HELSINKI UNIVERSITY OF TECHNOLOGY

Department of Electrical and Communications Engineering

Tuukka Taipale

Comparison of Routing Software in Linux

Thesis submitted in partial fulfillment of the requirements for the degree of Master of Science in Engineering

Espoo, Finland, September 27, 2006

Supervisor of the Thesis	Professor D.Sc. (Tech.)	Raimo Kantola
Instructor of the Thesis	Master of Science	Pasi Strengell

HELSINKI UNIVERSITY OF TECHNOLOGY | ABSTRACT OF THE MASTER'S THESIS

Author:	Tuukka Taipale	
Name of the Thesis:	Comparison of Routing Software in Linux	
Date:	September 27, 2006	**Number of pages:** 64
Department:	Department of Electrical and Communications Engineering	
Professorship:	S-38 Networking Technology	
Supervisor:	Professor D.Sc (Tech.)	Raimo Kantola
Instructor:	Master of Science	Pasi Strengell

Linux operating system is becoming more and more popular today. Network connections are becoming faster and the amount is increasing all the time. Today's networks need routing so that the messages can go towards their destinations in the Internet. The routing can be performed in the Linux systems. In this thesis we handle both the Linux operating system and routing functionality.

Our routing software is based on the FreeBSD operating system. This thesis studies how well that software works on Linux. The first step is to port this software on Linux. After that we examine the functionality of the software in Linux by comparing the routing daemon with two commercial routing solutions and an open source one. The comparison consists of performance and software complexity measurements.

The results of these measurements not only show that the software is capable to be run on Linux, but also give even more information on how different routing software packages perform the routing tasks. The output of the software complexity measurements shows the type of source code in the compared routing solutions. The complexity of the software is related to the easiness to maintain it.

Keywords: Routing software, Routing performance, Software complexity

TEKNILLINEN KORKEAKOULU DIPLOMITYÖN TIIVISTELMÄ

Tekijä:	Tuukka Taipale		
Työn nimi:	Reititysohjelmistojen vertailu Linuxissa		
Päivämäärä:	27 syyskuuta 2006	**Sivumäärä:** 64	
Osasto:	Sähkö- ja Tietoliikennetekniikan osasto		
Professuuri:	S-38 Tietoverkkotekniikka		
Valvoja:	Professori, TT	Raimo Kantola	
Ohjaaja:	Filosofian Maisteri	Pasi Strengell	

Linux-käyttöjärjestelmä yleistyy nykyään yhä enemmän ja enemmän. Verkkoyhteydet tulevat nopeammiksi ja niiden määrä kasvaa koko ajan. Nykypäivän verkot tarvitsevat reititystä, jotta viestit voidaan välittää Internetissä eteenpäin kohti vastaanottajaa. Linux-järjestelmät voivat toimia reitittiminä. Tässä työssä käsittelemme sekä Linux-käyttöjärjestelmää että reititystoiminnallisuutta.

Reititysohjelmistomme perustuu FreeBSD-käyttöjärjestelmään. Tässä työssä tutkimme, kuinka hyvin tämä ohjelmisto toimii Linuxissa. Ensimmäinen toimenpide on muokata reititysohjelmisto yhteensopivaksi Linuxin kanssa. Sen jälkeen tutkimme ohjelmiston toiminnallisuutta Linuxissa vertailemalla tätä reititysohjelmaa kahden kaupallisen ja yhden avoimeen lähdekoodiin perustuvan reititysratkaisun kanssa. Vertailu koostuu suorituskyky- ja ohjelmiston kompleksisuuden mittauksista.

Näiden mittausten tulokset eivät pelkästään näytä, että ohjelmaa voidaan ajaa Linuxissa, vaan antavat myös lisätietoa siitä, miten reititysohjelmistot suorittavat reititystehtäviä. Ohjelmiston kompleksisuusmittausten tuloksena näemme lähdekoodin laadun vertailluissa reititysohjelmissa. Ohjelmiston kompleksisuus liittyy siihen, kuinka helppoa ohjelmistoa on ylläpitää.

Avainsanat: Reititysohjelmisto, Reitityksen suorituskyky, Ohjelmiston kompleksisuus

PREFACE

This master’s thesis was carried out in Nokia Networks, IPSO Platform and FlexiPlatform departments. I appreciate the instructions and possibility to do this work along with my other responsibilities.

I would like to express my gratitude to my supervisor Master of Science Pasi Strengell for his guidance during my work for the thesis. I am also very grateful to the work mates with whom I have discussed issues concerning IP routing.

I would also like to thank my wife Marjut for her patience during this work. Special thanks also for my parents and siblings for being valuable support to me during my studies.

Helsinki, September 27, 2006

Tuukka Taipale

Table of Contents

List of Figures

1. Introduction

1.1 Background

Every network node in the Internet has to be capable to send and receive Internet Protocol (IP) packets. As all the endpoints cannot have a connection between each other, we need to form a structured network between all the Internet hosts. The network contains then endpoints and nodes between them. Those nodes between the endpoints are called routers. The task of a router is to forward Internet packets between the endpoints. Another task is to keep their internal forwarding table up to date.

Router is a special-purpose dedicated computer that connects several networks. Routers switch packets between these networks in a process known as forwarding. This process may be repeated several times on a single packet by multiple routers until the packet can be delivered to the final destination - switching the packet from router to router to router... until the packet gets to its destination. [BAK]

These routers are computers that have an operating system (OS). The OS runs different processes that take care of the hardware and provide the interface for the user to this hardware. One of these processes is a routing daemon that handles all the routing related work. In this thesis we start from a routing daemon that runs in a FreeBSD[1] based OS. Our goal is to determine: How suitable this routing daemon is for Linux?

1.2 Objective

First, we port this routing daemon to Linux. Regarding this work phase we study the differences between Linux and FreeBSD that are related to routing. These differences show us the needed modifications in porting.

[1] FreeBSD is open source variant of Berkeley Software Distribution (BSD), which is a UNIX variant. [MCK]

In addition we measure the complexity and maintainability of the ported routing daemon. To define if the ported routing daemon is of good quality, we compare the complexity measurement results with the corresponding results of a few other routing daemons. The software complexity comparison includes two commercial routing daemons (IpInfusion ZebOS and NextHop GateD) and an open source one (Quagga).

Performance plays a big role in routers. So we compare also the performance of the routing daemons mentioned above. Performance measurements include for example scalability, memory usage and convergence times.

As the ported routing daemon is not optimized anyway, we exclude extensive stress testing and stability evaluation from this thesis. These kinds of tests are more useful in a later phase, when the ported routing daemon is optimized for Linux. We do not study the internal structure of the routing daemons either. Additionally, we do not introduce the different routing protocols, as this thesis concentrates more on the software than to the functionality of the routing protocols.

1.3 Structure

This thesis starts with introducing the routing software in chapter 2. That chapter introduces the routing solutions compared in this thesis. Chapter 3 describes the porting work needed to migrate the routing daemon from FreeBSD to Linux. We discuss the software complexity measurements in chapter 4. That chapter introduces the different ways to measure the complexity from the source code.

Chapter 5 describes the work done in porting. After that we compare the performance of the routing daemons in chapter 6. Chapter 7 presents the results and conclusions of this thesis.

2. Routing Software

Today’s operating systems usually run many processes. Daemon is a special type of process. It is a process which goes around in the background and does routine work [LEH]. This description suits well with this study. In this case the routine work the daemon process does consist of routing tasks. This chapter introduces the different routing solutions we investigate in this thesis.

2.1 The Purpose of Routing Software

An IP router has basically two tasks. Routing is generally accomplished by maintaining a routing table in each end system and each router, that gives, for each possible destination network, the next router to which the internet datagram should be sent [STA]. This long description of router tasks summarizes the two parts. Firstly, the router forwards the datagrams from sender towards the destination in the network. Secondly, the router keeps the routing table synchronized between the other routers. In the following subsections we describe these two tasks in detail.

2.1.1 Traffic Forwarding

Forwarding is the process a router goes through for each packet it receives. The packet may be consumed by the router, it may be output on one or more interfaces of the router, or both. Forwarding includes the process of deciding what to do with the packet as well as queuing it up for (possible) output or internal consumption. [BAK]

We use the above definition for forwarding in this thesis. The process which decides where the packets should be forwarded uses specific rules for that. These rules are stored in the Forwarding Information Base (FIB).

The table containing the information necessary to forward IP Datagrams, in this document, is called the Forwarding Information Base. At minimum, this contains the interface identifier and next hop information for each reachable destination network prefix. [BAK]

Both forwarding and FIB are very important terms when we speak about routing. If a router was a customer servant, IP Datagrams would be its customers. The customer serving job would then be called forwarding. Even if the packets were coming to the router itself, they should be forwarded to a suitable receiving process. To continue more this example of customers and their servant, we can imagine the situation in a hotel’s reception. The receptionist receives customers. The receptionist is now our forwarder. He/she looks the customer room from booking list, or FIB in our terms. Finally the receptionist points the way to the room, or a next hop, to the customer. This was a different situation, but the same forwarding functionality.

What would we then do if one of our links goes down? At least we do not receive any packets for forwarding from there anymore. The following subsection explains more that case.

2.1.2 Routing Table Management

The second task of a router is to keep its routing table up to date. Routing table can also be described by route database. The term "router" derives from the process of building this route database; routing protocols and configuration interact in a process called routing [BAK]. This sentence tells the core idea behind the route database. Router gathers information of the network via routing protocols and saves that to the database. Another alternative is to modify the configuration by hand. Both these ways makes the router able to adapt to the network changes dynamically.

The hand made changes are easy to understand, but how can the router adjust its internal database when network changes? This case was above in the previous subsection where we thought about the case when one link goes down. This failure can be noticed by routing protocols. A router uses them to adjust its database automatically to the network changes. We have different kinds of routing protocols, but in this case their functionality is similar. Routing protocols constantly send polling messages over the link. Default time interval for the polling packets for example in Routing Information Protocol (RIP) is 30 seconds [BAK]. Then, if the router has not received the polling packet from the link for 180 seconds, the link is marked as failed.

All these kinds of routing information is then in the routing table. We can assume that in the beginning, when the network is initially set up, the network has also other routing traffic. These messages include route add and probably also route remove messages. But after this initial routing table synchronization messaging, the network transfers only these link polling packets. This means, from the routing table perspective, that routing table has minor alternating needs after initialization.

To continue the above example from hotel reception, we connect the routing table also to it. We have the customer names and corresponding rooms in our FIB. Routing table now has more information. This information is not defined in any RFC yet, but we can imagine some relevant facts we need to know from each customer. These can include for example phone number, home address and customer age, to name but a few. We see that these facts are not so important in the forwarding process. However, they are real and can be used for example for customer classification purposes. In routing we can similarly classify the traffic according to packet information.

2.2 Evaluated Routing Daemons

In this section we describe more the evaluated routing daemons. As all these applications are routing daemons, it means that they do basically the work described in the previous section. We have four routing daemons: Nokia proprietary ipsrd, ZebOS from IpInfusion, GateD from Nexthop and open source Quagga. Each of them is introduced in the different subsections below. For easier comparison, we have organized the subsections of each routing daemon in the same form. The introductions are in the following order: Overview, Architecture, Routing protocols and Supported operating systems. As ipsrd is the routing daemon that we are modifying during the porting work, we give the most detailed introduction to its architecture.

The routing protocols are not introduced here. The traditional routing protocols mentioned in the following subsections can be studied from the references in the table below. The different routing daemons may support also other routing protocols. They are not introduced here either.

Table 1: Traditional Routing Protocols

Abbreviation	Name	Reference
BGP	Border Gateway Protocol	RFC 1771
OSPF	Open Shortest Path First	RFC 2328
RIP	Routing Information Protocol	RFC 2453

2.2.1 Nokia Ipsrd

Overview

Ipsrd is a Nokia proprietary routing daemon. The name of ipsrd comes from its predecessor: Ipsilon Routing Daemon.

Ipsrd is a modular user level process consisting of core services, a routing database, and protocol modules supporting multiple routing protocols. Ipsrd is based on GateD 3.5 Release Beta 3. [IPS]

Architecture

The following figure describes the ipsrd architecture:

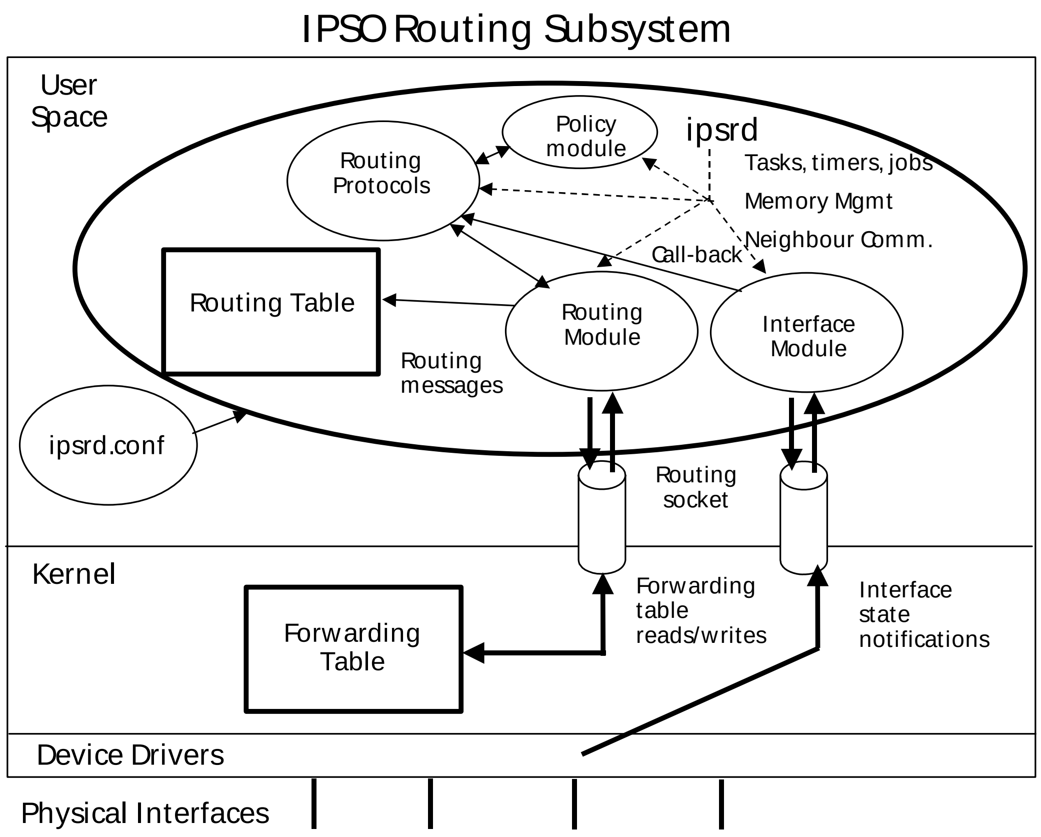


Figure 1: IPSO Routing Subsystem Architecture [IPS]

Ipsrd core specifically provides mechanisms for scheduling protocol computation, memory management mechanisms, neighbor communication management mechanisms, route storage mechanisms, and configuration and reconfiguration support.

The routing table is built from various routing protocols that are enabled and also from information obtained from the kernel about connectivity the router has to a network topology. Forwarding is performed by the operating system kernel and thus the instructions for forwarding are also there. Ipsrd only updates these instructions when needed.

Events handled by ipsrd can be summarized as:

- Receive and send protocol messages,
- Protocol functionality processing,

- Receive interface state change and corresponding protocol actions,
- Fire timers and generate resulting actions,
- Respond to the received signals, including reconfiguring itself based on changes in system configuration,
- Add/delete routes and protocol-installed Address Resolution Protocol (ARP) entries to and from the kernel forwarding table,
- Monitoring.

The following paragraphs describe the different parts of the ipsrd architecture in figure. Ipsrd is the big circle in the user space of figure and the different parts are inside that circle.

Protocol scheduling

As per GateD architecture, ipsrd provides a common substrate, which includes abstractions such as tasks, timers and jobs, for implementing primitive operations (read/write protocol message, perform route computation) efficiently. [IPS] The next paragraphs have the definitions for these abstractions [IPS].

A *task* is a separately schedulable thread of protocol computation. All ipsrd processing is done within a single process, and routing protocols are implemented as one or several tasks. Different protocol implementations use tasks differently. The simplest way to use tasks is to allocate a single task to all computation that happens within a protocol. Each protocol listens on a socket for messages of its protocol. Similarly, ipsrd interface task listens on an interface socket, and the forwarding table task has access to the forwarding table by operations on a routing socket.

A *timer* is an event scheduled for a future instant, which causes some non-preemptable computation to be performed at that instant. This computation is associated with a specific task. Routing protocols use timers for periodic tasks such as monitoring connection status, timing out connection opens and so on. The ipsrd timer module as per GateD implementation allows specification of both one-shot timers and interval timers. Also ipsrd allows protocols to specify higher priority timers, drifting timers, as well as non-drifting timers. Time specification granularity is 1 second.

A *job* (which can either be foreground or background) is non-preemptable protocol computation that can be scheduled anytime in the future. Foreground jobs are time-limited computations run when it is safe to modify the ipsrd routing database. Background jobs are longer running computations scheduled when no timers or I/O is pending. For example the OSPF implementation sets a timer after an SPF computation to ensure that the SPF computation is not done within that time period; at the expiry of the timer, it sets up a background job to run the next SPF computation. Each background job has a priority between zero and seven; background jobs are scheduled in priority order.

Memory management

Ipsrd uses the operating system memory management functionality for some situations. However, to optimize memory usage for the most common case of allocating control blocks, ipsrd provides its own memory management routines.

Neighbour communication

Every routing protocol instance communicates with one or more neighbors for exchange of routing information. Ipsrd provides an interface for protocol implementations to send/receive routing messages (transport mechanism APIs), and some abstract data types for physical entities involved in such communication (structs for neighbour communication, one per protocol used to store neighbour’s address, state info etc).

Interface module

Ipsrd uses a 4-level hierarchical structure to store router's connectivity information. These are physical interface, logical interface, address family and address. This is designed to be consistent with the operating system kernel. On start-up ipsrd receives all state from the kernel through a SYSCTL call. It also receives interface up/down/delete/add and address add/delete events asynchronously in interface routing socket messages. For each received message ipsrd then notifies the protocol modules of the changes through protocol's registered call-back functions.

Routing module

Route storage mechanism in ipsrd routing module provides a central repository for different protocol instances to maintain routing information.

Ipsrd uses a routing socket message to modify the kernel forwarding table as and when ipsrd’s routing table is modified by the protocols. It learns the view of the kernel's forwarding table on start-up through a SYSCTL call and then uses that information to modify routes. Ipsrd can add, delete or change existing routes in the forwarding table.

Routing policy

The ipsrd policy module provides a mechanism for storing ipsrd routing policy descriptions. Ipsrd supports setting of rules for routing policy. *Route filtering* allows the user to define the list of routes the router will accept from or propagate to its neighbours; ipsrd supports only inbound route filtering (list of routes that will be accepted from the neighbours).

Route precedence is the value ipsrd uses to prioritize routes to the same destination from one protocol or peer over another.

With ipsrd *route aggregation* it is possible to generate a more general route given the presence of a one or more specific routes, and by that way reduce the amount of routing information passed around.

Additionally it is possible to redistribute the routes between the different protocols that ipsrd supports.

Routing protocols

Ipsrd supports EGP (exterior gateway protocol), IGP (interior gateway protocol), and multicast routing protocols as separate modules.

Interior routing protocols supported are RIP (v1, v2), RIPng, OSPFv2, and OSPFv3. Additionally Cisco specific IGRP is supported.

BGPv4 is the supported exterior routing protocol.

DVMRPv3 and PIM (sparse and dense mode) are supported multicast protocols.

The following router services are also available:

- Router discovery: ipsrd implements the server portion (v4 and v6)
- VRRP v2
- bootp relay functionality

Additionally ICMP and IGMP (v2 and mtrace) are supported, latter for multicast protocols and for reporting multicast group membership.

Supported operating systems

As Nokia proprietary routing daemon, ipsrd supports only Nokia proprietary Ipsilon Operating System[2] (IPSO). However, the goal of the porting work of this thesis is to make ipsrd compatible with Linux kernel 2.6.x.

2.2.2 IpInfusion ZebOS

Overview

ZebOS is routing software evolved on top of open source GNU Zebra [ZEB]. Zebra software is commercially produced as ZebOS Server Routing Suite [RAM]. In this thesis we measure ZebOS Advanced Routing Suite (ARS), which has more features than Zebra or ZebOS Server Routing Suite. Despite the fact that ZebOS has more

[2] The IPSO is based on FreeBSD 2.1.5, and IPSO still shares similarities with other UNIX-style operating systems. FreeBSD itself is derived from BSD4.4-Lite, a version of UNIX developed at the University of California, Berkeley. [IPS]

features, the software architecture is still of the same type in both Zebra and ZebOS [IPI].

Architecture

ZebOS has a modular platform independent architecture [IPI]. ZebOS manages its various protocols as separate daemon processes. This is the major difference for example between ipsrd and ZebOS. When ipsrd performs the routing work in a single process, ZebOS has its own process for each protocol. The following figure describes the ZebOS architecture.

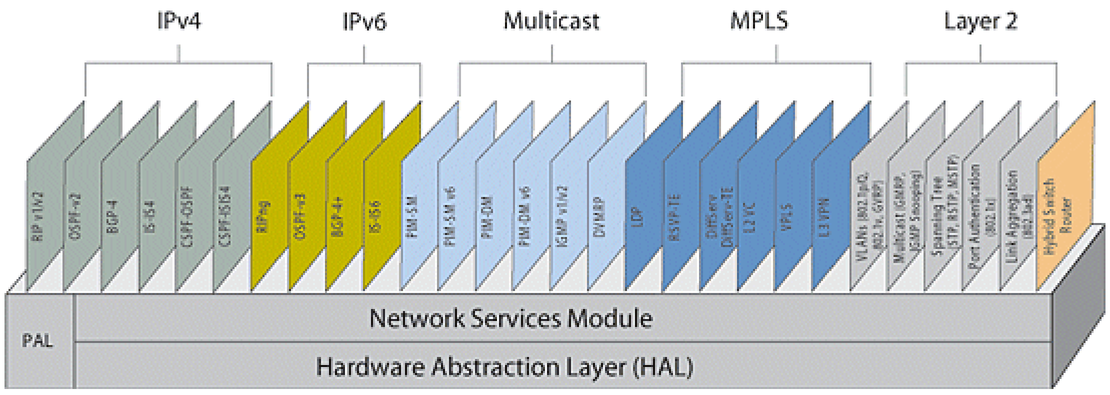


Figure 2: ZebOS Modular Routing and Switching Software Building Blocks [IPI]

Each protocol module is built on the ZebOS Network Services Module (NSM). It is the base module that simultaneously and independently communicates with every ZebOS ARS routing and switching process. The NSM manages both the route table and each of the enabled protocols; performs route conversion and redistribution; and manages the interface state, routing policies and filtering. The NSM communicates through the Platform Abstraction Layer (PAL) to the underlying operating system or network processor for forwarding table updates. [IPI]

Routing protocols

The ZebOS ARS supports both IPv4 and IPv6 versions of OSPF, BGP, IS-IS and RIP. It also offers virtual routing support and Traffic Engineering (TE) extensions and Constrained Shortest Path First (CSPF) topology support for the OSPF and IS-IS Protocol Modules. ZebOS also provides MPLS modules. [IPI] As we do not measure

MPLS routing modules, we skip the MPLS introduction here. The details can be investigated from [IPI]. Supported multicast routing protocols are IPv4 and IPv6 versions of PIM-SM and PIM-DM, IGMP v1/v2 and DVMRP.

Supported operating systems

ZebOS supports Linux, MontaVista Professional Edition, NetBSD and VxWorks. [IPI]

2.2.3 NextHop GateD

Overview

For over fifteen years, GateD has been the standard starting point for anyone who needed routing in the Internet, for everything from server redundancy to the most scalable, core IP routers. [NHT] This marketing text from NextHop GateD datasheet shows the long history of GateD. The first GateD was developed at Cornell University by the Cornell GateDaemon Consortium [COR].

GateD (GateDaemon) is traditional [RAM] routing software. Traditional means here that GateD has the longest history of the routing daemons compared in this thesis. In the beginning of its history GateD was open source software (through GateD Consortium), but today NextHop has commercialized the software. GateD is a single daemon that can run multiple protocols at the same time [FEN]. This is different from ZebOS, which has multiple processes performing routing tasks.

Architecture

NextHop GateD and ipsrd are based on the same GateDaemon and thus share the same architecture. They both have all the functionality in a single process. GateD is a modular software program consisting of core services, a routing database, and protocol modules supporting multiple routing protocols [G35]. The following figure illustrates the GateD architecture and modules.

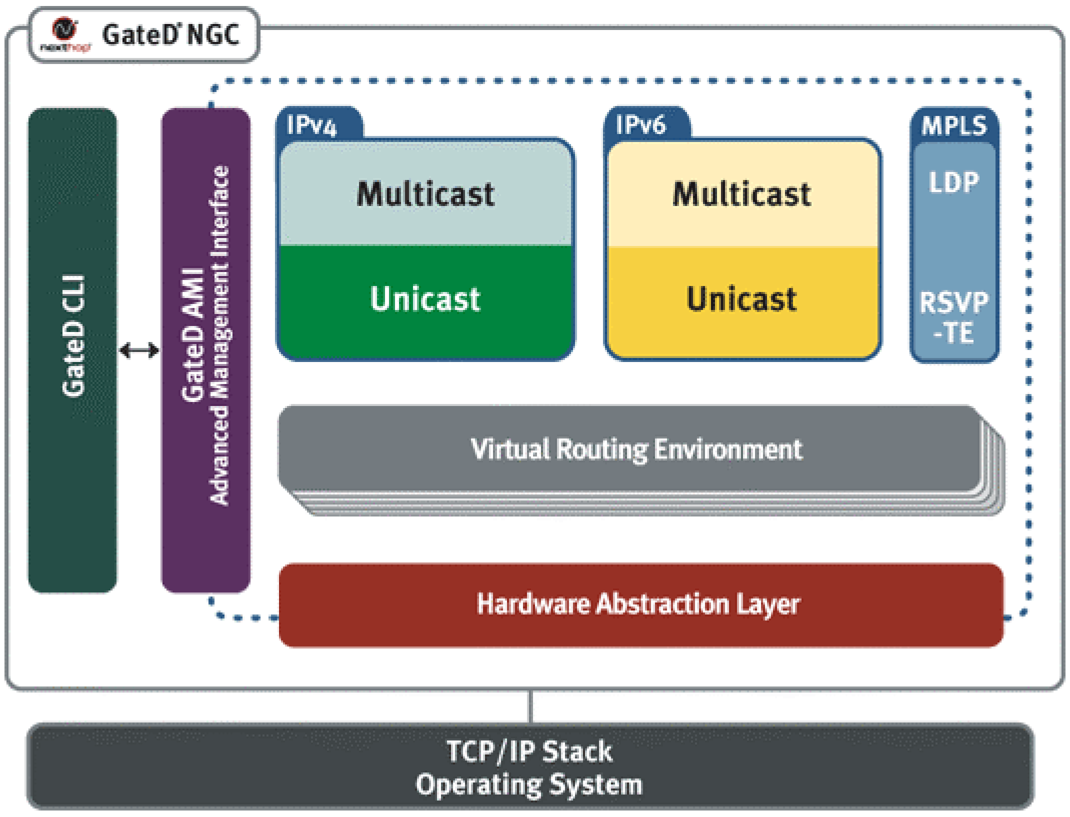


Figure 3: GateD Architecture [NHT]

As we can see from the figure, the majority of the modules are inside the box marked with dashed line. That box is the single process of the routing daemon. The CLI part and the TCP/IP Stack and Operating System are the parts outside that box. The different modules of the figure are introduced below.

The basic features on which the routing protocols rely are on the borders of the box in the architecture figure (GateD AMI and Hardware Abstraction Layer). The core of GateD consists of the following features [NHT]: GateD AMI (Advanced Management Interface), OS adaptation layer, Sophisticated Policy Engine, Memory Management, Static Route Support, Timer Facilities, Cooperative Multitasking, Checksum Generation and Verification and MIBs. So this long list contains the basis of GateD. This basis is complemented by various routing protocols in the other modules of the core box.

Routing protocols

The routing protocol packages are additional available components for the basic GateD core. Available components include GateD Fast OSPF, GateD BGP, GateD IS-IS, GateD DVMRP, GateD PIM-SM, GateD PIM-SSM, GateD PIM-DM, GateD MSDP, GateD MP-BGP for IPv6, GateD IS-IS for IPv6, GateD Fast OSPF3, GateD MPLS, GateD VRE (Virtual Routing Environment), and GateD VPN (layer-3 MPLS-BGP virtual private networking). RIP v1 and v2 are included to the core package [NHT].

Supported operating systems

NextHop’s GateD software has been ported to many software and hardware environments including Solaris, LynxOS, Nucleus, Linux, HP-UX, Tru64 UNIX and proprietary OSes. NextHop can supply a pre-ported version of GateD for a number of these environments including: [NHT]

- NetBSD

- Monta Vista Carrier Grade Linux

- Red Hat Enterprise Linux

- Wind River Systems VxWorks, PNE

- Green Hills Integrity

- ENEA OSE

2.2.4 Quagga

Overview

Quagga is a routing software package that provides TCP/IP based routing services with routing protocols support. Quagga is an open source routing daemon, distributed under the GNU General Public License. Quagga is a fork of GNU Zebra which was developed by Kunihiro Ishiguro. [QUA] It means that the Quagga is based on the same code base than the ZebOS introduced earlier. This results in many similarities in Quagga and ZebOS.

Architecture

As this routing software is developed from GNU Zebra, the core process is named zebra. The other routing protocols run in their own daemons. The following figure illustrates the architecture of the Quagga routing daemon, along with the associated daemons of RIP, BGP and OSPF [RAM].

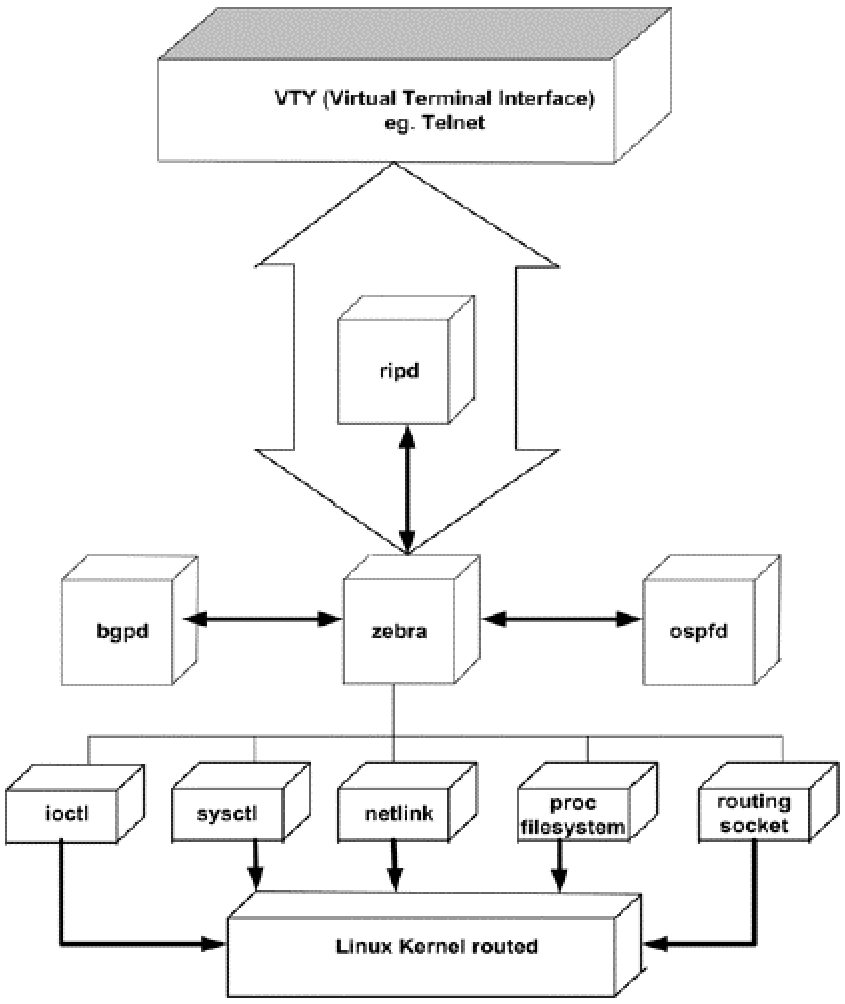


Figure 4: Quagga Architecture [RAM]

The Protocol daemons interact with the kernel routing table using Zebra daemon as the intermediary. Zebra defines its own TCP-based protocol to handle inter-process communication between the Zebra daemon and the protocol daemons. Each protocol daemon sends selected routes to Zebra daemon, which is responsible for interacting, and managing the routes to be installed in the forwarding table. [RAM]

Zebra daemon effectively serves as a moderator for allocation and distribution of services and resources to the various protocol daemons. Each daemon has its own routing table. Zebra daemon maintains the kernel routing table, and is also responsible for redistributing information between the various routing protocol daemons. [RAM]

Routing protocols

Quagga supports RIP v1 and v2, RIPng, OSPFv2, OSPFv3 and BGP. [QUA]

Supported operating systems

Quagga supports gnu/Linux 2.4.x and higher, FreeBSD 4.x and higher, NetBSD 1.6 and higher, OpenBSD 2.5 and higher, Solaris 8 and higher [QUA].

2.3 Feature Comparison

The comparison is based on the above introductions of each routing daemon. According with them, we divide the comparison into three parts: architecture, available routing protocols and supported operating systems. Generally, the more the routing software has features the bigger it is. Bigger software can be more expensive and more difficult to use and configure than a smaller one.

2.3.1 Architecture

Architecture has a significant effect on the design of the software. Architectural similarities as well as differences can be identified from each routing daemon. Different architectures can result in more or less efficient software performance. Architectural design has also a big effect on the software complexity.

We divide the routing daemons into two categories: single process and multiprocess. That property has a big effect on the architecture of the routing software. Based on the traditional GateDaemon codebase, Nokia ipsrd and NextHop GateD belong to the single process category. Similarly, IpInfusion ZebOS and Quagga belong to the multiproces category. The latter two routing daemons are derived from the open source GNU Zebra software, which is designed to have multiple processes. Here we can also see that architectural design decisions have a longstanding status in the software.

As the biggest architectural difference between the routing daemons is the above process division, we can still compare some other properties. Operating systems usually provide an interface to handle the kernel internal data. Due to this design of the operating system kernel, each daemon centralizes the routing messages through a single software module. Multiprocess daemons transfer the routing messages from protocols to the operating system kernel via the *zebra* daemon process. In single process daemons these messages are transferred to the kernel via the core part. The routing software has to share this kind of conformance to external interfaces.

2.3.2 Routing Protocols

The most important features of a routing daemon are of course routing protocols. We compare also them. By comparing the set of routing protocols we can for example see how large networks the router can support.

We have organized the routing protocols to the table below. The table contains also the routing daemons. The routing protocols mentioned in the introductions of the routing daemons are in the left side column of the table. Letter *x* in each row stands for that the routing daemon includes the protocol in question. An empty box means that the routing protocol is not included to the software. Some protocols like MPLS not so important in this thesis are excluded from the table.

Table 2: Routing Protocols in the Different Routing Solutions

	Nokia ipsrd	IpInfusion ZebOS	NextHop GateD	Quagga
IPv4 Protocols				
RIP v1/v2	x	x	x	x
OSPFv2	x	x	x	x
BGP4	x	x	x	x
IS-IS		x	x	
Multicast Protocols				
DVMRP	x	x	x	
PIM-SM	x	x	x	
PIM-DM	x	x	x	
IGMPv2	x	x	x	
IPv6 Protocols				
RIPng	x	x	x	x
OSPFv3	x	x	x	x
BGP4++		x	x	
IS-IS IPv6		x	x	

2.3.3 Operating Systems

By comparing the supported operating systems we can see how portable the software is. The more operating systems the routing software supports the more we have possibilities to run it.

The target operating system for the ported ipsrd is Linux. We measure the routing software in Linux and all the routing daemons support it. That is the most important requirement for us. According to our overview of the supported operating systems, NextHop GateD supports the widest range of different operating systems. Quagga, as open source software, provides the most up-to-date support to the most common open source operating systems. IpInfusion ZebOS supports fewer OSes than GateD. Ipsrd does not support many OSes, only IPSO. Linux is also considered as the supported OS for ipsrd because as result of this thesis we provide a Linux version of ipsrd.

3. Porting Work Requirements

This chapter describes the requirements for the software porting work. The first section gives an overview of the porting work. The second section clarifies the differences between the initial and the goal operating systems.

3.1 Software Porting

Porting software can also be called migrating software to another system. In the beginning, we have an application in our initial operating system. The goal is to make the application compatible with the other operating system we are willing to use. We call this migration process porting. Porting requires adjustments and modifications to the application because the initial and target systems have differences. When we modify the original application according to the differences, we port the application.

Porting work is more effective when we divide the work into different steps. The following subsections describe the different work phases suggested by Imperial Software Technology (IST), a company making commercial software migrating projects.

3.1.1 Before Porting

The difference between migration and development project must be noticed. When we port an application, it does not mean that we rewrite the code. In the migrating process, we try to do the smallest amount of work possible to preserve the code while replacing the needed interfaces. If the porting work has a clear strategy, and moves forward in small, safe steps, the migration has a good chance of finishing on time and within budget. This means that for every change we make, we should be able to prove that we have not broken anything. [IST]

The above advice is very useful. Proceeding with small steps takes time. In case of finding no problems that time could be spent on something else. However, if we can track a fault in our work, it is best to know the fault as soon as possible after making it. Then, the backtracking of the problem will be much faster and easier than it would be in case where we migrate many parts of the application simultaneously.

All in all, the porting work should be prepared well beforehand. That way the work itself is done much faster. The following subsection gives some practical suggestions for preparing for porting.

3.1.2 Preparing for Porting

Migrating the code is the last thing in a migration. The essential work before that is to make the structure safe. This includes also that we make us familiar with the application. The level of detail should be that when we take the application to pieces, we know where every bit goes. The following five-step instructions should cut the migration time at least in half [IST]:

1. Rewrite the project makefiles:

   This helps us to clean our application. Rewriting the makefiles makes us to think what needs to be built and how. This also assists us to be sure what program lines we can drop away from code. It is often possible that we have even dead or obsolete code in our application. This step confirms us if the application includes some code we should get rid of. When we remove unnecessary code, we have also less code lines to check in the porting phase.

2. Move things around:

   Our application can have such a part that is spread over many files. Especially, if that is the part we want to update, we must reorganize our files. [IST] suggests that if we were updating for example the GUI of our application, we could move all GUI code out of files that are 80% or more non-GUI files and move all application code out of files that are 80% or more GUI code files. This way we should get a much cleaner divide.

3. Look for unnecessary use of the API:

    This usually means look for contamination of non-API code with API data structures simply because they are there. In other words, if we do not really need some API structures, we must not use them.

4. Look for generated code:

    All the generated code should always get thrown away. Instead the design file is far easier to migrate to an appropriate design for the new toolkit than map generated code into a new toolkit.

5. Do all the above things before migrating a single line of code:

    The last step notices us to check that after the previous enhancements the application still builds and runs. Shortcutting any of these tasks will risk the migration process later. At least it is possible that we do some unnecessary work.

The above instructions apply partly for our project. IST has made the instructions for general type porting projects. They have the GUI migrating as an example in the instructions. In our case we are not updating any GUI to the application. Instead we replace the low level interfaces. In spite of this, the above model can be used as a reference. The first two steps are well relevant to our case as well. All in all, it is good to recognize the other steps, too.

3.1.3 The Migration

The migration process itself depends heavily on the context of the project. In our case we are just porting the application from FreeBSD to Linux. Both these are UNIX variants, and thus not much difference exists. In addition, our application does not have GUI, which should be updated. This makes the project a little less complex. The differences between FreeBSD and Linux are described in the following sections 3.2 and 3.3. They show us the requirements that our porting work must fulfill.

3.2 Routing in FreeBSD and Linux

Even if FreeBSD and Linux are both UNIX variants, there are routing related differences between them. FreeBSD has a little more complex routing interface between kernel and user than Linux has. Also network interfaces have differences. The following subsections describe more these differences. From the differences we can see the required code modifications in porting work.

3.2.1 Routing in FreeBSD

Routing requires that the network node has multiple network interfaces [MCK]. With multiple interfaces the node can take packets from one and forward them to another interface. The requirement to have multiple interfaces is kind of physical. We have also functional properties. The following description gives a good introduction to FreeBSD routing system.

The routing facilities were designed for use by single homed and multihomed hosts, as well as for routers. There are several components involved in routing, illustrated in the following figure. The design of the routing system places some components within the operating system and others at user level. The routing facilities included in the kernel do not impose routing policies, but instead support a routing mechanism by which externally defined policies can be implemented. By a routing mechanism, we mean a table lookup that provides a first-hop route (a specific network interface and immediate destination) for each destination. [MCK]

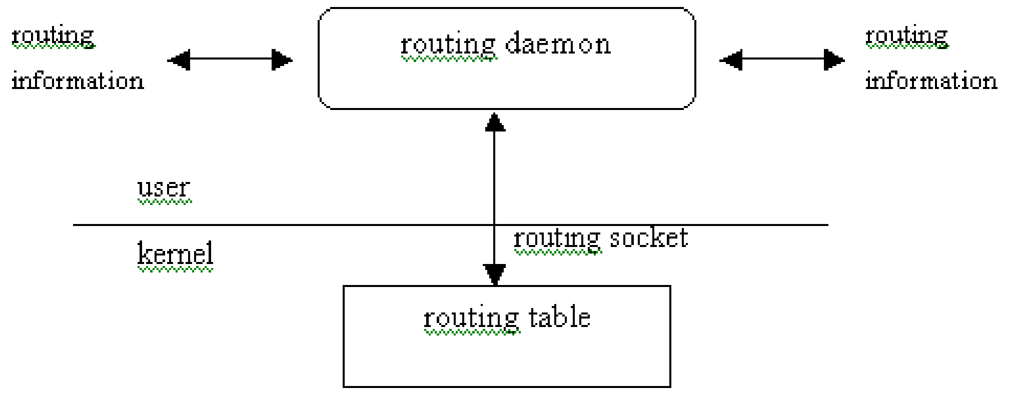


Figure 5: Routing Design [MCK]

Figure 5 shows the FreeBSD routing facilities. These include the kernel level routing table and a routing socket. The user level consists of the routing daemon and routing information next to it. As these components are described in the previous chapters if this thesis, we give here the description of the routing socket. It is not mentioned before.

Routing socket

The user-level routing interface is the *routing socket* [MCK]. It is used to add, delete and change the routes in the kernel routing table. It is the way the routing daemon writes the route data to the kernel. Basically any user-level process that works with a routing table has to use the *routing socket*. Routing daemon is a userland process and sends and receives routing information. Routing table is the database for routing, and it is inside the kernel. Routing socket is the way to transfer information between these two parts.

3.2.2 Routing in Linux

The routing socket mentioned in the previous subsection is replaced in Linux by Linux Netlink [SAL]. Netlink is the messaging system between the kernel and the user space. The routing design figure in the previous subsection applies also to Linux context. Netlink supports many IP services. One of them is the routing service. As this thesis describes routing, we do not describe here the other IP services supported by Netlink.

Linux Netlink has several routing message types. They can be used to create and remove routing related settings in the kernel. We can for example add a route to the kernel routing table with Netlink. Netlink route socket offers us also means to configure, gather statistics and listen to changes in shared resources [SAL]. These resources include IP addresses and network interfaces.

3.2.3 Routing Differences between FreeBSD and Linux

Based on the previous two subsections, there are not many differences between Linux and FreeBSD. The most noticeable difference is that in FreeBSD we have message types to change for example nexthop for a route already in the routing table. Linux does not provide such a message type. Another difference is that Linux Netlink supports multiple IP services when the BSD routing socket is only for routing purposes.

3.2.4 Interface Types in FreeBSD and Linux

As mentioned above, interface changes are handled in both FreeBSD and Linux by the routing socket between the user space and the kernel. That is why we also take a look at the interfaces in these operating systems. The differences in them have an influence on the porting work itself.

FreeBSD network interfaces are organized hierarchically in four layers: physical interface, logical interface, address family and address. On the other hand, Linux uses a two layer hierarchical interface structure: link and address. This mapping has to be done in the porting work.

4. Measuring Software Complexity

There are several attributes we can use to measure software complexity. Complexity measurement is a static way of measuring software. In these measurements we do not need to run the software. Instead we only browse through the source files and check them in various ways. The different measurement ways are introduced in this chapter. Software Engineering Institute of Carnegie Mellon University has published excellent descriptions of these metrics in their web pages [SEI]. We use these pages as a reference in this chapter.

4.1 Lines-of-code Metrics

The lines-of-code measures are the most traditional measures used to quantify software complexity. They are simple, easy to count, and very easy to understand. They do not, however, take into account the intelligence content and the layout of the code. [VER]

The lines can be calculated from the source code in different ways. In this thesis we are using the following lines-of-code metrics:

- LOCphy: number of physical lines
- LOCbl: number of blank lines (a blank line inside a comment block is considered to be a comment line)
- LOCpro: number of program lines (declarations, definitions, directives, and code)
- LOCcom: number of comment lines

The following recommendations are given for the lines-of-code measures: [VER]

Function length should be 4 to 40 program lines. A function definition contains at least a prototype, one line of code, and a pair of braces, which makes 4 lines. A function longer than 40 program lines probably implements many functions. Functions containing one selection statement with many branches are an exception to this rule. Decomposing them into smaller functions often decreases readability.

File length should be 4 to 400 program lines. The smallest entity that may reasonably occupy a whole source file is a function, and the minimum length of a function is 4 lines. Files longer than 400 program lines (10..40 functions) are usually too long to be understood as a whole.

At least 30 percent and at most 75 percent of a file should be **comments**. If less than one third of a file is comments the file is either very trivial or poorly explained. If more than 75% of a file are comments, the file is not a program but a document. In a well-documented header file percentage of comments may sometimes exceed 75%.

4.2 McCabe's Cyclomatic Number

Cyclomatic complexity is the most widely used member of a class of static software metrics. Cyclomatic complexity may be considered a broad measure of soundness and confidence for a program. Introduced by Thomas McCabe in 1976, it measures the number of linearly-independent paths through a program module. This measure provides a single ordinal number that can be compared to the complexity of other programs. Cyclomatic complexity is often referred to simply as program complexity, or as McCabe's complexity. It is often used in concert with other software metrics. As one of the more widely-accepted software metrics, it is intended to be independent of language and language format. [CYC]

As the cyclomatic number makes us an easy way to compare software, we have taken it also to this thesis. By comparing this number we can see the relative complexity of the whole software. Below, we have more careful introduction to this subject. The descriptions are from our reference pages [CYC].

4.2.1 Techical Detail

The cyclomatic complexity of a software module is calculated from a connected graph of the module (that shows the topology of control flow within the program):

Cyclomatic complexity (CC) = E - N + p

where E = the number of edges of the graph

N = the number of nodes of the graph

p = the number of connected components

To actually count these elements requires establishing a counting convention, i.e. tools to count cyclomatic complexity contain these conventions. The complexity number is generally considered to provide a stronger measure of a program's structural complexity than is provided by counting lines of code. The following figure is a connected graph of a simple program with a cyclomatic complexity of seven. Nodes are the numbered locations, which correspond to logic branch points; edges are the lines between the nodes. The conditions increasing the cyclomatic complexity are the nodes that have more than one path beginning from it.

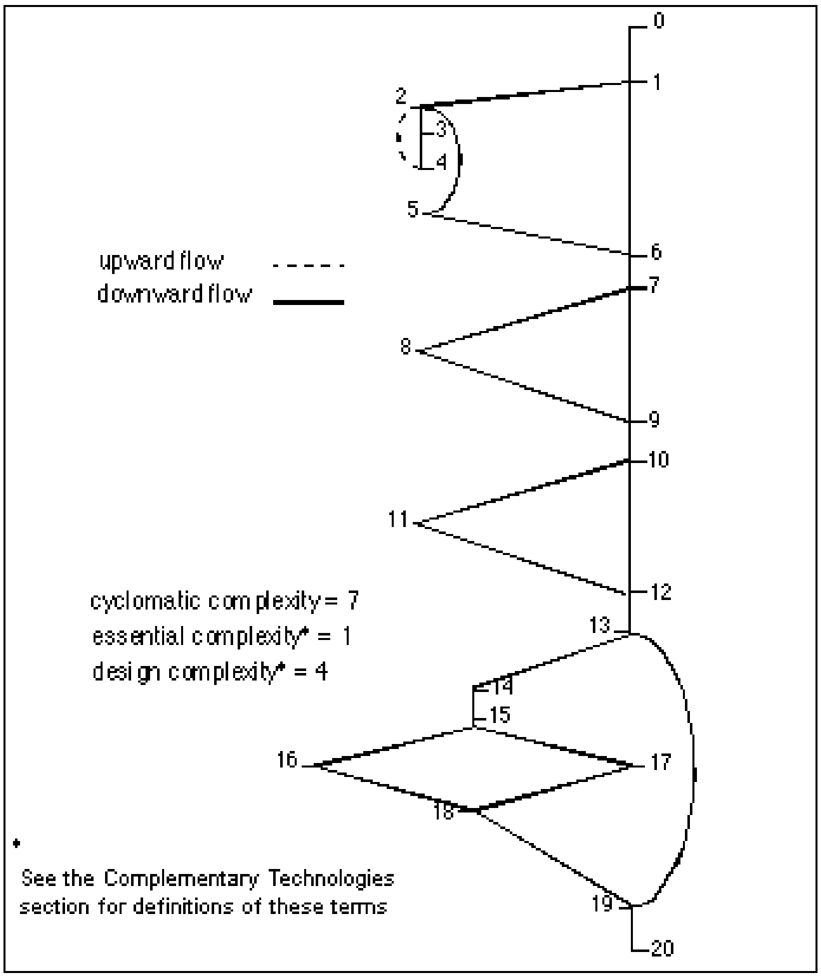

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
upward flow
downward flow
cyclomatic complexity = 7
essential complexity* = 1
design complexity* = 4
*
See the Complementary Technologies section for definitions of these terms

Figure 6: Connected Graph of a Simple Program [CYC]

In this thesis we calculate the cyclomatic complexity on the entire source files. Basically each conditional statement increases the cyclomatic complexity by one. These language constructs are: if (...), for (...), while (...), case ..., catch (...), &&, ||, ?, #if, #ifdef, #ifndef, #elif. It should be noted that the cyclomatic complexity is insensitive to unconditional branches like goto-, return- and break-statements although they surely increase the complexity [VER].

A large number of programs have been measured, and ranges of complexity have been established that help the software engineer determine a program's inherent risk and stability. The resulting calibrated measure can be used in development, maintenance, and reengineering situations to develop estimates of risk, cost, or program stability. Studies show a correlation between a program's cyclomatic complexity and its error frequency. A low cyclomatic complexity contributes to a program's understandability and indicates that it is amenable to modification at a lower risk than a more complex program. A module's cyclomatic complexity is also a strong indicator of its testability.

A common application of cyclomatic complexity is to compare it against a set of threshold values. One such threshold set is in the table below:

Table 3: Cyclomatic Complexity [CYC]

Cyclomatic Complexity	**Risk Evaluation**
1-10	a simple program, without much risk
11-20	more complex, moderate risk
21-50	complex, high risk program
greater than 50	untestable program (very high risk)

4.2.2 Usage Considerations

Cyclomatic complexity can be applied in several areas, including

- *Code development risk analysis*. While code is under development, it can be measured for complexity to assess inherent risk or risk buildup.

- *Change risk analysis in maintenance*. Code complexity tends to increase as it is maintained over time. By measuring the complexity before and after a proposed change, this buildup can be monitored and used to help to decide how to minimize the risk of the change.

- *Test Planning*. Mathematical analysis has shown that cyclomatic complexity gives the exact number of tests needed to test every decision point in a program for each outcome. Thus, the analysis can be used for test planning. An excessively complex module will require a prohibitive number of test steps; that number can be reduced to a practical size by breaking the module into smaller, less-complex sub-modules.

- *Reengineering*. Cyclomatic complexity analysis provides knowledge of the structure of the operational code of a system. The risk involved in reengineering a piece of code is related to its complexity. Therefore, cost and risk analysis can benefit from proper application of such an analysis.

Cyclomatic complexity can be calculated manually for small program suites, but automated tools are preferable for most operational environments. For automated graphing and complexity calculation, the technology is language-sensitive; there must be a front-end source parser for each language, with variants for dialectic differences.

Cyclomatic complexity is usually only moderately sensitive to program change. Other measures (see Complementary Technologies) may be very sensitive. It is common to use several metrics together, either as checks against each other or as part of a calculation set.

4.2.3 Maturity

Cyclomatic complexity measurement, an established but evolving technology, was introduced in 1976. Since that time it has been applied to tens of millions of lines of code in both Department of Defense (DoD) and commercial applications. The resulting base of empirical knowledge has allowed software developers to calibrate measurements of their own software and arrive at some understanding of its complexity. Code graphing and complexity calculation tools are available as part (or as options) of several commercial software environments.

4.2.4 Costs and Limitations

Cyclomatic complexity measurement tools are typically bundled inside commercially-available CASE toolsets. It is usually one of several metrics offered. Application of complexity measurements requires a small amount of training. The fact that a code module has high cyclomatic complexity does not, by itself, mean that it represents excess risk, or that it can or should be redesigned to make it simpler; more must be known about the specific application.

4.2.5 Alternatives

Cyclomatic complexity is one measure of structural complexity. Other metrics bring out other facets of complexity, including both structural and computational complexity, as shown in the table below.

Table 4: Other Facets of Complexity [CYC]

Complexity Measurement	**Primary Measure of**
Halstead Complexity Measures	Algorithmic complexity, measured by counting operators and operands
Henry and Kafura metrics	Coupling between modules (parameters, global variables, calls)

Bowles metrics	Module and system complexity; coupling via parameters and global variables
Troy and Zweben metrics	Modularity or coupling; complexity of structure (maximum depth of structure chart); calls-to and called-by
Ligier metrics	Modularity of the structure chart

4.2.6 Complementary Technologies

The following three metrics are specialized measures that are used in specific situations:

1. *Essential complexity*. This measures how much unstructured logic exists in a module (e.g., a loop with an exiting GOTO statement).
2. The program in Figure 6 has no such unstructured logic, so its essential complexity value is one.
3. *Design complexity*. This measures interaction between decision logic and subroutine or function calls.
4. The program in Figure 6 has a design complexity value of 4, which is well within the range of desirability.
5. *Data complexity*. This measures interaction between data references and decision logic.

Other metrics that are "related" to Cyclomatic complexity in general intent are also available in some CASE toolsets.

The metrics listed in Alternatives are also complementary; each metric highlights a different facet of the source code.

4.3 Halstead's Metrics

Halstead complexity measurement was developed to measure a program module's complexity directly from source code, with emphasis on computational complexity. The measures were developed by the late Maurice Halstead as a means of determining a quantitative measure of complexity directly from the operators and operands in the module. Among the earliest software metrics, they are strong indicators of code complexity. Because they are applied to code, they are most often used as a maintenance metric. There are widely differing opinions on the worth of Halstead measures, ranging from "convoluted... [and] unreliable" to "among the strongest measures of maintainability". The material in this technology description is largely based on the empirical evidence found in the Maintainability Index work, but there is evidence that Halstead measures are also useful during development, to assess code quality in computationally-dense applications. [HAL]

4.3.1 Technical Detail

The Halstead measures are based on four scalar numbers derived directly from a program's source code:

n1 = the number of distinct operators

n2 = the number of distinct operands

N1 = the total number of operators

N2 = the total number of operands

From these numbers, five measures are derived:

Measure	**Symbol**	**Formula**
Program length	N	N= N1 + N2
Program vocabulary	n	n= n1 + n2

Volume	V	V= N * (LOG2 n)
Difficulty	D	D= (n1/2) * (N2/n2)
Effort	E	E= D * V

These measures are simple to calculate once the rules for identifying operators and operands have been determined. The extraction of the component numbers from code requires a language-sensitive scanner, which is a reasonably simple program for most languages.

4.3.2 Usage Considerations

Applicability

The Halstead measures are applicable to operational systems and to development efforts once the code has been written. Because maintainability should be a concern during development, the Halstead measures should be considered for use during code development to follow complexity trends. A significant complexity measure increase during testing may be the sign of a brittle or high-risk module. Halstead measures have been criticized for a variety of reasons, among them the claim that they are a weak measure because they measure lexical and/or textual complexity rather than the structural or logic flow complexity exemplified by Cyclomatic Complexity measures. However, they have been shown to be a very strong component of the Maintainability Index measurement of maintainability. In particular, the complexity of code with a high ratio of calculational logic to branch logic may be more accurately assessed by Halstead measures than by Cyclomatic Complexity, which measures structural complexity.

Relation to other complexity measures

Marciniak describes all of the commonly-known software complexity measures and puts them in a common framework [MAR]. This is helpful background for any complexity measurement effort. Most measurement programs benefit from using several measures, at least initially; discarding those that do not suit the specific environment; and combining those that work (see subsection Complementary Technologies).

4.3.3 Maturity

Halstead measures were introduced in 1977 and have been used and experimented with extensively since that time. They are one of the oldest measures of program complexity. Because of the criticisms mentioned above, they have seen limited use. However, their properties are well-known and, in the context explained in Usage Considerations, they can be quite useful.

4.3.4 Costs and Limitations

The algorithms are free; the tool described in Technical Detail, contains Halstead scanners for Pascal and C, and some commercially-available CASE toolsets include the Halstead measures as part of their metric set. For languages not supported, standalone scanners can probably be written inexpensively, and the results can be exported to a spreadsheet or database to do the calculations and store the results for use as metrics. It should be noted that difficulties sometimes arise in uniquely identifying operators and operands. Adding Halstead measures to an existing maintenance environment's metrics collection effort and then applying them to the software maintenance process will require not only the code scanner, but a collection system that feeds the resulting data to the metrics effort. Halstead measures may not be sufficient by themselves as software metrics (see subsection Complementary Technologies).

4.3.5 Alternatives

A common practice today is to combine measures to suit the specific program environment. Most measures are amenable for use in combination with others (although some overlap). Thus, many alternative measures are to some degree complementary. Oman presents a very comprehensive list of code metrics that are found in maintainability analysis work, and orders them by degree of influence on the maintainability measure being developed in that effort [OMA]. Some examples are (all are averages across the set of programs being measured)

- lines of code per module
- lines of comments per module
- variable span per module
- lines of data declarations per module

4.3.6 Complementary Technologies

Cyclomatic Complexity and its associated complexity measures measure the structural complexity of a program. Maintainability Index technique for measuring program maintainability combines cyclomatic complexity with Halstead measures to produce a practical measure of maintainability.

Function point measures provide a measure of functionality, with some significant limitations (at least in the basic function point enumeration method); the variant called engineering function points adds measurement of mathematical functionality that may complement Halstead measures.

Lines-of-code (LOC) metrics offer a gross measure of code, but do not measure content well. However, LOC in combination with Halstead measures may help relate program size to functionality.

4.4 Maintainability Index

Quantitative measurement of an operational system's maintainability is desirable both as an instantaneous measure and as a predictor of maintainability over time. Efforts to measure and track maintainability are intended to help reduce or reverse a system's tendency toward "code entropy" or degraded integrity, and to indicate when it becomes cheaper and/or less risky to rewrite the code than to change it. *Software Maintainability Metrics Models in Practice* is the latest report from an ongoing, multi-year joint effort (involving the Software Engineering Test Laboratory of the University of Idaho, the Idaho National Engineering Laboratory, Hewlett-Packard, and other companies) to quantify maintainability via a Maintainability Index (MI) [WEL]. Measurement and use of the MI is a process technology, facilitated by simple tools, that in implementation becomes part of the overall development or maintenance process. These efforts also indicate that MI measurement applied during software development can help reduce lifecycle costs. The developer can track and control the MI of code as it is developed, and then supply the measurement as part of code delivery to aid in the transition to maintenance.

4.4.1 Technical Detail

In the MI technology, a program's maintainability is calculated using a combination of widely-used and commonly-available measures to form a MI. The basic MI of a set of programs is a polynomial of the following form (all are based on average-per-code-module measurement):

$$171 - 5.2 * \ln(aveV) - 0.23 * aveV(g') - 16.2 * \ln(aveLOC) + 50 * \sin(\text{sqrt}(2.4 * perCM))$$

The coefficients are derived from actual usage (see Usage Considerations). The terms are defined as follows:

aveV = average Halstead Volume V per module

aveV(g') = average extended cyclomatic complexity per module

aveLOC = the average count of lines of code (LOC) per module; and, optionally

perCM = average percent of lines of comments per module

The module is, in this thesis, a function or a struct definition in a single source code file. In this thesis we measure the source code files. The average is calculated over the number of these modules in a single file.

Oman develops the MI equation forms and their rationale [O92]; the Oman study indicates that the above metrics are good and sufficient predictors of maintainability. Oman builds further on this work using a modification of the MI and describing how it was calibrated for a specific large suite of industrial-use operational code [OMA]. Oman describes a prototype tool that was developed specifically to support the capture and use of maintainability measures for Pascal and C [O91]. The aggregate strength of this work and the underlying simplicity of the concept make the MI technique potentially very useful for operational Department of Defense systems.

4.4.2 Usage Considerations

Calibration of the equations.

The coefficients shown in the equation are the result of calibration using data from numerous software systems being maintained by Hewlett-Packard. The authors claim that follow-on efforts show that this form of the MI equation generally fits other industrial-sized software systems [OMA and WEL], and the breadth of the work tends to support this claim. It is advisable to test the coefficients for proper fit with each major system to which the MI is applied.

Effects from comments in code.

The user must analyze comment content and quality in the specific system to decide whether the comment term perCM is useful.

4.4.3 Ways of Using MI

1. The system can be checked periodically for maintainability, which is also a way of calibrating the equations.

2. It can be integrated into a development effort to screen code quality as it is being built and modified; this could yield potentially significant life cycle cost savings.

3. It can be used to drive maintenance activities by evaluating modules either selectively or globally to find high-risk code.

4. MI can be used to compare or evaluate systems: Comparing the MIs of a known-quality system and a third-party system can provide key information in a make-or-buy decision.

4.4.4 Maturity

Oman tested the MI approach by using production operational code containing around 50 000 lines of code to determine the metric parameters, and by checking the results against subjective data gathered using the 1989 AFOTEC maintainability evaluation questionnaire [AFO, OMA]. Other production code of about half that size was used to check the results, with apparent consistency.

Welker applied the results to analyses of a US Air Force (USAF) system, the Improved Many-On-Many (IMOM) electronic combat modeling system. The original IMOM (in FORTRAN) was translated to C and the C version was later reengineered into Ada. The maintainability of both newer versions was measured over time using the MI approach [WEL]. Results were as follows:

- The reengineered version's MI was more than twice as high as the original code (larger MI = more maintainable), and declined only slightly over time (note that the original code was not measured over time for maintainability, so change in its MI could not be measured).
- The translated baseline's MI was not significantly different from the original. This is of special interest to those considering translation, because one of the primary objectives of translation is to reduce future maintenance costs. There was also evidence that the MI of translated code deteriorates more quickly than reengineered code.

4.4.5 Costs and Limitations

Calculating the MI is generally simple and straightforward, given that several commercially-available programming environments contain utilities to count code lines, comment lines, and even Cyclomatic Complexity. Other tools to calculate Halstead Complexity Measures are less common than the one described in Oman [O91] because the measure is not used as widely. However, once conventions for the counting have been established, it is generally not difficult to write language-specific code scanners to count the Halstead components (operators and operands) and calculate the E and V measures. In relating that removal of unused code in a single module did not affect the MI, Pearse highlights the fact that MI is a system measurement; its parameters are average values [PEA]. However, measuring the MI of individual modules is useful because changes in either structural or computational complexity are reflected in a module's MI. A product/process measurement program not already gathering the metrics used in MI could find them useful additions. Those metrics already being gathered may be useful in constructing a custom MI for the system. However, it would be advisable to consult the references for their findings on the effectiveness of metrics, other than Halstead E and V and cyclomatic complexity, in determining maintainability.

4.4.6 Dependencies

The MI method depends on the use of Cyclomatic Complexity and Halstead Complexity Measures. To realize the full benefit of MI, the maintenance environment must allow the rewriting of a module when it becomes measurably unmaintainable. The point of measuring the MI is to identify risk; when unacceptably risky code is identified, it should be rewritten.

4.4.7 Alternatives

The process described by Sittenauer is designed to assist in deciding whether or not to reengineer a system [SIT]. There are also many research and analytic efforts that deal with maintainability as a function of program structure, design, and content, but none was found that was as clearly appropriate as MI to current Department of Defense systems.

4.4.8 Complementary Technologies

The MI technology takes into account many metrics. Thus, there are not many complementary technologies for the MI. The test in Sittenauer is meant to verify generally the condition of a system, and would be useful as a periodic check of a software system and for comparison with the MI [SIT].

4.5 Software Complexity Measurements

This section introduces the software complexity measurements. In the beginning we describe the tool used for the measuring. Then we select some complexity metrics we are checking from each routing daemon.

4.5.1 Measurement Tool

The measurements are performed with the CMT++ tool from Verifysoft [VER]. This tool calculates McCabe’s cyclomatic number, Lines-of-code metrics, Halstead’s metrics and Maintainability Index. As we have discussed, these metrics combined are a good indication of the quality of software. We compare the routing daemons also with these metrics.

CMT++ browses through all the code and gathers the metric values from the files. We collect the complexity measurement results to Excel sheets and calculate also the relative values. We can also utilize the Lines-of-code metrics to calculate the average values. This helps us to compare the software of different size.

4.5.2 Selected Complexity Metrics for Comparison

We compare the routing daemons primarily with the Maintainability Index. It takes into account many other metrics as well. As MI is quite an abstract number, we also compare several other metrics and their combinations. From the Lines-of-code metrics we compare the file length and the percentage of comments. From the other metrics, we compare the cyclomatic number v(G) and information volume V.

All the metrics are measured from each file of the routing software. We compare the result of each file to the recommended values discussed in chapter 4. Then we check the result values if they are inside the recommendations or if they are too much or too small. This produces us the number of files in each category. To compare different routing software, we divide the amount of files in each category by the total number of files in the routing software in question. The final output is a relative value which can easily be compared among the different routing solutions.

5. Porting Work

This chapter describes the work done in porting. After the porting work description we take a look at the encountered problems. Finally, in the next chapter we validate that our porting is successful. The porting starts with the FreeBSD version of routing daemon *ipsrd*. The goal is to have that software running and performing routing tasks in Linux. The subsections below introduce the steps performed during the porting work. This chapter gives a basis for the routing daemon comparison in the following chapters. Because we do not initially have the routing daemon ipsrd running in Linux, the porting work is required to make that possible.

5.1 Porting of Ipsrd

The very first task we do is to take the source code of the routing software from FreeBSD to Linux environment. As this is done, we rewrite the makefiles as recommended in [IST]. The next step according to the instructions is to move things around. We do not need to reorganize our code because we decide to keep the sources as much unchanged as possible. That would cause fewer problems. We do not follow the other instructions of [IST] either. Those are not applicable to our project. We do not have for example generated code in the routing software.

5.1.1 Modules

Clearly the biggest part of the code in the modules can be as they are. We do not need to touch other places than those which were directly connected to the kernel. The need to modify the code lines related to the kernel is due to the different interfaces described in more detail in chapter 3.

The way we do the porting is by creating a new layer between the Linux kernel and the routing daemon. The layer converts the *route* routing socket messages from ipsrd to *Netlink* route socket messages for the Linux kernel. Also the rest of the differences are translated. The layer converts the network interface socket traffic to the compatible format for both the ipsrd and the Linux kernel. As the routing interface specifications are publicly available with the source codes of FreeBSD and Linux, we do not include them into this thesis.

5.1.2 Referenced Libraries

As FreeBSD and Linux have mostly the same libraries, we just need to do a few adjustments to the source code. The differing libraries are related to the routing sockets. That is why we can remove include statements for FreeBSD route socket, and add the corresponding Linux Netlink socket include statements instead.

5.1.3 Needed Actions to Run the Process in Linux

After the needed adjustments are ready, we can compile the program. Compilation goes well when the software is compatible with the system it is compiled in. The problems during the compilation should be solved, otherwise the program does not compile at all. The problems encountered in porting are discussed below.

Due to the design of the routing daemon we have to give a configuration file as an argument when we start the process. When the configuration file is formatted right, we can start the process with no pain.

5.2 Encountered Porting Problems

This subsection discusses the problems encountered in the porting work. The problems can be used to track typical errors the developers do, not only in porting but also in regular software development.

5.2.1 Compilation Errors

The compilation errors are not always a bad thing. In our project we made a good use of these errors. They showed usually the place where we had forgotten to make an update in the porting work. Of course some errors put us to really think what we can do with the error. For example having included some FreeBSD library instead of the needed one from Linux caused difficult problems.

Notable in the compiler output was also quite a big amount of warnings. Those warnings were usually created because of deprecated casting on left values. For some reason the related warning was not shown in the FreeBSD compilation.

5.2.2 Solutions for the Errors

All the compiler errors were successfully corrected. Sometimes it required a search from the Internet to see from which library some variables are referenced. Even if we were able to solve all the errors by browsing the source code, we did that rather rarely. Linux has also good command line programs for searching a pattern from a file. We were using those standard programs very much.

5.2.3 Kernel Assertion Failures in the Beginning

Ipsrd is designed to check that the internal data of the process is correct. Therefore, there are such places in the code where the values and variables are checked with an assertion function. If this check fails, the process is terminated. Especially in the beginning we got some assertion failures. A typical reason for a failure was the same as earlier; we had forgotten to port something. It was still good that we got these failures, as we could correct them already during the development phase.

6. Routing Daemon Performance

This chapter compares the performance of the routing daemons investigated in this thesis. They are Nokia ipsrd, IpInfusion ZebOS, NextHop GateD and open source Quagga. The comparison is made by testing the routing daemons in an identical test setup. That is described in the following section.

6.1 Test Environment

We use the same hardware and Linux version in all tests. This makes the results comparable among the routing daemons. The test network is identical as well. This section explains the test setup.

We use the Nokia IP740i hardware. It is a standard rack mounted computer. IP740i can be used for easy installation of Linux and the routing software. The used Linux distribution is Red Hat Enterprise Linux 4.0 (RHEL) with kernel version 2.6.14. The additional software consists of each routing daemon in turn. The testing of Quagga is done with a Fedora Core Linux kernel version 2.6.16. This kernel is newer than the one used in the other tests. The testing for Quagga was done first. We had the different kernel version at that point.

The network setup includes the test machine IP740i, IXIA [IXI] traffic generator/analyzer, a router, a switch and a PC. The PC is used for configuration. The test network setup is illustrated in figure 7. The router in the figure can be any OSPF and RIP capable router. We used an IPSO router here.

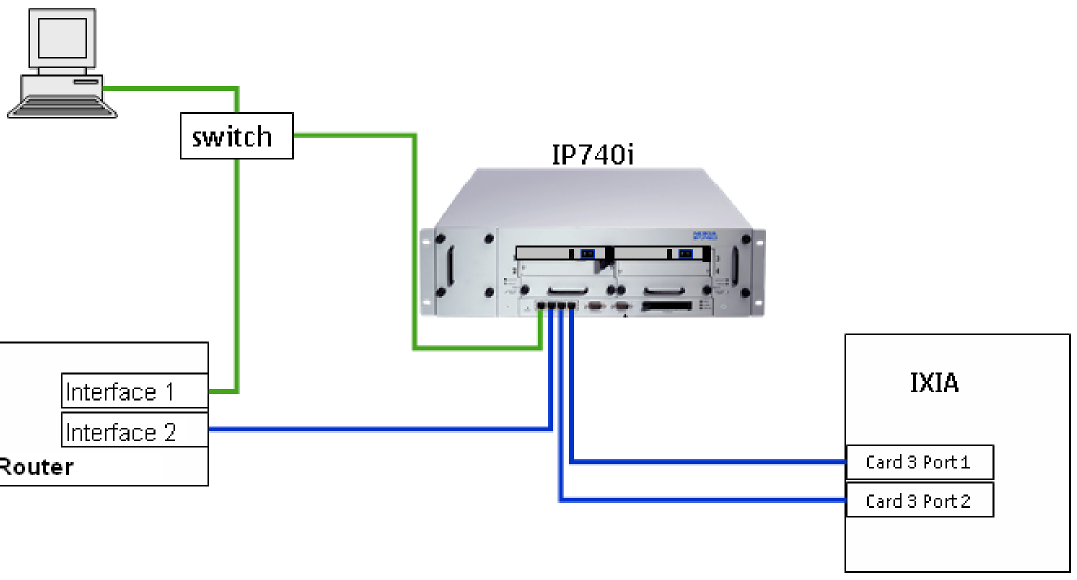


Figure 7: Test Network

6.2 Test Contents

This section describes the router features we are testing. We are mainly testing the performance of the routing daemons. The performance tests include the memory usage of the routing daemons with a changing number of OSPF and RIP routes. We insert also 100000 routes to the kernel through the routing daemon and measure the time it takes to complete that. The last test measures the OSPF convergence time with a varying number of routes. The following sections describe these tests.

6.2.1 Inserting 100000 Routes

This is the simplest test. To see how fast the routing daemon can put the routes to the kernel, we use the OSPF protocol to insert 100000 routes at once. Then we measure the time it takes to put all the routes to the kernel routing table. Finally we can compare the results and see how fast the routing daemons are in this kind of quick add.

6.2.2 Memory Usage

Memory usage is tested with both OSPF and RIP protocols. We start from 10000 routes, and add 10000 routes more at a time until we have a total of 100000 routes installed. During this we measure the memory usage of the routing daemon. This helps to see how efficiently the routing daemon can use the operating system memory.

6.2.3 OSPF Convergence

Route convergence test measures how fast the router can adapt to the network state changes. This test is performed using OSPF and testing the convergence time with three different route amounts: 100, 1000 and 10000. The IXIA traffic generator inserts the wanted amount of routes to the router under test. It uses the OSPF protocol and gives the same routes via two different networks. The other network has a smaller cost for the routes. When the tested routing daemon has updated the information and has set the default routes to go via the cheaper network, the IXIA removes these routes suddenly from the cheaper network. Finally, the routing daemon has updated its routing database and forwards the traffic via the only connected network. The time from the removal of the original routes to the complete updating of the new routes is the measured OSPF convergence time.

7. Results and Suggestions

This chapter presents the results of this thesis. Firstly, we describe the results of the performance comparison. Secondly, we present the complexity measurement results. After that, we analyze the ipsrd results compared with the other routing solutions. Finally we think about this thesis and give our own opinion of this work and suggestions for further investigation. This chapter concludes the thesis.

7.1 The Performance of the Routing Daemons

This section presents the results of the performance tests described in chapter 6. We present the result of each test in its own section. They are: Inserting 100000 Routes, Memory Usage and OSPF Convergence.

7.1.1 Inserting 100000 Routes

The simplest test has the shortest results. The results can be found in the following table. Nokia ipsrd has a known problem of inserting 100000 OSPF routes. This is why the corresponding result in the table is N/A.

Table 5: Time to Insert 100000 OSPF Routes

	Nokia ipsrd	**IpInfusion ZebOS**	**NextHop Gated**	**Quagga**
time (s)	N/A	539	567	550

The results show that with these values ZebOS seems to perform the best in this test. However, all the result times are between 539 and 567 seconds. This is from 8 minutes 59 seconds to 9 minutes 27 seconds. The delay of 28 seconds in about 9 minutes is still quite small.

7.1.2 Memory Usage

We divide the memory usage results into OSPF and RIP parts. This follows the testing we performed. The next figures illustrate the memory usage in both cases. For ipsrd we tested OSPF routes until 60000. For Quagga we did not test the memory usage with RIP routes as the version we tested did not support that many RIP routes. The supported number of RIP routes was only about 2000 for an interface in Quagga.

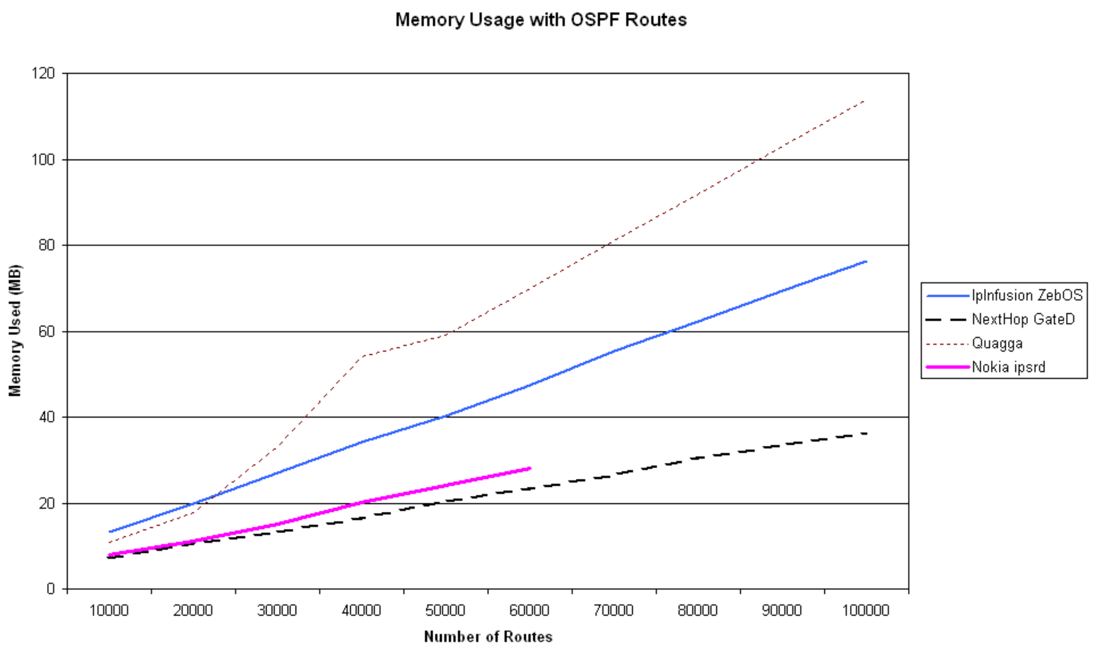


Figure 8: Memory Usage with OSPF Routes

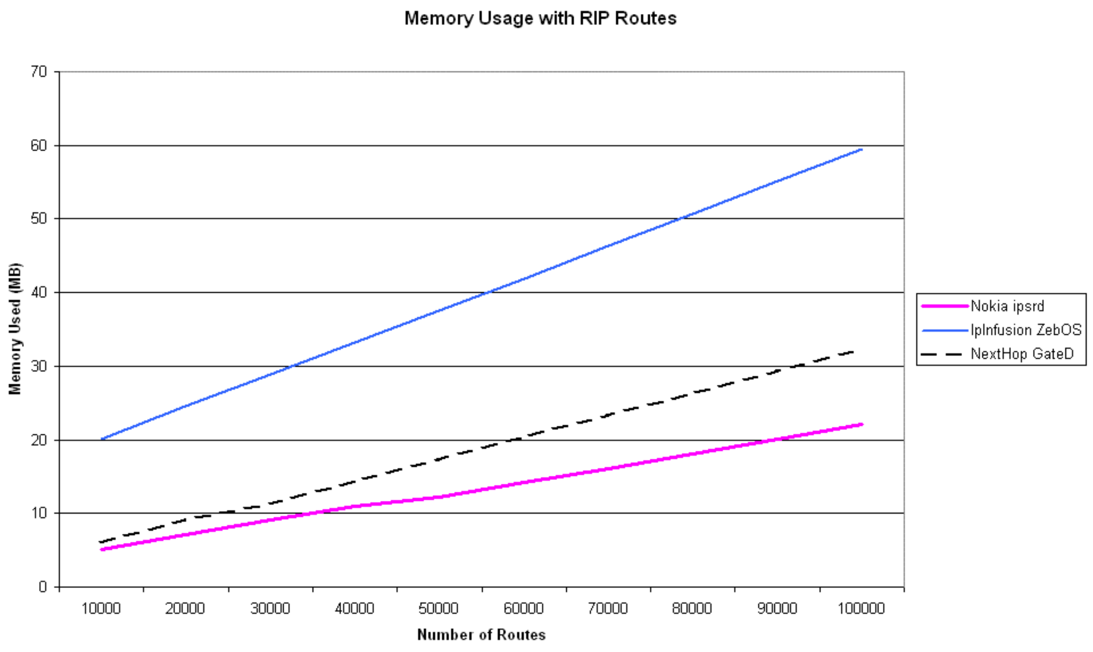


Figure 9: Memory Usage with RIP Routes

From these figures we can see that the software with multiple processes uses clearly more memory than the ones with a single process design. Both Quagga and ZebOS are above ipsrd and GateD in the memory usage results. We can see that the memory usage is increasing in a linear fashion when the number of routes grows bigger.

7.1.3 OSPF Convergence

The OSPF convergence test results show us the quickness of the routing solutions to apply the new settings of the network. The following table presents the results for this test. We show three different cases with a different number of routes. The time is measured in seconds.

Table 6: OSPF Convergence Time Results

	Nokia ipsrd	IpInfusion ZebOS	NextHop Gated	Quagga
Routes	**time (s)**	**time (s)**	**time (s)**	**time (s)**
100	8.70	5.81	10.09	1.63
1000	9.57	7.50	15.68	3.07
10000	33.62	13.61	34.02	10.93

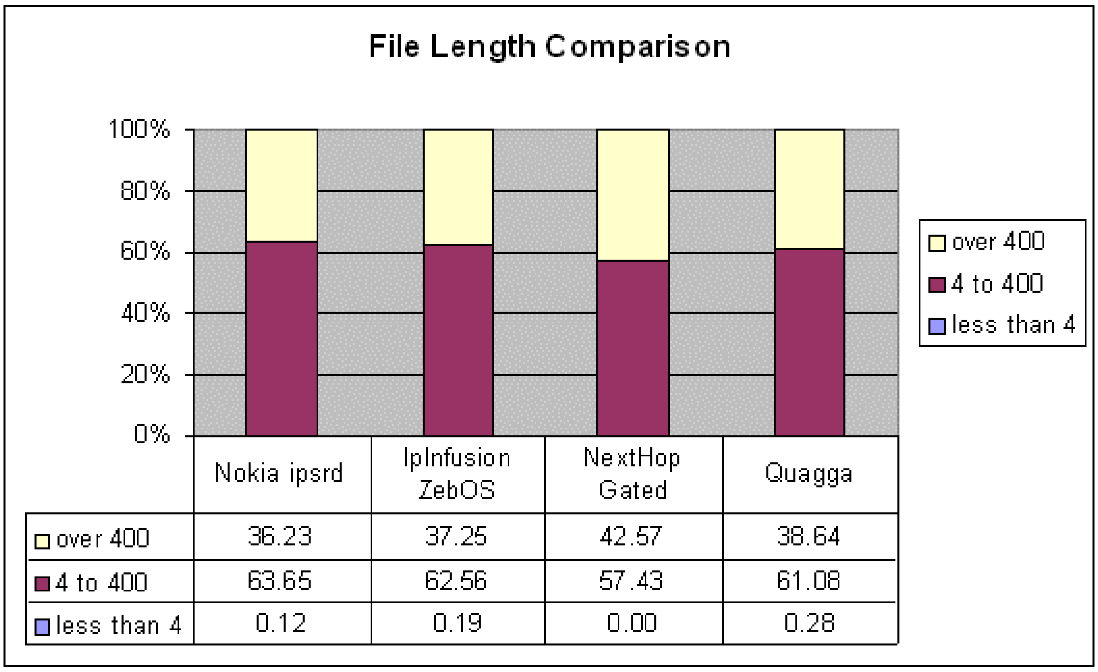
The results show that the fastest in this test was Quagga. ZebOS comes next. The division to the multiprocess and single process architecture can be noticed from these results. The multiprocess architecture enables the routing daemon to calculate the needed next hops faster than the single process daemons do. Ipsrd is a slightly faster than GateD. However, comparing the ipsrd and GateD results to the corresponding results of Quagga and ZebOS, we notice an amazing performance gap with a big number of routes.

7.2 Complexity Measurement Results

The results from the complexity measurements are presented in this section. The results are divided so that we present each metric in a different subsection. We compare the metric results between each routing solution. These results have a strong influence on how difficult the software is to understand and modify. For these results we have used a similar format in all metrics. All the result values are presented as percents.

7.2.1 File Length

The recommended values used in the comparison are introduced in chapter 4. We put them also here for completeness. The file length should be from 4 to 400 lines. The figure below presents the proportion of files in each routing solution in the recommended range and outside that.



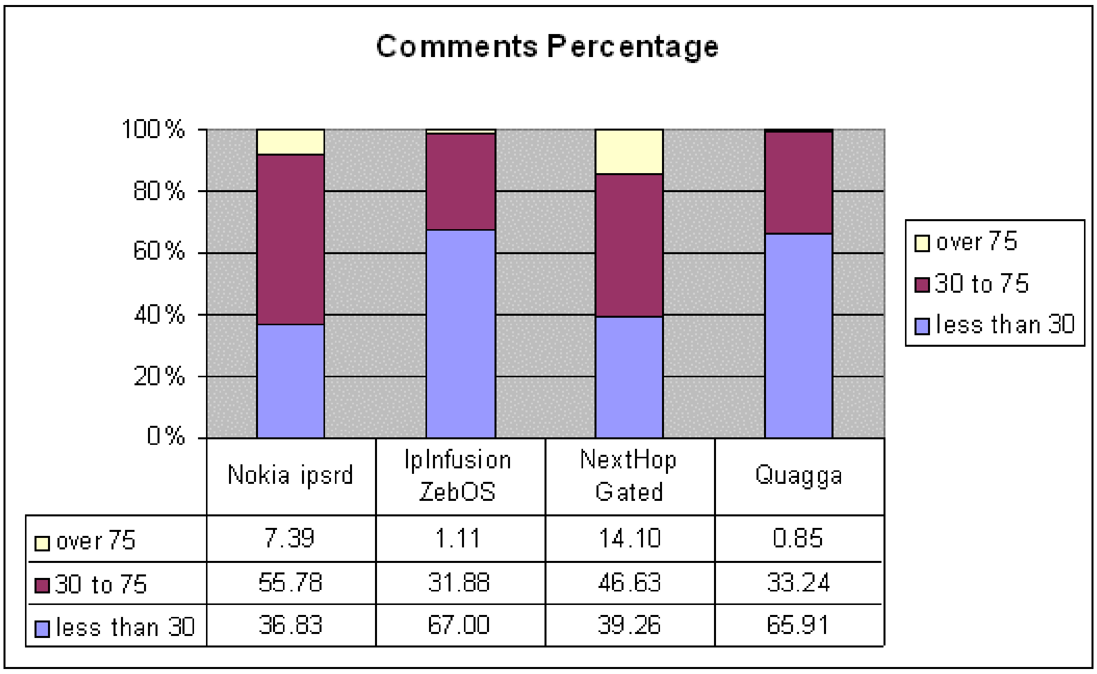
	Nokia ipsrd	IpInfusion ZebOS	NextHop Gated	Quagga
over 400	36.23	37.25	42.57	38.64
4 to 400	63.65	62.56	57.43	61.08
less than 4	0.12	0.19	0.00	0.28

Figure 10: File Length Comparison

All the routing daemons seem to have about 60% of all files in the recommended range. GateD has little more long files than the other programs. The proportion of very short files is near zero. This comparison does not result in any big difference between the routing daemons. The GateD has more (5%) too long files than the other solutions. Thus, GateD is the worst in this comparison. The other routing solutions are on about the same level.

7.2.2 Comments Percentage

The percentage of comments in the source code files should show how well the code is explained. The comment percentage should be between 30% and 75%. These values are used as reference limits to the result values. The following figure presents the results of this calculation.



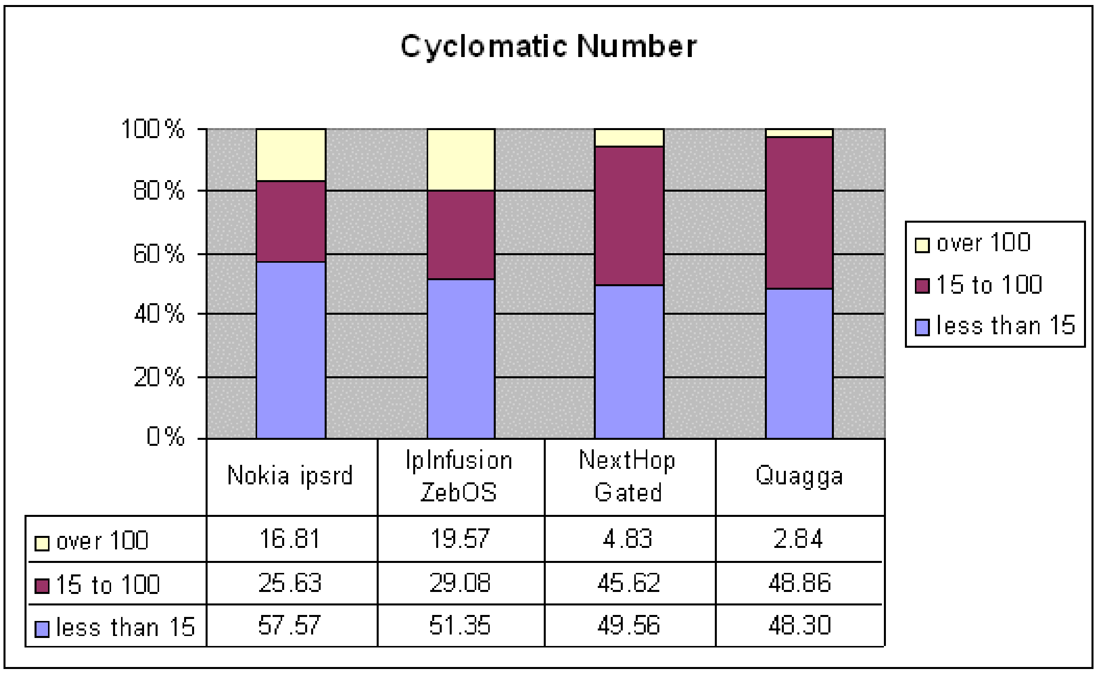
	Nokia ipsrd	IpInfusion ZebOS	NextHop Gated	Quagga
□ over 75	7.39	1.11	14.10	0.85
■ 30 to 75	55.78	31.88	46.63	33.24
□ less than 30	36.83	67.00	39.26	65.91

Figure 11: Comments Percentage

The routing daemons have quite many differences from the commenting perspective. GateD seems to have the most (14.10%) files with over 75% of lines commented. Ipsrd has about half of that proportion (7.39%) with highly commented lines. Both ZebOS and Quagga have very few those files. Ipsrd has the biggest proportion (55,78%) in the recommended range. Also GateD has about half (46,63%) of the files in the recommended range. ZebOS and Quagga have only one third of all their files between 30% and 75% comments. A significant amount of ZebOS and Quagga files are commented less than the recommended limit is. When this proportion is about two thirds with ZebOS and Quagga, both ipsrd and GateD have fewer than 40% of all files in the too few commented category. This comparison shows that ZebOS and Quagga are the worst solutions. Ipsrd is the best and GateD is also good.

7.2.3 Cyclomatic Number

The comparison of recommended cyclomatic complexity should inform us about how complex the code is from the perspective of loops and conditions in the code. The measurements are based on the limits recommended in chapter 4. Cyclomatic number of a source code file should be between 15 and 100. The following figure illustrates the results of this metric.



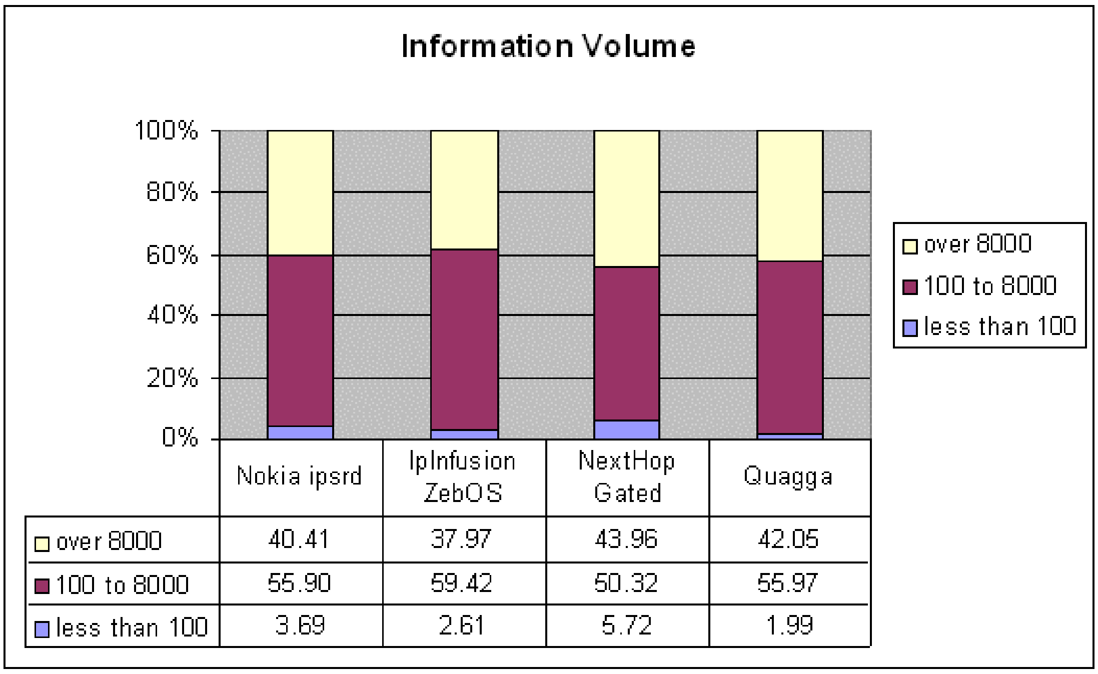
	Nokia ipsrd	IpInfusion ZebOS	NextHop Gated	Quagga
🟨 over 100	16.81	19.57	4.83	2.84
🟪 15 to 100	25.63	29.08	45.62	48.86
🟦 less than 15	57.57	51.35	49.56	48.30

Figure 12: Cyclomatic Number Measurement Results

The results show that ipsrd and ZebOS have over 15% of the files more complex than recommended. GateD and Quagga do not have a significant fraction of too complex files. However, ipsrd has the biggest proportion of files in the less than 15 category. The other routing daemons have nearly 10% less those files. A common thing for GateD and Quagga is that they have nearly half of their files in the recommended complexity range. For the rest two routing daemons, the recommended range includes only slightly over one fourth of the files. If we compare the proportion of too complex files, Quagga and GateD are the best.

7.2.4 Information Volume

The information volume should describe the amount of information inside a single file. These results are calculated from each routing solution. The recommended values here are from 100 to 8000. Similarly to the previous sections, we present these results here in percentages. The following figure illustrates the results of this measurement.



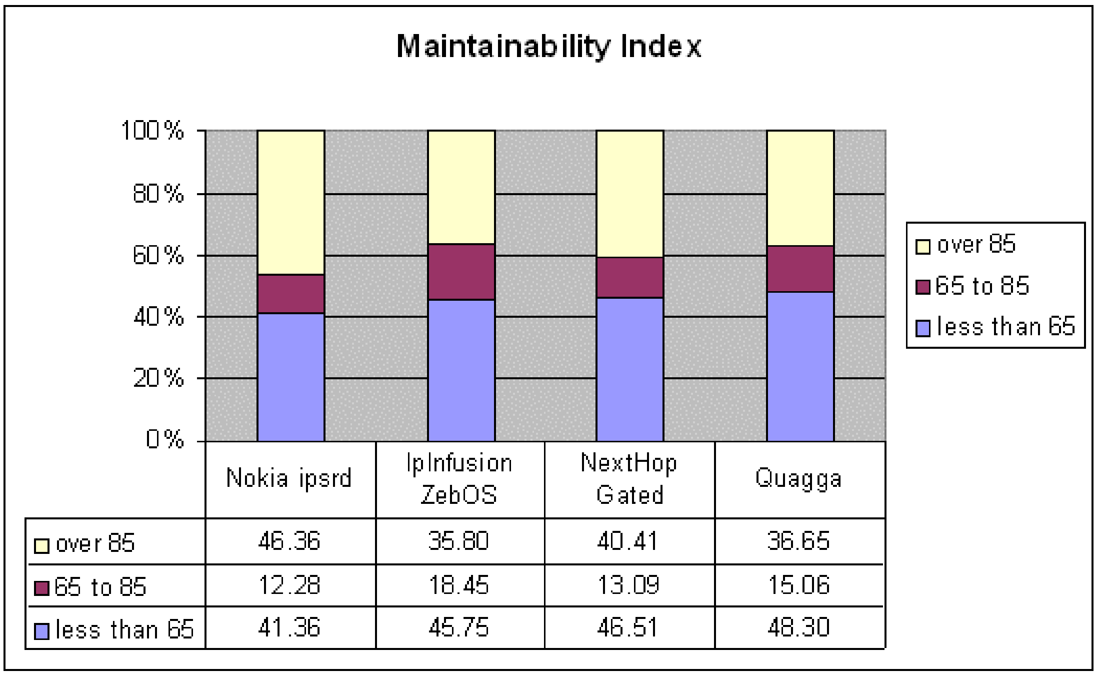
	Nokia ipsrd	IpInfusion ZebOS	NextHop Gated	Quagga
over 8000	40.41	37.97	43.96	42.05
100 to 8000	55.90	59.42	50.32	55.97
less than 100	3.69	2.61	5.72	1.99

Figure 13: Information Volume Measurement Results

This measurement did not show any big differences between the routing solutions. Each of them has a similar distribution of files in the Information Volume measurements. The recommended value 8000 is overstepped in about 40% of all files in all the routing daemons. The biggest part of files is in the recommended range in all the routing solutions. There are only few files in the category of less than 100 which is the recommended lowest limit.

7.2.5 Maintainability Index

This measurement combines the distinct metrics measured in the previous subsections. This result is calculated also automatically with the tool we use. According to the recommendations, Maintainability Index value under 65 shows that the related code is difficult to maintain. If the value is between 65 and 85 the code is moderately difficult to maintain. The Maintainability Index value should always be over 85 to prove that the code is easy to maintain. The following figure illustrates the results of this measurement.



	Nokia ipsrd	IpInfusion ZebOS	NextHop Gated	Quagga
□ over 85	46.36	35.80	40.41	36.65
■ 65 to 85	12.28	18.45	13.09	15.06
■ less than 65	41.36	45.75	46.51	48.30

Figure 14: Maintainability Index Measurement Results

The above results on the Maintainability Index measurements show that the routing solutions have differences in their maintainability. Nokia ipsrd has clearly the most files in the best category. With nearly half (46.36%) of all the files of good maintainability ipsrd beats ZebOS and Quagga by over 10%. GateD has also a good amount (40%) of files in this category. All the routing solutions have quite a big proportion (40%) of files under the low limit. Ipsrd has the least (41.36%) of these difficult-to-maintain files. ZebOS and GateD have more (about 46%) files in this category. Quagga has the biggest proportion (48.3%) of these difficult files. ZebOS has the most (18.45%) files in the moderate-to-maintain category. The other routing solutions have less (12% - 15%) of these files. Ipsrd performs the best in this comparison. ZebOS and GateD are equally good. Quagga is the worst routing solution because it has the most files in the difficult-to-maintain category.

7.3 Comparison Summary

We have presented the results of the different measurements of each routing solution. This subsection summarizes the earlier measurements. Now we compare the overall results of the routing daemons.

The following table summarizes the comparison results. Each routing solution gets points about how good it is in the different measurement. The evaluated number equals with the amount of beaten routing solutions. For example if ipsrd is the best in a measurement it beats three other routing solutions, and thus gets three points. The complexity measurement results are checked in a following way:

File Length:	The proportion of files in the recommended range. Bigger is better.
Comments Percentage:	The proportion of files in the recommended range. Bigger is better.
Cyclomatic Number:	The proportion of files in the recommended range. Bigger is better.
Information volume:	The proportion of files in the recommended range. Bigger is better.
Maintainability index:	The proportion of files under the low limit. Smaller is better.

Table 7: Comparison Results Summary

	Nokia ipsrd	IpInfusion ZebOS	NextHop GateD	Quagga
Performance				
Inserting 100000 OSPF routes	0	3	1	2
Memory Usage (OSPF)	2	1	3	0
Memory Usage (RIP)	3	1	2	0
OSPF Convergence	1	2	0	3
Complexity Measurements				
File Length	3	2	0	1
Comments Percentage	3	0	2	1
Cyclomatic Number	0	1	2	3
Information Volume	1	3	0	2
Maintainability Index	3	2	1	0
Sum				
	16	15	11	12

Ipsrd gets the most points from this comparison. ZebOS is second with only one point less than ipsrd. GateD and Quagga are a little further from the top two routing solutions. This evaluation is done with a weight of one in all the results. We can also put different weights to the above results.

For example the memory usage is measured two times. If we take an average of these two results and put a double weight to the other performance results, the multiprocess routing solutions would perform much better than the single process solutions. The memory effectiveness can still relate to the scalability of the routing solution. The more memory the program uses the less routes it will support. In future, the performance might be more important than the memory usage, as the memory is becoming cheaper and cheaper all the time. It means that the multiprocess routing solutions might require less performance enhancement work.

We can put different weights also to the complexity measurements results. The initial results contain the values that are used in the MI calculation. The last result is the MI value itself. This calculation puts a double weight to the MI. It is possible to use only this final MI result. The factors of the MI are weighted already in the MI calculation. We see from the MI results that ipsrd is the best in this comparison.

When we combine the weighted results of the previous paragraphs, we can get interesting overall results. We can conclude the measurement results here. We calculate the final points so that we combine the memory usage into a single value. The other performance results are summed up as they are already. The complexity measurement results are doubled. The final results are summarized in the following table.

Table 8: Weighted Comparison Results Summary

	Nokia ipsrd	IpInfusion ZebOS	NextHop GateD	Quagga
Performance				
Inserting 100000 OSPF routes	0	3	1	2
Memory Usage Average	2.5	1	2.5	0
OSPF Convergence	1	2	0	3
Sum	*3.5*	*6*	*3.5*	*5*
Complexity Measurements				
Maintainability Index	3	2	1	0
Double MI	*6*	*4*	*2*	*0*
Sum				
	9.5	10	5.5	5

This table shows that finally ZebOS gets 0.5 points more than ipsrd. This gap is small but means that ZebOS is the best routing solution in this weighted comparison. GateD and Quagga get about half of the points given to the best two. If we did not take into account the complexity measurements, ZebOS would be the best. Quagga would be the second. The original comparison results make the different weight setups possible. Basically, by weighting any of the routing solutions would get the best points. The weighting used in this thesis is only one way of measuring the alternatives.

7.3.1 Performance Summary

The first performance measurement of inserting routes shows us no critical differences. The measurements of the memory usage test show that the single process routing daemons use significantly less memory than the multiprocess ones. Also open source Quagga is using the most memory. That is a problem of open source development. The performance of the routing solution is not as important as the support of different features.

Another notable difference between open source and commercial routing solutions can be seen from the OSPF convergence measurement results. The open source Quagga is the fastest in those. The supremacy in convergence can be explained with a slightly different kernel version used in Quagga measurements.

The multiprocess routing solutions are faster in the convergence time. The separation of tasks between multiple processes makes the routing daemon able to update its routing table faster. That can be seen especially with a big number of routes.

7.3.2 Complexity Differences

The complexity measurements show the biggest differences between the routing daemons in the comments percentage and cyclomatic complexity. As these are considered also in the Maintainability Index calculation the results of it show also difference between the routing solutions.

Ipsrd has the best results in the Maintainability Index measurements. Comments are also done the best in ipsrd. Cyclomatic complexity is the best in Quagga. Both ipsrd and ZebOS perform the worst in the cyclomatic complexity measurements. The Maintainability Index measurement summarizes many metrics into a single value. We suppose that ipsrd is the best routing daemon from the overall software complexity viewpoint.

7.4 Conclusion

This thesis studied the routing daemon based on the FreeBSD operating system. The goal was to solve how suitable the routing daemon is for Linux. We ported this software to Linux. Then we compared the performance and software complexity between the ported routing software and a few other routing solutions. The results show that the ported routing daemon suits well to Linux.

This thesis has required more work than I thought in the beginning. Especially the theoretical background was surprisingly demanding to learn and find references. However, the work made me learn about the routing software. That was one of my initial goals for this work.

The objectives stated in the beginning were fulfilled quite well in this thesis. The only setback we encountered was that Quagga did not support as many RIP routes as we required for memory usage testing. All in all, this thesis gives a good basis for further testing and comparison of these or other routing solutions.

The initial decision to exclude routing protocol introductions from this thesis proved to be good. Those introductions would have made this thesis too large. The decision to concentrate only on the software issues was beneficial to the progress of this thesis. It clarified the objectives and gave a good basis for the writing in the very beginning.

Future work will include a more careful performance testing. The work is needed to solve the reason for the noticeable performance difference in the OSPF convergence of the multiprocess and single process routing software. We should also solve the problem of ipsrd with a big amount of OSPF routes. The work with ipsrd should continue by optimizing and stabilizing the software for Linux. After that ipsrd has a good future also on Linux.

8. Bibliography

[AFO] Software Maintainability Evaluation Guide 800-2, Volume 3. Kirtland AFB, NM: HQ Air Force Operational Test and Evaluation Center (AFOTEC), 1989.

[BAK] F. Baker, “Requirements for IP Version 4 Routers”, RFC 1812, June 1995

[BRA] Internet Engineering Task Force (R. Braden, Editor), “Requirements for Internet Hosts -- Communication Layers”, RFC 1122, October 1989

[COR] Cornell University, “MESL Techical Report”, http://cidc.library.cornell.edu/reports/mesl.htm

[CYC] E. VanDoren, ”Cyclomatic Complexity”, July 2000, http://www.sei.cmu.edu/str/descriptions/cyclomatic.html

[FEN] M. Feng, R. Leung, A. D. Jun, “Linux Network”, Summer Report 1999, Network Architecture Lab, Electrical and Computer Engineering, University of Toronto, September 1999, http://dcn.ssu.ac.kr/~softgear/prog_sw_report_summer_99.pdf

[G35] GateD R3_5Alpha_5 Documentation, April 1994, http://sunse.jinr.ru/local/gated/main.html

[GIL] R. Gilligan, E. Nordmark, “Transition Mechanisms for IPv6 Hosts and Routers”, RFC 1933, April 1996

[HAL] E. VanDoren, ”Halstead Complexity Measures”, January 1997, http://www.sei.cmu.edu/str/descriptions/halstead.html

[IPI] IP Infusion Inc., ” ZebOS Advanced Routing Suite”, 2003, http://www.ipinfusion.com/products/advanced/products_advanced.html

[IPS] Nokia, “IPSO-NET System Architecture Specification”, Nokia internal document, October 2005

[IST] Imperial Software Technology, ”An ABC of Software Migration”, http://www.softwaremigration.com/abc.html

[IXI] IXIA, ”IXIA - Leader in IP Performance Testing”,2006, http://www.ixiacom.com/

[LEH] G. Lehey, “The Complete FreeBSD”, 3rd ed., Walnut Creek CDROM, Walnut Creek CA, May 1999

[MAR] J. J. Marciniak, ed., Encyclopedia of Software Engineering, 131-165. New York, NY: John Wiley & Sons, 1994.

[MCK] M. K. McKusick, K. Bostick, M. J. Karels, J. S. Quarterman, “The Design and Implementation of the 4.4 BSD Operating System”, 9th ed., Addison-Wesley, August 2000

[MIT] E. VanDoren, “Maintainability Index Technique for Measuring Program Maintainability”, March 2002, http://www.sei.cmu.edu/str/descriptions/mitmpm.html

[NHT] NextHop Technologies, Inc., ”GateD Next Generation Carrier (NGC)”, 2005, http://www.nexthop.com/products/ngc.html

[NOK] Nokia, “Ipsrd, The Ipsilon Routing Daemon, User Manual”, Nokia internal document, September 1999

[O91] P. Oman, “HP-MAS: A Tool for Software Maintainability, Software Engineering” (#91-08-TR). Moscow, ID: Test Laboratory, University of Idaho, 1991.

[O92] P. Oman, J. Hagemeister, “Construction and Validation of Polynomials for Predicting Software Maintainability” (92-01TR). Moscow, ID: Software Engineering Test Lab, University of Idaho, 1992.

[OMA] P. Oman, J. Hagemeister, "Constructing and Testing of Polynomials Predicting Software Maintainability." Journal of Systems and Software 24, 3 (March 1994): 251-266.

[PEA] T. Pearse, P. Oman, "Maintainability Measurements on Industrial Source Code Maintenance Activities," 295-303. Proceedings of the International Conference on Software Maintenance. Opio, France, October 17-20, 1995. Los Alamitos, CA: IEEE Computer Society Press, 1995.

[QUA] Quagga Routing Software Suite, July 2006, http://www.quagga.net/

[RAM] A. Ramanath, ”A Study of the interaction of BGP/OSPF in Zebra/ZebOS/Quagga”, August 2004, http://www.quagga.net/docs/BGP_OSPF_Interaction_Report.pdf

[SAL] J. Salim, H. Khosravi, A. Kleen, A. Kuznetsov, “Linux Netlink as an IP Services Protocol”, RFC 3549, July 2003

[SEI] Software Engineering Institute, Carnegie Mellon University, http://www.sei.cmu.edu/about/index.html

[SIT] C. Sittenauer, M. Olsem, "Time to Reengineer?" Crosstalk, Journal of Defense Software Engineering 32 (March 1992): 7-10.

[STA] William Stallings, “Data and Computer Communications”, 5th ed., Prentice Hall, Inc., Upper Saddle River, New Jersey, 1997

[TRO] G. Trotter, “Terminology for Forwarding Information Base (FIB) based Router Performance”, RFC 3222, December 2001

[VER] Verifysoft Technology, “CMT++ Complexity Measures C/C++”, October 2005, http://www.verifysoft.com/en_cmtpp.html

[WEL] K. D. Welker, P. W. Oman, "Software Maintainability Metrics Models in Practice." Crosstalk, Journal of Defense Software Engineering 8, 11 (November/December 1995): 19-23.

[YUJ] J. Yu, “Scalable Routing Design Principles”, RFC 2791, July 2000

[ZEB] IP Infusion Inc., ”GNU Zebra -- routing software”, 2003, http://www.zebra.org/