define FRP_CONFIG 0x41    /* IPv4 static configuration */
struct frp_config {
    struct frp_mhdr mh;
    u_int16_t cost;     /* Cost of link */
    u_int16_t poll;     /* Poll time */
    u_int16_t fail;     /* Retry time */
    IPADDR id;      /* Router-ID of peer */
};

#define FRP_PATH    0x42    /* IPv4 path to gateway */
struct frp_path {
    struct frp_mhdr mh;
    u_int16_t gwcost;   /* Cost from peer to gateway */
    IPADDR path[];  /* Path from peer to gateway */
};
#define FRP_NOGATEWAY 0xffff   /* gwcost value if gateway not reachable */

#define FRP_ROUTE  0x43    /* IPv4 route update */
struct frp_route {
    struct frp_mhdr mh;
    u_int8_t flags;     /* See below */
    u_int8_t bits;      /* Prefix length */
    u_int16_t cost;     /* Route cost */
    u_int16_t gwcost;   /* Cost from originator to gateway */
    IPADDR ip;      /* IP prefix */
};
```



```c
#define FRP_FLAG_BEGIN   1 /* First route of batch update */
#define FRP_FLAG_COMMIT  2 /* Last route of batch update, commit */
#define FRP_FLAG_NULL    4 /* Null route (do not add) */
//#define FRP_FLAG_UPDATE  8 /* Add/change single route */
//#define FRP_FLAG_DELETE 16 /* Delete route */
#define FRP_FLAG_GATEWAY 128/* Route is a gateway route */
#define FRP_NULLRT_SIZE 4 /* Size of null route message */


/*
 IPv6 message types
 */
#define FRP_CONFIG6   0x61    /* IPv6 configuration */
struct frp_config6 {
    struct frp_mhdr mh;
    u_int16_t cost;     /* Cost of link */
    struct in6_addr gw; /* Gateway address to use */
};

#define FRP_PATH6  0x62    /* IPv6 path to gateway */
struct frp_path6 {
    struct frp_mhdr mh;
    u_int16_t gwcost;   /* Cost from peer to gateway */
    struct in6_addr path[]; /* Path from peer to gateway */
};

#define FRP_ROUTE6 0x63    /* IPv6 route update */
struct frp_route6 {
    struct frp_mhdr mh;
    u_int8_t flags;     /* See above */
    u_int8_t bits;      /* Prefix length */
    u_int16_t cost;     /* Route cost */
    u_int16_t gwcost;   /* Cost from originator to gateway */
    struct in6_addr ip6; /* IPv6 prefix, truncate to nearest 32 bits */
};
```

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <stdarg.h>
#include <unistd.h>
#include <ctype.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/select.h>
#include <net/if.h>
#include <net/if_dl.h>
#include <net/if_types.h>
#include <net/route.h>
#include <netinet/in.h>
#include <netinet/if_ether.h>
#include <arpa/inet.h>
#include <sys/sysctl.h>
#include <fcntl.h>
#include <signal.h>
#include <ifaddrs.h>
#include <syslog.h>

#define IPADDR u_int32_t
#define TIMETEN u_int32_t

#define WHITESPACE (" \t\n\r")

struct ace {
    struct ace *next;
    int permit;
    IPADDR ip;
    int bits;
    int maxbits;
    int rtype;
    struct acl *acl;
    int ifindex;
};
#define RTYPE_ANY       0
#define RTYPE_LAYER2    1
#define RTYPE_STATIC    2
#define RTYPE_PROTOCOL  4
#define RTYPE_LOCAL     8
#define RTYPE_REMOTE    16

#define MAX_PACKET      1360
#define MAX_PATH        63
#define DEFAULT_POLL    50   /*  5 seconds */
#define DEFAULT_FAIL    150  /* 15 seconds */
#define DEFAULT_RETRY   20   /*  2 seconds */

struct acl {
    struct acl *next;
    char *name;
    struct ace *head;
    struct ace *tail;
};

struct peer {
    struct peer *next;
    int cloned;    /* If 1, object was cloned */
```



```c
    int state;
    int fd;

    struct sockaddr_in lsa; /* Local & remote peer sockaddrs */
    struct sockaddr_in rsa;

    IPADDR routerid;   /* Router ID of peer */
    IPADDR nexthop;    /* IP address for next hop advertisement */
    int ttl;      /* TTL of packets */
    char *secret;      /* Shared secret */
    int confcost;    /* Configured link cost */
    TIMETEN confpoll; /* Configured poll interval */
    TIMETEN conffail; /* Configured fail timeout */
    TIMETEN confretry;  /* Configured retry timeout */
    int cost;      /* Link cost (maximised with peer) */
    TIMETEN poll;      /* Poll interval (minimised with peer) */
    TIMETEN fail;      /* Timeout to fail (minimised with peer) */
    TIMETEN retry;     /* Timeout to retry packet */
    int ifindex;     /* Interface index */
    IPADDR localroute;   /* Network base address of containing subnet */
    int localbits;     /* Prefix length of containing network */
    int localannounce; /* Announce local net, 0 = if winner, */
            /* 1 = always, 2 = never */

    struct acl *remote;   /* Permit connections from (if listen) */
    struct acl *announce;   /* Announce routes to peer */
    struct acl *accept;   /* Accept routes from peer */

    u_int32_t lseq;     /* Send sequence number */
    u_int32_t aseq;       /* Last acknowleged sequence number */
    u_int32_t rseq;       /* Expected sequence number */
    u_int32_t xseq;       /* Re-sync candidate sequence number */
    int pathlen;      /* Length of path to gateway via this link */
    IPADDR path[MAX_PATH];    /* Path to gateway via this link */
    int gwcost;      /* Cost to gateway via this link */

    struct iproute *routes; /* Inbound peer routing table */
    struct iproute *newrts;    /* Assemble new routing table here */
    struct iproute *annrts; /* Routing announcement */

    u_char *lastpacket;   /* Packet in flight */
    int lastpktlen;      /* Length of packet in flight */
    u_char *nextpacket; /* Packet buffer */
    int nextpktlen;     /* Pointer into packet buffer */
    int ackreq;      /* Acklnowlegement required by peer */

    TIMETEN lasttime; /* Time last message received */
    TIMETEN nexttime;   /* Time to transmit next retry */
    int quickstart;     /* Quick retries on startup */

    int synreq;      /* Neet to synchronise */
    int configreq;      /* Need to send config */
    int pathreq;      /* Need to send path */
    int pollreq;      /* Need to send poll resquest */
    int respreq;      /* Need to send poll response */
    int sendack;      /* Inbound message requires ACK */
    int shutdown;      /* Need to shut down peer */
    struct iproute *nextrt; /* Next route to add to outbound packet */
    int nullrt;     /* Send null update */
};
#define LOCIP(peer) ((peer)->lsa.sin_addr.s_addr)
```

```c
#define REMIP(peer) ((peer)->rsa.sin_addr.s_addr)


struct iproute {
    struct iproute *next;
    IPADDR ip;        /* Base IP address of route */
    int bits;         /* Mask length */
    int isgw;         /* 1 if reverse path is via default gateways */
    int cost;         /* Cost to originator via this link */
    int gwcost;       /* Cost from originator to gateway */
    int rtype;        /* Route type */
    struct peer *peer;    /* Link associated with route */
    int ifindex;      /* Interface index of local route */
    int inuse;        /* Flag used to detect unused entries */
};


struct acl *acls;       /* List header of named ACLs */

IPADDR maskbits[33];      /* Map prefix length to mask */

struct iproute *freelist;     /* Free list of route objects */
struct iproute *localroutes;   /* Local routing table header */

struct peer *peers;       /* Running peers */
struct peer *listens;     /* "Meta" peers */
struct peer *gwpeer;        /* Gateway peer (if not a gateway) */
int gwcost;           /* 0=gw, -1=no gw, >0 = gw cost */

int checkroutes;        /* Routing update required */
int rtsocket;         /* Routing socket fd */
int udpport;          /* Default port number */
int isgateway;          /* Set if router is a gateway */
int gwalways;           /* Advertise gateway route if no default */
TIMETEN now;          /* Current time */
int routeflag;          /* Kernel route flag (RTF_PROTO<n>) */
int otherflag;          /* Additional route flags */
int overrflag;          /* Flags to override */
IPADDR defaultroute;        /* Default route to apply if none available */
IPADDR defgateway;      /* Current default gateway */

IPADDR routerid;        /* Router ID for paths */

char *statusfile;       /* File to dump status information to */
char *pidfile;          /* File to write PID into */
pid_t pid;         /* Our PID */

int debug;         /* Debug level: 0 = no messages */
#define LOG0 msg        /* unconditional */
#define LOG1 if(debug) msg  /* Topology changes, major events */
#define LOG2 if(debug >= 2) msg  /* Routing changes */
#define LOG3 if(debug >= 3) msg /* Exceptions */
#define LOG4 if(debug >= 4) dbg   /* Routing announcements */
#define LOG5 if(debug >= 5) dbg   /* Kernel routes added/deleted */
#define LOG6 if(debug >= 6) dbg   /* Message processing */
#define LOG7 if(debug >= 7) dbg   /* Packets sent/received */
#define LOG8 if(debug >= 8) dbg   /* Internal decisions */
```



```c
#define FOREACH(item, head) for((item) = (head); (item); (item) = (item)->next)
#define FREEROUTE(ipr) { (ipr)->next = freelist; freelist = (ipr); }
#define NEWROUTE(ipr) { if(((ipr) = freelist)) freelist = (ipr)->next;    \
        else (ipr) = malloc(sizeof(struct iproute));   }
#define ADDROUTEAFTER(ipr, table, after) {    NEWROUTE(ipr)    \
        if(after) { (ipr)->next = (after)->next;  \
            (after)->next = (ipr); } \
        else {      (ipr)->next = (table);       \
            (table) = (ipr); }      }
#define KILLROUTES(table) { struct iproute *ipr,  *nipr;           \
      for(ipr= (table); ipr; ipr = nipr) {        \
        nipr = ipr->next; FREEROUTE(ipr) }      \
      (table) = 0;                    }

#define NEXTSEQ(r) ( ((r) + 1) == 0 ? 1 : (r) + 1 )

char *formatip(IPADDR ip);
char *formattime(TIMETEN t);

struct iproute *findroute(struct iproute *ipr, IPADDR ip, int bits,
                 struct iproute **prev);
struct acl *findacl(char *name, int create);
int checkacl(struct acl *acl, IPADDR ip, int bits, int rtype, int ifindex);
void dumproutes(struct iproute *ipr);

int getroutes();

int parse_config(char *file);
void parse_acl_std(void);
char *parseip(char *s, IPADDR *ip_p, int *bits_p, int *maxbits_p);
char *getlocaladdr(struct peer *peer, IPADDR peerip, int listen);

void msg(char *fmt, ...);
void dbg(char *fmt, ...);

/* EOF */
```

```c
#include "frpd.h"
#include "frp.h"

/*
 Globals
 */
struct acl *acls = 0;       /* List header of named ACLs */

IPADDR maskbits[33];        /* Map prefix length to mask */

struct iproute *freelist = 0;   /* Free list of route objects */
struct iproute *localroutes = 0;   /* Local routing table header */

struct peer *peers = 0;         /* Running peers */
struct peer *listens = 0;     /* "Meta" peers */

int gwcost = -1;          /* 0 = we are a gateway;    */
                /* -1 = no gateway, 1-65534 = cost */
int rtsocket = -1;        /* Routing socket */

int udpport = 343;        /* Default port number */
int isgateway = 0;        /* Is a gateway */
int gwalways = 0;         /* Always a gateway */
IPADDR routerid = 0;        /* Router ID for paths */
struct peer *gwpeer = 0;    /* Gateway peer */
char *statusfile = 0;       /* Dump status infor to here */
int checkroutes = 0;        /* Routing update required */
TIMETEN now;              /* Current time, 100 ms increments */
int debug = 1;            /* Debug messages */
char *pidfile = 0;        /* Write PID here */

void
msg(char *fmt, ...) {
    va_list ap;
    va_start(ap, fmt);
    vsyslog(LOG_INFO, fmt, ap);
    va_end(ap);
}
void
dbg(char *fmt, ...) {
    va_list ap;
    va_start(ap, fmt);
    vsyslog(LOG_DEBUG, fmt, ap);
    va_end(ap);
}


/*
 Find a route in a routing table, sorted by ip, bits
 Update prev (if provided) to be previous routing entry, useful for adding
 a route if not found.
 */
struct iproute *
findroute(struct iproute *ipr, IPADDR ip, int bits, struct iproute **prev) {
    if(prev) *prev = 0;
    IPADDR ri;

    ip = ntohl(ip);
    FOREACH(ipr, ipr) {
        ri = ntohl(ipr->ip);
        if(ri > ip)
```



```c
            return 0;
        if(ip == ri) {
            if(bits == ipr->bits)
                return ipr;
            if(bits > ipr->bits)
                return 0;
        }
        if(prev) *prev = ipr;
    }
    return 0;
}


static void
dumproute(FILE *file, struct iproute *ipr) {
    char ib[IFNAMSIZ];
    fprintf(file, "%15s/%-2d %1s%1s %-15s %-8s rc %-5d gc %d\n",
        formatip(ipr->ip), ipr->bits,
        (ipr->rtype & RTYPE_LAYER2)  ? "2" :
        (ipr->rtype & RTYPE_STATIC)  ? "S" :
        (ipr->rtype & RTYPE_PROTOCOL) ? "P" :
        (ipr->rtype & RTYPE_LOCAL)   ? "L" : "R",
        ipr->isgw ? "G" : " ",
        ipr->peer ? formatip(REMIP(ipr->peer))  : "local",
        ipr->ifindex ? if_indextoname(ipr->ifindex,  ib) : "-",
        ipr->cost, ipr->gwcost);
}
void
dumproutes(struct iproute *ipr) {
    char ib[IFNAMSIZ];
    FOREACH(ipr, ipr)
        dbg("%s/%d %s%s -> %s iface %s rc %d gc %d",
        formatip(ipr->ip), ipr->bits,
        (ipr->rtype & RTYPE_LAYER2)  ? "2" :
        (ipr->rtype & RTYPE_STATIC)  ? "S" :
        (ipr->rtype & RTYPE_PROTOCOL) ? "P" :
        (ipr->rtype & RTYPE_LOCAL)   ? "L" : "R",
        ipr->isgw ? "G" : "",
        ipr->peer ? formatip(REMIP(ipr->peer))  : "local",
        ipr->ifindex ? if_indextoname(ipr->ifindex,  ib) : "-",
        ipr->cost, ipr->gwcost);
}


/*
 Dump internal state to a dump file
 */
static void
braindump(void) {
    FILE *file;
    struct peer *peer;
    struct iproute *ipr;
    int i;
    static char *statustemp = 0;

    /*
     Open status file. If it didn't work say so and don't try again
     */
    if(!statusfile) return;
    if(!statustemp) {
        statustemp = malloc(strlen(statusfile)  + 5);
```

```
        sprintf(statustemp, "%s.tmp", statusfile);
    }
    file = fopen(statustemp, "w");
    if(!file) {
        LOG1("Could not write %s: %m", statusfile);
        statusfile = 0;
        return;
    }

    /*
     Local status
     */
    fprintf(file, "RouterID %s Gateway %s GWCost %d%s\nPath",
            formatip(routerid),
            defgateway ? formatip(defgateway) : "unset",
            gwcost, isgateway ? " gateway" : "");
    if(isgateway)
        fprintf(file, " local\n");
    else if(gwcost != -1 && gwpeer) {
        for(i = 0; i < gwpeer->pathlen; i++)
            fprintf(file, " %s", formatip(gwpeer->path[i]));
        putc('\n', file);
    }
    else fprintf(file, " none\n");

    /*
     Listens
     */
    FOREACH(peer, listens)
        fprintf(file, "Listening on %s\n", formatip(LOCIP(peer)));

    /*
     Main routing table
     */
    fprintf(file, "\nRouting table:\n");
    FOREACH(ipr, localroutes)
        dumproute(file, ipr);

    /*
     Peers, including received and announced routes
     */
    FOREACH(peer, peers) {
        fprintf(file, "\nPeer %s Source %s NextHop %s\n",
                formatip(REMIP(peer)), formatip(LOCIP(peer)),
                formatip(peer->nexthop));
        fprintf(file, "Cost %d GWCost %d Poll %s Retry %s Fail %s"
                " Type %s Status %s\n",
                peer->cost, peer->gwcost,
                formattime(peer->poll),
                formattime(peer->retry),
                formattime(peer->fail),
                (peer->cloned) ? "dynamic" : "static",
                (peer == gwpeer) ? "up-gw" :
                (peer->synreq)   ? "down" : "up");
        fprintf(file, "Path");
        if(peer->gwcost == -1)
            fprintf(file, " none\n");
        else {
            for(i = 0; i < peer->pathlen; i++)
                fprintf(file, " %s", formatip(peer->path[i]));
            putc('\n', file);
```



```
        }
        fprintf(file, "Received routes:\n");
        FOREACH(ipr, peer->routes)
            dumproute(file, ipr);
        fprintf(file, "Announced routes:\n");
        FOREACH(ipr, peer->annrts)
            dumproute(file, ipr);
    }
    fclose(file);

    /*
     Done
     */
    if(rename(statustemp, statusfile) == -1) {
        LOG1("Could not write %s: %m", statusfile);
        statusfile = 0;
        unlink(statustemp);
    }
}


/*
 Dump a packet
 */
#define DFP_S(pkt,len,peer) if(debug >= 6) dp(1, pkt, len, peer)
#define DFP_R(pkt,len,peer) if(debug >= 6) dp(0, pkt, len, peer)
static void
dp(int outbound, void *p, int len, struct peer *peer) {
    u_char *ptr = p;
    u_char *end;
    struct frp_hdr *hdr;
    struct frp_mhdr *mh;
    struct frp_ctrl   *cml;
    struct frp_config *cmc;
    struct frp_path   *cmp;
    struct frp_route  *cmr;
    int i;
    char *t;
    char buf[16 * MAX_PATH];

    if(len == -1) {
        dbg("From: %s error: %m", formatip(REMIP(peer)));
        return;
    }

    end = ptr + len;
    hdr = (struct frp_hdr *) ptr;
    ptr += sizeof(struct frp_hdr);

    if(outbound)
        dbg("S %s -> %s lseq=%08x rseq=%08x len=%d",
              formatip(LOCIP(peer)), formatip(REMIP(peer)),
              ntohl(hdr->lseq), ntohl(hdr->rseq), len);
    else dbg("R %s <- %s rseq=%08x lseq=%08x",
              formatip(LOCIP(peer)), formatip(REMIP(peer)),
              ntohl(hdr->rseq), ntohl(hdr->lseq), len);

    while(ptr < end) {
        mh = (struct frp_mhdr *) ptr;
        ptr += mh->len * sizeof(u_int32_t);
        if(!mh->len) return;
```

```c
        if(ptr > end) return;

        switch(mh->type) {
        case FRP_CTRL:
            cml = (struct frp_ctrl *) mh;
            switch(cml->ctrl) {
            case FRP_CTRL_POLL:  t = "poll";    break;
            case FRP_CTRL_ACK:   t = "ack";     break;
            case FRP_CTRL_NAK:   t = "nak";     break;
            default:    t = "unknown";   break;
            }
            dbg("CTRL C=%d (%s) P=%x", cml->ctrl, t, cml->param);
            break;
        case FRP_CONFIG:
            cmc = (struct frp_config *) mh;
            dbg("CONFIG id=%s c=%d poll=%d fail=%d",
                formatip(cmc->id), ntohs(cmc->cost),
                ntohs(cmc->poll), ntohs(cmc->fail));
            break;
        case FRP_PATH:
            cmp = (struct frp_path *) mh;
            t = buf;
            for(i = 0; i < mh->len - 1; i++) {
                sprintf(t, "%s%s", i ? "," : "",
                    formatip(cmp->path[i]));
                t = strchr(t,0);
            }
            dbg("PATH gd=%d p=%s", ntohs(cmp->gwcost), buf);
            break;
        case FRP_ROUTE:
            cmr = (struct frp_route *) mh;
            if(cmr->flags & FRP_FLAG_NULL)
                dbg("ROUTE NULL flags=%x", cmr->flags);
            else dbg("ROUTE %s/%u c=%d gc=%d flags=%x (%s%s%s)",
                formatip(cmr->ip), cmr->bits,
                ntohs(cmr->cost), ntohs(cmr->gwcost),
                cmr->flags,
                cmr->flags & FRP_FLAG_GATEWAY ? "G" : "",
                cmr->flags & FRP_FLAG_BEGIN   ? "B" : "",
                cmr->flags & FRP_FLAG_COMMIT  ? "C" : "");
            break;
        case FRP_CONFIG6:
            dbg("CONFIG6 unsupported");
            break;
        case FRP_PATH6:
            dbg("PATH6 unsupported");
            break;
        case FRP_ROUTE6:
            dbg("ROUTE6 unsupported");
            break;
        default:
            dbg("UNKNOWN T=%02x L=%d", mh->type, mh->len * 4);
        }
    }
}


/*
 Process an ACL
 */
int
```



```c
checkacl(struct acl *acl, IPADDR ip, int bits, int rtype, int ifindex) {
    struct ace *ace;

    FOREACH(ace, acl->head) {
        if(ace->bits > 0 && (ip & maskbits[ace->bits]) != ace->ip)
            continue;
        if(bits != -1 && (bits < ace->bits || bits > ace->maxbits))
            continue;
        if(rtype && ace->rtype && !(rtype & ace->rtype))
            continue;
        if(ifindex && ace->ifindex && ace->ifindex != ifindex)
            continue;
        if(ace->acl && !checkacl(ace->acl, ip, bits, rtype, ifindex))
            continue;
        return ace->permit;
    }
    return 0;
}


/*
 Security stuff
 Hash calculation: SHA1 over packet, IP addresses & secret
 Note that only the first 64 bits (FRP_HASHZIE) of the hash are used.
 */
#include <sha.h>
static u_int8_t *
dohash(u_char *buf, int len, char *secret, IPADDR sa, IPADDR da) {
    static u_int8_t hash[SHA_DIGEST_LENGTH];
    SHA_CTX shctx;

    SHA_Init(&shctx);
    SHA_Update(&shctx, buf, len);
    SHA_Update(&shctx, (u_char *)&sa, sizeof(IPADDR));
    SHA_Update(&shctx, (u_char *)&da, sizeof(IPADDR));
    SHA_Update(&shctx, secret, strlen(secret));
    SHA_Final(hash, &shctx);
    return hash;
}

/*
 Check that hash matches packet hash.
 Do not call if peer->secret is null.
 */
static int
checksecure(u_char *pkt, int len, struct peer *peer) {
    u_int8_t *hash;

    hash = dohash(pkt + FRP_HASHSIZE, len - FRP_HASHSIZE, peer->secret,
                REMIP(peer), LOCIP(peer));
    if(!memcmp(hash, pkt, FRP_HASHSIZE))
        return 1;
    LOG2("packet security failure");
    return 0;
}

/*
 Compute packet hash
 Or zero hash field if secret is null
 */
static void
```

```c
secure(struct peer *peer, struct frp_hdr *pkt, int len) {
    u_int8_t *hash;

    if(!peer->secret) {
        memset(pkt->hash, 0, FRP_HASHSIZE);
        return;
    }
    hash = dohash(&pkt->hash[FRP_HASHSIZE], len-FRP_HASHSIZE, peer->secret,
                    LOCIP(peer), REMIP(peer));
    memcpy(pkt->hash, hash, FRP_HASHSIZE);
}


/*
 Return a random sequence number (that isn't 0)
 */
static u_int32_t
initseq(void) {
    static u_int32_t r = 0xbabefee1;
    int fd;
    struct timeval tv;

    do {
        fd = open("/dev/urandom", O_RDONLY);
        if(fd != -1) {
            read(fd, &r, sizeof(r));
            close(fd);
        }
        gettimeofday(&tv, 0);
        r = r ^routerid ^tv.tv_sec ^tv.tv_usec ^(getpid() << 16)
                      ^getppid();
    }
    while(r == 0);
    return r;
}


/*
 Send a SYN packet
 */
static void
sendsyn(struct peer *peer) {
    struct frp_hdr syn;

    syn.lseq = htonl(peer->lseq);
    syn.rseq = 0;
    secure(peer, &syn, sizeof(syn));
    DFP_S(&syn, sizeof(syn), peer);
    write(peer->fd, &syn, sizeof(syn));
    return;
}


/*
 Send a NAK
 */
static void
sendnak(struct peer *peer, int fd, u_int32_t lseq, u_int32_t rseq) {
    struct {
        struct frp_hdr hdr;
        struct frp_ctrl ctrl;
```



```c
    } nak;
    static TIMETEN nextnak = 0;

    /*
     Rate limit NAKs, stop NAK wars.
     */
    if(now < nextnak)
        return;
    nextnak = now + 10;

    /*
     Format & send a NAK
     */
    nak.hdr.lseq = htonl(ntohl(rseq) + 1);
    if(nak.hdr.lseq == 0)
        nak.hdr.lseq = htonl(1);
    nak.hdr.rseq = lseq;
    nak.ctrl.mh.type = FRP_CTRL;
    nak.ctrl.mh.len = 1;
    nak.ctrl.ctrl = FRP_CTRL_NAK;
    nak.ctrl.param = 0;
    secure(peer, &nak.hdr, sizeof(nak));
    DFP_S(&nak, sizeof(nak), peer);
    write(fd, &nak, sizeof(nak));
}


/*
 Turn everything off on a peer
 If "full" is 1, completely shut down the peer. If zero, just shut the
 session down, but don't reset routing state.
 Note that clearing out the tables of routes received and announced is
 done in the mainline (as it's common to deletion of both static and
 dynamic peers).
 */
static void
resetpeer(struct peer *peer, int full) {
    /*
     Reset stuff
     */
    peer->lastpktlen = 0;
    peer->nextpktlen = 0;
    peer->lseq      = initseq();
    peer->aseq      = 0;
    peer->rseq      = 0;
    peer->nextrt    = 0;
    peer->nullrt    = 0;
    peer->ackreq      = 0;
    peer->synreq      = 1;
    peer->configreq      = 0;
    peer->pathreq     = 0;
    peer->pollreq      = 0;
    peer->respreq     = 0;
    peer->sendack     = 0;
    peer->nexttime     = 0;
    peer->poll   = peer->confpoll;
    peer->fail   = peer->conffail;
    peer->cost      = peer->confcost;
    peer->retry     = peer->confretry;
    peer->quickstart  = 0;

```

```c
    /*
     If peer is being shut down, kill off the path and gw info, and
     reset all the timers.
     */
    if(full) {
        peer->lasttime   = 0;
        peer->shutdown    = 0;
        peer->routerid   = 0;
        peer->gwcost     = -1;
        peer->pathlen    = 0;
        checkroutes = 1;
    }
}


static void *
addmessage(struct peer *peer, int size, int type) {
    u_char *ptr;
    struct frp_mhdr *hdr;

    if(!peer->nextpacket) {
        peer->nextpacket = malloc(MAX_PACKET);
        peer->nextpktlen = 0;
    }
    if(!peer->nextpktlen)
        peer->nextpktlen = sizeof(struct frp_hdr);
    if(!size || peer->nextpktlen + size > MAX_PACKET)
        return 0;
    ptr = peer->nextpacket + peer->nextpktlen;
    peer->nextpktlen += size;
    peer->ackreq = 1;
    hdr = (struct frp_mhdr *) ptr;
    hdr->len  = size / sizeof(u_int32_t);
    hdr->type = type;
    return ptr;
}



#define DROP(msg) { LOG3("dropping packet: %s", msg); return; }

static void
process_packet(u_char *ptr, int len, struct peer *peer) {
    u_char *end;
    struct frp_hdr *hdr;
    struct frp_mhdr *mh;
    struct frp_ctrl    *cml;
    struct frp_config *cmc;
    struct frp_path   *cmp;
    struct frp_route  *cmr;
    int c;
    int i;
    struct frp_hdr ack;
    u_int32_t lseq, rseq, xseq;
    struct iproute *ipr;

    end = ptr + len;
    hdr = (struct frp_hdr *) ptr;
    lseq = ntohl(hdr->lseq);
    rseq = ntohl(hdr->rseq);
    ptr += sizeof(struct frp_hdr);
```



```c

    LOG8("from=%s:%d  to=%s:%d  syn=%d conf=%d path=%d"
            " lseq=%x  aseq=%x  rseq=%x  xseq=%x\n",
        formatip(REMIP(peer)),  ntohs(peer->rsa.sin_port),
        formatip(LOCIP(peer)),  ntohs(peer->lsa.sin_port),
        peer->synreq,  peer->configreq,  peer->pathreq,
        peer->lseq,  peer->aseq,  peer->rseq,  peer->xseq);

    if(!hdr->lseq)
        DROP("zero in lseq")

    /*
     If this is a synchronisation  packet, note the remote sequence
     number in xseq.  Drop any such packets with payload, as these are
     not to be trusted.
     If it's a kosher-looking  sync, send an ACK and return.  Note that
     state won't get updated until we get a data packet.
     */
    if(!hdr->rseq) {
        if(len != sizeof(struct frp_hdr))
            DROP("null rseq with payload")
        peer->xseq = NEXTSEQ(lseq);
        ack.lseq = htonl(peer->lseq);
        ack.rseq = htonl(lseq);
        secure(peer, &ack, sizeof(ack));
        DFP_S(&ack, sizeof(ack), peer);
        write(peer->fd,  &ack, sizeof(ack));
        return;
    }

    /*
     If this isn't a SYN packet (rseq != 0), check that both the
     local and remote sequence numbers are in the expected ranges
     */
    if(rseq != peer->lseq &&
       rseq != peer->aseq) {
        sendnak(peer, peer->fd, hdr->lseq, hdr->rseq);
        DROP("unexpected packet rseq")
    }
    xseq = peer->rseq;
    if(xseq) {
        for(i = 2; i > 0; i--) {
            if(lseq == xseq)
                break;
            xseq = NEXTSEQ(xseq);
        }
        if(!i && (!peer->xseq || peer->xseq != lseq))
            DROP("unexpected packet lseq")
    }

    /*
     If the packet acknowledges one of ours, update the acknowledged
     sequence.
     If it was in response to one of our syn requests, we need to send
     path and config details.
     */
    if(rseq == peer->lseq && peer->aseq != peer->lseq) {
        LOG8("ack received");
        peer->aseq = peer->lseq;
        peer->lasttime = now;
        peer->lastpktlen = 0;
```

```
        if(peer->synreq) {
            LOG6("%s: SYN acknowledged", formatip(REMIP(peer)));
            peer->synreq = 0;
            peer->configreq = 1;
            peer->pathreq = 1;
            peer->nextrt = 0;
            peer->nullrt = 0;
        }
    }

    /*
     If the remote sequence number is different from last time,
     force a response.
     If this is the result of a new connection (xseq != 0), send config
     and path info
     */
    xseq = peer->xseq;
    peer->xseq = 0;
    if(lseq != peer->rseq) {
        peer->sendack = 1;
        peer->rseq = lseq;
        LOG8("ACK flag set");
        if(xseq) {
            peer->synreq = 0;
            peer->configreq = 1;
            peer->pathreq = 1;
            peer->pollreq = 0;
            peer->nextrt = 0;
            peer->nullrt = 0;
        }
    }

    /*
     If it isn't, then we have a duplicate or a simple ack.
     If there is data, re-ack the packet; if there isn't, just drop it.
     */
    else {
        if(len > sizeof(struct frp_hdr))
            peer->sendack = 1;
        return;
    }

    while(ptr < end) {
        mh = (struct frp_mhdr *) ptr;

        ptr += mh->len * sizeof(u_int32_t);
        if(!mh->len) DROP("bad mh->len") /* Paranoia */
        if(ptr > end) DROP("mh->len past end") /* More paranoia */

        switch(mh->type) {
        case FRP_CTRL:
            cml = (struct frp_ctrl *) mh;
            if(cml->ctrl == FRP_CTRL_POLL)
                peer->respreq = 1;
            else if(cml->ctrl == FRP_CTRL_NAK) {
                LOG3("%s: NAK received for peer",
                    formatip(REMIP(peer)));
                resetpeer(peer, 0);
                break;
            }
            break;
```



```

        /*
         Configuration message -- agree on configuration
         */
        case FRP_CONFIG:
            cmc = (struct frp_config *) mh;
            LOG6("%s: router ID = %s cost = %d",
                formatip(REMIP(peer)),
                formatip(cmc->id), ntohs(cmc->cost));

            /*
             If the router-ID has changed, signal a routing re-run
             But if it's ours, drop this connection on the floor
             */
            if(peer->routerid != cmc->id) {
                if(cmc->id == routerid) {
                    peer->shutdown = 1;
                    DROP("duplicate router ID")
                }
                peer->routerid = cmc->id;
                checkroutes = 1;
            }

            /*
             Maximise the cost. If it changes, bump the routing
             */
            c = ntohs(cmc->cost);
            if(c < peer->confcost)
                c = peer->confcost;
            if(c != peer->cost) {
                peer->cost = c;
                checkroutes = 1;
            }

            /*
             Minimise the fail time. Minimum of one second to
             allow for poll and retry
             */
            c = ntohs(cmc->fail);
            if(c >= 10 && c < peer->conffail)
                peer->fail = c;
            else peer->fail = peer->conffail;

            /*
             Now the poll time
             */
            c = ntohs(cmc->poll);
            if(c >= 1 && c < peer->confpoll)
                peer->poll = c;
            else peer->poll = peer->confpoll;

            /*
             Compute the retry time to require three retries
             before giving up (min 1 second). Can have more by
             specifying a smaller retry time.
             */
            c = (peer->fail - peer->poll - 1) / 3;
            if(c < 1) c = 1;
            if(peer->confretry > c)
                peer->retry = c;
            else peer->retry = peer->confretry;
```

```
            /*
             If we are the smaller routerid, bump the poll
             time by one to avoid duelling polls
             */
            if(ntohl(routerid) < ntohl(peer->routerid))
                peer->poll++;

            LOG1("%s: Configured: id=%s cost=%d poll=%d"
                        " retry=%d fail=%d",
                formatip(REMIP(peer)),
                formatip(peer->routerid), peer->cost,
                peer->poll, peer->retry, peer->fail);
            break;

            /*
             Path message
             Copy the path and length
             */
        case FRP_PATH:
            cmp = (struct frp_path *) mh;
            c = mh->len - 1;
            if(c >= MAX_PATH) {
                peer->shutdown = 1;
                break;
            }
            peer->pathlen = c;
            for(i = 0; i < c; i++)
                peer->path[i] = cmp->path[i];

            c = ntohs(cmp->gwcost);
            if(c == FRP_NOGATEWAY)
                c = -1;
            if(peer->gwcost != c)
                peer->gwcost = c;
            checkroutes = 1;
            if(peer->annrts) {
                peer->nullrt = 0;
                peer->nextrt = peer->annrts;
            }
            else peer->nullrt = 1;
            break;

            /*
             Route message
             */
        case FRP_ROUTE:
            cmr = (struct frp_route *) mh;
            /*
             BEGIN indicates that we should scrub anything we
             have already
             */
            if(cmr->flags & FRP_FLAG_BEGIN)
                KILLROUTES(peer->newrts)

            /*
             Add the route, unless it's a null route
             */
            if(cmr->flags & FRP_FLAG_NULL) {
                LOG6("%s: Null route",formatip(REMIP(peer)));
            }
```



```
            else {
                NEWROUTE(ipr)
                ipr->ip = cmr->ip;
                ipr->bits = cmr->bits;
                ipr->isgw = !!(cmr->flags & FRP_FLAG_GATEWAY);
                ipr->cost = ntohs(cmr->cost);
                ipr->gwcost = ntohs(cmr->gwcost);
                ipr->peer = peer;
                ipr->ifindex = 0;
                ipr->rtype = 0;
                ipr->next = peer->newrts;
                peer->newrts = ipr;
                LOG6("%s: Route %s/%d flags=%x gw=%d c=%d"
                            " gc=%d",
                    formatip(REMIP(peer)),
                    formatip(cmr->ip), ipr->bits,
                    ipr->bits, ipr->isgw, ipr->cost,
                    ipr->gwcost);
            }

            /*
             COMMIT says the update is done, so delete the old
             current routes list, and move the new list to the
             current routes.
             Signal the change.
             */
            if(cmr->flags & FRP_FLAG_COMMIT) {
                KILLROUTES(peer->routes)
                peer->routes = peer->newrts;
                peer->newrts = 0;
                checkroutes = 1;
            }
            break;

            /*
             IPv6 messages are not supported
             */
        case FRP_CONFIG6:
        case FRP_PATH6:
        case FRP_ROUTE6:
            DROP("IPv6 not supported")

            /*
             Nor are unknown messages ...
             */
        default:
            DROP("bad msg type")
        }
    }
}



/*
 Connect or listen to a peer/listen
 Return fd on success, -1 on fail
 */
static int
connectpeer(struct peer *peer, int doconnect) {
    int fd;
    int i;
```

```c

    fd = socket(PF_INET, SOCK_DGRAM, 0);
    if(fd < 0)
        goto oops;
    i = 1;
    if(setsockopt(fd, SOL_SOCKET, SO_REUSEPORT, &i, sizeof(int)))
        goto oops;
    if(setsockopt(fd, IPPROTO_IP, IP_TTL, &peer->ttl, sizeof(int)))
        goto oops;
    LOG8("bind %s:%d", formatip(LOCIP(peer)), ntohs(peer->lsa.sin_port));
    if(bind(fd, (struct sockaddr *)&peer->lsa, sizeof(struct sockaddr_in)))
        goto oops;
    if(doconnect) {
        LOG8("connect %s:%d", formatip(REMIP(peer)),
                ntohs(peer->rsa.sin_port));
        if(connect(fd, (struct sockaddr *) &peer->rsa,
                sizeof(struct sockaddr_in)))
            goto oops;
    }
    return fd;

oops:   LOG1("Could not %s to %s: %m", doconnect ? "connect to" : "listen on",
             formatip(doconnect ? REMIP(peer) : LOCIP(peer)));
    if(fd != -1)
        close(fd);
    return -1;
}



static int restart = 0;
static int shutdwn = 0;
static int exitnow = 0;

static void
ouch(int sig) {
    switch(sig) {
    case SIGUSR1:   if(debug < 8)
                LOG0("Debug level raised to %d", ++debug);
            break;
    case SIGUSR2:   if(debug > 0)
                LOG1("Debug level lowered to %d", --debug);
            break;
    case SIGHUP:    exitnow = restart = 1; /* Restart program */
            LOG1("SIGHUP received, restarting ...");
            break;
    case SIGTERM:   exitnow = shutdwn = 1;
            LOG1("SIGTERM received, full shutdown");
            break;
    case SIGINT:  exitnow = 1;
            LOG1("SIGINT received, shutting down");
            break;
    }
}

int
main(int argc, char **argv)  {
    int i, j;
    socklen_t sal;
    fd_set fds;
    int fdc;
```



```c
    struct timeval tv;
    u_char buf[65536];
    struct rt_msghdr *rtm;
    struct frp_hdr *hdr;
    TIMETEN lasttime;
    int len;
    struct peer *peer, *lastpeer, *nextpeer;
    struct peer *listen;
    struct frp_ctrl   *cml;
    struct frp_config *cmc;
    struct frp_path   *cmp;
    struct frp_route  *cmr;
    struct iproute *ipr, *pipr, *nipr;
    u_int32_t rseq;
    int pfd;
    FILE *pf;
    extern char *optarg;
    char *configfile;
    int foreground;
    int debuglvl;
    int forceexit;
    time_t basetime;

    /*
     Parse arguments
     */
    configfile = "frpd.conf";
    foreground = 0;
    debuglvl = -1;
    forceexit = 0;
    while((i = getopt(argc, argv, "c:d:fp:xX")) != -1) switch(i) {
    case 'c':   configfile = optarg;      break;
    case 'd':   debuglvl = atoi(optarg); break;
    case 'f':   foreground = 1;        break;
    case 'p':   pidfile = optarg;       break;
    case 'x':   forceexit = SIGINT;      break;
    case 'X':   forceexit = SIGTERM;       break;
    default: return 1;
    }

    /*
     Compute mask bits array
     */
    for(i = 0; i <= 32; i++)
        maskbits[i] = htonl(0xffffffff << (32 - i));
    parse_acl_std();

    /*
     If we were asked to kill the running process, do so now.
     */
    if(forceexit) {
        if(!pidfile)
            parse_config(configfile);
        if(!pidfile) {
            fprintf(stderr, "No PID file defined\n");
            return 1;
        }
        pf = fopen(pidfile, "r");
        if(!pf) {
            fprintf(stderr, "Could not open PID file %s: %s\n",
                  pidfile, strerror(errno));
```

```
            return 0;
        }
        i = -1;
        fscanf(pf, "%d", &i);

        if(i > 1 && flock(fileno(pf), LOCK_EX | LOCK_NB) == -1
                && errno == EWOULDBLOCK) {
            fclose(pf);
            if(kill(i, forceexit) == 0)
                return 0;
            fprintf(stderr, "Could not kill PID %d: %s\n",
                    i, strerror(errno));
        }
        flock(fileno(pf), LOCK_UN);
        fclose(pf);
        fprintf(stderr, "Existing daemon not running\n");
        return 1;
    }

    /*
     Parse the configuration
     */
    if(!parse_config(configfile))
        return 1;
    if(!peers && !listens) {
        fprintf(stderr, "No peers defined\n");
        return 1;
    }
    if(debuglvl >= 0 && debuglvl <= 8)
        debug = debuglvl;

    /*
     Go into the background
     */
    if(!foreground)
        daemon(1, 1);
    pid = getpid();

    /*
     Open the routing socket
     Just listen to IPv4 routing changes
     Don't bother with our own messages (like, we probably know)
     */
    rtsocket = socket(PF_ROUTE, SOCK_RAW, AF_INET);
    if(rtsocket == -1) {
        perror("routing socket");
        return 1;
    }
    i = 0;
    if(setsockopt(rtsocket, SOL_SOCKET, SO_USELOOPBACK, &i, sizeof(i)) <0){
        perror("setsockopt");
        return 1;
    }

    /*
     Set up syslog
     */
    openlog("frpd", (foreground ? LOG_PERROR : 0), LOG_DAEMON);

    /*
     Set up signal handler
```



```
     */
    signal(SIGHUP, ouch);
    signal(SIGINT, ouch);
    signal(SIGTERM, ouch);
    signal(SIGUSR1, ouch);
    signal(SIGUSR2, ouch);

    /*
     Write the PID file
     Open it read/write
     If we could, see if it's locked.
     If it is, read the pid, and knock the existing process on the head
     If any of that failed, bail, otherwise try the whole procedure again.
     If we got the lock, just overwrite the PID with ours (keeping the lock)
     If the file was not there, lock it and write our PID.
     Note that when we exit, we always unlink the file before dropping
     the lock.
     */
    pf = 0;
    if(pidfile) {
        while((pf = fopen(pidfile, "r+"))) {
            if(flock(fileno(pf), LOCK_EX | LOCK_NB) == -1
                    && errno == EWOULDBLOCK) {
                i = -1;
                fscanf(pf, "%d", &i);
                if(i > 1) {
                    LOG1("Killing existing process %d", i);
                    if(kill(i, SIGINT) == 0) {
                        flock(fileno(pf), LOCK_EX);
                        flock(fileno(pf), LOCK_UN);
                        fclose(pf);
                        sleep(1);
                        pf = 0;
                        continue;
                    }
                }
                LOG0("Could not lock PID file");
                return 1;
            }
            rewind(pf);
            fprintf(pf, "%d\n", (int) pid);
            fflush(pf);
            break;
        }
        if(!pf) {
            pf = fopen(pidfile, "w");
            if(pf) {
                flock(fileno(pf), LOCK_EX);
                fprintf(pf, "%d\n", (int) pid);
                fflush(pf);
            }
            else LOG1("Could not write PID file %s: %m",
                        pidfile);
        }
    }

    /*
     Initialise time.
     */
    gettimeofday(&tv, 0);
    basetime = tv.tv_sec;
```

```
    now = tv.tv_usec / 100000;


    /*
     Initialise the peers & listens. If anything failed, drop them
     on the floor. And stamp on them.
     */
    lastpeer = nextpeer = 0;
    for(peer = peers; peer; peer = nextpeer) {
        nextpeer= peer->next;
        resetpeer(peer, 1);
        peer->fd = connectpeer(peer, 1);
        if(peer->fd == -1) {
            if(lastpeer)
                lastpeer->next = nextpeer;
            else peers = nextpeer;
            free(peer);
        }
        else lastpeer = peer;
    }
    for(peer = listens; peer; peer = nextpeer) {
        nextpeer = peer->next;
        peer->fd = connectpeer(peer, 0);
        if(peer->fd == -1) {
            if(lastpeer)
                lastpeer->next = nextpeer;
            else listens = nextpeer;
            free(peer);
        }
        else lastpeer = peer;
    }
    if(!peers && !listens) {
        LOG1("No active peers/listens");
        exitnow = 1;
    }

    /*
     Set the router ID, it it hasn't been manually set
     */
    if(!routerid) {
        if(listens)
            routerid = LOCIP(listens);
        else if(peers)
            routerid = LOCIP(peers);
    }
    LOG1("Starting, routerid=%s, PID=%d", formatip(routerid), pid);

    checkroutes = 1;
    lasttime = 0;
    while(!exitnow) {
        /*
         Rate-limit this
         */
        if(now != lasttime) {
            lasttime = now;

            /*
             Time related protocol stuff with each peer:
             - Timeouts;
             - Startup polls;
             - Retransmission;
```



```
             - Regular keepalives.
             */
            FOREACH(peer, peers) {
                /*
                 If the peer has timed out, shut it dowm
                 */
                if(peer->lasttime &&
                   now >= peer->lasttime + peer->fail) {
                    peer->shutdown = 1;
                    continue;
                }

                /*
                 Only do the protocol retries if nexttime
                 has been reached
                 */
                if(now < peer->nexttime)
                    continue;

                /*
                 If we need to send a SYN packet, do so
                 now.
                 */
                if(peer->synreq) {
                    sendsyn(peer);
                    if(peer->quickstart < peer->poll)
                        peer->nexttime = now
                            + (++peer->quickstart);
                    else peer->nexttime = now
                            + peer->poll;
                }

                /*
                 Otherwise, if we need to retransmit an
                 un-acknowledged packet, do so now.
                 Note that if rseq has advanced since our
                 last retry, we need to update (and re-secure)
                 the packet accordingly to acknowledge the
                 received data.
                 */
                else if(peer->lseq != peer->aseq
                        && peer->lastpktlen) {
                    DFP_S(peer->lastpacket,peer->lastpktlen,
                                peer);
                    hdr = (struct frp_hdr *)
                            peer->lastpacket;
                    rseq = ntohl(peer->rseq);
                    if(hdr->rseq != rseq) {
                        hdr->rseq = rseq;
                        secure(peer, hdr,
                                 peer->lastpktlen);
                    }
                    write(peer->fd, peer->lastpacket,
                            peer->lastpktlen);
                    peer->nexttime = now + peer->retry;
                }

                /*
                 If we've reached time to poll, do so
                 */
                else if(now >= peer->lasttime + peer->poll)
```

```
                peer->pollreq = 1;
            }

            /*
             Check the routing table for changes. Only do this
             once per second to allow things to settle.
             Run this again if the gateway status changed and
             we are a gateway.
             If it failed prematurely, getroutes() returns
             1, else 0, which sets checkroutes appropriately.
             */
            if(checkroutes) {
                i = gwcost;
                checkroutes = getroutes();
                if(isgateway && i != gwcost)
                    checkroutes = getroutes();
                if(statusfile)
                    braindump();
            }
        }

        /*
         Shut down requested peers
         Return to pristine state; if cloned, delete completely
         */
        lastpeer = nextpeer = 0;
        for(peer = peers; peer; peer = nextpeer) {
            nextpeer = peer->next;
            if(peer->shutdown) {
                LOG1("Peer %s shut down",
                        formatip(REMIP(peer)));
                /*
                 Free route records
                 */
                KILLROUTES(peer->routes);
                KILLROUTES(peer->newrts);
                KILLROUTES(peer->annrts);
                checkroutes = 2;

                /*
                 If it's been shut down, and it's a cloned
                 peer, kill it off completely.
                 Make sure we don't have any dangling pointers
                 either in gwpeer or in the routing table
                 */
                if(peer->cloned) {
                    LOG8("Cloned peer deleted");
                    close(peer->fd);
                    if(peer == gwpeer)
                        gwpeer = 0;
                    pipr = nipr = 0;
                    for(ipr=localroutes; ipr; ipr = nipr) {
                        nipr = ipr->next;
                        if(ipr->peer == peer) {
                            if(pipr)
                                pipr->next = nipr;
                            else localroutes = nipr;
                            FREEROUTE(ipr)
                        }
                        else pipr = ipr;
                    }
```



```
                    if(lastpeer)
                        lastpeer->next = nextpeer;
                    else peers = nextpeer;
                    if(peer->lastpacket)
                        free(peer->lastpacket);
                    if(peer->nextpacket)
                        free(peer->nextpacket);
                    free(peer);
                    continue;
                }

                /*
                 Otherwise just clean things out
                 */
                resetpeer(peer, 1);
            }
            lastpeer = peer;
        }

        /*
         Process outbound packets
         */
        FOREACH(peer, peers) {
            /*
             Don't bother with any of this if we're still waiting
             for our last packet
             */
            if(peer->lseq != peer->aseq || !peer->rseq
                    || peer->synreq)
                continue;

            /*
             If we need to send a config packet, do so now;
             */
            if(peer->configreq) {
                cmc = addmessage(peer, sizeof(*cmc),
                            FRP_CONFIG);
                if(cmc) {
                    cmc->cost = htons(peer->cost);
                    cmc->id   = routerid;
                    cmc->poll = htons(peer->confpoll);
                    cmc->fail = htons(peer->conffail);
                    peer->configreq = 0;
                    LOG6("%s: Added config",
                        formatip(REMIP(peer)));
                }
            }

            /*
             Ditto the path packet
             */
            if(peer->pathreq) {
                if(gwpeer)
                    j = gwpeer->pathlen;
                else j = 0;

                cmp = addmessage(peer, sizeof(*cmp)
                            + sizeof(IPADDR) * (j+1),
                            FRP_PATH);
                if(cmp) {
                    cmp->gwcost = htons(gwcost);
```

```
                cmp->path[0] = routerid;
                for(i = 0; i < j; i++)
                    cmp->path[i+1] = gwpeer->path[i];
                peer->pathreq = 0;
                LOG6("%s: Added path gwcost=%d",
                  formatip(REMIP(peer)),
                  gwcost);
            }
        }

        /*
         And if we're to send some routing info, do that
         now too
         */
        while(peer->nextrt) {
            cmr = addmessage(peer,sizeof(*cmr), FRP_ROUTE);
            if(!cmr) break;
            ipr = peer->nextrt;
            cmr->flags = ((ipr == peer->annrts) ?
                    FRP_FLAG_BEGIN   : 0)
                  | ((ipr->next == 0) ?
                    FRP_FLAG_COMMIT  : 0)
                  | ((ipr->isgw) ?
                    FRP_FLAG_GATEWAY : 0);
            cmr->bits   = ipr->bits;
            cmr->cost   = htons(ipr->cost);
            cmr->gwcost = htons(ipr->gwcost);
            cmr->ip     = ipr->ip;
            peer->nextrt = ipr->next;
                LOG6("%s: Added route %s/%d",
                  formatip(REMIP(peer)),
                  formatip(ipr->ip), ipr->bits);
        }

        /*
         If we have to send a null route list, do so now
         */
        if(peer->nullrt) {
            cmr = addmessage(peer, FRP_NULLRT_SIZE,
                      FRP_ROUTE);
            if(cmr) {
                cmr->flags = FRP_FLAG_NULL
                    | FRP_FLAG_BEGIN
                    | FRP_FLAG_COMMIT;
                cmr->bits = 0;
                peer->nullrt = 0;
                LOG6("%s: Added null route list",
                  formatip(REMIP(peer)));
            }
        }

        /*
         Send a poll request if required
         */
        if(peer->pollreq) {
            cml = addmessage(peer, sizeof(struct frp_ctrl),
                      FRP_CTRL);
            if(cml) {
                cml->ctrl  = FRP_CTRL_POLL;
                cml->param = 0;
                peer->pollreq = 0;
```



```
            }
        }

        /*
         If we need to respond to a poll request, do so.
         Don't worry if there's no room in the packet.
         */
        if(peer->respreq) {
            cml = addmessage(peer, sizeof(struct frp_ctrl),
                      FRP_CTRL);
            if(cml) {
                cml->ctrl  = FRP_CTRL_ACK;
                cml->param = 0;
            }
            peer->respreq = 0;
        }

        /*
         Finally, if we haven't already ACKed an inbound
         packet, do so now
         */
        if(peer->sendack) {
            addmessage(peer, 0, 0);
            peer->sendack = 0;
        }

        /*
         If we have an outbound packet, and we're not still
         waiting for the last one to be acknowledged, send
         the packet now
         */
        if(peer->nextpktlen) {
            if(peer->ackreq) {
                peer->lseq = NEXTSEQ(peer->lseq);
                peer->ackreq = 0;
            }
            hdr = (struct frp_hdr *)peer->nextpacket;
            hdr->lseq = htonl(peer->lseq);
            hdr->rseq = htonl(peer->rseq);
            secure(peer, hdr, peer->nextpktlen);
            DFP_S(hdr, peer->nextpktlen, peer);
            write(peer->fd, hdr, peer->nextpktlen);
            peer->nextpacket = peer->lastpacket;
            peer->lastpacket = (u_char *) hdr;
            peer->lastpktlen = peer->nextpktlen;
            peer->nextpktlen = 0;
            peer->nexttime = now + peer->retry;
        }
    }


    fdc = rtsocket + 1;
    FD_ZERO(&fds);
    FD_SET(rtsocket, &fds);

    FOREACH(peer, listens) {
        if(peer->fd >= fdc)
            fdc = peer->fd + 1;
        FD_SET(peer->fd, &fds);
    }
    i = 10;
```

```
    if(checkroutes)
        i = 1;
    FOREACH(peer, peers) {
        if(peer->fd >= fdc)
            fdc = peer->fd + 1;
        FD_SET(peer->fd, &fds);
        if(peer->lseq != peer->aseq) {
            j = peer->nexttime - now;
            if(j >= 0 && j < i)
                i = j;
        }
        else {
            j = peer->lasttime + peer->poll - now;
            if(j >= 0 && j < i)
                i = j;
        }
        LOG8("now=%d nexttime=%d lasttime=%d poll=%d"
            " retry=%d fail=%d i=%d",
            now, peer->nexttime, peer->lasttime,
            peer->poll, peer->retry, peer->fail, i);
    }
    LOG8("Waiting %s seconds", formattime(i));
    tv.tv_sec = i / 10;
    tv.tv_usec = (i % 10) * 100000;
    if(!i) tv.tv_usec = 20000;
    i = select(fdc, &fds, 0, 0, &tv);

    gettimeofday(&tv, 0);
    now = tv.tv_sec - basetime;
    now = now * 10 + tv.tv_usec / 100000;

    if(i <= 0) {
        if(i == 0 || errno == EINTR)
            continue;
        perror("select");
        return 1;
    }

    /*
     If we got a routing socket message, read it and see if it's
     interesting. Adds, changes & deletes might interest us,
     but only if they don't involved cloned routes (ARP
     activity), broadcasts, multicasts, ICMP
     redirects/unreachables etc
     */
    if(FD_ISSET(rtsocket, &fds)) {
        len = read(rtsocket, buf, sizeof(buf));
        rtm = (struct rt_msghdr *) buf;
        if(!checkroutes && len >= sizeof(struct rt_msghdr)
           && ( rtm->rtm_type == RTM_ADD
            || rtm->rtm_type == RTM_DELETE
            || rtm->rtm_type == RTM_CHANGE )
           && !( rtm->rtm_flags & ( RTF_REJECT
                    | RTF_DYNAMIC
                    | RTF_MODIFIED
                    | RTF_WASCLONED
                    | RTF_BROADCAST
                    | RTF_MULTICAST )))
            checkroutes = 1;
        LOG8("Route message type %d flags %x check=%d",
            rtm->rtm_type, rtm->rtm_flags, checkroutes);
```



```
    }

    /*
     See if we got any unsolicited requests on any of the listen
     sockets.
     */
    FOREACH(listen, listens) if(FD_ISSET(listen->fd, &fds)) {
        sal = sizeof(struct sockaddr_in);
        len = recvfrom(listen->fd, buf, sizeof(buf), 0,
                (struct sockaddr *) &listen->rsa, &sal);
        DFP_R(buf, len, listen);

        /*
         Check the remote ACL (if there is one). Just drop
         the packet if so
         */
        if(listen->remote) {
            if(!checkacl(listen->remote,
                    REMIP(listen), -1, 0, 0)) {
                LOG2("rejected packet from %s\n",
                    formatip(REMIP(listen)));
                continue;
            }
        }

        /*
         Check if the packet has the right secret
         */
        if(listen->secret && !checksecure(buf, len, listen))
            continue;

        /*
         Having got this far, establish a new connection.
         */
        pfd = connectpeer(listen, 1);
        if(pfd == -1)
            continue;

        /*
         If this is not a SYN packet (rseq == 0), NAK it
         and drop the connection. Typically, this happens
         if we have restarted, and the other end thinks it
         has a session open. By giving it a sensible looking
         NAK, we tell it that it should drop the connection and
         restart from scratch.
         */
        hdr = (struct frp_hdr *) buf;
        if(len != sizeof(struct frp_hdr) || hdr->rseq != 0) {
            LOG6("NAK %s", formatip(REMIP(listen)));
            sendnak(listen, pfd, hdr->lseq, hdr->rseq);
            close(pfd);
            continue;
        }

        /*
         We're happy at this point. Create a new peer
         object, copy the template listen object onto
         it, mark it as cloned (so we know to get rid of
         it on shutdown), do the initialisation and link
         it to the list of peers.
         */
```

```
            peer = malloc(sizeof(struct peer));
            *peer = *listen;
            peer->next = peers;
            peers = peer;

            getlocaladdr(peer, REMIP(listen), 0);
            peer->lsa       = listen->lsa;
            peer->rsa       = listen->rsa;
            if(!peer->nexthop)
                peer->nexthop = REMIP(peer);
            peer->cloned   = 1;
            peer->fd       = pfd;
            peer->shutdown = 0;
            resetpeer(peer, 1);

            /*
             Process the received packet
             */
            LOG1("Connect from %s established",
                        formatip(REMIP(peer)));
            process_packet(buf, len, peer);
        }

        /*
         See if we got any packets on established peers ...
         Check basic sanity.  If it's sane, and it is encoded with
         the right secret, process it.
         */
        FOREACH(peer, peers) if(FD_ISSET(peer->fd, &fds)) {
            len = read(peer->fd, buf, sizeof(buf));
            DFP_R(buf, len, peer);

            if(len < (int) sizeof(struct frp_hdr)) {
                if(len != -1)
                    LOG3("%s: short packet",
                        formatip(REMIP(peer)));
                continue;
            }
            if(!peer->secret || checksecure(buf, len, peer))
                process_packet(buf, len, peer);
        }
    }

    /*
     If shutdown request, do brutal shutdown stuff
     Basically,  stuff up the peers list etc so that the routing code
     thinks there are no peers and therefore no default route, so
     all routes get deleted.
     */
    if(shutdwn) {
        peers = 0;
        getroutes();
    }

    /*
     Last gasp packets
     Send a NAK packet to every active peer.  This should cause the
     peer to restart.
     */
    FOREACH(peer, peers) if(!peer->synreq)
        sendnak(peer, peer->fd, htonl(peer->rseq), htonl(peer->lseq));
```



```

    /*
     If there's a PID file, drop it now
     Ditto the status file
     */
    if(pf) {
        unlink(pidfile);
        flock(fileno(pf), LOCK_UN);
        fclose(pf);
    }
    if(statusfile)
        unlink(statusfile);

    /*
     If we got asked to restart,  have a go at staring ourselevs
     */
    if(restart)
        execv(argv[0], argv);

    return 0;
}

/* EOF */
```

```c
#include "frpd.h"


static char *secret = 0;



static char *
gettmp(int size) {
    static char buf[64];
    static char *ptr = buf;
    char *s;
    if(ptr + size >= buf + sizeof(buf))
        ptr = buf;
    s = ptr;
    ptr += size;
    return s;
}


char *
formatip(IPADDR ip) {
    union { u_int8_t b[4]; IPADDR a; } ipa;
    char *s;

    s = gettmp(16);
    ipa.a = ip;
    sprintf(s, "%u.%u.%u.%u", ipa.b[0], ipa.b[1], ipa.b[2], ipa.b[3]);
    return s;
}

char *
formattime(TIMETEN t) {
    char *s = gettmp(16);
    int d = t % 10;
    sprintf(s, "%u%c%c", t / 10, d ? '.' : 0, d + '0');
    return s;
}

static char *
parsetime(char *s, TIMETEN *tp) {
    TIMETEN t;
    char *p;

    t = strtol(s, &p, 10);
    if(t > 1000000)
        return "Invalid interval";
    t *= 10;
    if(*p == '.' && isdigit(p[1]) && p[2] == 0)
        t += p[1] - '0';
    else if(p == s || *p != 0)
        return "Invalid interval";
    *tp = t;
    return 0;
}


char *
parseip(char *s, IPADDR *ip_p, int *bits_p, int *maxbits_p) {
    int bits;
    int maxbits;
```



```c
    union { u_int8_t b[4]; IPADDR a; } ipa;
    int i, j;
    char *t;

    ipa.a = 0;
    for(i = 0; i < 4; i++) {
        j = strtol(s, &t, 10);
        if(t == s) return "Invalid IP address";
        s = t + 1;
        ipa.b[i] = (unsigned) j;
        if(*t != '.') break;
    }

    if(*t == '/') {
        if(!bits_p) return "Invalid IP address";
        bits = strtol(s, &t, 10);
        if(s == t || bits < 0 || bits > 32)
            return "Invalid prefix length";
        if(*t == '-') {
            if(!maxbits_p) return "Invalid prefix length";
            s = t + 1;
            maxbits = strtol(s, &t, 10);
            if(s == t || maxbits < bits || maxbits > 32)
                return "Invalid prefix range";
        }
        else maxbits = bits;
    }
    else maxbits = bits = 32;
    if(*t)   return "Invalid IP address";
    if((ipa.a & maskbits[bits]) != ipa.a)
        return "IP address mask mismatch";
    *ip_p = ipa.a;
    if(bits_p) *bits_p = bits;
    if(maxbits_p)   *maxbits_p = maxbits;
    return 0;
}


struct acl *
findacl(char *name, int create) {
    struct acl *acl;
    FOREACH(acl, acls)
        if(!strcmp(acl->name, name))
            return acl;
    if(!create) return 0;
    acl = calloc(1, sizeof(struct acl));
    acl->next = acls;
    acls = acl;
    acl->name = strdup(name);
    return acl;
}


static char *
parse_acl(char *name, char *expression) {
    struct acl *acl;
    struct ace *ace, proto;
    char *s;
    char buf[4096];

    if(expression) {
```

```
        strcpy(buf, expression);
        s = strtok(buf, WHITESPACE);
    }
    else s = strtok(0, WHITESPACE);

    if(!s) return "Expected 'permit' or 'deny'";

    memset(&proto, 0, sizeof(proto));
    if(!strcmp(s, "permit"))
        proto.permit = 1;
    else if(strcmp(s, "deny"))
        return "Expected 'permit' or 'deny'";
    proto.maxbits = 32;

    while((s = strtok(0, WHITESPACE))) {
        if(!strcmp(s, "ip")) {   // n.n.n.n[/bits[-mbits]]
            s = strtok(0, WHITESPACE);
            if(!s) return "Expected IP prefix";
            s = parseip(s, &proto.ip, &proto.bits, &proto.maxbits);
            if(s) return s;
        }
        else if(!strcmp(s, "default")) {
            proto.ip = 0;
            proto.maxbits = proto.bits = 0;
        }
        else if(!strcmp(s, "interface")) {
            s = strtok(0, WHITESPACE);
            if(!s) return "Expected interface name";
            proto.ifindex = if_nametoindex(s);
            if(!proto.ifindex)
                return "Unknown interface";
        }
        else if(!strcmp(s, "acl")) {
            s = strtok(0, WHITESPACE);
            if(!s) return "Expected ACL name";
            proto.acl = findacl(s, 0);
            if(!proto.acl) return "ACL not found";
        }
        else if(!strcmp(s, "layer2"))
            proto.rtype |= RTYPE_LAYER2;
        else if(!strcmp(s, "static"))
            proto.rtype |= RTYPE_STATIC;
        else if(!strcmp(s, "local"))
            proto.rtype |= RTYPE_LOCAL;
        else if(!strcmp(s, "protocol"))
            proto.rtype |= RTYPE_PROTOCOL;
        else if(!strcmp(s, "remote"))
            proto.rtype |= RTYPE_REMOTE;

        else return "Unrecognised keyword";
    }

    acl = findacl(name, 1);
    ace = malloc(sizeof(struct ace));
    *ace = proto;
    if(!acl->head)
        acl->head = ace;
    else acl->tail->next = ace;
    acl->tail = ace;

    return 0;
```



```
}


/*
static void
dumpacl(char *name) {
    struct acl *acl;
    struct ace *ace;
    union { u_int8_t b[4]; IPADDR a; } ipa;

    acl = findacl(name, 0);
    if(!acl) {
        printf("No such ACL %s\n", name);
        return;
    }
    FOREACH(ace, acl->head) {
        ipa.a = ace->ip;
        printf("%-6s %u.%u.%u.%u/%d-%d rtype=%x, acl=%s\n",
            ace->permit ? "permit" : "deny",
            ipa.b[0], ipa.b[1], ipa.b[2], ipa.b[3],
            ace->bits, ace->maxbits, ace->rtype,
            ace->acl ? ace->acl->name : "(none)");
    }
}
*/


/*
 Find interface information associated with IP address
 */
char *
getlocaladdr(struct peer *peer, IPADDR peerip, int listen) {
    static struct ifaddrs *ifphdr = 0;
    struct ifaddrs *ifp;
    IPADDR a, m;
    int b;
    IPADDR localaddr;
    static time_t lasttime = 0;
    time_t t;

    localaddr = 0;
    peer->localbits = 0;

    t = time(0);
    if(ifphdr && t != lasttime) {
        freeifaddrs(ifphdr);
        ifphdr = 0;
    }
    if(!ifphdr) if(getifaddrs(&ifphdr)) {
        LOG1("Error getting interface addresses: %m");
        return "Could not obtain interface addresses";
    }
    lasttime = t;

    for(ifp = ifphdr; ifp; ifp = ifp->ifa_next)
            if(ifp->ifa_addr->sa_family == AF_INET) {
        a = ((struct sockaddr_in *)(ifp->ifa_addr))->sin_addr.s_addr;
        m = ((struct sockaddr_in *)(ifp->ifa_netmask))->sin_addr.s_addr;
        b = (m & 0x55555555) + ((m >> 1) & 0x55555555);
        b = (b & 0x33333333) + ((b >> 2) & 0x33333333);
        b = (b & 0x0f0f0f0f) + ((b >> 4) & 0x0f0f0f0f);
```

```c
        b = (b & 0x00ff00ff) + ((b >> 8) & 0x00ff00ff);
        b = (b & 0x0000ffff) + ((b >>16) & 0x0000ffff);
        if((listen && a == peerip) ||
           (!listen && (peerip & m) == (a & m)
               && b > peer->localbits) ||
           (!listen && peerip ==
              ((struct sockaddr_in *)(ifp->ifa_dstaddr))->
                       sin_addr.s_addr)) {
            localaddr = a;
            peer->localroute = peerip & m;
            peer->localbits = b;
            peer->ifindex = if_nametoindex(ifp->ifa_name);
        }
    }
    if(!localaddr && listen)
        return "Interface address not local";
    else if(!localaddr)
        return "IP address is not directly connected";

    if(listen)
        LOCIP(peer) = peerip;
    else {
        REMIP(peer) = peerip;
        LOCIP(peer) = localaddr;
    }
    return 0;
}



static char *
addpeer(IPADDR peerip, int listen) {
    char *s;
    struct peer *peer;
    int p;
    int new;

    if(listen) {
        FOREACH(peer, listens)
            if(LOCIP(peer) == peerip)
                break;
    }
    else {
        FOREACH(peer, peers)
            if(REMIP(peer) == peerip)
                break;
    }
    if(!peer) {
        new = 1;
        peer = calloc(1, sizeof(struct peer));
        peer->lsa.sin_family = AF_INET;
        peer->lsa.sin_port = htons(udpport);
        peer->rsa.sin_family = AF_INET;
        peer->rsa.sin_port = htons(udpport);

        if(listen)
            peer->nexthop = 0;
        else peer->nexthop = peerip;
        peer->ttl = 1;
        peer->secret = secret;
        peer->confcost = 1;
```



```c
        peer->confpoll = DEFAULT_POLL;
        peer->conffail = DEFAULT_FAIL;
        peer->confretry = DEFAULT_RETRY;
        peer->localannounce = 2;

        s = getlocaladdr(peer, peerip, listen);
        if(s) return s;
    }
    else new = 0;

    while((s = strtok(0, WHITESPACE))) {
        if(!strcmp(s, "port")) {
            if(!new) return "Port must be on first line";
            s = strtok(0, WHITESPACE);
            if(!s) return "Expected port number";
            p = atoi(s);
            if(p < 1 || p > 65535)
                return "Invalid port number";
            if(listen)
                peer->lsa.sin_port = htons(p);
            else peer->rsa.sin_port = htons(p);
        }
        else if(!strcmp(s, "source-port") && !listen) {
            if(!new) return "Source port must be on first line";
            s = strtok(0, WHITESPACE);
            if(!s) return "Expected port number";
            p = atoi(s);
            if(p < 1 || p > 65535)
                return "Invalid port number";
            peer->lsa.sin_port = htons(p);
        }
        else if(!strcmp(s, "cost")) {
            s = strtok(0, WHITESPACE);
            if(!s) return "Expected cost";
            peer->confcost = atoi(s);
            if(peer->confcost < 1 || peer->confcost > 256)
                return "Invalid cost";
            peer->cost = peer->confcost;
        }
        else if(!strcmp(s, "ttl")) {
            if(!new) return "TTL must be on first line";
            s = strtok(0, WHITESPACE);
            if(!s) return "Expected TTL";
            peer->ttl = atoi(s);
            if(peer->ttl < 1 || peer->ttl > 254)
                return "Invalid TTL";
        }
        else if(!strcmp(s, "accept")) {
            s = strtok(0, WHITESPACE);
            if(!s) return "Expected ACL name";
            peer->accept = findacl(s, 0);
            if(!peer->accept)
                return "Unknown ACL";
        }
        else if(!strcmp(s, "announce")) {
            s = strtok(0, WHITESPACE);
            if(!s) return "Expected ACL name";
            peer->announce = findacl(s, 0);
            if(!peer->announce)
                return "Unknown ACL";
        }
```

```
        else if(!strcmp(s, "accept-connect") && listen) {
            s = strtok(0, WHITESPACE);
            if(!s) return "Expected ACL name";
            peer->remote = findacl(s, 0);
            if(!peer->remote)
                return "Unknown ACL";
        }
        else if(!strcmp(s, "nexthop") && !listen) {
            s = strtok(0, WHITESPACE);
            if(!s) return "Expected IP address";
            s = parseip(s, &peer->nexthop, 0, 0);
            if(s) return s;
        }
        else if(!strcmp(s, "source-ip") && !listen) {
            if(!new) return "Source IP must be on first line";
            s = strtok(0, WHITESPACE);
            if(!s) return "Expected IP address";
            s = parseip(s, &LOCIP(peer), 0, 0);
            if(s) return s;
        }
        else if(!strcmp(s, "secret")) {
            s = strtok(0, WHITESPACE);
            if(!s) return "Expected secret";
            peer->secret = strdup(s);
        }
        else if(!strcmp(s, "poll")) {
            s = strtok(0, WHITESPACE);
            if(!s) return "Expected poll interval";
            s = parsetime(s, &peer->confpoll);
            if(s) return s;
            peer->poll = peer->confpoll;
        }
        else if(!strcmp(s, "fail")) {
            s = strtok(0, WHITESPACE);
            if(!s) return "Expected timeout";
            s = parsetime(s, &peer->conffail);
            if(s) return s;
            peer->fail = peer->conffail;
        }
        else if(!strcmp(s, "retry")) {
            s = strtok(0, WHITESPACE);
            if(!s) return "Expected retry interval";
            s = parsetime(s, &peer->confretry);
            if(s) return s;
            peer->retry = peer->confretry;
        }
        else if(!strcmp(s, "announce-local")) {
            s = strtok(0, WHITESPACE);
            if(!s) return "Expected yes, no or choose";
            if(!strcmp(s, "choose"))
                peer->localannounce = 0;
            else if(!strcmp(s, "yes"))
                peer->localannounce = 1;
            else if(!strcmp(s, "no"))
                peer->localannounce = 2;
            else return "Expected yes, no or choose";
        }
        else return "Unknown keyword";
    }

    if(peer->confpoll + 3 * peer->confretry > peer->conffail)
```



```
        peer->conffail = peer->confpoll + 3 * peer->confretry;

    if(new) {
        if(listen) {
            peer->next = listens;
            listens = peer;
        }
        else {
            peer->next = peers;
            peers = peer;
        }
    }
    return 0;
}


void
parse_acl_std(void) {
    parse_acl("rfc1918", "permit ip 10.0.0.0/8-32");
    parse_acl("rfc1918", "permit ip 172.16.0.0/12-32");
    parse_acl("rfc1918", "permit ip 192.168.0.0/16-32");

    parse_acl("bogons", "permit ip 0.0.0.0/8-32");      /* Local host */
    parse_acl("bogons", "permit ip 127.0.0.0/8-32");    /* Loopback */
    parse_acl("bogons", "permit ip 169.254.0.0/16-32"); /* Link-local */
    parse_acl("bogons", "permit ip 224.0.0.0/3-32");    /* Non-unicast */

    parse_acl("public", "deny acl bogons");
    parse_acl("public", "deny acl rfc1918");
    parse_acl("public", "permit");

    parse_acl("any", "permit");
    parse_acl("none", "deny");
}


int
parse_config(char *file) {
    FILE *cf;
    char buf[4096];
    char *s, *t;
    int i;
    IPADDR peerip;
    int lc, lcc;
    int ok;

    if(file) {
        cf = fopen(file, "r");
        if(!cf) {
            perror(file);
            return 0;
        }
    }
    else cf = stdin;

#define CFOOPS(msg) { printf("%s:%d: Error: %s\n", file, lc, msg); \
            ok = 0; goto next;}
    lcc = 1;
    lc = 0;
    ok = 1;
    while(fgets(buf, sizeof(buf), cf)) {
```

```c
    lc += lcc;
    lcc = 1;
    t = 0;
    for(s = buf; *s; ) {
        if(*s == '#' || *s == '\n' || !*s) {
            *s = 0;
            if(t) {
                i = sizeof(buf) - (t - buf);
                if(i >= sizeof(buf) || !fgets(t, i, cf))
                    break;
            }
            else break;
            lcc++;
        }
        if(*s == '\\') t = s;
        else if(!isspace(*s))
            t = 0;
        s++;
    }

    s = strtok(buf, WHITESPACE);
    if(!s) continue;

    if(!strcmp(s, "acl")) {
        s = strtok(0, WHITESPACE);
        if(!s) CFOOPS("Expected ACL name")
        s = parse_acl(s, 0);
        if(s) CFOOPS(s)

    }
    else if(!strcmp(s, "listen")) {
        s = strtok(0, WHITESPACE);
        if(!s) CFOOPS("Expected peer address")
        s = parseip(s, &peerip, 0, 0);
        if(s || !peerip)
            CFOOPS(s);
        s = addpeer(peerip, 1);
        if(s) CFOOPS(s);
    }
    else if(!strcmp(s, "peer")) {
        s = strtok(0, WHITESPACE);
        if(!s) CFOOPS("Expected peer address")
        s = parseip(s, &peerip, 0, 0);
        if(s || !peerip)
            CFOOPS(s);
        s = addpeer(peerip, 0);
        if(s) CFOOPS(s);
    }
    else if(!strcmp(s, "id")) {
        s = strtok(0, WHITESPACE);
        if(!s) CFOOPS("Expected router ID")
        s = parseip(s, &routerid, 0, 0);
        if(s) CFOOPS(s)
    }
    else if(!strcmp(s, "port")) {
        s = strtok(0, WHITESPACE);
        if(!s) CFOOPS("Expected port number")
        udpport = atoi(s);
        if(udpport < 1 || udpport >= 65535)
        CFOOPS("Invalid port number")
    }
```



```c
    else if(!strcmp(s, "gateway")) {
        s = strtok(0, WHITESPACE);
        if(!s) CFOOPS("Expected yes, no or always")
        else if(!strcmp(s, "always"))
            gwalways = isgateway = 1;
        else if(!strcmp(s, "yes"))
            isgateway = 1;
        else if(!strcmp(s, "no"))
            isgateway = 1;
        else CFOOPS("Expected yes, no or always")
    }
    else if(!strcmp(s, "statusfile")) {
        s = strtok(0, WHITESPACE);
        if(!s) CFOOPS("Expected status file")
        statusfile = strdup(s);
    }
    else if(!strcmp(s, "debug")) {
        s = strtok(0, WHITESPACE);
        if(!s) CFOOPS("Expected debug level");
        debug = atoi(s);
        if(debug < 0 || debug > 8)
            CFOOPS("Bad debug level");
    }
    else if(!strcmp(s, "pidfile")) {
        s = strtok(0, WHITESPACE);
        if(!s) CFOOPS("Expected PID file")
        pidfile = strdup(s);
    }
    else if(!strcmp(s, "secret")) {
        s = strtok(0, WHITESPACE);
        if(!s) CFOOPS("Expected secret")
        secret = strdup(s);
    }
    else if(!strcmp(s, "route-flag")) {
        s = strtok(0, WHITESPACE);
        if(!s) CFOOPS("expected routing flag")
        if(!strcmp(s, "proto1"))
            routeflag = RTF_PROTO1;
        else if(!strcmp(s, "proto2"))
            routeflag = RTF_PROTO2;
        else if(!strcmp(s, "proto3"))
            routeflag = RTF_PROTO3;
        else if(!strcmp(s, "static"))
            otherflag |= RTF_STATIC;
        else CFOOPS("invalid routing keyword")
    }
    else if(!strcmp(s, "default-gateway")) {
        s = strtok(0, WHITESPACE);
        if(!s) CFOOPS("Expected default gateway address")
        s = parseip(s, &defaultroute, 0, 0);
        if(s)   CFOOPS(s)
    }
    else if(!strcmp(s, "override")) {
        s = strtok(0, WHITESPACE);
        if(!s)  CFOOPS("expected routing override flag")
        if(!strcmp(s, "proto1"))
            overrflag |= RTF_PROTO1;
        else if(!strcmp(s, "proto2"))
            overrflag |= RTF_PROTO2;
        else if(!strcmp(s, "proto3"))
            overrflag |= RTF_PROTO3;
```

```
        else if(!strcmp(s, "static"))
            overrflag |= RTF_STATIC;
        else CFOOPS("invalid routing override flag")
    }


    else CFOOPS("Unrecognised keyword")

next:   continue; /* FORTRAN IV, anyone? */
}
if(cf != stdin) fclose(cf);
return ok;
}
```

```c
#include "frpd.h"

int routeflag = RTF_PROTO2;    /* Route flag RTF_PROTO<n> */
int otherflag = 0;          /* Other routing flags */
int overrflag  = 0;         /* Override routes with this flag */

IPADDR defaultroute = 0;     /* Default default route */
IPADDR defgateway = 0;        /* Current default route */

/*
 Add an IP route to the kernel routing table
 */
static void
addroute(IPADDR ip, int bits, IPADDR gw) {
    u_char buf[256];
    u_char *saptr;
    struct rt_msghdr *rtm;
    struct sockaddr_in *ipsa;
    struct sockaddr_in *gwsa;
    struct sockaddr_in *mask;
    int size;

    /*
     Don't do this if we're quiescent
     */
    LOG5("add route %s/%d -> %s routeflag=%x",
            formatip(ip), bits, formatip(gw), routeflag);
    if(gwcost == -1 && ip)
        return;

    /*
     Zap the buffer.
     Assemble message header
     Note that the RTF_HOST flag and the RTA_NETMASK address flag
     are added conditionally further down.
     */
    memset(buf, 0, sizeof(buf));
    rtm = (struct rt_msghdr *)buf;
    rtm->rtm_version  = RTM_VERSION;
    rtm->rtm_type  = RTM_ADD;
    rtm->rtm_pid     = pid;
    rtm->rtm_flags      = RTF_GATEWAY | RTF_UP | routeflag | otherflag;
    rtm->rtm_addrs      = RTA_DST | RTA_GATEWAY;
    saptr = buf + sizeof(struct rt_msghdr);

    /*
     Now the IP address
     */
    ipsa = (struct sockaddr_in *) saptr;
    ipsa->sin_len      = sizeof(struct sockaddr_in);
    ipsa->sin_family  = AF_INET;
    ipsa->sin_addr.s_addr  = ip;
    saptr += SA_SIZE(saptr);

    /*
     And the gateway address
     */
    gwsa = (struct sockaddr_in *) saptr;
    gwsa->sin_len    = sizeof(struct sockaddr_in);
    gwsa->sin_family  = AF_INET;
    gwsa->sin_addr.s_addr  = gw;
```



```c
    saptr += SA_SIZE(saptr);

    /*
     And finally the mask
     Otherwise add a mask address.
     The following is based on observation. (Ugh!)

     Masks are a sockaddr_in, but the length is oddball, and the family
     is unset.

     Default masks have the length byte set to 0.
     Otherwise, the length (mask->sin_len) byte is the mask length
     divided by 8 plus 4, which leaves just the "interesting" bytes
     of mask->sin_addr.

     Note that the SA_SIZE() macro rounds up to 4 byte increments, so the
     actual size of the mask sockaddr is 4 for a default mask, and 8
     for other mask lengths. Note that a sockaddr (of any description)
     is usually pasdded to 16 bytes, as it is in the case of the address
     and gateway sockaddrs.

     So, we get:
          len family  port   addr
       /0   00  00   00 00       Default
       /1   05  xx   xx  xx   80 xx  xx  xx   xx = anything
       /2   05  xx   xx  xx   C0 xx  xx  xx   (we leave it as 0)
       /8   05  xx   xx  xx   FF xx  xx  xx
       /9   06  xx   xx  xx   FF 80 xx  xx
       /16  06  xx   xx  xx   FF FF xx  xx
       /24  07  xx   xx  xx   FF FF FF xx
       /28  08  xx   xx  xx   FF FF FF F0
       /32  08  xx   xx  xx   FF FF FF FF  Or omit and set HOST
     */
    if(bits > 0) {     /* Leave as all zeroes if bits == 0 */
        mask = (struct sockaddr_in *) saptr;
        mask->sin_len  = (bits + 7) / 8 + 4;
        mask->sin_addr.s_addr  = maskbits[bits];
    }
    saptr += SA_SIZE(saptr);
    rtm->rtm_addrs  |= RTA_NETMASK;

    /*
     Update the message size and send the routing message to the kernel.
     */
    size = saptr - buf;
    rtm->rtm_msglen  = size;
    write(rtsocket,  buf, size);
}


/*
 Main routing engine
 The steps are as follows:
 (1) Compute the best path to the gateway; this sets the new gateway peer
     and gateway cost values.
 (2) Construct the local routing table from the set of routes learned
     from our peers.
 (3) Parse the kernel routing table, inserting (or replacing) locally
     originated routes. At this point, delete stale routes from the
     kernel table.
 (4) Insert any routes learned from peers into the routing table (if it
```

```
   wasn't there already).
 (5) Check the gateway status, trigger a path update if necessary.
 (6) Deal with local routes associated with inactive peer interfaces.
 (7) Generate route announcements to all active peers; if the announcement
   to a peer has changed, send the update.
 */
int
getroutes() {
    int mib[6] = { CTL_NET, PF_ROUTE, 0, 0, NET_RT_DUMP, 0 };
    static size_t oldbufsiz = 0;
    static u_char *buf = 0;
    static IPADDR oldpath[MAX_PATH];
    static int oldpathlen = 0;
    size_t bufsiz;
    u_char *ptr, *end, *saptr;
    struct rt_msghdr *rtm;
    struct sockaddr *sa;
    struct sockaddr_in *sin;
    struct sockaddr_dl *sdl;
    u_int u;
    int i, j;
    int valid;
    IPADDR ip;
    IPADDR gw;
    int bits;
    int ifindex;
    int rtype;
    struct iproute *ipr;
    struct iproute *r;
    struct iproute *pipr;
    struct iproute *annh;
    struct iproute *annt;
    struct iproute *oldr;
    struct peer *peer;
    struct peer *gp;
    int gc;
    IPADDR defaultgw;
    IPADDR oldgw;
    int doann;
    int newpath;

    /*
     Compute gateway path on non-gateway nodes
     */
    newpath = 0;
    if(!isgateway) {
        /*
         Search peers for the best route to the gateway
         */
        gp = 0;
        gc = -1;
        FOREACH(peer, peers) {
            /*
             Ignore peers that haven't figured out their gateway,
             and peers that include our router-id in their path to
             the gateway.
             The latter avoids counting to infinity.
             */
            if(peer->gwcost == -1)
                continue;
            for(i = 0; i < peer->pathlen; i++)
```



```
                if(peer->path[i] == routerid)
                    break;
            if(i < peer->pathlen)
                continue;

            /*
             Gateway selection:
             Pick this one if it's the first valid one we found;
             or if it's better than the best we've found so far;
             or if it's as good as the best so far and it's the
             existing gateway peer (this buys us stability).
             */
            i = peer->gwcost + peer->cost;
            if(gc == -1 || i < gc || (i == gc && peer == gwpeer)) {
                gp = peer;
                gc = i;
            }
        }

        /*
         If the route has changed, log the change and set
         gwpeer & gwcost accordingly
         */
        if(gp != gwpeer || gc != gwcost) {
            LOG1("Gateway changed from %s (cost %d)"
                 " to %s (cost %d)",
                gwpeer      ? formatip(gwpeer->nexthop)  :
                defaultroute ? formatip(defaultroute) : "none",
                gwcost,
                gp          ? formatip(gp->nexthop)  :
                defaultroute ? formatip(defaultroute) : "none",
                gc);
            gwpeer = gp;
            gwcost = gc;
            newpath = 1;
        }

        /*
         Find the default route
         Check if the path has changed
         */
        if(gwpeer) {
            if(oldpathlen != gwpeer->pathlen)
                newpath = 1;
            else if(!newpath) {
                for(i = 0; i < gwpeer->pathlen; i++)
                    if(oldpath[i] != gwpeer->path[i])
                        newpath = 1;
            }
            defaultgw = gwpeer->nexthop;
        }
        else defaultgw = 0;

        /*
         If the path, gateway cost etc has changed, tell all the
         other nodes of this fact
         */
        if(newpath) {
            FOREACH(peer, peers)
                if(!peer->synreq)
                    peer->pathreq = 1;
```

```
            if(gwpeer) {
                for(i = 0; i <= gwpeer->pathlen; i++)
                    oldpath[i] = gwpeer->path[i];
                oldpathlen = gwpeer->pathlen;
            }
            else oldpathlen = 0;
        }
    }
    else {
        gwpeer = 0;
        defaultgw = 0;
        oldpathlen = 0;
    }

    /*
     Blast the local routing table
     */
    KILLROUTES(localroutes)

    /*
     Add all the peer routes to the local routing table
     Mark them all as new; the subsequent pass through the routing table
     will mark the existing ones as such.
     */
    FOREACH(peer, peers) {
        LOG8("Searching for routes from peer %s",
                formatip(REMIP(peer)));
        FOREACH(ipr, peer->routes) {
            if(peer->accept && !checkacl(peer->accept,
                    ipr->ip, ipr->bits, ipr->rtype, 0))
                continue;
            r = findroute(localroutes, ipr->ip, ipr->bits, &pipr);
            /*
             If we found the route, don't update it unless it's
             better than the old one
             */
            if(r) {
                if(ipr->cost > r->cost ||
                   (ipr->cost == r->cost &&
                    ipr->isgw < r->isgw))
                    continue;
                LOG8("Updated %s/%d -> %s c=%d",
                    formatip(ipr->ip), ipr->bits,
                    formatip(peer->nexthop), ipr->cost);

            }
            /*
             If the route doesn't exist, add it
             */
            else {
                ADDROUTEAFTER(r, localroutes, pipr)
                LOG8("Added %s/%d -> %s c=%d",
                    formatip(ipr->ip), ipr->bits,
                    formatip(peer->nexthop), ipr->cost);
                r->ip       = ipr->ip;
                r->bits     = ipr->bits;
            }
            r->isgw     = ipr->isgw;
            r->cost     = ipr->cost;
            r->gwcost   = ipr->gwcost;
            r->rtype    = RTYPE_REMOTE;
```



```
            r->peer     = peer;
            r->ifindex  = peer->ifindex;
            r->inuse    = 0;
        }
    }

    /*
     Dump the routing table. First find out how big it is, make sure
     the buffer is big enough, then actually go and get it.
     */
    LOG8("Retrieving routing table");
    if(sysctl(mib, 6, NULL, &bufsiz, NULL, 0) < 0) {
        LOG1("error obtaining routing table size: %m");
        return 1;
    }
    if(!oldbufsiz || bufsiz > oldbufsiz) {
        if(buf) free(buf);
        buf = malloc(bufsiz);
        oldbufsiz = bufsiz;
    }
    i = sysctl(mib, 6, buf, &bufsiz, NULL, 0);
    if(i == -1 && errno == ENOMEM) {
        LOG2("Routing table size error");
        return 1;
    }
    else if(i == -1) {
        LOG1("Error retrieving routing table: %m");
        return 1;
    }

    /*
     For each route in the returned bunch of routes ...
     */
    oldgw = defgateway;
    defgateway = 0;
    ptr = buf;
    for(end = buf + bufsiz; ptr < end; ptr += rtm->rtm_msglen) {

        /*
         Peel apart the route header
         */
        rtm = (struct rt_msghdr *)ptr;
        saptr = ptr + sizeof(struct rt_msghdr);
        valid = 0;
        ifindex = 0;
        ip = 0;
        gw = 0;
        if(rtm->rtm_flags & RTF_HOST)
            bits = 32;
        else bits = -1;

        /*
         Ignore down, dynamic, cloned, broadcast & multicast routes
         */
        rtype = 0;
        if(!(rtm->rtm_flags & RTF_UP) ||
            (rtm->rtm_flags & (RTF_DYNAMIC |
                    RTF_WASCLONED |
                    RTF_BROADCAST |
                    RTF_MULTICAST)))
            continue;
```

```c

    /*
     Set the route type flags based on the route flags for ACL
     matching
     */
    if(rtm->rtm_flags & RTF_LLINFO)
        rtype = RTYPE_LAYER2;
    else if(rtm->rtm_flags & RTF_STATIC)
        rtype = RTYPE_STATIC;
    else if(rtm->rtm_flags & RTF_PROTO1)
        rtype = RTYPE_PROTOCOL;
    else rtype = RTYPE_LOCAL;

    /*
     Now parse each of the sockaddr objects from the route
     */
    for(i = 0; i < RTAX_MAX; i++) if(rtm->rtm_addrs & (1 << i)) {
        sa = (struct sockaddr *) saptr;

        /*
         Destination IP address. Check that it is in fact
         an IP address; just ignore it if it isn't
         Have a first cut at determining what the prefix length
         is from the route class. (CIDR? What's CIDR?)
         */
        if(i == RTAX_DST) {
            if(sa->sa_family != AF_INET)
                break;
            sin = (struct sockaddr_in *) saptr;
            ip = sin->sin_addr.s_addr;
            if(ip == ntohl(INADDR_LOOPBACK))
                break;
            j = *((u_char *) &sin->sin_addr);
            if(bits == -1) {
                if(j >= 128)  bits = 24;
                else if(j >= 64) bits = 16;
                else      bits = 8;
            }
            valid = 1;
        }

        /*
         Gateway IP address, if specified and it's IPv4.
         */
        else if(i == RTAX_GATEWAY && sa->sa_family == AF_INET) {
            sin = (struct sockaddr_in *) saptr;
            gw = sin->sin_addr.s_addr;
        }

        /*
         The netmask. Why they couldn't just use a sockaddr_in
         for this I don't know. See comments in addroute()
         above for more info on this ugly thing.
         */
        else if(i == RTAX_NETMASK) {
            bits = 0;
            for(j = 4; j < sa->sa_len; j++) {
                u = saptr[j];
                u = (u & 0x55) + ((u >> 1) & 0x55);
                u = (u & 0x33) + ((u >> 2) & 0x33);
                u = (u & 0x0f) + ((u >> 4) & 0x0f);
```



```c
                bits += u;
            }
        }

        /*
         Interface specification. We're just interested in
         the ifindex for ACL matching
         */
        else if(i == RTAX_IFP) {
            sdl = (struct sockaddr_dl *) saptr;
            ifindex = sdl->sdl_index;
        }
        saptr += SA_SIZE(saptr);
    }
    if(!valid) continue;

    /*
     Check the general sanity of the route
     */
    LOG8("Kernel route %s/%d -> %s flags=%x ifindex=%d",
        formatip(ip), bits, formatip(gw), rtm->rtm_flags,
        ifindex);
    if(ip != (ip & maskbits[bits]))
        continue;

    /*
     If we found a default route ...
     If it's a learned route, and it doesn't point to the correct
     default gateway, then delete it.
     Otherwise note that we have a default route.
     */
    if(bits == 0) {
        if(gw != defaultgw && (rtm->rtm_flags & routeflag)) {
            LOG2("delete stale default route -> %s",
                    formatip(gw));
            rtm->rtm_type = RTM_DELETE;
            write(rtsocket, ptr, rtm->rtm_msglen);
        }
        else defgateway = gw;
        continue;
    }

    /*
     See if this route is one we (or a previous incarnation of we)
     inserted; if so, see if it's in our routing table.
     If it isn't, or the route should be a local one, then it's a
     stray, so put it down humanely.
     */
    if(rtm->rtm_flags & routeflag) {
        ipr = findroute(localroutes, ip, bits, 0);
        if(!ipr || !ipr->peer || gw != ipr->peer->nexthop) {
            if(gwcost == -1)
                continue;
            LOG2("delete stale route %s/%d -> %s",
                formatip(ip), bits, formatip(gw));
            rtm->rtm_type = RTM_DELETE;
            write(rtsocket, ptr, rtm->rtm_msglen);
        }
        else ipr->inuse = 1;
        continue;
    }
```

```c

        /*
         Find the route in our routing table
         If it's not there, add it
         But, if it's to be overridden, delete it
         */
        ipr = findroute(localroutes, ip, bits, &pipr);
        LOG8("Route %s/%d type=%d ifindex=%d %s",
            formatip(ip), bits, rtype, ifindex,
            ipr ? "found" : "new");
        if(!ipr) {
            ADDROUTEAFTER(ipr, localroutes, pipr)
            LOG2("Added %s/%d local", formatip(ip), bits);
            ipr->ip = ip;
            ipr->bits = bits;
        }
        else if(rtm->rtm_flags & overrflag) {
            if(gwcost == -1)
                continue;
            LOG2("Overriding %s/%d", formatip(ip), bits);
            rtm->rtm_type = RTM_DELETE;
            write(rtsocket, ptr, rtm->rtm_msglen);
            continue;
        }
        /*
         Here is where we fill out a route as an originated route
         Prime the cost to 0, isGW flag to true, and GWcost to the
         local gateway cost.
         */
        ipr->isgw    = 1;
        ipr->cost    = 0;
        ipr->gwcost  = gwcost;
        ipr->rtype   = rtype;
        ipr->peer    = 0;
        ipr->ifindex = ifindex;
        ipr->inuse   = 1;
    }

    /*
     For each route not found in the kernel, add it to the kernel table
     */
    FOREACH(ipr, localroutes) if(!ipr->inuse)
        addroute(ipr->ip, ipr->bits, ipr->peer->nexthop);

    /*
     If this is a gateway, set the gateway flag
     Do this if we actually have a default gateway, or if the "always
     gateway" flag is set.
     */
    if(isgateway) {
        if(defgateway || gwalways)
            gwcost = 0;
        else gwcost = -1;
    }

    /*
     Otherwise, if no default route was found (or one was found that
     didn't point to the correct peer, and was therefore deleted),
     set the gateway to the address of the gateway peer.
     If there is no gateway peer, set the default to the configured
     default (if any).
```



```c
     If we set a gateway, put it into the routing table.
     If the result of this is a change, force a PATH update to be
     sent to all peers.
     */
    else if(!defgateway) {
        if(gwpeer)
            defgateway = gwpeer->nexthop;
        else if(defaultroute)
            defgateway = defaultroute;
        if(defgateway)
            addroute(0, 0, defgateway);
        if(defgateway != oldgw)
            FOREACH(peer, peers)
                peer->pathreq = 1;
    }

    LOG4("Local table:");
    if(debug >= 4) dumproutes(localroutes);

    /*
     If there are no routes, purge the announced routes and we're done
     */
    if(gwcost == -1) {
        LOG4("Gateway not reachable, not announcing routes");
        FOREACH(peer, peers) {
            KILLROUTES(peer->annrts);
            peer->nextrt = 0;
            peer->nullrt = 0;
        }
        return 0;
    }

    /*
     Search the routing table for local routes which are the
     subject of a peer relationship.
     For these routes, we need to pick winners & losers; winners get
     to announce the route to their peers.
     The winner is the peer with the lowest gwcost; failing that the
     lower IP address.
     Note that if the remote peer is down, we count ourselves as
     a loser. This deals with peer/listen relationships where the
     peer disappears when it goes down.
     Reminder: peer->localannounce: 0 = pick winner; 1 = always announce;
                        2 = never announce.
     */
    FOREACH(ipr, localroutes) {
        ipr->inuse = 1;
        FOREACH(peer, peers) {
            if(ipr->ip   == peer->localroute &&
               ipr->bits == peer->localbits &&
               peer->localannounce != 1 &&
               ( peer->localannounce == 2 ||
                 peer->gwcost < gwcost   ||
                 ( peer->gwcost == gwcost &&
                   ntohl(LOCIP(peer)) > ntohl(REMIP(peer)) ))) {
                LOG8("Route %s/%d is local loser",
                        formatip(ipr->ip), ipr->bits);
                ipr->inuse = 0;
            }
        }
    }
```

```

    /*
     Prepare the peer announcements
     Ignore peers that are down
     */
    FOREACH(peer, peers) {
        if(peer->gwcost == -1)
            continue;
        LOG4("Routing for %s", formatip(REMIP(peer)));

        annh = annt = 0;
        doann = 0;
        if(peer->nextrt || peer->nullrt)
            doann = 1;
        oldr = peer->annrts;
        FOREACH(ipr, localroutes) {
            /*
             Don't announce losing routes
             */
            if(!ipr->inuse)
                continue;

            /*
             Split horizon, allowing for multiple links
             Don't advertise routes back to where we got them from
             Don't advertise local routes up the interface
             they point at.
             ??? May need to restrict this to just iface routes
             */
            if(ipr->peer && ipr->peer->routerid == peer->routerid)
                continue;
            if(!ipr->peer && ipr->ifindex == peer->ifindex)
                continue;

            /*
             Check announcement ACL, skip if not permitted
             */
            if(peer->announce && !checkacl(peer->announce,
                        ipr->ip, ipr->bits,
                        ipr->rtype, ipr->ifindex))
                continue;

            /*
             See if route would reach "over the horizon"
             That is, if it would be shorter to route via
             our gateway and back via the route originator's
             gateway, then don't bother advertising the route
             */
            LOG8("%s/%d rc=%d + lc=%d"
                    " < rgc=%d + pgc=%d + isgw=%d",
                formatip(ipr->ip), ipr->bits,
                ipr->cost, peer->cost, ipr->gwcost,
                peer->gwcost, (ipr->isgw && peer == gwpeer));
            if(ipr->cost + peer->cost >=
               ipr->gwcost + peer->gwcost
                    + (ipr->isgw && peer == gwpeer))
                continue;

            /*
             Add the route to the announcement
             */
```



```
            ADDROUTEAFTER(r, annh, annt)
            r->ip      = ipr->ip;
            r->bits    = ipr->bits;
            r->isgw    = (ipr->isgw && (peer == gwpeer));
            r->cost    = ipr->cost + peer->cost;
            r->gwcost  = ipr->gwcost;
            r->rtype   = 0;
            r->ifindex = 0;
            r->peer    = 0;
            annt = r;

            LOG8("new: %s/%d gw=%d cost=%d gwcost=%d rtype=%d",
                    formatip(r->ip), r->bits, r->isgw,
                    r->cost, r->gwcost, r->rtype);
            /*
             If this route is different to the corresponding
             previous announcement, set the "do announcement"
             flag.
             */
            if(!doann) {
                if(!oldr || oldr->ip     != r->ip
                    || oldr->bits   != r->bits
                    || oldr->isgw   != r->isgw
                    || oldr->cost   != r->cost
                    || oldr->gwcost != r->gwcost
                    || oldr->rtype  != r->rtype)
                    doann = 1;
                if(oldr) oldr = oldr->next;

            }
        }

        /*
         If there are any unchecked previous announcement records
         left, then the announcement has changed and we need to
         send it out.
         */
        if(oldr) doann = 1;

        /*
         If the routing table has changed, force an update
         If there are no routes to be announced to this peer,
         set the nullrt flag to force a null route announcement
         If the announcement hasn't changed, just blow the newly
         constructed announcement away.
         */
        LOG4("Announcements for %s: (update %srequired)",
            formatip(REMIP(peer)), doann ? "" : "not ");
        if(debug >= 4) dumproutes(annh);
        if(doann) {
            KILLROUTES(peer->annrts)
            peer->annrts = annh;
            peer->nextrt = annh;
            if(annh)
                peer->nullrt = 0;
            else peer->nullrt = 1;
        }
        else KILLROUTES(annh)
    }
    return 0;
}
```

```

/* EOF */

```