UNIVERSITY OF OSLO
Department of Informatics

A Promising Cfengine Linux Router

Nakarin Phooripoom

Network and System Administration

Oslo University College

May 19, 2008

A Promising Cfengine Linux Router

Nakarin Phooripoom

Network and System Administration
Oslo University College

May 19, 2008

Abstract

With the multiplicity usage of computer networking devices called router, it is becoming common practice for everybody who would like to be online making this technology be the most responsible for allowing one of the 20th century's greatest communications developments, the internet, to exist and become very popular in these days. Network management is important and necessary when dealing with a load of routers from different manufacturers because they have very different configuration languages which are proprietary and completely separate from server configuration. To discover whether these incompatible languages can be unified into a single open standard that can be integrated into server management by using promise theory is our goal.

This thesis considers both practical and theoretical parts. It consists of building a linux router, modeling a set of routing configurations using promise theory and designing a set of promises for cfengine 3 which can configure the router directly from the cfengine 3 promise language.

Acknowledgements

First, I would like to express my genuine appreciation and special thanks to my advisor, Professor Mark Burgess for teaching, encouraging, guidance, support, passion and dedication to my research. From my admiration, I am very proud to have been his student and advisee.

This thesis work is the conclusion of a challenging two years master's degree in network and system administration at Oslo University College in collaboration with Oslo University. I have learned much from the instructors in this program and I thank them for their excellent efforts.

I am grateful to be in the good company of my fellow classmates and alumni in the program; they have been excellent companions throughout the study here. Karim Sani Ntieche, Iman Dagnew, Stian Østen and Saerda Halifu have been my closest comrades and colleagues at school and I am grateful for their friendship, good will, collaboration and association.

Thanks to Kyrre Begnum and Ismail Hassan for showing interest, inspiring and for general, much valued, conversations and input on about everything regarding this project work.

Finally, I will never forget their encouragement, apprehension and love from my parents, Dr. Chamras and Mrs. Lekha Phooripoom. They always have been on my side and hearten me during these two years far aboard from home.

Special thought for my dearest girlfriend, Yo Wanida Napredakul. Many encouraging words have helped when anxiety was ruling.

Oslo, May 2008

Nakarin Phooripoom

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This chapter serves as an introduction to the subject of the thesis. It gives an overview of the ideas and motivations behind this project.

1.1 Introduction

The Internet is one of the 20th century's greatest communications developments. It allows people around the world to send e-mail to one another in a matter of seconds, and it lets you distribute and read, among other things all around the world. We are all used to seeing the various parts of the Internet that come into our homes and offices – the Web pages, e-mail messages and downloaded files that make the Internet a dynamic and valuable medium. But none of these parts would ever make it to your computer without a piece of the Internet that you have probably never seen. In fact, most people have never stood "face to machine" with the technology most responsible for allowing the Internet to exist at all: the router. Routers are specialized computers that send your messages and those of every other Internet user speeding to their destinations along thousands of pathways.

When you send e-mail to a friend on the other side of the country, how does the message know to end up on your friend's computer, rather than on one of the millions of other computers in the world? Much of the work to get a message from one computer to another is done by routers, because they are the crucial devices that let messages flow between networks, rather than within networks.

Let's look at what a very simple router might do. Imagine a small company that makes animated 3-D graphics for local television stations. There are 10 employees of the company, each with a computer. Four of the employees are animators, while the rest are in sales, accounting and management. The animators will need to send lots of very large files back and forth to one another as they work on projects. To do this, they will use a network. When one animator sends a file to another, the very large file will use up most of the network's capacity, making the network run very slowly for other users. One of the reasons

that a single intensive user can affect the entire network stems from the way that Ethernet works. Each information packet sent from a computer is seen by all the other computers on the local network. Each computer then examines the packet and decides whether it was meant for its address. This keeps the basic plan of the network simple, but has performance consequences as the size of the network or level of network activity increases. To keep the animators' work from interfering with that of the folks in the front office, the company sets up two separate networks, one for the animators and one for the rest of the company. A router links the two networks and connects both networks to the Internet.

The router is the only device that sees every message sent by any computer on either of the company's networks. When the animator in our example sends a huge file to another animator, the router looks at the recipient's address and keeps the traffic on the animator's network. When an animator, on the other hand, sends a message to the bookkeeper asking about an expense-account check, then the router sees the recipient's address and forwards the message between the two networks.

One of the tools a router uses to decide where a packet should go is a routing table. A routing table is a collection of information, including:

- Information on which connections lead to particular groups of addresses
- Priorities for connections to be used
- Rules for handling both routine and special cases of traffic

A routing table can be as simple as a half-dozen lines in the smallest routers, but can grow to massive size and complexity in the very large routers that handle the bulk of messages.

A router, then, has two separate but related jobs:

- The router ensures that information does not go where it is not needed. This is crucial for keeping large volumes of data from clogging the connections of "innocent bystanders."
- The router makes sure that information does make it to the intended destination.

In performing these two jobs, a router is extremely useful in dealing with two separate computer networks. It joins the two networks, passing information from one to the other and, in some cases, performing translations of various protocols between the two networks. It also protects the networks from one another, preventing the traffic on one from unnecessarily spilling over to the other. As the number of networks attached to one another grows, the configuration table for handling traffic among them grows, and the processing power of the router is increased. Regardless of how many networks are attached,

though, the basic operation and function of the router remains the same. Since the Internet is one huge network made up of tens of thousands of smaller networks, its use of routers is an absolute necessity.

1.2 Problem Description

When there are too many routers in the network, a problem that would be discussed is about *"Network Management"* which is refered to the activities, methods, procedures, and tools that pertain to the operation, administration, maintenance, and provisioning of networked systems. *Operation* deals with keeping the network (and the services that the network provides) up and running smoothly. It includes monitoring the network to spot problems. *Administration* deals with keeping track of resources in the network and how they are assigned and keep the network under control. *Maintenance* is concerned with performing repairs and upgrades to make the managed network to run better. *Provisioning* is concerned with configuring resources in the network to support a given service.

There are several approaches to accomplish the goal of network management which can be described as

1. Command Line Interface (CLI): a mechanism for interacting with a router by typing commands to conduct the system using *telnet* or *ssh* protocol.
2. Text User Interface (TUI): this way of instructing a device to perform a given task is referred to as "entering" a command: the system waits for the user to conclude the submitting of the text command by pressing the "Enter" key.
3. Web Graphical User Interface (GUI): more user-friendly and convenient approach comparing to CLI as contrasts with the use of a mouse pointer.
4. Simple Network Management Protocol (SNMP) [1]: an application-layer protocol that provides a message format for communication between SNMP managers and agents. The SNMP agent contains Management Information Base (MIB) variables, a collection of managed objects, whose values the SNMP manager can request or change through Get or Set operations.
5. YANG [28]: a data modeling language used to model configuration and state data manipulated by the NETCONF protocol, NETCONF remote procedure calls, and NETCONF notifications.

Other than SNMP and YANG, the way to deal with a router is relied on a very specific configuration language from different vendors/router manufacturers causing it is hard to handle if in the network uses many diffferent kind of routing products.

1.3 Motivation

Different vendors/router manufacturers have very different configuration langauges which are proprietary and completely separate from server configuration. Our aim is to discover whether these apparently incompatible languages can be unified into a single open standard that can be integrated into server management. All the main routing platforms today are based on some flavour of Unix, so we can imagine that in the future all management will simply be unix configuration management. We do this by using promise theory - a generic approach to modelling policy.

Since we use promise theory, cfengine is the best tool to handle this task in order to prove of concept in practice.

1.4 Document Structure

This thesis will be constructed as follow:

Chapter 2, the *background* chapter, will provide the reader with the appropriate background information about dynamic routing protocols and selected router vendors/manufacturers. The goal is introducing the reader to the aspects and challenges on the research topic.

Chapter 3 explains promise theory, assumptions, relationship to cfengine and example of how to represent configurations as promises.

Chapter 4 typically presents an approach to modelling routing promises explaining what different between Cisco/Juniper/Vyatta is and what the agents and the promise types of routing are.

Chapter 5 shows how to implement a Vyatta linux router and a model for the set of promises that can be applied to all three platforms.

Chapter 6 contains the conclusion of the thesis. This chapter will discuss interesting element in finding and suggests some future work.

Hereafter follows a Bibliography and Appendix.

Chapter 2

Background on Routing Protocols and Vendors

This chapter provides the reader with the appropriate background information about dynamic routing protocols and selected router vendors/manufacturers.

2.1 Introduction to IP Routing

Routing is the act of forwarding packets toward a given destination from one network segment or interface to the next. *Routing tables* are the databases that routers use to route traffic toward their destination. These tables contain the network addresses and prefixes can be:

1. learned from dynamic routing protocols such as RIP, OSPF, and BGP.
2. learned statically from static (configured) routing-table entries.
3. learned from the router's network interfaces.

Each address and prefix in a routing table has a next hop associated with it that takes the packet one hop closer to its destination. IP packets that a router receives contains two types of information: the packet data itself (payload) and information that identifies the packet. In IP packets, the identifying information is at the beginning of the packet, in the header. One of these header fields is the source address, which states the packet's origin; another, which is key to the routing tables, is the destination address, which tells where the packet is going when the router uses standard destination-based forwarding. When the router is determining the path toward the destination, it checks the routing table for a route that matches the packet's destination and then sends the packet to the next hop associated with that route. If there is no exact match, the router locates a more general route, selecting the *longest match*, which is the route that matches the most bits in the network portion of the address. For example, if the packet's destination is *10.0.16.2* and the routing table contains a route to *10.0.16.2/32*, which is the address of the specific host, the packet is sent using the next hop associated with that route. If the only matching routes

in the table are *10.0.0.0/8* and *10.0.16.0/24*, the latter route is used because it is the longest match.

If no match is found in the routing table, the default route of *0.0.0.0/0* is used if it exists. If no default route is configured or learned, the traffic is dropped.

Routing protocols are broken up into a few different categories, in two senses. First, we have IGP, or Interior Gateway Protocols. RIP, OSPF, and ISIS are a few IGP which you may have heard about. These are routing protocols that deal with intra-domain routing. EGP, Exterior Gateway Protocols, deal with inter-domain routing, between enterprises. Now defunct, EGP was actually a protocol, but BGP is now the standard inter-domain protocol.

Second, routing protocols are said to be of two categories in another sense: link-state, or vector-distance. The vector-distance approach is: "tell your neighbors about the world." This means that you will broadcast your entire routing table, to all your neighbors. The "vector" is the destination, and the "distance" is really a metric, or hop count. Link-state routing protocols "tell the world about your neighbors." The idea is to figure out who is "up" and broadcast that information about their link's state to all other routers. Link-state is very computationally intensive, but it provides an entire view of the network to all routers.

Most people prefer link-state protocols because they converge faster, which means that all of the routers have the same information. Link-state calculations take a long time though, and happen every time we get an update, so they cannot be used Internet-wide.

In a Nutshell:

- *Routers send packets toward their destination, normally by shipping it toward a router that knows a bit more about the destination topology.*
- *Routing is two one-way problems; it is very common for your packets to take asymmetric routes.*
- *Link-state: fast convergence, eats CPU. Vector-distance: slow convergence, easier on the silicon.*

2.1.1 RIP

The Routing Information Protocol (RIP) [13, 15, 22, 21]; was developed as part of the ARPANET project and was included in the Unix BSD operating system in the early 1980s. RIP was widely deployed in the 1980s and became the industry standard for interior routing. It was standardized by the IETF in 1988,

in RFC 1058. This version is referred to as RIP Version 1. RIP Version 2, defined in RFC 2453, added support for Classless Interdomain Routing (CIDR) and authentication. RIP Version 2 MD5 authentication is defined in RFC 2082. RFCs 2080 and 2081 define RIPng, which is designed for IPv6 networks.

RIP is the simplest unicast routing protocol in widespread use today. RIP is very simple, both in configuration and protocol design, so it is widely used in simple topologies. However, RIP does not scale well to larger networks, where OSPF are generally more appropriate, because it uses a distance-vector algorithm (also called the Bellman-Ford algorithm) to determine the best route to a destination. The distance is measured in hops, which is the number of routers that a packet must pass through to reach the destination. The best route is the one with the shortest number of hops. In the routing table, the router maintains two basic pieces of information for RIP routes: the IP address of the destination network or host and the hop count (metric) to that destination.

Every 30 seconds, devices on a RIP network broadcast RIP route information, which describes their view of the network topology and generates a lot of traffic on the network. RIP uses two techniques to reduce the amount of traffic:

- Split horizon – A device receives a route advertisement on an interface but does not retransmit that advertisement back on the same interface. This limits the amount of RIP traffic by eliminating information that its RIP neighbor has already learned.
- Poison reverse – If a RIP device learns from an interface that a device is no longer connected or reachable, it advertises that device's route back on the same interface, setting the number of hops to 16, which means infinite or unreachable. Poison reverse improves the convergence time on a RIP network.

If you use RIP, you should remember that the protocol itself has some inherent limitations. RIP can be used only in small networks because the maximum number of hops to a destination is 16. If a RIP device is more than 15 hops away, it is considered to be unreachable. In practice, this is often a serious limitation. From a route convergence point of view, you should use RIP only if your network is small, with no devices more than four hops from each other. If the network diameter is larger than this, the route convergence time increases to about two to four minutes, which can lead to network instabilities and routers becoming unreachable. In comparison, OSPF typically converge in about 40 seconds.

RIP Version 1 has two additional limitations. First, it uses only classful routing, so it cannot handle subnet and network mask information. Second, it uses clear-text password authentication, which is vulnerable to attack. RIP Version 2 was developed to address these two limitations, supporting CIDR and MD5 authentication. However, the hop-count limit of 15 was retained to maintain interoperability with Version 1.

In a Nutshell:

- *RIP is a distance-vector interior gateway routing protocol: it uses a hop count and next-hop router to specify routes.*
- *RIP Version 1 is uses broadcast and does not support CIDR. RIP Version 2 is classless and uses multicast.*
- *Even though it converges slowly and suffers from certain bugs, RIP is well-suited for small to medium environments.*

2.1.2 OSPF

The Open Shortest Path First (OSPF) protocol [14, 23, 24]; is an IGP that routes packets within a single AS, or domain. The IETF began work on OSPF in the late 1980s to develop a replacement for RIP, which was the only routing protocol at the time, because people felt that a stronger routing protocol was needed and the link-state algorithm looked promising. OSPF was implemented by router vendors in the early 1990s and was eventually standardized by the IETF in 1997 as OSPF Version 1. The current standard is Version 2, defined in RFC 2328. Much of the OSPF design was lifted from IS-IS, which is an ISO routing-protocol standard developed at the same time. OSPF was designed specifically for TCP/IP and explicitly supports IP subnetting and the tagging of externally derived routing information. OSPF also provides for the authentication of routing updates. RFC 2740 defines OSPF for IPv6.

OSPF is a link-state protocol and uses link-state advertisements (LSAs) to describe the network topology. Each OSPF router generates LSAs that describe the topology it sees and floods the LSAs throughout the domain. As a result, each router ends up with a link-state database that describes the same network topology. Once the router has the complete network topology, it runs the Dijkstra SPF calculation to determine the shortest path to each destination in the network. The calculation results in destination/next-hop pairs that are placed in the OSPF routing database. Each router performs the SPF calculation independently, and the result is that each OSPF router has an identical routing database (though each router has different next hops for the destinations).

People use OSPF when they discover that RIP just is not going to work for their larger network, or when they need very fast convergence. OSPF is the most widely used IGP. When we discuss IGPs, we are talking about one routing domain, or Autonomous System (AS). Imagine a medium-sized company with multiple buildings and departments, all connected together and sharing two redundant Internet links. All of the buildings on-site are part of the same AS. But with OSPF we also have the concept of an Area, which allows further segmentation, perhaps by department in each building.

Perhaps the most important reasons for OSPF's popularity are that it is both an open standard and a mature protocol. Virtually every vendor of routing hardware and software supports it. This makes it the routing protocol of choice in multivendor enterprise networks. It is also frequently found in ISP networks for the same reasons.

To understand the design needs for areas in OSPF, we need to know how OSPF works. There is some terminology including:

- **Router ID** - In OSPF this is a unique 32-bit number assigned to each router. This is chosen as the highest IP address on a router, and can be set large by configuring an address on a loopback interface of the chosen router.
- **Neighbor Routers** - Two routers with a common link that can talk to each other.
- **Adjacency** - A two-way relationship between two neighbor routers. Neighbors don't always form adjacencies.
- **LSA** - Link State Advertisements are flooded; they describe routes within a given link.
- **Hello Protocol** - This is how routers on a network determine their neighbors and form LSAs.
- **Area** - A hierarchy. A set of routers that exchange LSAs, with others in the same area. Areas limit LSAs and encourage aggregate routes.

To get this information distributed, OSPF does three things.

First, when a router running OSPF comes up it will send hello packets to discover its neighbors and elect a designated router (DR). The hello packet includes link-state information, as well as a list of neighbors. Providing information about your neighbor to that neighbor serves as an ACK, and proves that communication is bi-directional. OSPF is smart about the layer 2 topology: if you're on a point-to-point link, it knows that this is enough, and the link is considered "up." If you're on a broadcast link, the router must wait for an election before deciding if the link is operational.

The election vote can be stuffed, with a priority ID, so that you can ensure that your beefiest router is the DR. Otherwise, the largest IP address wins. The key idea with a DR and backup DR (BDR) is that they are the ones to generate LSAs, and they must do database exchanges with other routers in the subnet. So, non-designated routers form adjacencies with the DR. The whole DR/BDR design is used to keep the protocol scalable. The only way to ensure that all routers have the same information is to make them synchronize their databases. If you have 15 routers, and want to bring another one up, then you would have to form 15 new adjacencies. If you centralize the database, with

a backup (just in case), then adding more becomes an easy to manage linear problem.

The database exchange is part of bringing up adjacencies after the hello packets are exchanged, and it is very important. If the databases are out of synchronization, we could risk routing loops, blackholes and other dangers. The third part of bringing up an adjacency is Reliable Flooding, or LSA exchange.

The details of an LSA, as well as a more advanced discussion of areas will not be discussed here. For now, just know that area zero is special, and if you have multiple areas, they must all touch area zero. This is also called the Backbone Area. There are different types of areas in OSPF, and it can get really absurd when you throw in *virtual Links* to allow two areas to speak without hitting area zero.

OSPF defines several different types of areas. The core of an OSPF network is the *backbone area*, which is the area 0 (written as the 32-bit *0.0.0.0*). All ABRs are attached to the backbone area, as are any networks that have an area ID of *0.0.0.0*. The backbone area is a transit area that distributes traffic between other areas. The routers that make up the backbone must be physically contiguous. If they are not, you create OSFP *virtual links* so that the backbone routers appear to be contiguous.

In a straightforward OSPF network, all areas connect directly to the backbone area. All these areas, including the backbone, are referred to as *regular areas*.

OSPF *stub areas* are areas through which or into which AS external advertisements are not flooded. A stub area receives detailed or summarized routing information about other areas but receives no information about external ASs. It can receive a default summary from an ABR to reach external ASs. Because a stub area has no external routes, it cannot connect to an external area (that is, it cannot contain an ASBR) and you cannot redistribute routes from another protocol into the stub area. You might use stub areas when much of the topological database consists of AS external advertisements because it reduces the size of the topological databases and therefore the amount of memory required on the internal routers in the stub area. Another restriction on stub areas is that you cannot create a virtual link through them.

Not-so-stubby areas (NSSAs) are a variant of stub areas that allows a stub area to connect to an external network. This allows external routes originated by ASBRs within the areas to be flooded in Type 7 LSAs and then leaked into other areas. However, external routes from other areas are not flooded into the NSSA.

Finally, there also are different types of routers in OSPF.

- **ABR** - An Area Border Router is a router that is in area zero, and one or

more other areas.

- **DR, BDR** - A Designated Router, as we said, is the router that keeps the database for the subnet. It sends and receives updates (via multicast) from the other routers in the same network.
- **ASBR** - The Autonomous System Boundary Router is very special, but confusing. The ASBR connects one or more AS, and exchanges routes between them. The ASBR's purpose is to redistribute routes from another AS into its own AS.

In a Nutshell:

- *OSPF is a fast-converging, link-state IGP.*
- *OSPF forms adjacencies with neighbors and shares information via the DR and BDR using Link State Advertisements.*
- *Areas in OSPF are used to limit LSAs and summarize routes. Everyone connects to area zero, the backbone.*

2.1.3 BGP

The IGPs, RIP and OSPF maintain the mapping for the topology within a single administrative domain or AS, along with the set of best paths between systems within the domain. Each AS uses one or more common IGPs and common metrics to determine how to route packets within the AS. The administration of an AS appears to other ASs to have a single coherent interior routing scheme and presents a consistent picture of what destinations are reachable through it.

To handle inter-AS routing, IGPs use an EGP. EGPs keep track of how routing domains are connected to each other and the sequence of domains that must be crossed to reach a particular destination. Although a number of EGPs were developed in the late 1980s, the Border Gateway Protocol (BGP) is the only one currently being used on IP networks and the Internet. Version 1 of BGP was introduced in 1989, and the current usage, Version 4, is defined in RFC 1771 and has been in use since 1995.

BGP [12] is the routing protocol that holds the Internet together, providing the mesh-like connectivity of Internet service provider (ISP) networks. ISPs use BGP to connect to each other, forming the virtual backbone of the Internet. Large enterprises also sometimes use BGP to connect to their ISPs, as well as to connect portions of their internal corporate network.

BGP uses a path vector algorithm to determine network topology and paths

to destinations. This algorithm defines a route as a pairing between a destination and the attributes of the path to that destination. It considers multiple attributes of the path in order to choose the best route to the destination. In comparison, a distance-vector protocol uses a single distance metric to choose the best route. BGP routing updates carry path information, which is a full list of the transit ASs that must be crossed between the AS receiving the update and the AS that can forward the packet using its IGP. BGP uses this list to eliminate loops in the path because a router can check the list of ASs to see whether a route has already passed through it. BGP treats each AS equally when considering the path, no matter how big or small it is. BGP does not know how many routers or what type of links are in an AS.

The key features of the protocol are the notion of path attributes and aggregation of network layer reachability information (NLRI). Path attributes provide BGP with flexibility and expandability which are partitioned into well-known and optional. The provision for optional attributes allows experimentation that may involve a group of BGP routers without affecting the rest of the Internet. New optional attributes can be added to the protocol in much the same fashion as new options are added to the Telnet protocol, for instance.

One of the most important path attributes is the AS-PATH. AS reachability information traverses the Internet, this information is augmented by the list of autonomous systems that have been traversed thus far, forming the AS-PATH. The AS-PATH allows straightforward suppression of the looping of routing information. In addition, the AS-PATH serves as a powerful and versatile mechanism for policy-based routing.

In a Nutshell:

- *BGP is the routing protocol to handle inter-AS routing, unlike RIP and OSPF.*
- *BGP uses an algorithm that cannot be classified as either a pure distance vector, or a pure link state.*
- *BGP by itself is very complicated. No one understand it clearly. To be expert in BGP, you need to know BGP Attributes very well.*

2.2 Router Vendors/Manufacturers

2.2.1 Cisco Systems, Inc.

Cisco Systems, Inc. [16] is a multinational corporation company with a huge amount of employees all around the world. The company headquarter is in San Jose, California. Cisco designs and sells networking and communications technology and services under five brands, namely Cisco, Linksys, WebEx,

IronPort, and Scientific Atlanta. Everybody should know that Cisco is the largest vendor in the market these days. It is said that Cisco was not the first company to develop and sell a router, but it was one of the first to sell commercially successful multi-protocol routers, to allow previously incompatible computers to communicate using different network protocols. Moreover, a reputation of Cisco brand is very high. For someone who would like to do a career in networking computer, might have at least one of Cisco Career Certifications to be quilified.

2.2.2 Juniper Networks, Inc.

Juniper Networks, Inc. [17] is an information technology company where the headquarter is located in Sunnyvale, California and founded in 1996. The company designs and sells Internet Protocol network products and services. Juniper also partners with Nokia Siemens Networks, Ericsson, and Alcatel-Lucent to provide IP/MPLS network solutions to customers. Juniper might be the second largest routing vendor after Cisco as they have been competing for a market share to each other for the whole time. The Juniper Operating System (JUNOS) was developed based on FreeBSD, a Unix-like free operating system descended from AT&T UNIX via the Berkeley Software Distribution (BSD) branch. To have the Juniper to be a case study, would be an advantage for a future work on this field especially for the cfengine.

2.2.3 Vyatta Inc.

Vyatta Inc. [20] manufactures the open source software routers and firewalls. Their main product is a Linux distribution with specialized networking applications and functionality, and management interfaces for those applications based on XORP [27], or Extensible Open Router Platform, an open source routing software suite. In my opinion, XORP requires someone to download the code, compile it on a linux machine, and then integrate a lot of different parts. Vyatta taking on that job, by pulling things together into a more user-friendly distribution. Furthermore, the differences between Vyatta and XORP are huge. XORP is simply a routing stack that runs on Unix-like systems. Vyatta is a complete system with numerous other features (firewall, VPN, DHCP, VLANs, etc., etc.) that are not part of XORP. Vyatta also integrates the features into a system. Their product is also intended and marketed as a replacement for Cisco with a strong emphasis on cost and the flexibility inherent in an open source, Linux-based system running on commodity x86 hardware by providing a Cisco Replacement Guide on their website which shows various Cisco products and the comparable Vyatta/x86 solutions. [18]

Chapter 3

Introduction to Promise Theory and Cfengine

We are looking for a way to manage routing in a vendor-independent manner. In this chapter we look at a model using promises that allows us to describe routing independently of any technology.

3.1 Promise Theory

Promise theory begins with the idea of completely autonomous agents (components) some of which might be able to communicate or interact through the promises they make to one another. Promise theory is quite a new theory to describe and understand behaviour of systems for what can be happened in a network of entirely components.[7, 3, 6]; Rather than assuming the belief that "only that which is programmed happens", it takes the opposite viewpoint: "only that which is promised can be predicted". The components in Promise theory are honestly autonomous. That means they decide their own behavior, cannot normally be forced to do something against its will, but can voluntarily cooperate with one another. Every agent has its own viewpoints and understands it based on its own information. For instance: a network interface that has its own private IP address, netmask and broadcast address, will usually see a different set of packets than another which is configured differently.

In computer science, most models are based on the idea of state machine, or chage of state. We use the idea of promise to talk about properties without having to talk about actions. When we talk about promises behavior, we do not refer to just what we see happening at the moment, but what we expect to happen in the future because we can ensure that a system will behave in predictable ways. Expected behaviour is declared by agents in advance. As a result, we can predict the behaviour from the declarations or "promises".

It is important to have a clear concept when modeling. It is defined by Mark Burgess for talking about promises and the behaviour of agents form here [9]:

- **Behaviour** - The observable properties and actions exhibited by an agent.
- **Agent** - Any component or entity that has autonomous or independent be- haviour and/or information.
- **Promise** - A voluntary declaration made by one agent (the promiser) to another agent (the promisee) about a fact or actions which the promiser believes the promisee has not (yet) observed.
- **Command** - A message requesting that an agent perform an action.
- **Obligation** - A feeling of compulsion to make a promise.
- **Action** - An event during which a single agent (and its private resources) change from one state to another.
- **Configuration** - An arrangement or pattern formed by several independent states.

These are quite abstract, but it is usually an advantage in modelling general concepts. The most important at least you need to know is most theories of computer science assume that commands that are issued are complied regardless of whether they triump. In promise theory, no agents are obliged to make a promise.

According to an agent definition, every part of a system which can give a promise, receive a promise or evaluate the value of a promise independently should be a agent. The main idea is to reduce a system to its basic components. This could lead to a large number of agents, but we try to keep it simple and do not introduce agents that we do not need to refer to. Some agents will make the same promises as others that we might not concern about different agents in a group that all behave the same, but some agents might need to be separate.

Promises are made by agents. We write a promise

$$Promiser \xrightarrow{body} promisee \tag{3.1}$$

For example:

$$a_1 \xrightarrow{b} a_2 \tag{3.2}$$

The body of the promise contains a description of what the promise is about (type of promises) which we can tag bodies b for simplicity. For instance:

- Promise to tell my current status.
- Promise to log all changes.
- Promise to process a batch file.

We assume for simplicity that the promise types do not overlap with one another. There is no problem if an agent makes two promises of different types to another agent, e.g. promising to log all changes and promising to process a batch file. Nor is there a problem in making two completely identical promises, e.g. promising to tell the current status and promising to tell the current status (this is the same thing). However, we shall not make two promises of the same type with different constraints, e.g. promising to response a request in 10 ms and promising to response a request in 20 ms. This is a contradiction, or we shall say broken promise. So, to summarize, repeating the same promise twice is okay. Promises are *idempotent*: repetition confirms but they dont add up cumulatively.

Promises are not *transitive* in general. If a promises b to a' and a' promises b' to a'', then it is not true that a has promised anything whatsoever to a''. We write it down like this:

$$a \xrightarrow{b} a' \xrightarrow{b'} a'' \tag{3.3}$$

We should be careful not to confuse promises with communication on a point-to-point between two agents. We can make a promise with body $+b$ to be a specification to give behaviour from one agent to another, while a promise with body $-b$ is a specification of what behaviour would be received by one agent from another also (see table 3.1 or from here [9]).

Symbol	Interpretation
$a \xrightarrow{+b} a'$	Promise from a to a' with body b
$a' \xrightarrow{-b} a$	Promise to accept b
$\mathrm{v}_a(a \xrightarrow{b} a')$	The value of promise to a
$\mathrm{v}_{a'}(a \xrightarrow{b} a')$	The value of promise to a'
$\oplus$	Combination of promises in parallel
$\otimes$	Combination of promises in series

Table 3.1: Summary or promise notation

With reference to a value of promise, if a promise is valuable, it is a reason why a promise will be kept. If a promise has a negative value, it might be kept unless it is exchanged something in return.

3.2 Cfengine

Cfengine [2, 10]; is a policy-based server/client configuration management system written by Mark Burgess at Oslo University College. Its primary function is to provide automated configuration and maintenance of computers,

from a policy specification. It is available for all major UNIX and UNIX-like operating systems, and it also run under recent Windows operating systems via the Cygwin UNIX-compatibility environment/libraries.

Cfengine consists of a number of components, separate programs that work together. The major components of cfengine are:

- **cfagent** - This is a program which actually interprets policy promises and implements them in a convergent way, which is specified in a file *update.conf* and *cfagent.conf*. This program can also use data generated by the statistical monitoring engine **cfenvd** and it can fetch data from **cfservd** running on local or remote hosts.
- **cfexecd** - This is a program which can execute **cfagent** and logs its output. It can either be run in daemon (standalone) mode or in non-daemon mode using a crontab on a UNIX-like system according to policy setting in *cfagent.conf*.
- **cfservd** - This is a deamon which deals with two purposes. One is to act as a file server for the other cfengine hosts to which would like to copy files and the other purpose is to listen a request from the other cfengine hosts to start **cfagent** on receipt of a connection from **cfrun**. Its configuration file is *cfservd.conf*.
- **cfrun** - This is a command used to initiate cfagent on other cfengine hosts. We might say **cfrun** contacts remote hosts and requests that they run **cfagent**.
- **cfenvd** - This is a deamon used to collect statistics about resource usage on the hosts which it runs. The resource usage, for example, are users, load, processes and sockets.

One of the main innovations of cfengine is the idea that changes in computer configuration should be carried out in a *convergence* and *voluntary cooperation*.

Convergence means that each change operation made by the agent should have the character of a fixed point. Rather than describing the steps needed to make a change, cfengine describes the final state in which one wants to end up. The agent then ensures that the necessary steps are taken to end up in this ”policy compliant state”. Thus, cfengine can be run again and again, whatever the initial state of a system, and it will end up with a predictable result.

Voluntary cooperation is a cooperative solution based on indiviual autonomy. It means that no cfengine component is capable to receive information which it has not clearly asked for itself. According to this idea, it allows strong security, easy adaptability and resilience to change environments.

With regard to the most recent version of cfengine, cfengine 3 is a complete rewrite of cfengine front-end. It is said on the cfengine website [4] that the

aim of cfengine 3 is to remove the limitations from the organically grown of cfengine 2. *Convergence*, however, is still central to the design and functioning of cfengine since cfengines goal is to bring the system to a state of stable equilibrium and to thereafter maintain it in that state. Tool like cfengine are maintenance systems, not really like change management systems. Maintenance is a process of small changes or corrections to a model. Models that talk about change management tend to forget everything after every change, but maintenance is necessary to repair the system or return it to its intended state like cfengine does.

Cfengine 3's new language [11, 8]; is an implementation of the theoretical model developed at Oslo University College over the past four year, known as "promise theory". Promises were originally introduced by Mark Burgess as a way to describe cfengine's model of autonomy. From the idea that all the parts of the system are independent policy decision points. The most visible component of cfengine 3 is the new language interface comparing to cfengine 2.

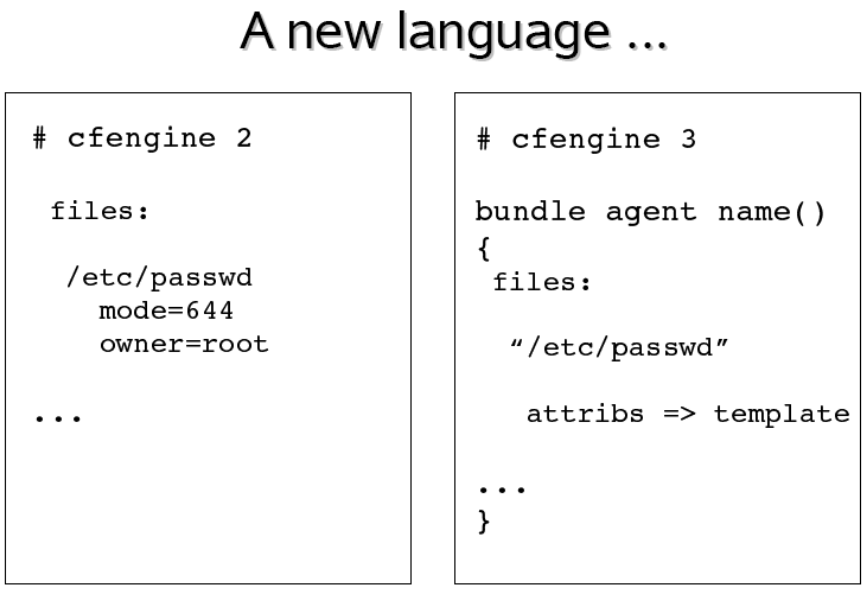

Figure 3.1: Compare cfengine 2 and 3 language interface

Cfengine is developed based on promises. A promise is madefrom one autonomous entity to another. Two essential things to make promises to the system are promise object (or promiser) and recipient (or promisee). Thus we can imagine a promise as an arrow from one entity to another.

```
"promiser" -> "promisee"
```

After that, we distinguish different promises from one another by defining the *promise body*, which is a label on the arrow describing something about what is

being promised. We write this down as follow:

```
subject => decision/constraint
```

A single promise might consist of many subdivisions. In cfengine 3, they use the term *promise bundle* to describe this and also simply list related promise bodies after the promiser, making a general pattern of a statement in cfengine 3 looks like Figure 3.2 [5]:

```
bundle name and details
{
main_subject:

  context_for_promise::

      "promiser object" -> "promisee"

              subsubject_1 => choice_1,
              subsubject_2 => choice_2,
              ...
              subsubject_n => choice_n;

...
}
```

Figure 3.2: The general form of a statement in cfengine 3

Cfengine arranges promises into *"bundles"*. However, sometimes a decision or a given constraint might involve with many attributes that are grouped together. In this case, these are grouped into a *"body"* which is simply an aggregate of subsubject, choice pairs [5].

```
body name
{
subsubject_1 => choice_1;
subsubject_2 => choice_2;
subsubject_3 => choice_3;
}
```

Figure 3.3: "Body", function-like in cfengine 3

We should not think of bundles as private functions in the sense of an object programming language, although they can have private variables. The promise within do not act on private workspace, but they can act on any part of the system on which the agent is running. For ease of understanding, the organization is absolutely formal and merely cosmetic. We used all these concepts to design a cfengine promise routing language.

Chapter 4

Modelling Routing Using Promises

Promises fit the way routers work quite well - autonomous behaviour, configured at a point. We now want to use promises to model a routing scenario and compare it to vendor languages.

4.1 Commonalities and Differences between Cisco, Juniper and Vyatta

The project itself concerns both practical and theoretical skills. In order to do some experiments, we need to have a clear concept in theory before doing some practical works. To achieve this idea, we started by creating a simple network scenario to compare and find commonalities and differences aspects between various vendors who provide routing products. Due to resource constraints, we made a decision to choose routers from reliable vendors in the market these days: *Cisco Systems, Inc*, *Juniper Networks, Inc.* and *Vyatta*. (The reason to make this decision is mentioned in Chapter 2.2.)

A research scenario is simple and straightforward. It consists of three routers connected to each other like a triangle, as illustrated in Figure 4.1.

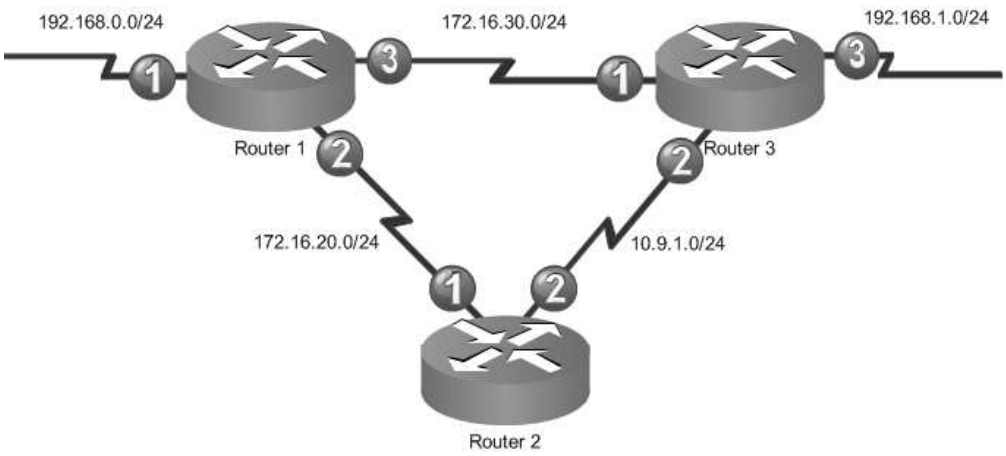

Figure 4.1: A simple research scenario

Router 1 has three network interfaces, which a network interface 2 connects to Router 2's network interface 1 and a network interface 3 connects to Router 3's network interface 1. In the same way, Router 3 has three network interfaces connected to router 1 and 2. Only Router 2 has two network interfaces. Router 2's network interface 2 is connected to Router 3's network interface 2.

Router 1	**IP Address**	**Subnet Mask**
Interface 1	192.168.0.1	255.255.255.0
Interface 2	172.16.20.1	255.255.255.0
Interface 3	172.16.30.1	255.255.255.0
Loopback	10.0.1.1	255.255.255.0
Router 2	**IP Address**	**Subnet Mask**
Interface 1	172.16.20.2	255.255.255.0
Interface 2	10.9.1.2	255.255.255.0
Loopback	10.0.2.2	255.255.255.0
Router 3	**IP Address**	**Subnet Mask**
Interface 1	172.16.30.3	255.255.255.0
Interface 2	10.9.1.3	255.255.255.0
Interface 3	192.168.1.3	255.255.255.0
Loopback	10.0.3.3	255.255.255.0

Table 4.1: Summary of all routers' network interfaces for the research scenario

After that we implemented the network according to the information in Table 4.1 using static route, RIP, OSPF and BGP. There were 6 cases have been studied, being expressed by:

4.1.1 Static route network

A static route network is a network which there is no dynamic routing protocol implemented. A routing table is created by simple adding it line by line. We manually configured the routers following a Figure 4.1 and told the routers about to which interfaces or next-hop IP addresses to forward the traffic. We implemented by using these commands regarding each vendor product to make a routing table in the router 2 to forward traffic to network *192.168.0.0/24* by the next-hop IP address *172.16.20.1*.

- **Cisco**:
  Cisco@R2(config)# ip route 192.168.0.0 255.255.255.0 172.16.20.1
- **Juniper**:
  Juniper@R2# set routing-options static route 192.168.0.0/24 next-hop 172.16.20.1
- **Vyatta**:
  Vyatta@R2# set protocols static route 192.168.0.0/24 next-hop 172.16.20.1

The outcome of the command was the same for all. For any traffic, if the destination network is *192.168.0.0* with subnet mask *255.255.255.0*, forward the traffic to an next-hop address which its IP address is *172.16.20.1*.

All configurations for static route networks are located in Appendix A.

4.1.2 RIP network

A RIP network is a network which is used a dynamic routing protocol called RIP to make a routing table and route traffic. Instead of manually defining a way to route traffic, RIP will deal with this task by its own.

We implemented RIP version 2 for all routers to ensure that RIP would work perfectly without any classless inter domain routing (CIDR) problems. This was an example how to implement RIP network depending on different vendor products. As the result, the 2nd router broadcasts its information to all interfaces and when the network is converged, it would have a fine routing table, ready to operate.

- **Cisco**:
  Cisco@R2(config)# router rip
  Cisco@R2(config-router)# version 2
  Cisco@R2(config-router)# network 172.16.20.0
  Cisco@R2(config-router)# network 10.9.1.0
- **Juniper**:
  Juniper@R2# set policy-statement advertise-rip-routes term 1 from protocol direct
  Juniper@R2# set policy-statement advertise-rip-routes term 1 from protocol rip

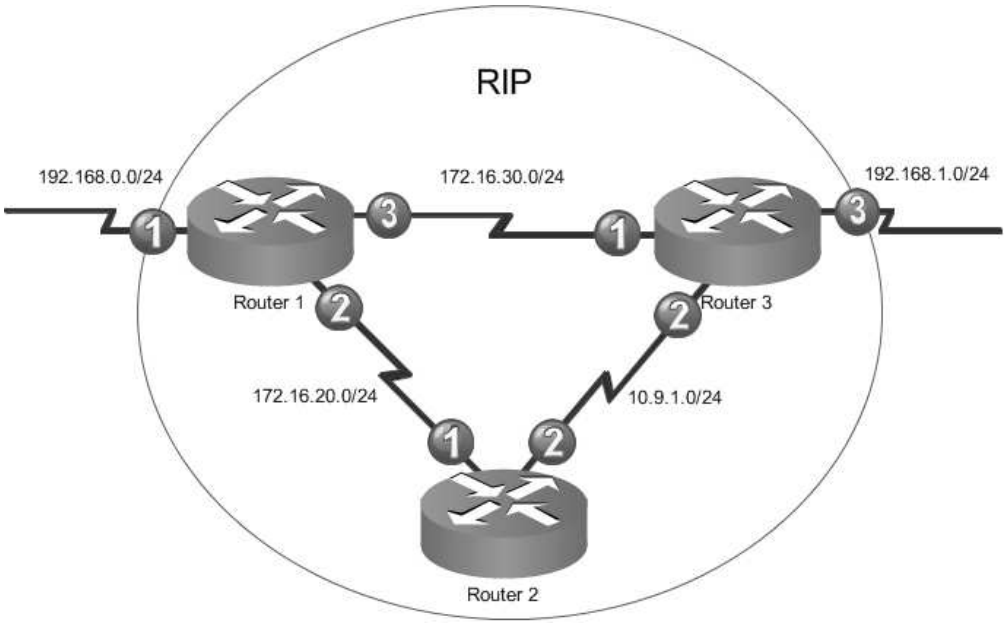

Figure 4.2: A RIP scenario

Juniper@R2# set policy-statement advertise-rip-routes term 1 then accept
Juniper@R2# set protocol rip
Juniper@R2# edit group rip-group
Juniper@R2# export advertise-rip-routes
Juniper@R2# set neighbor fe-0/0/0
Juniper@R2# set neighbor fe-0/0/1

- **Vyatta**:
  Vyatta@R2# set policy policy-statement EXPORT_CONNECTED term 1 from protocol connected
  Vyatta@R2# set policy policy-statement EXPORT_CONNECTED term 1 then action accept
  Vyatta@R2# set protocols rip interface eth0 address 172.16.20.2
  Vyatta@R2# set protocols rip interface eth1 address 10.9.1.2
  Vyatta@R2# set protocols rip export EXPORT_CONNECTED

Cisco, typically enable RIP for an interface by associating its IP address with a network, but Juniper and Vyatta are different. To have RIP systems communicate with the rest of the network for Juniper and Vyatta routers, we need to enable RIP on each interface which is directly connected to a RIP neighbor and then create a policy to share routes with its neighbors and learn routes from them.

All routers' configurations for RIP networks are located in Appendix A.

4.1.3 Single-area OSPF network

A single-area OSPF network is a network which is used a dynamic routing protocol called OSPF to make a routing table and route traffic. It is called a single-area OSPF because it has only one OSPF area. (See the explanation of OSPF in Chapter 2.1.2.)

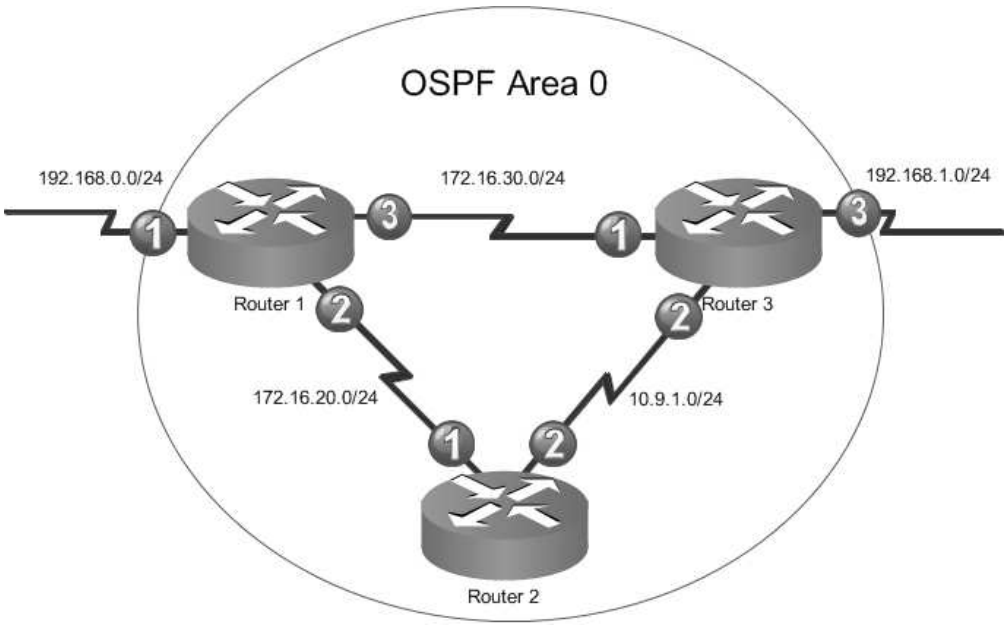

Figure 4.3: A single-area OSPF scenario

- **Cisco**:
  Cisco@R3(config)# router ospf 100
  Cisco@R3(config-router)# network 172.16.30.0 0.0.0.255 area 0
  Cisco@R3(config-router)# network 10.9.1.0 0.0.0.255 area 0
  Cisco@R3(config-router)# network 192.168.1 0.0.0.255 area 0

- **Juniper**:
  Juniper@R3# set router-id 10.0.3.3
  Juniper@R3# edit protocols ospf
  Juniper@R3# set area 0.0.0.0 interface lo0 passive
  Juniper@R3# set area 0.0.0.0 interface fe-0/0/0
  Juniper@R3# set area 0.0.0.0 interface fe-0/0/1
  Juniper@R3# set area 0.0.0.0 interface fe-0/0/2

- **Vyatta**:
  Vyatta@R3# set policy policy-statement EXPORT_CONNECTED term 1 from protocol connected
  Vyatta@R3# set policy policy-statement EXPORT_CONNECTED term 1 then action accept
  Vyatta@R3# set protocols ospf4 router-id 10.0.3.3

Vyatta@R3# set protocols ospf4 area 0.0.0.0 interface eth0 address 172.16.30.3
Vyatta@R3# set protocols ospf4 area 0.0.0.0 interface eth1 address 10.9.1.3
Vyatta@R3# set protocols ospf4 area 0.0.0.0 interface eth2 address 192.168.1.3
Vyatta@R3# set protocols ospf4 export EXPORT_CONNECTED

The basic setup for configuring a single OSPF area was straightforward. For Cisco, we enabled OSPF on a router by defining an OSPF process (100 in this case) and assigning an address range to an area. However we enabled the protocol on all router interfaces that would participate in the OSPF domain and specify to which area the interfaces should belong in for Juniper and Vyatta. (Area 0 in Cisco equals to area 0.0.0.0 in Juniper and Vyatta.)

All routers' configurations for single OSPF area networks are located in Appendix A.

4.1.4 Multi-area OSPF network

A multi-area OSPF network differs from a single-area OSPF network only there are more than one OSPF areas in the network. The setup for configuring is moderately similar. We slightly changed the OSPF area from area 0 in Cisco or area 0.0.0.0 in Juniper and Vyatta to another area number leading to the interfaces works in different OSPF areas.

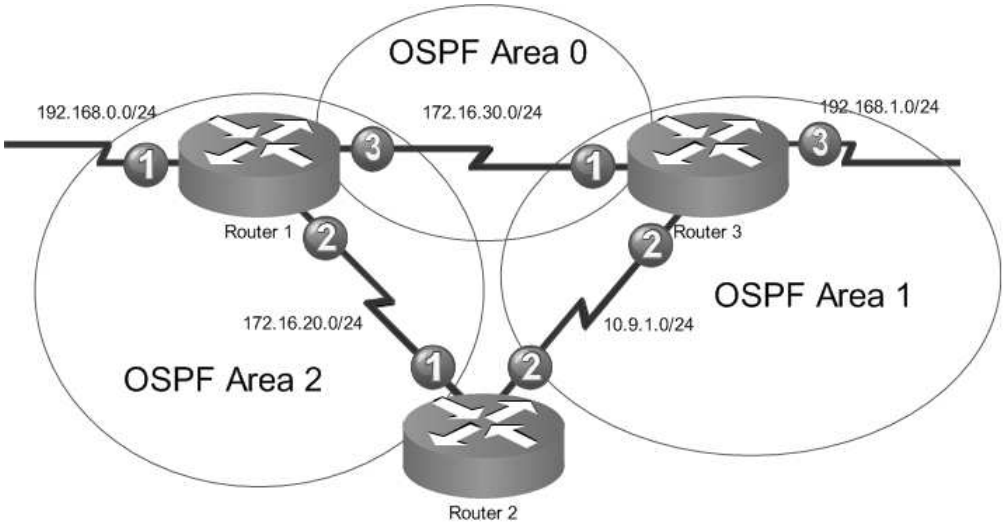

Figure 4.4: A multi-area OSPF scenario

- **Cisco**:
  Cisco@R3(config)# router ospf 100
  Cisco@R3(config-router)# network 172.16.30.0 0.0.0.255 area 0
  Cisco@R3(config-router)# network 10.9.1.0 0.0.0.255 area 1
  Cisco@R3(config-router)# network 192.168.1 0.0.0.255 area 1

- **Juniper**:
  Juniper@R3# set router-id 10.0.3.3
  Juniper@R3# edit protocols ospf
  Juniper@R3# set area 0.0.0.0 interface lo0 passive
  Juniper@R3# set area 0.0.0.0 interface fe-0/0/0
  Juniper@R3# set area 0.0.0.1 interface fe-0/0/1
  Juniper@R3# set area 0.0.0.1 interface fe-0/0/2

- **Vyatta**:
  Vyatta@R3# set policy policy-statement EXPORT_CONNECTED term 1 from protocol connected
  Vyatta@R3# set policy policy-statement EXPORT_CONNECTED term 1 then action accept
  Vyatta@R3# set protocols ospf4 router-id 10.0.3.3
  Vyatta@R3# set protocols ospf4 area 0.0.0.0 interface eth0 address 172.16.30.3
  Vyatta@R3# set protocols ospf4 area 0.0.0.1 interface eth1 address 10.9.1.3
  Vyatta@R3# set protocols ospf4 area 0.0.0.1 interface eth2 address 192.168.1.3
  Vyatta@R3# set protocols rip export EXPORT_CONNECTED

All routers' configurations for multi-area OSPF are located in Appendix A.

4.1.5 iBGP network

An iBGP network is a network which is used a dynamic routing protocol called BGP to make a routing table and route traffic especially inside the same Autonomous System (AS). The BGP protocol requires that all iBGP peers within an AS have a connection to one another to create a full-mesh of iBGP peering connections.

We assigned all router to AS 65000. Since BGP does not provide reachability information, you must make sure that each iBGP peer knows how to reach other peers. To be able to reach one another, each peer must have some sort of Interior Gateway Protocol (IGP) route, such as a connected route, a static route, or a route through a dynamic routing protocol such as RIP or OSPF, which tells them how to reach the opposite router. In this case, we do not need that as all network interfaces are directly connected. For an unconnected area, we did static route.

- **Cisco**:
  Cisco@R1(config)# router bgp 65000
  Cisco@R1(config-router)# network 192.168.0.0
  Cisco@R1(config-router)# network 172.16.20.0 mask 255.255.255.0
  Cisco@R1(config-router)# network 172.16.30.0 mask 255.255.255.0
  Cisco@R1(config-router)# neighbor 172.16.20.2 remote-as 65000
  Cisco@R1(config-router)# neighbor 172.16.30.3 remote-as 65000

- **Juniper**:
  Juniper@R1# set routing-options autonomous-system 65000

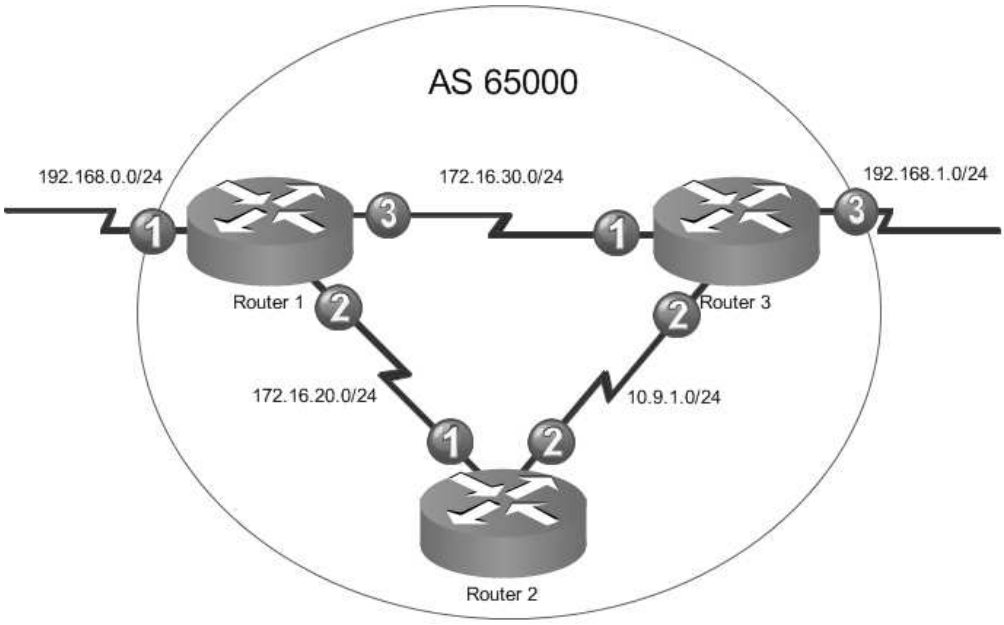

Figure 4.5: An iBGP scenario

Juniper@R1# set routing-options router-id 10.0.1.1
Juniper@R1# set protocols bgp group internal-within-AS65000 type internal neighbor 172.16.20.2
Juniper@R1# set protocols bgp group internal-within-AS65000 type internal neighbor 172.16.30.3

- **Vyatta**:
  Vyatta@R1# set protocols bgp bgp-id 10.0.1.1
  Vyatta@R1# set protocols bgp local-as 65000
  Vyatta@R1# set protocols bgp peer 10.0.2.2 as 65000
  Vyatta@R1# set protocols bgp peer 10.0.2.2 local-ip 10.0.1.1
  Vyatta@R1# set protocols bgp peer 10.0.2.2 next-hop 10.0.1.1
  Vyatta@R1# set protocols bgp peer 10.0.3.3 as 65000
  Vyatta@R1# set protocols bgp peer 10.0.3.3 local-ip 10.0.1.1
  Vyatta@R1# set protocols bgp peer 10.0.3.3 next-hop 10.0.1.1

See Appendix A. for iBGP configurations for all routers.

4.1.6 eBGP network

External BGP is the method that different Autonomous Systems (ASs) use to interconnect with one another. eBGP usually takes place over WAN links, where there may be a single physical path between eBGP peers. For instance, Figure 4.6. There are three ASs trying to connect to each other.

Instead of refering to the same AS number, we assigned the neighbor interfaces to another AS number.

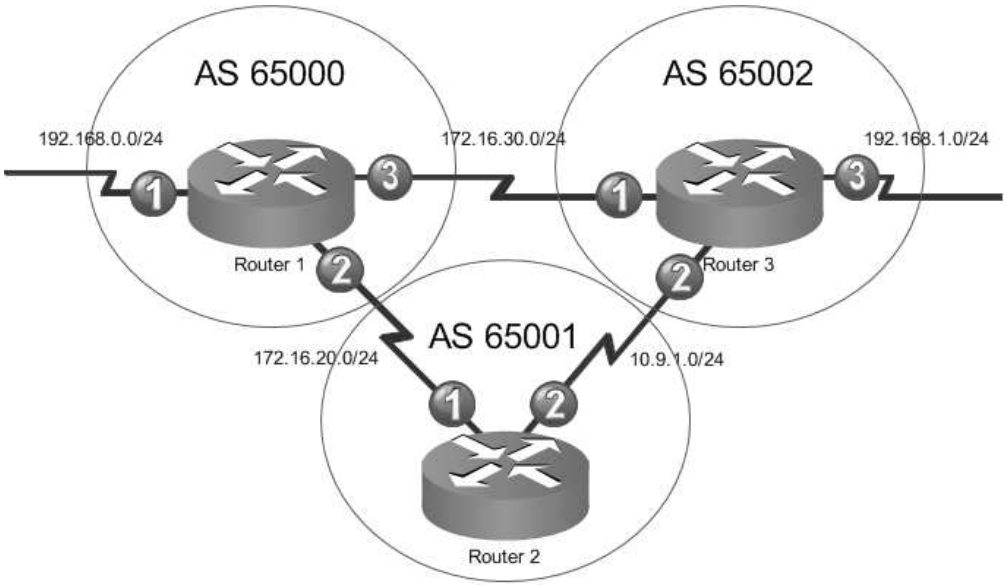

Figure 4.6: An eBGP scenario

- **Cisco**:
  Cisco@R1(config)# router bgp 65000
  Cisco@R1(config-router)# network 192.168.0.0
  Cisco@R1(config-router)# network 172.16.20.0 mask 255.255.255.0
  Cisco@R1(config-router)# network 172.16.30.0 mask 255.255.255.0
  Cisco@R1(config-router)# neighbor 172.16.20.2 remote-as 65001
  Cisco@R1(config-router)# neighbor 172.16.30.3 remote-as 65002

- **Juniper**:
  Juniper@R1# set routing-options autonomous-system 65000
  Juniper@R1# set routing-options router-id 10.0.1.1
  Juniper@R1# set protocols bgp group session-to-AS65001 type external neighbor 172.16.20.2
  Juniper@R1# set protocols bgp group EBGP2 peer-as 65001
  Juniper@R1# set protocols bgp group session-to-AS65002 type external neighbor 172.16.30.3
  Juniper@R1# set protocols bgp group EBGP3 peer-as 65002

- **Vyatta**:
  Vyatta@R1# set protocols bgp bgp-id 10.0.1.1
  Vyatta@R1# set protocols bgp local-as 65000
  Vyatta@R1# set protocols bgp peer 172.16.20.2 as 65001
  Vyatta@R1# set protocols bgp peer 172.16.20.2 local-ip 172.16.20.1
  Vyatta@R1# set protocols bgp peer 172.16.20.2 next-hop 172.16.20.1
  Vyatta@R1# set protocols bgp peer 172.16.30.3 as 65002
  Vyatta@R1# set protocols bgp peer 172.16.30.3 local-ip 172.16.30.1
  Vyatta@R1# set protocols bgp peer 172.16.30.3 next-hop 172.16.30.1

See Appendix A. for eBGP configurations for all routers.

We found that different vendors have different ideas and different approaches. So that we looked into some basic charateristics which can be found on their products. We observed and presented it as shown in Table 4.2.

	RFC	**Cisco**	**Juniper**	**Vyatta**
Software Type		Cisco IOS	JUNOS	JUNOS-like (XORP)
Opensource		No	No	Yes
Installation		No	No	Yes
Static route	-	Yes	Yes	Yes
RIP	2453	Yes	Yes	Yes
OSPF	2328	Yes	Yes	Yes
BGP	1771/4271	Yes	Yes	Yes
NAT	2766	Yes	Yes	Yes
VRRP	3768	Yes	Yes	Yes
Access lists		Yes	Yes	Yes
Route maps		Yes	Yes	Yes
VPN IPSec	4301	Yes	Yes	Yes
VPN SSL		No	Yes/No	No/Linux
FTP client	959	Yes	Yes	Yes
TFTP client	1350	Yes	Yes	Yes
TELNET server	854	Yes	Yes	Yes
SSH server	4251	Yes/No	Yes/No	Yes
HTTP server	2068	Yes	Yes	Yes
DHCP server	2131	Yes	Yes	Yes
DHCP relay		Yes	Yes	Yes
NTP server	1305	Yes/No	Yes/No	No/Linux
NTP client	1305	Yes	Yes	Yes
SNMP	3412/1157	Yes	Yes	Yes
ping		Yes	Yes	Yes
traceroute		Yes	Yes	Yes

Table 4.2: Comparision of some characteristics for Cisco, Juniper and Vyatta

It is to say that some major characteristics of all specific vendors are fairly similar which can be applied to use instead of each other. These are short descriptions for the keywords we used according to the table.

- Yes: The protocol is supported.
- No: The protocol is not supported.
- Yes/No: Available on specific products only.
- No/Linux: Not supported on Vyatta router but it can be enabled at the Linux level. For instance; OpenVPN can be used to build SSL VPNs on Linux, likewise ntp and ntp-server can be used to build NTP server on Linux for Vyatta.

One advantage of Cisco routers beyond the others is Cisco router supports more routing protocols than Juniper routers and Vyatta software routers because Cisco has created two routing protocols called Interior Gateway Routing Protocol (IGRP) [25] and Enhanced Interior Gateway Routing Protocol (EIGRP) [26]. Both routing protocols can only be used and understood by Cisco products. For example if you uses either IGRP or EIGRP to be a main routing protocol to route all traffic inside your network and you would like to add more routers to the network, you have no chance to use products from the others vendors as their products have completely no idea concerning IGRP and EIGRP. The only approch to scale you network in this situation is definitely to use Cisco products. However, if your network is operating with the other standard routing protocols such as RIP, OSPF, BGP and so on, it will not be a problem to use the products from only Cisco system as all products are compattible. They can be used together in the same network and exist together successfully.

IGRP is a distance-vector routing protocol used by routers to exchange routing data within an autonomous system. IGRP was created in part to overcome the limitations of RIP (maximum hop count of only 15, and a single routing metric) when used within large networks. IGRP supports multiple metrics for each route, including bandwidth, delay, load, MTU, and reliability; to compare two routes these metrics are combined together into a single metric, using a formula which can be adjusted through the use of pre-set constants. and IGRP is considered a classful routing protocol. As the protocol has no field for a subnet mask the router assumes that all interface addresses have the same subnet mask as the router itself. This contrasts with classless routing protocols that can use variable length subnet masks. Classful protocols have become less popular as they are wasteful of IP address space. EIGRP is a routing protocol loosely based on their original IGRP. EIGRP is an enhanced version of IGRP. The same distance vector technology found in IGRP is also used in EIGRP, and the underlying distance information remains unchanged. The convergence properties and the operating efficiency of this protocol have improved significantly. This allows for an improved architecture while retaining existing investment in IGRP. We will not discuss more in both Cisco proprietary routing protocols, IGRP and EIGRP, in details. It is out of scope of the purpose of the thesis.

Imagaine that after you configure the routers and then save it before exit, the network devices would create a configuration file and store it in their storage. All of them, Cisco, Juniper and Vyatta are using a file-based configuration. A file-based configuration can be edited directly with any text editors but you need to make sure that you are following the correct language syntax with unique commands depending on the vendor products. An advantage of this kind of configuration is very convenient to back up or restore the configuration. You can setup a TFTP or FTP system to store the configuration files. When you want to use the configuration files in the TFTP or FTP system, you can copy it accross the network to replace to the current one then ask the routers

to use the new one afterwards. All routing models are supported this function.

Syntax is one of the most important thing in the IT world. Different vendors'products use different language syntaxes, but one thing that they use exactly alike is a structure. Cisco, Juniper and Vyatta have hierarchical structure with the way to achieve the goal is similar. Juniper and Vyatta configuration are fairly closed. It would be by the reason of their operating systems are look alike. For Juniper routers, they use curly brackets ({ }), double quotation marks (" ") and semincolon (;) to define the structure. However, Vyatta routers uses curly brackets, double quotation marks, and using colon (:) instead of semincolon. Cisco routers are a bit different from both because Cisco uses only exclamation mark (!) and space to assemble its configuration file. To make it more understandable, an example of configuration file structure for Cisco, Juniper and Vyatta are located in Table 4.3.

Cisco	Juniper	Vyatta
!	protocols {	protocols {
router rip	rip {	rip {
version 2	group rip-group {	interface eth0 {
network 192.168.0.0	export advertise-rip-routes;	address 192.168.0.1 {
network 172.16.0.0	neighbor fe-0/0/0.0;	}
!	neighbor fe-0/0/1.0;	}
	neighbor fe-0/0/2.0;	interface eth1 {
	}	address 172.16.20.1 {
	}	}
	}	}
		interface eth2 {
		address 172.16.30.1 {
		}
		}
		export: "EXPORT_CONNECTED"
		}
		}

Table 4.3: Heirarchical structure for Cisco describing with RIP

A router can run multiple routing protocols simultaneously. The longest prefix match rule does not help us because the prefix lengths are the same. How then do we decide which route to take? There is a concerned value called *"administrative distance"* which A router uses the concept to determine which route wins. Administrative distance specifies a distance value that indicates how good this route is. The router will use this distance value to help it to decide between routes to the same destination prefix from different sources. For example, if you have more than one static route to the same destination, or if the router has learned another route to this destination via RIP, OSPF or whatever, it will compare this administrative distances and use the route with

the lowest distance value.

Every routing protocol has an administrative distance number that indicates how much the router trusts the information it receives by this method. A Vyatta router uses the concept of administrative distance to determine which route wins. In addition to, this concept is the same as that used by Cisco routers. Basically each routing protocol has a configured *"distance"*, and if a route is heard from two protocols, then the version with the smallest distance wins. Juniper routers, however, are different in terms of values and a name since JUNOS refers to administrative distance as *"route preferences"*, but still using the same concept. When a Juniper router learns about routes to the same destination from different sources, including routing protocols, it chooses the one that has the lowest preference value as the active route and installs it in the forwarding table.

The build-in table of administrative distances, Table 4.4, shows the default values for these administrative distances for all specificed routers.

Routing protocol or source	Cisco and Vyatta	Juniper
Connected interface	0	0
Static route	1	5
External BGP	20	170
OSPF	110	100
RIP	120	150
Internal BGP	200	170

Table 4.4: Cisco, Juniper and Vyatta default administrative distances

4.2 Modelling Routing Configurations

The basic active parts of routers are the interfaces. Each interface can operate independently, or be connected together. Traffic arriving at an interface can be forwarded to another interface. To modelling routing protocols, we would like a model by which to compare routing in three vendor routing models. Promise theory is a good approach because it offers a method for capturing and monitoring the autonomy of routers and their interfaces.

The general approach to using promises is:

- Separating all components in the system into "Agents" which are the autonomous agents of promises. We call these agents autonomous, meaning that each agent can freely reject any impulse or suggestion from another agent. No agent can be forced or controlled against its will.
- Defining the promises for each agent seems to make in order for the system to function (Promise types or the bodies of promises). In principle,

it should contain two things: a *promise type*, which explains the subject of the promise, and a *constraint* or *variation*, which describes the extent of what is being promised.

Consequently, we broke the router components down into its smallest parts and then model the promised behaviours using the ideas how to make promises which we mentioned earlier in the Chapter 3.1.

To compare the behaviours between three routers from different vendors is a bit tricky because they are using their own commands, syntaxes and methods to make all modules functioning. Typically, routers are configured with commands, but these are not promises. To know promises, we must understand the effects of the routing commands what describe the behaviours they lead to.

To write down promises more formally, we need to start with a set of "agents", which are the elements of a system capable of making promises either implicitly or explicitly. We can consider *"network interfaces"* of the routers to be agents in promises because they can be configured differently, meaning that each agent can freely reject any impulse or suggestion from another agent. No agent can be forced or controlled against its "will" as they can behave and exist differently depending on IP addresses, subnet masks and broadcast addresses and also can collaborate together with internal and external network interfaces during the process of forwarding traffic.

Based on the idea that everything, which can be configured differently, would possibly be selected to be an agent causing a routing table should be considered as well. We know a fact that when the network is converged, a routing table on each router in the network would look unsimilarly according to dynamic routing protocols it used. When the network topology is changed, all routers will recalculate the routing information and make a new routing table over and over again.

It is to say that routing tables have independent behaviour and/or information. Our aim, however, is to get rid of the complexity and model a simplicity one. We know that the routing table is one of the most important part of the routers because without routing tables, routers cannot desire to where the packets should go. Everytime network interfaces receive and merely want to forward the packets to another interface, they have to use the information where the packets can be reached the destination in the routing table. Anyhow, when you think at this level, you are thinking about a communication protolcol, not about promises. Making promises is way above that level of thinking. It does not matter that we have to make a promise from a network interface to a routing table to make the model work. We can simply make a promise directly between network interfaces with no concerns about how routers forward the packets. We can leave it as the technical level, not management level like we are interested in. Thus, we waived the routing table not to mention and selected only network interfaces to be only one agent in the model.

Let's see how promise theory helps us to understand the routers. We can use Figure 4.7 to describe as an organization of routers which can be the only one description standing for all routers from different vendors in cfengine promise language.

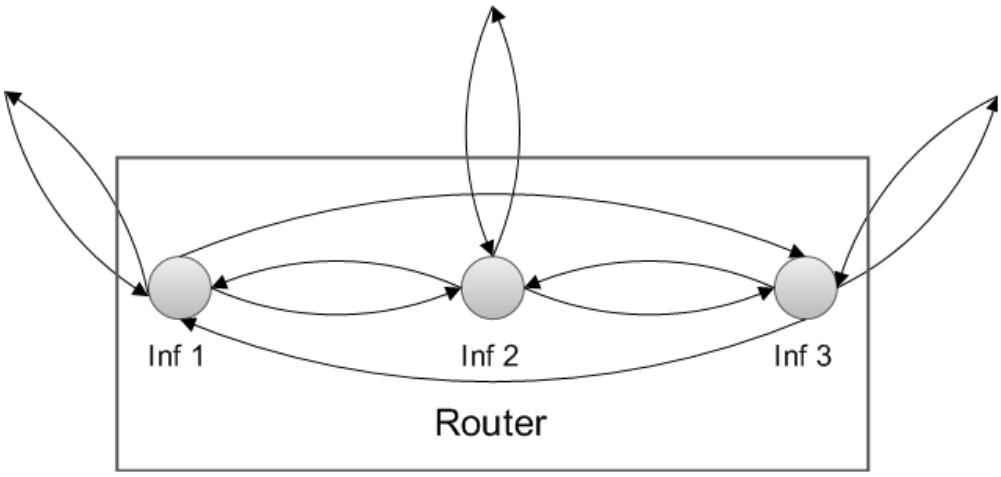

Figure 4.7: An Organization of Routers

It is true that there is only one type of agents in a model, nevertheless we can separate it into two cases to help us to understand the organization easier: promises for network interfaces between routers and another case is promises for network interfaces inside a router.

4.2.1 Promises: Network interfaces between routers

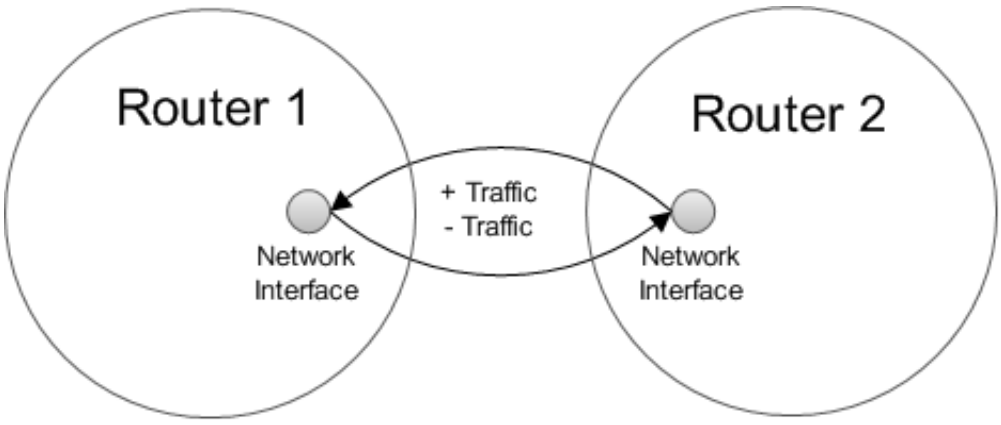

Figure 4.8: Promises for Network interfaces between routers

During the operating time of routers, they connect to the other routers for sure by using a network cable attached to two network interfaces. On the con-

dition that they are connected, the routers can forward whatever traffic across their network interfaces especially dynamic routing protocols in order to build routing tables before being converged. We might think that it is trivial to model this situation by making the same promises to each other because they are both the same kind of agents, network interfaces. See Figure 4.8.

We can label the bodies of the promise for simplicity as:

- Promise to forward all traffic to a connected network interface.
- Promise to receive all traffic from a connected network interface.

4.2.2 Promises: Network interfaces inside a router

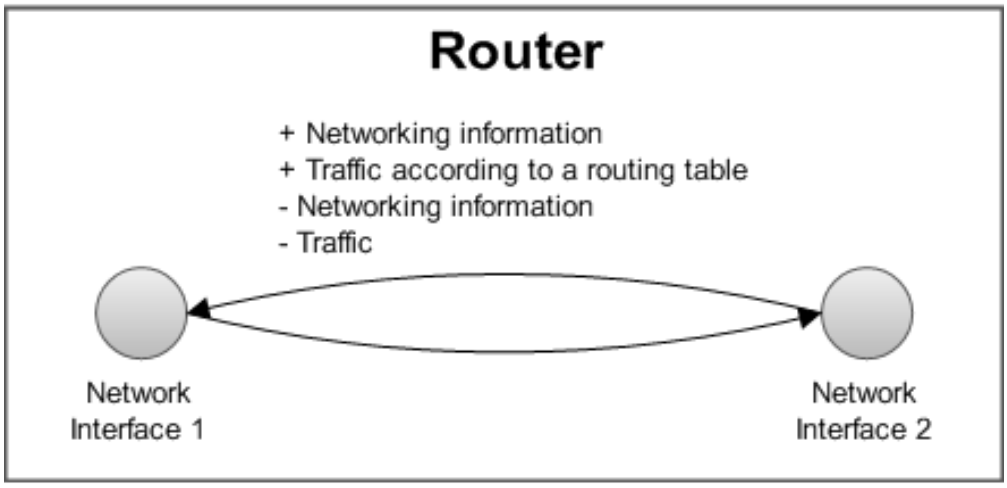

Figure 4.9: Promises for Network interfaces inside a router

Apart from having a relationship between two network interfaces on different routers, they also have a promise to network interfaces inside a router itself. The main reason to use a router in the network is to separate networks into a small one to reduce unwanted broadcast traffic and to usually tailored to the tasks of routing and forwarding packets between minimum two networks. For each network interface describes networking information which is an IP address, a subnet mask and a broadcast address would be dissimilar. That is to say, they would know all information in order to keep in touch to the others.

We can label the bodies of the promise for simplicity as:

- Promise to tell the other network interfaces the ip address, the subnet mask, the broadcast address and so on.

- Promise to receive information concerning the ip address, the subnet mask, the broadcast address and so on., from the other network interfaces.
- Promise to forward all traffic to a destination network interface relying on the information in a routing table.
- Promise to receive all traffic which is sent by the other agents.

Does a routing table involve in a process of forwarding packets? As a routing table or Routing Information Base (RIB) is a kind of electronic table or database type object which stores the routes (and in some cases, metrics associated with those routes) to particular network destinations. Whenever a node needs to send data to another node on a network, a routing table must be used to decide the way for the packet to be forwarded. Hence, the answer is definitely "Yes", but forget about that. Like we mentioned before, when you think at this level, you are thinking about a communication protocol, not about promises. In promise theory we are way above that level of thinking, as a promise is not a message.

Chapter 5

Implementation of Linux Router and a Set of Promises

5.1 Vyatta software router

To implement a *proof of concept*, we decided to build a linux router by using Vyatta software. Unlike traditional closed source venders which required to run their proprietary on in an expensive hardware platform, Vyatta open source software is able to install and run on many different environments. There are several approaches provided by Vyatta Inc.. In fact, Vyatta does not even need to be installed. Vyatta can run on a LiveCD environment where it boots from a CD and run on memory or Vyatta can be installed on internal hard drive or run in a vitural machine either on a LiveCD or installed to a virtual disk or installed to USB flash drive. You might select the operating environment to the best suits your need which have different pros and cons depended on the approaches.

1. **LiveCD**
   A LiveCD is a CD containing a bootable image. Vyatta LiveCD is created from ISO image available from Vyataa website. [19] The usage of LiveCD is simple. First, to make sure that the CD Drive is configured in the hardware BIOS to be the first boot device, then insert the CD and boot. The software will be loaded into and runs in memory. One advantages of the LiveCD is it does not touch the machine kernel as it is not loaded into the hard drive. That makes it is ideal for evaluations and lab environments. Once you finish testing, simply remove the CD, reboot and you will be to the way you were before testing. Another advantage is it is secure. We can change system files by phisically accessing but the CD is not writable causing no one can change it. In this case, the only way to save a configuration file is to use a floppy disk where we considered such as a huge disadvantage.

2. **Hard drive**
   Installing to the hard drive is the most common way to do. There are several options for hard drive today. Of course, there is a traditional

internal single hard drive which is available in all PCs. For protecting from drive failure, hardward RAID, where RAID appears as a signle disk to Linux, can be used. If moving part is concerned, then compact flash or solid state disk can be used. An enormous advantage to install Vyatta software to the hard drive is booting from hard drive is generraly faster that booting from CD. If the drive is large enogh, you may collect log files whatever you want and can be installed more additinal application if you need. Futhermore, if you install in hardware RAID environment, it can help you to protect from drive failure.

3. **Virtual machine**
   Vyatta can run on most common hypervisors including VMware, XEN or the others, especially VMware, Vyatta has certified VMware applicance available. Vyatta can be used in the virtualized environment in two ways. First, a Vyatta ISO LiveCD can be assigned to a virtual CD drive or point the virtual CD drive to the LiveCD image, vice versa. That means Vyatta will run as LiveCD. Alternatively, it can be installed to the virtual hard drive. A several advantages for running Vyatta in a virtualized environment, one approach is to combine with the other applications to make a better remote office solution. For instance, using Vyatta plus IP/PBX, e-mail, etc. Virtualization improves management flexibility with Snapshots and other virtualization tools which are available from the hypervisor vendors making hardware manangement easier. If the space and power consumption are concerned, solving with virtualized environment would be a good idea and lastly, it is easy to build complex network with multiple Vyatta nodes all on single machine for training and testing purpose.

4. **Compact Flash Drive**
   This is another technology which we can see in many operating system in these days. Instead of running by CD/DVD, it can be operateed by USB flash drives (USB key) or PMCIA compact flash drives. The main advantages to use this approach is portability. it make excellent for a lab environment for many users share the same hardware. However, you must be awared that some hardware may not support booting off of USB devices.

See Appendix B. for how to build a Linux router in detials.

Once the system has been installed, it is a good idea to perform a few check to verify everything has been installed and work properly as expected.

1. **Verity root login**
   To make sure that the root login would successfully be installed. As long as you can login as root, it guarantees that you can do everything, using user id "*root*" and a default password "*vyatta*". Successful login receives Linux bash shell prompt.

2. **Verify the router shell is operational**
   To check that the router shell is active by typing the *"xorpsh"* command to enter the router shell. If the router shell is operational, the router manager process will start and you will receive the default prompt: *"root@vyatta¿"*. If not, the router manager process will eventually time out and the user will be dropped back to the Linux shell prompt: *"root:-#"*

3. **Verify version and boot source information**
   Type *"show version"* command at the router shell prompt. This command will response a piece of important information. *"Baseline Version"* is a version of the Vyatta software and *"Booted From"* would tell you that the system is booted from disk. A common installation mistake is to forget to remove the LiveCD in the CD drive. If that line shows you "livecd", that means you are using a wrong source. To sort it out by just removing the LiveCD and rebooting the system.

4. **Check the configuration file to verify all interfaces are discovered**
   Type *"show configuration"* at the router shell prompt and look at the interfaces in the configuration file. Ethernet interfaces will be identified as *"eth#"*

5. **Check the configuration file to verify the default users**
   To check that the configuration file has been installed properly by using a [Spacebar] key to page down when you see a *"–More–"* indicator. As we know that we are logging in as root, so that user ID root is already verified by loggin in. The reason is to simply verify a user called *"vyatta"* has been pre-set up in the system and an encrypted password for vyatta suppose to be the same as an encrpyted password for the root user.

```
login {
    user root {
        authentication {
            encrypted-password: "$1$$Ht7gBYnxI1xCdO/JOnodh."
        }
    }
    user vyatta {
        authentication {
            encrypted-password: "$1$$Ht7gBYnxI1xCdO/JOnodh."
        }
    }
}
```

Figure 5.1: Example: To verify the default users

The Vyatta software is based on Debian linux. If we would like to install more debian packages, Vyatta allows us to do that. A nice and easy way to install some applications is to use a software called *"apt"*. First, we need to check a version of debian by looking into a file *"/etc/debian_version"* and then add debian repositories according to the version into *"/etc/apt/source.list"* file. See Figure 5.2 for an example of debian repositories.

```
vyatta:~# cat /etc/apt/sources.list
deb http://archive.vyatta.com/vyatta/ community main

# Line commented out by installer because it failed to verify:
deb http://security.debian.org/ etch/updates main contrib
# Line commented out by installer because it failed to verify:
deb http://http.us.debian.org/debian/ etch main contrib non-free
deb-src http://security.debian.org/ etch/updates main contrib
vyatta:~# _
```

Figure 5.2: Example: contents inside "/etc/apt/source.list"

After that we can install some essential debian packages to make the linux router be prepared avoiding problems which is going to be happend when we install cfengine afterwards.

Debian Packages	Descriptions
build-essential	Informational list of build-essential packages.
gcc	The GNU C compiler.
g++	The GNU C++ compiler.
libdb4.4-dev	Berkeley v4.4 Database Libraries [development].
libssl-dev	SSL development libraries, header files and documentation.
openssl	Secure Socket Layer (SSL) binary and related cryptographic tools.
byacc	Public domain Berkeley LALR Yacc parser generator.
flex	A fast lexical analyzer generator.

Table 5.1: Essential Debian packages to install Cfengine

You might visit cfengine website [4] to see how to get started with and to install cfengine into the linux router.

5.2 The Set of Promises

In cfengine 3, promises belong to bundles and attributes of promises belong in bodies. Hence, control information is now the province of promise *"bodies"*. Which means, a body is exactly a collection of control attributes, belonging to a promise. We can have more than one body like a template and can call to use it in *"bundle"*.

To design cfengine's bodies and bundle, we started by exploring behaviours of the promises and configurations in depth and then we tried to declare important control attributes of the promises as much as possible. For example, when you want to make a promise about network information for network interfaces in routers, we need to think in general about what the network information the routers uses, what kind of the network interfaces are, what kind of values that can be controlled. The network information would be a IP address, a subnet mask and a broadcast address. The interface types would be

ethernet interfaces, serial interfaces or WAN interfaces. The controlled values would be Media Access Control (MAC), Maximum transmission unit (MTU) and maximun global link speed. After that we compose them nicely into the body to which they should belong in the simplest way to easily use them later.

Based on our model in Chapter 4.2, let's see the results we have.

5.2.1 Body - Network interface

```
# Network interface
body ethernet myethinf(ethinfaddress,ethinfprefixlength,ethinfbroadcast)
{
 vyatta::
  ethinf_address          => "$(ethinfaddress)";
  ethinf_prefix_length    => "$(ethinfprefixlength)";
  ethinf_broadcast        => "$(ethinfbroadcast)";
  ethinf_disable          => "false";
}

body ethernet myethvif(ethvifnumber,ethvifaddress,
                                    ethvifprefixlength,ethvifbroadcast)
{
 vyatta::
  ethvif_number           => "$(ethvifnumber)";
  ethvif_address          => "$(ethvifaddress)";
  ethvif_prefix_length    => "$(ethvifprefixlength)";
  ethvif_broadcast        => "$(ethvifbroadcast)";
  ethvif_disable          => "false";
}

body ethernet myethchar()
{
 vyatta::
  ethchar_mac         => "AA-BB-CC-DD-EE-FF";
  ethchar_mtu         => "1500";
  ethchar_duplex      => "auto";
  ethchar_speed       => "auto";
}

body loopback myloinf(loinfaddress,loinfprefixlength,loinfbroadcast)
{
 vyatta::
  loinf_address            => "$(loinfaddress)";
  loinf_prefix_length      => "$(loinfprefixlength)";
  loinf_broadcast          => "$(loinfbroadcast)";
  loinf_disable            => "false";
}
```

When the routers starts up, it will automatically discovers the physical interfaces available on the system and creates a loobback interface. Apart from the interface automatically created by the system, each level of interface, IP address, prefix-length, broadcast address and vifs (virtual interfaces) to be used must explicitly created through configuration. Since our linux router has only ethernet interfaces, we can use *myethinf()* to define an IP address on the interface, *myethvif()* to create a virtual interface on the physical ethernet interface and *myloinf()* to define an IP adddress on the loopback interface which is widely used for advertising OSPF or BGP as the loopback interface is the most reliable interface of routers. For more characteristics of ethnernet interface, we allow to change them by the body, *myethchar()*.

5.2.2 Body - Static route

```
# Static route
body static mystatic(staticnetwork,staticnetmask,staticnexthop,staticmatric)
{
 vyatta::
  static_network      => "$(staticnetwork)";
  static_netmask      => "$(staticnetmask)";
  static_next_hop     => "$(staticnexthop)";
  static_matric       => "$(staticmatric)";
}
```

This body *mystatic()* is straight forward as it contains four variables, network address, subnet mask, next-hop address and matric number. The idea to design this body is to allow cfengine to do static route, routing without using any dynamic routing protocols. Be careful that you are allowed to add many route but can add only one default route for each router.

5.2.3 Body - RIP

```
# RIP IPv4
body rip myrip(ripaddress)
{
 vyatta::
  rip_address                  => "$(ripaddress)";
  rip_metric                   => "1";
  rip_horizon                  => "split-horizon-poison-reverse";
  rip_disable                  => "false";
  rip_passive                  => "false";
  rip_accept-non-rip           => "true";
  rip_accept_default_route     => "true";
  rip_advertise_default_route  => "ture";
  rip_update                   => "30";
  rip_timeout                  => "180";
```

```
  rip_expiry                    => "120";
  rip_deletion_delay            => "120";
  rip_triggered_delay           => "3";
  rip_triggered_jitter          => "66";
  rip_update_jitter             => "35";
  rip_request_interval          => "30";
  rip_interpacket_delay         => "50";
}

body rip myripauthentication(rippassword)
{
 vyatta::
  rip_simple_password     => "$(rippassword)";
}

body rip myripauthenticationmd5(ripmd5key,ripmd5pass)
{
 vyatta::
  rip_md5_key             => "$(ripmd5key)";
  rip_md5_password        => "$(ripmd5pass)";
  rip_md5_start_time      => "2008-01-01.00:00";
  rip_md5_end_time        => "2008-12-31.23:59";
}

body rip myripexport(ripexporttext)
{
 vyatta::
  rip_export      => "$(ripexporttext)";
}

body rip myripimport(ripimporttext)
{
 vyatta::
  rip_import      => "$(ripimporttext)";
}

# RIP IPv6
body rip myripng(ripngaddress)
{
 vyatta::
  ripng_address                    => "$(ripngaddress)";
  ripng_metric                     => "1";
  ripng_horizon                    => "split-horizon-poison-reverse";
  ripng_disable                    => "false";
  ripng_passive                    => "false";
  ripng_accept_non_rip             => "true";
  ripng_accept_default_route       => "true";
```

```
  ripng_advertise_default_route     => "ture";
  ripng_update                      => "30";
  ripng_timeout                     => "180";
  ripng_expiry                      => "120";
  ripng_deletion_delay              => "120";
  ripng_triggered_delay             => "3";
  ripng_triggered_jitter            => "66";
  ripng_update_jitter               => "35";
  ripng_request_interval            => "30";
  ripng_interpacket_delay           => "50";
}

body rip ripngauthentication(ripngpassword)
{
 vyatta::
  ripng_simple_password     => "$(ripngpassword)";
}

body rip ripngauthenticationmd5(ripngmd5key,ripngmd5pass)
{
 vyatta::
  ripng_md5_key             => "$(ripngmd5key)";
  ripng_md5_password        => "$(ripngmd5pass)";
  ripng_md5_start_time      => "2008-01-01.00:00";
  ripng_md5_end_time        => "2008-12-31.23:59";
}

body rip myripngexport(ripngexporttext)
{
 vyatta::
  ripng_export      => "$(ripngexporttext)";
}

body rip myripngimport(ripngimporttext)
{
 vyatta::
  ripng_import      => "$(ripngimporttext)";
}
```

As you can notice that there are two types of RIP bodies. One *myrip()* is for IPv4 and another one *myripng()* is for IPv6. The reason why we need to separate it because they use different approach to achieve the goal. It is obvious that they both are RIP but the old RIP for IPv4 networks does not understand IPv6 network addresses so that they created RIPNG (RIP New Generation) to resolve this problem.

Inside the RIP body contains many fixed values especially many RIP timers.

For example: update timer, timeout timer, expiry timer, etc. Typically, we do not need to change those but for further objective, we allowed to change by modifying directly in the body. An only variable we must declare when you call this body is an network address running RIP and a policy in order to exchange RIP information with the adjacent routers. In addition to secure the RIP connections, you can use the authenication mechanism to authorize RIP updates sent and received via the addresses either plain text password or MD5 authentication key.

5.2.4 Body - OSPF

```
# OSPF
body ospf myospf(ospfrouterid)
{
 vyatta::
  ospf_router_id                => "$(ospfrouterid)";
  ospf_rfc1583_compatibility    => "false";
  ospf_ip_router_alert          => "false";
}

body ospf myospfarea(ospfarea,ospfaddress)
{
 vyatta::
  ospf_area                       => "$(ospfarea)";
  ospf_area_type                  => "normal";
  ospf_default_lsa_disable        => "false";
  ospf_default_lsa_metric         => "0";
  ospf_summaries_disable          => "false";
  ospf_link_type                  => "broadcast";
  ospf_address                    => "$(ospfaddress)";
  ospf_priority                   => "128";
  ospf_hello_interval             => "10";
  ospf_router_dead_interval       => "40";
  ospf_interface_cost             => "1";
  ospf_retransmit_interval        => "5";
  ospf_transit_delay              => "1";
  ospf_passive                    => "false";
  ospf_disable                    => "false";
}

body ospf myospfareainterfaceneighbor(ospfneighboraddress,
                                                    ospfneighborrouterid)
{
 vyatta::
  ospf_neighbor_address       => "$(ospfneighboraddress)";
  ospf_neighbor_router_id     => "$(ospfneighborrouterid)";
```

```
}

body ospf myospfarearange(ospfnetwork)
{
 vyatta::
  ospf_area_range       => "$(ospfnetwork)";
  ospf_advertise        => "true";
}

body rip ospfauthentication(ospfpassword)
{
 vyatta::
  ospf_simple_password      => "$(ospfpassword)";
}

body rip ospfauthenticationmd5(ospfmd5key,ospfmd5pass)
{
 vyatta::
  ospf_md5_key              => "$(ospfmd5key)";
  ospf_md5_password         => "$(ospfmd5pass)";
  ospf_md5_start_time       => "2008-01-01.00:00";
  ospf_md5_end_time         => "2008-12-31.23:59";
}

body ospf myospfvirtuallink(ospfvirtualaddress,ospftransitarea)
{
 vyatta::
  ospf_backbone_area                  => "0.0.0.0";
  ospf_virtual_link_address           => "$(virtualaddress)";
  ospf_transit_area                   => "$(ospftransitarea)";
  ospf_virtual_link_hello             => "10";
  ospf_virtual_link_router_dead       => "40";
  ospf_virtual_link_retransmit        => "5";
  ospf_virtual_link_trasit_delay      => "1";
}

body ospf myospfexport(ospfexporttext)
{
 vyatta::
  ospf_export     => "$(ospfexporttext)";
}

body ospf myospfimport(ospfimporttext)
{
 vyatta::
  ospf_import     => "$(importtext)";
}
```

```
body ospf myospftraceoption()
{
 vyatta::
  ospf_trace_all_disable     => "false";
}
```

Two main bodies which can be used at least to configure OSPF network to be operated are *myospf()* and *myospfarea()*. In *myospf()*, we can configure OSPF global attributes and can specify the other characteristics of OSPF in *myospfarea()*. In OSPF, the network is broken up into areas. Within each area, routers prossess only local routing information. Routing information about other areas is calculated using routes exchanged between areas. This reduce the amount of network topology information routers have to generate and maintain making OSPF a better choice for larger networks.

No interface can belong to more than one area. If all the interfaces on a router belong to the same area, the router is said to be an internal router. If the router has OSPF-enableed interfaces that belong to more than one area, it is said to be an Area Border Router (ABR). If you define more than one area in your OSPF network, one of the areas must be designated as the backbone with *ospf_area_type* equals *"normal"* and an ABR must be connected to the backbone area (0.0.0.0).

OSPF requires a single contiguous backbone area which all other areas must connect to the backbone and all inter-area traffic passes through it. If you cannot design your network to have a single contiguous backbone, or if a failed link splits the backbone, you can extend a backbone by creating a virtual link between two non-contiguous areas by using *myospfvirtuallink()*.

5.2.5 Body - BGP

```
# BGP
body bgp mybgp(bgpid,bgplocalas)
{
 vyatta::
  bgp_id            => "$(bgpid)";
  bgp_local_as      => "$(localas)";
}

body bgp mybgpdamping()
{
 vyatta::
  bgp_damping_half_life       => "15";
  bgp_damping_max_suppress    => "60";
  bgp_damping_reuse           => "750";
```

```
  bgp_damping_suppress        => "3000";
  bgp_damping_disable         => "false";
}

body bgp mybgpconfederation(identifier)
{
 vyatta::
  bgp_confederation_identifier    => "$(identifier)";
  bgp_confederation_disable       => "false";
}

body bgp mybgpexport(bgpexporttext)
{
 vyatta::
  bgp_export      => "$(bgpexporttext)";
}

body bgp mybgpimport(bgpimporttext)
{
 vyatta::
  bgp_import      => "$(bgpimporttext)";
}

body bgp mybgppeer(bgppeer,bgplocalip,bgpas,bgpnexthop)
{
 vyatta::
  bgp_peer_address                => "$(bgppeer)";
  bgp_peer_multihop               => "1";
  bgp_peer_local_ip               => "&(bgplocalip)";
  bgp_peer_as                     => "$(bgpas)";
  bgp_peer_next_hop               => "$(bgpnexthop)";
  bgp_peer_holdtime               => "90";
  bgp_peer_delay_open_time        => "0";
  bgp_peer_client                 => "false";
  bgp_peer_confederation          => "false";
  bgp_peer_prefix_maximum         => "250000";
  bgp_peer_prefix_disable         => "false";
  bgp_peer_ipv4_unicast           => "true";
  bgp_peer_ipv4_multicast         => "false";
  bgp_peer_ipv6_unicast           => "true";
  bgp_peer_ipv6_multicast         => "false";
}

body bgp mybgproutereflector(bgpclusterid)
{
 vyatta::
  bgp_route_reflector_cluster_id    => "$(bgpclusterid)";
```

```
  bgp_route_reflector_disable       => "false";
}

body bgp mybgptrace()
{
 vyatta::
  bgp_trace_verbose_disable         => "false";
  bgp_trace_all_disable             => "false";
  bgp_trace_messagein_disable       => "false";
  bgp_trace_messageout_disable      => "false";
  bgp_trace_statechange_disable     => "false";
  bgp_trace_policy_disable          => "false";
}
```

To enable BGP on the router, we need to use the body, *mybgp()* to set its BGP ID and autonomous system and *mybgppeer()* to define an iBGP or eBGP peer. An BGP peer can be one of two types: Internal BGP (iBGP) peers are peers that are configured with the same AS number and external BGP (eBGP) peers are peers that are configured with different AS numbers. The BGP ID is normally given the loopback address, the most reliable interface of the router. Note that the BGP ID does not actually provide any reachability information, but just gives the BGP speaker a unique identifier.

The bodies, *mybgpconfederation()*, *mybgpdamping()* and *mybgproutereflector()* are optional. Confederations enable you to reduce the size and complexity of the iBGP mesh and *mybgpdamping()* is for setting the characteristics of route flap damping. Route flapping is a situation where a route fluctuates repeatedly between being announced, then withdrawn, then announced, then withdrawn, and so on. In this situation, a BGP system will send a excessive number of update messages advertising network reachability information. Route damping is intended to minimize the propagation of update messages between BGP peers for flapping routes. This reduces the load on these devices without unduly impacting the route convergence time for stable routes. Route reflection, *mybgproutereflector()*, is another technology designed to help ASs with large numbers of iBGP peers. In a standard BGP implementation, all iBGP peers must be fully meshed. When an iBGP peer learns a route from another iBGP peer, the receiving router does not forward the route to any of its iBGP peers, since these routers should have learned the route directly from the announcing router. In a route reflector environment, the iBGP peers are no longer fully meshed. Instead, each iBGP peer has an iBGP connection to one or more route reflectors which are the routers configured to be router reflector servers.

5.2.6 Bundle

When we finished defining the body parts, it is a time to apply them to our promises which we mentioned in Chapter 4.2. An approach in cfengine 3 to

use the bodies is to call them in the bundle part. Recall Figure 4.7 in the Chapter 4.1. Imagine that we are going to ask cfengine to directly configure and maintain promises for a router number 2 to use RIP and a default route. There are two agents in this case, eth0 and eth1. Eth0 promises to have an IP address 172.16.0.2 and eth1 promises to have an IP address 10.9.1.2. Both of them are in class C network (subnet mark equals 255.255.255.0). When they want to forward traffic but there in no match for a route in the routing table, they will use the eth1 interface as a default route. As the result, we can do it like an example below.

```
# Router 2
bundle agent routing()
{
 routing:
  "eth0"
    ethernet  => myethinf("172.16.20.2","24","172.16.20.255"),
    rip       => myrip("172.16.20.2"),
    rip       => myripexport("EXPORT_CONNECTED"),
    rip       => myripauthentication("password");
  "eth1"
    ethernet  => myethinf("10.9.1.2","24","10.9.1.255"),
    rip       => myrip("10.9.1.2"),
    rip       => mystatic("0.0.0.0/0", "172.16.20.1", "1");
}
```

Chapter 6

Conclusion and Future work

This study of incompatible routing languages from different manufacturers has proven to be instructive work. The primary goal of the thesis was to discover whether these routing languages can be unified into a single open standard that can be intregated into server management. The main focus was to model routing configuration by using promise theory, to design a set of promises for cfengine 3 and to build a linux router.

Particular routers which were selected to be studied were from Cisco Systems, Inc., Juniper Networks Inc. and Vyatta Inc. Operationally, Cisco, Juniper and Vyatta routers are based on different hardware and software concepts. Cisco routers use Cisco IOS, Juniper routers use JUNOS and Vyatta based on XORP, Extensible Open Route Platform. Applications wise, Cisco, Juniper and Vyatta cater to similar market segments. However, they have some advantages in certain areas compared to the other. Cisco, being the first to enter the market, has better market penetration. One of the strong areas of Cisco is the low-end router market including remote-office, and branch-office connectivity solutions. Vyatta is a ground-breaking brand with a strong emphasis on the cost and flexibility inherent in an open source, Linux-based system to be intended as a replacement for some Cisco products. All of them provide high-end router solutions that offer gigabit transfers and advanced security routing solutions.

Since Vyatta is a open source software router, the only prerequisites to use Vyatta are to have a working PC or appliance with an Intel or Intel-like processor and at least one network interface. Even it is only facultative, it is better to have a floppy disk drive or a hard drive to save the configurations. For better performances, however, it is recommended to install on a hard drive.

Promise theory is a way of describing and understanding the behaviour of systems that are composed of many parts some of which might be able to communicate with one another. Cfengine, a policy-based server/client configuration management system, was the outcome of the promises. In addition to, convergence and voluntary cooperation are central to the design and func-

tioning of cfengine.

Two considerations that we must concern before disigning a model for the set of promises for cfegine are the cfengine 3 language components (syntax) and behaviours of the routing protocols. We also need to know the functions in details of the each routing protocol in order to predict the behaviours to cover all actions of the routing protocols to be placed as control information in cfengine 3's body. The cfengine 3 language components were introduced in Chapter 3.2. Differ from cfengine 2, there is a separate part between bundle (promises) and body (attributes of promises) as the goal of cfengine 3 is to remove the limitations from the organically grown of cfengine 2.

There were three dynamic routing protocols, RIP, OSPF and BGP, were thoroughly studied in the thesis. RIP is widely used in simple topologies as it is very simple to implement and deploy. However, it does not scale well to larger networks, where OSPF are generally more appropriate. People use OSPF when they discover that RIP just is not going to work for their larger network, or when they need very fast convergence. OSPF is the most widely used IGP. BGP is a routing protocol maintaining of mapping topology between ASs. It is different from RIP and BGP which can be used only inside an AS. That is to say, BGP is the major protocol that holds the internet together in these days.

Over the long term research in modelling routing using promises, we introduced the organization of routers which can be described and represented the incompatible routing languages from different vendors. Be grateful for promise theory, it makes a complicated thing in routing much easier to understand. The model for the set of promises that can be applied to all routing platform was presented in Chapter 5.2. We would not say it was a perfect design as it was not completed yet, but satisfactory and serviceable. There are many features that we can add up in the future.

A problem which can be issued might be: is it acceptable to be only one type of promise agent in the model? We would say "Yes", because an agent can be any component or entity that has autonomous or independent behaviour and/or information. Regarding our previous design, we had two agents in a model, a network interface and a routing table. It is obvious when routers wants to forward traffic from one destination to another destination, it uses a routing table to desire where the packets should go by comparing a destination address to a destination network information field inside the routing table. If they match up, then forward the packet to a specific next-hop address, otherwise using a common route, a default route, to forward the packet. When you think at this level, you are thinking about a communication protocol, not about promises. In promise theory we are way above that level of thinking. The promise made by the network interface do not need to ask the routing table for a direction becasue we can leave it as the technical level (how it works, how to implement it, etc.) but what we are interested in promises is at the management level so that it does not matter to turn high level promises into low level details to help

us to understand systems from the top down. Consequently, having only one type of agent is fine.

The corresponding study showed that all routing configurations were used hierachical structure causing it was impossible for cfengine to directly configure the routers at this time. Cfengine is a configuration system management tools which is available for all major UNIX and UNIX-like and all major UNIX are used plain text system configurations which cfengine can understand and manage it. There was a big challenge for cfengine to directly configure the linux router.

Due to limitations of time, directly configuring could not be done. Future research on how to make it work would be interested and necessary. We thought about the way to solve this problem and would like to inspire a future researcher to follow this guideline below. It might help more or less.

How to solve

1. *To certainly understand the configurations for those particular routers.*
2. *To make configuration templates as regard to a configuration structure of router vendors.*
3. *To let cfengine to edit the configuration template, copy and then replace the completed configuration file to a configuration path in a linux router.*
4. *To reboot a routing service in order to use a new configuration.*

With this approach, it is not only routing module but aslo can be used to manage the whole router configuration including system, policy and so on. Furthermore, when everything is under control, they will possibly apply the same approach and idea to switching products afterwards.

Bibliography

[1] Indra Widjaja Alberto Leon-Garcia. *Communication Networks: Fundamental Concepts and Key Architectures, Second Edition*. McGraw-Hill, 2004. ISBN 0-07-119848-2.

[2] M. Burgess. Recent developments in cfengine. *Unix.nl Conference*, 2001.

[3] M. Burgess and S. Fagernes. Pervasive computing management: A model of network policy with local autonomy. *IEEE Transactions on Networking*, (summitted).

[4] Mark Burgess. "introduction to cfengine 3". *http://www.cfengine.org/*. (visited April 2008).

[5] Mark Burgess. Promises and cfengine 3. *http://www.cfengine.org/cfengine3.phtml?page=cf3.html*. (visited April 2008).

[6] Mark Burgess. Promise you a rose garden: An essay about system management. 1 January 2007.

[7] Mark Burgess. An approach to understanding policy based on autonomy and voluntary cooperation. *In IFIP/IEEE 16th international workshop on distributed systems operations and management (DSOM), in LNCS 3775*, pages 97–108, 2005.

[8] Mark Burgess. Technical challenges: Cfengine.com. April 2008.

[9] Mark Burgess. Computing promises. March 2008.

[10] Mark Burgess and Aeleen Frisch. *A System Engineer's Guide to Host Configuration and Maintenance Using Cfengine*. USENIX Association, 2007. Short Topics in System Administration.

[11] Mark Burgess and Aeleen Frisch. *Promises and Cfenine: A working specification for cfengine 3*. November 29, 2005.

[12] The free encyclopedia: Wikipedia. "border gateway protocol (bgp)". *http://en.wikipedia.org/wiki/BGP*, Febuary 2008. (visited Febuary 2008).

[13] The free encyclopedia: Wikipedia. "routing information protocol (rip)". *http://en.wikipedia.org/wiki/Routing_Information_Protocol*, Febuary 2008. (visited Febuary 2008).

[14] The free encyclopedia: Wikipedia. "open shortest path first (ospf)". *http://en.wikipedia.org/wiki/Ospf*, March 2008. (visited March 2008).

[15] C. Hedrick. "rfc 1058: Routing information protocol". *http://tools.ietf.org/html/rfc1058*, June 1988. (visited Febuary 2008).

[16] Cisco System Inc. Cisco's official website. *http://www.cisco.com/*, 2008. (visited May 2008).

[17] Juniper Networks Inc. Juniper's official website. *http://www.juniper.net/*, 2008. (visited May 2008).

[18] Vyatta Inc. Cisco replacement guide. *http://vyatta.com/documentation/general/Vyatta_Cisco_Replacement_Guide.pdf*. (visited March 2008).

[19] Vyatta Inc. Link to download vyatta livecd. *http://www.vyatta.com/download/index.php*. (visited April 2008).

[20] Vyatta Inc. Vyatta's official website. *http://www.vyatta.com/*, 2008. (visited May 2008).

[21] Gary Scott Malkin. "rfc 2453: Rip version 2". *http://tools.ietf.org/html/rfc2453*, November 1998. (visited Febuary 2008).

[22] Gary Scott Malkin and Robert E. Minnear. "rfc 2080: Ripng for ipv6". *http://tools.ietf.org/html/rfc2080*, January 1997. (visited Febuary 2008).

[23] John Moy. "rfc 2328: Ospf version 2". *http://tools.ietf.org/html/rfc2328*, April 1998. (visited Febuary 2008).

[24] Dennis Ferguson Rob Coltun and John Moy. "rfc 2740: Ospf for ipv6". *http://tools.ietf.org/html/rfc2740*, December 1999. (visited Febuary 2008).

[25] Cisco System. Introduction to igrp. *http://www.cisco.com/warp/public/103/5.html*, August 1991. (visited April 2008).

[26] Cisco System. Introduction to eigrp. *http://www.cisco.com/warp/public/103/1.html*, Febuary 2005. (visited April 2008).

[27] XORP. Open source ip routing. *http://www.xorp.org/*. (visited April 2008).

[28] YANG. A modelling language for netconf. *http://www.yang-central.org/twiki/bin/view/Main/WebHome*. (visited May 2008).

Appendix A

Experimental Router Configurations

All routing configurations of the routers in Chapter 4.1 are shown below here.

A.1 Cisco

A.1.1 Network interfaces

- **Router 1:**

```
!
 interface Serial0/0/1
  description R1 Interface 1
  ip address 192.168.0.1 255.255.255.0
!
 interface Serial0/0/2
  description R1 Interface 2
  ip address 172.16.20.1 255.255.255.0
!
 interface Serial0/0/3
  description R1 Interface 3
  ip address 172.16.30.1 255.255.255.0
!
 interface Loopback0
  description R1 Loopback interface
  ip address 10.0.1.1 255.255.255.255
!
```

- **Router 2:**

```
!
 interface Serial0/0/1
```

```
 description R2 Interface 1
 ip address 172.16.20.2 255.255.255.0
!
 interface Serial0/0/2
 description R2 Interface 2
 ip address 10.9.1.2 255.255.255.0
!
 interface Loopback0
 description R2 Loopback interface
 ip address 10.0.2.2 255.255.255.255
!
```

- **Router 3:**

```
!
 interface Serial0/0/1
 description R3 Interface 1
 ip address 172.16.30.3 255.255.255.0
!
 interface Serial0/0/2
 description R3 Interface 2
 ip address 10.9.1.3 255.255.255.0
!
 interface Serial0/0/3
 description R3 Interface 3
 ip address 192.168.1.3 255.255.255.0
!
 interface Loopback0
 description R3 Loopback interface
 ip address 10.0.3.3 255.255.255.255
!
```

A.1.2 Static route

- **Router 1:**

```
!
 ip classless
 ip route 192.168.1.0 255.255.255.0 172.16.30.3
 ip route 10.9.1.0 255.255.255.0 172.16.20.2
 ip route 0.0.0.0 0.0.0.0 Serial0/0/1
!
```

- **Router 2:**

```
!
 ip classless
```

```
 ip route 172.16.30.0 255.255.255.0 172.16.20.1
 ip route 192.168.0.0 255.255.255.0 172.16.20.1
 ip route 192.168.1.0 255.255.255.0 10.9.1.3
 ip route 0.0.0.0 0.0.0.0 Serial0/0/1
!
```

- **Router 3:**

```
!
 ip classless
 ip route 192.168.0.0 255.255.255.0 172.16.30.1
 ip route 172.16.20.0 255.255.255.0 10.9.1.2
 ip route 0.0.0.0 0.0.0.0 Serial0/0/1
!
```

A.1.3 RIP

- **Router 1:**

```
!
 router rip
  version 2
  network 192.168.0.0
  network 172.16.0.0
!
```

- **Router 2:**

```
!
 router rip
  version 2
  network 10.0.0.0
  network 172.16.0.0
!
```

- **Router 3:**

```
!
 router rip
  version 2
  network 192.168.1.0
  network 10.0.0.0
  network 172.16.0.0
!
```

A.1.4 Single-area OSPF

- **Router 1:**

```
!
 router ospf 100
  network 192.168.0.0 0.0.0.255 area 0
  network 172.16.20.0 0.0.0.255 area 0
  network 172.16.30.0 0.0.0.255 area 0
!
```

- **Router 2:**

```
!
 router ospf 100
  network 172.16.20.0 0.0.0.255 area 0
  network 10.9.1.0 0.0.0.255 area 0
!
```

- **Router 3:**

```
!
 router ospf 100
  network 172.16.30.0 0.0.0.255 area 0
  network 172.16.20.0 0.0.0.255 area 0
  network 192.168.1.0 0.0.0.255 area 0
!
```

A.1.5 Multi-area OSPF

- **Router 1:**

```
!
 router ospf 100
  network 192.168.0.0 0.0.0.255 area 2
  network 172.16.20.0 0.0.0.255 area 2
  network 172.16.30.0 0.0.0.255 area 0
!
```

- **Router 2:**

```
!
 router ospf 100
  network 172.16.20.0 0.0.0.255 area 2
  network 10.9.1.0 0.0.0.255 area 1
!
```

- **Router 3:**

```
!
 router ospf 100
  network 172.16.30.0 0.0.0.255 area 0
  network 172.16.20.0 0.0.0.255 area 1
  network 192.168.1.0 0.0.0.255 area 1
!
```

A.1.6 iBGP

- **Router 1:**

```
!
 router bgp 65000
  network 192.168.0.0
  network 172.16.20.0 mask 255.255.255.0
  network 172.16.30.0 mask 255.255.255.0
  neighbor 172.16.20.2 remote-as 65000
  neighbor 172.16.30.3 remote-as 65000
!
```

- **Router 2:**

```
!
 router bgp 65000
  network 172.16.20.0 mask 255.255.255.0
  network 10.9.1.0 mask 255.255.255.0
  neighbor 172.16.20.1 remote-as 65000
  neighbor 10.9.1.3 remote-as 65000
!
```

- **Router 3:**

```
!
 router bgp 65000
  network 172.16.30.0 mask 255.255.255.0
  network 10.9.1.0 mask 255.255.255.0
  network 192.168.1.0 mask 255.255.255.0
  neighbor 172.16.30.1 remote-as 65000
  neighbor 10.9.1.2 remote-as 65000
!
```

A.1.7 eBGP

- **Router 1:**

```
!
 router bgp 65000
```

```
  network 192.168.0.0
  network 172.16.20.0 mask 255.255.255.0
  network 172.16.30.0 mask 255.255.255.0
  neighbor 172.16.20.2 remote-as 65001
  neighbor 172.16.30.3 remote-as 65002
!
```

- **Router 2:**

```
!
 router bgp 65001
  network 172.16.20.0 mask 255.255.255.0
  network 10.9.1.0 mask 255.255.255.0
  neighbor 172.16.20.1 remote-as 65000
  neighbor 10.9.1.3 remote-as 65002
!
```

- **Router 3:**

```
!
 router bgp 65002
  network 172.16.30.0 mask 255.255.255.0
  network 10.9.1.0 mask 255.255.255.0
  network 192.168.1.0 mask 255.255.255.0
  neighbor 172.16.30.1 remote-as 65000
  neighbor 10.9.1.2 remote-as 65002
!
```

A.2 Juniper

A.2.1 Network interfaces

- **Router 1:**

```
interfaces {
    fe-0/0/0 {
        unit0 {
            family inet {
                address 192.168.0.1/32;
            }
        }
    }
    fe-0/0/1 {
        unit 0 {
            family inet {
```

```
                address 172.16.20.1/32;
            }
        }
    }
    fe-0/0/2 {
        unit 0 {
            family inet {
                address 172.16.30.1/32;
            }
        }
    }
    lo0 {
        unit 0 {
            family inet {
                address 10.0.1.1/32;
            }
        }
    }
}
```

- **Router 2:**

```
interfaces {
    fe-0/0/0 {
        unit0 {
            family inet {
                address 172.16.20.2/32;
            }
        }
    }
    fe-0/0/1 {
        unit 0 {
            family inet {
                address 10.9.1.2/32;
            }
        }
    }
    lo0 {
        unit 0 {
            family inet {
                address 10.0.2.2/32;
            }
        }
    }
}
```

- **Router 3:**

```
interfaces {
    fe-0/0/0 {
        unit0 {
            family inet {
                address 172.16.30.3/32;
            }
        }
    }
    fe-0/0/1 {
        unit 0 {
            family inet {
                address 10.9.1.3/32;
            }
        }
    }
    fe-0/0/2 {
        unit 0 {
            family inet {
                address 192.168.1.3/32;
            }
        }
    }
    lo0 {
        unit 0 {
            family inet {
                address 10.0.3.3/32;
            }
        }
    }
}
```

A.2.2 Static route

- **Router 1:**

```
routing-options {
    static {
        route 192.168.1.0/24 {
            next-hop 172.16.30.3;
        }
        route 10.9.1.0/24 {
            next-hop 172.16.20.2;
        }
        route 0.0.0.0/0 {
            next-hop 192.168.0.100;
        }
    }
```

```
}
```

- **Router 2:**

```
routing-options {
    static {
        route 172.168.30.0/24 {
            next-hop 172.16.20.1;
        }
        route 192.168.0.0/24 {
            next-hop 172.16.20.1;
        }
        route 192.168.1.0/24 {
            next-hop 10.9.1.3;
        }
        route 0.0.0.0/0 {
            next-hop 172.16.20.1;
        }
    }
}
```

- **Router 3:**

```
routing-options {
    static {
        route 192.168.0.0/24 {
            next-hop 172.16.30.1;
        }
        route 172.16.20.0/24 {
            next-hop 10.9.1.2;
        }
        route 0.0.0.0/0 {
            next-hop 172.16.30.1;
        }
    }
}
```

A.2.3 RIP

- **Router 1:**

```
policy-options {
    policy-statement advertise-rip-routes {
        term1 {
            from protocol [ direct rip ];
            then accept;
        }
```

```
        }
    }
    protocols {
        rip {
            group rip-group {
                export advertise-rip-routes;
                neighbor fe-0/0/0.0;
                neighbor fe-0/0/1.0;
                neighbor fe-0/0/2.0;
            }
        }
    }
```

- **Router 2:**

```
    policy-options {
        policy-statement advertise-rip-routes {
            term1 {
                from protocol [ direct rip ];
                then accept;
            }
        }
    }
    protocols {
        rip {
            group rip-group {
                export advertise-rip-routes;
                neighbor fe-0/0/0.0;
                neighbor fe-0/0/1.0;
            }
        }
    }
```

- **Router 3:**

```
    policy-options {
        policy-statement advertise-rip-routes {
            term1 {
                from protocol [ direct rip ];
                then accept;
            }
        }
    }
    protocols {
        rip {
            group rip-group {
                export advertise-rip-routes;
```

```
                neighbor fe-0/0/0.0;
                neighbor fe-0/0/1.0;
                neighbor fe-0/0/2.0;
            }
        }
    }
```

A.2.4 Single-area OSPF

- **Router 1:**

```
routing-options {
    router-id 10.0.1.1;
}
protocols {
    ospf {
        area0.0.0.0 {
            interface lo0.0 {
                passive;
            }
            interface fe-0/0/0.0;
            interface fe-0/0/1.0;
            interface fe-0/0/2.0;
        }
    }
}
```

- **Router 2:**

```
routing-options {
    router-id 10.0.2.2;
}
protocols {
    ospf {
        area0.0.0.0 {
            interface lo0.0 {
                passive;
            }
            interface fe-0/0/0.0;
            interface fe-0/0/1.0;
        }
    }
}
```

- **Router 3:**

```
routing-options {
```

```
        router-id 10.0.3.3;
    }
    protocols {
        ospf {
            area0.0.0.0 {
                interface lo0.0 {
                    passive;
                }
                interface fe-0/0/0.0;
                interface fe-0/0/1.0;
                interface fe-0/0/2.0;
            }
        }
    }
```

A.2.5 Multi-area OSPF

- **Router 1:**

```
routing-options {
    router-id 10.0.1.1;
}
protocols {
    ospf {
        area0.0.0.0 {
            interface lo0.0 {
                passive;
            }
            interface fe-0/0/2.0;
        }
        area 0.0.0.2 {
            interface fe-0/0/0.0;
            interface fe-0/0/1.0;
        }
    }
}
```

- **Router 2:**

```
routing-options {
    router-id 10.0.2.2;
}
protocols {
    ospf {
        area0.0.0.1 {
            interface lo0.0 {
                passive;
```

```
            }
            interface fe-0/0/1.0;
        }
        area 0.0.0.2 {
            interface fe-0/0/0.0;
        }
    }
}
```

- **Router 3:**

```
routing-options {
    router-id 10.0.3.3;
}
protocols {
    ospf {
        area0.0.0.0 {
            interface lo0.0 {
                passive;
            }
            interface fe-0/0/0.0;
        }
        area 0.0.0.1 {
            interface fe-0/0/1.0;
            interface fe-0/0/2.0;
        }
    }
}
```

A.2.6 iBGP

- **Router 1:**

```
routing-options {
    router-id 10.0.1.1;
    autonomous-system 65000;
}
protocols {
    bgp {
        group internal-within-AS65000 {
            type internal;
            neighbor 172.16.20.2;
            neighbor 172.16.30.3;
        }
    }
}
```

- **Router 2:**

```
routing-options {
    router-id 10.0.2.2;
    autonomous-system 65000;
}
protocols {
    bgp {
        group internal-within-AS65000 {
            type internal;
            neighbor 172.16.20.1;
            neighbor 10.9.1.3;
        }
    }
}
```

- **Router 3:**

```
routing-options {
    router-id 10.0.3.3;
    autonomous-system 65000;
}
protocols {
    bgp {
        group internal-within-AS65000 {
            type internal;
            neighbor 172.16.30.1;
            neighbor 10.9.1.2;
        }
    }
}
```

A.2.7 eBGP

- **Router 1:**

```
routing-options {
    router-id 10.0.1.1;
    autonomous-system 65000;
}
protocols {
    bgp {
        group session-to-AS65001 {
            type external;
            peer-as 65001;
            neighbor 172.16.20.2;
        }
        group session-to-AS65002 {
            type external;
```

```
            peer-as 65002;
            neighbor 172.16.30.3;
        }
    }
}
```

- **Router 2:**

```
routing-options {
    router-id 10.0.2.2;
    autonomous-system 65001;
}
protocols {
    bgp {
        group session-to-AS65000 {
            type external;
            peer-as 65000;
            neighbor 172.16.20.1;
        }
        group session-to-AS65002 {
            type external;
            peer-as 65002;
            neighbor 10.9.1.3;
        }
    }
}
```

- **Router 3:**

```
routing-options {
    router-id 10.0.3.3;
    autonomous-system 65002;
}
protocols {
    bgp {
        group session-to-AS65000 {
            type external;
            peer-as 65000;
            neighbor 172.16.30.1;
        }
        group session-to-AS65001 {
            type external;
            peer-as 65001;
            neighbor 10.9.1.2;
        }
    }
}
```

A.3 Vyatta

A.3.1 Network interfaces

- **Router 1:**

```
interfaces {
    loopback lo {
        address 10.0.1.1 {
            prefix-length: 32
        }
    }
    ethernet eth0 {
        address 192.168.0.1 {
            prefix-length: 24
        }
    }
    ethernet eth1 {
        address 172.16.20.1 {
            prefix-length: 24
        }
    }
    ethernet eth2 {
        address 172.16.30.1 {
            prefix-length: 24
        }
    }
}
```

- **Router 2:**

```
interfaces {
    loopback lo {
        address 10.0.2.2 {
            prefix-length: 32
        }
    }
    ethernet eth0 {
        address 172.16.20.2 {
            prefix-length: 24
        }
    }
    ethernet eth1 {
        address 10.9.1.2 {
            prefix-length: 24
        }
    }
}
```

- **Router 3:**

```
interfaces {
    loopback lo {
        address 10.0.3.3 {
            prefix-length: 32
        }
    }
    ethernet eth0 {
        address 172.16.30.3 {
            prefix-length: 24
        }
    }
    ethernet eth1 {
        address 10.9.1.3 {
            prefix-length: 24
        }
    }
    ethernet eth2 {
        address 192.168.1.3 {
            prefix-length: 24
        }
    }
}
```

A.3.2 Static route

- **Router 1:**

```
protocols {
    static {
        route 192.168.0.0/24 {
            next-hop: 172.16.30.3
        }
        route 10.9.1.0/24 {
            next-hop: 172.16.20.2
        }
        route 0.0.0.0/0 {
            next-hop: 192.168.0.100
        }
    }
}
```

- **Router 2:**

```
protocols {
    static {
```

```
            route 192.168.0.0/24 {
                next-hop: 172.16.20.1
            }
            route 192.168.1.0/24 {
                next-hop: 10.9.1.3
            }
            route 172.16.30.0/24 {
                next-hop: 172.16.20.1
            }
            route 0.0.0.0/0 {
                next-hop: 172.16.20.1
            }
        }
    }
```

- **Router 3:**

```
protocols {
    static {
        route 192.168.1.0/24 {
            next-hop: 172.16.30.1
        }
        route 172.16.20.0/24 {
            next-hop: 10.9.1.2
        }
        route 0.0.0.0/0 {
            next-hop: 172.16.30.1
        }
    }
}
```

A.3.3 RIP

- **Router 1:**

```
policy {
    policy-statement ”EXPORT_CONNECTED” {
        term1 {
            from{
                protocol: ”connected”
            }
            then {
                action: ”accept”
            }
        }
    }
}
```

```
protocols {
    rip {
        interface eth0 {
            address 192.168.0.1 {
            }
        }
        interface eth1 {
            address 172.16.20.1 {
            }
        }
        interface eth2 {
            address 172.16.30.1 {
            }
        }
        export: ″EXPORT_CONNECTED″
    }
}
```

- **Router 2:**

```
policy {
    policy-statement ″EXPORT_CONNECTED″ {
        term1 {
            from{
                protocol: ″connected″
            }
            then {
                action: ″accept″
            }
        }
    }
}
protocols {
    rip {
        interface eth0 {
            address 172.16.20.2 {
            }
        }
        interface eth1 {
            address 10.9.1.2 {
            }
        }
        export: ″EXPORT_CONNECTED″
    }
}
```

- **Router 3:**

```
policy {
    policy-statement ”EXPORT_CONNECTED” {
        term1 {
            from{
                protocol: ”connected”
            }
            then {
                action: ”accept”
            }
        }
    }
}
protocols {
    rip {
        interface eth0 {
            address 172.16.30.3 {
            }
        }
        interface eth1 {
            address 10.9.1.3 {
            }
        }
        interface eth2 {
            address 192.168.1.3 {
            }
        }
        export: ”EXPORT_CONNECTED”
    }
}
```

A.3.4 Single-area OSPF

- **Router 1:**

```
policy {
    policy-statement ”EXPORT_CONNECTED” {
        term1 {
            from{
                protocol: ”connected”
            }
            then {
                action: ”accept”

¿}
        }
    }
}
```

```
protocols {
    ospf4 {
        router-id: 10.0.1.1
        area 0.0.0.0 {
            interface eth0 {
                address 192.168.0.1 {
                }
            }
            interface eth1 {
                address 172.16.20.1 {
                }
            }
            interface eth2 {
                address 172.16.30.1 {
                }
            }
        }
        export: "EXPORT_CONNECTED"
    }
}
```

- **Router 2:**

```
policy {
    policy-statement "EXPORT_CONNECTED" {
        term1 {
            from{
                protocol: "connected"
            }
            then {
                action: "accept"
            }
        }
    }
}
protocols {
    ospf4 {
        router-id: 10.0.2.2
        area 0.0.0.0 {
            interface eth0 {
                address 172.16.20.2 {
                }
            }
            interface eth1 {
                address 10.9.1.2 {
                }
            }
```

```
            }
            export: ”EXPORT_CONNECTED”
        }
    }
```

- **Router 3:**

```
policy {
    policy-statement ”EXPORT_CONNECTED” {
        term1 {
            from{
                protocol: ”connected”
            }
            then {
                action: ”accept”
            }
        }
    }
}
protocols {
    ospf4 {
        router-id: 10.0.3.3
        area 0.0.0.0 {
            interface eth0 {
                address 172.16.30.3 {
                }
            }
            interface eth1 {
                address 10.9.1.3 {
                }
            }
            interface eth2 {
                address 192.168.1.3 {
                }
            }
        }
        export: ”EXPORT_CONNECTED”
    }
}
```

A.3.5 Multi-area OSPF

- **Router 1:**

```
policy {
    policy-statement ”EXPORT_CONNECTED” {
        term1 {
```

```
                from{
                    protocol: ”connected”
                }
                then {
                    action: ”accept”
                }
            }
        }
    }
    protocols {
        ospf4 {
            router-id: 10.0.1.1
            area 0.0.0.0 {
                interface eth2 {
                    address 172.16.30.1 {
                    }
                }
            }           area 0.0.0.2 {
                interface eth0 {
                    address 192.168.0.1 {
                    }
                }
                interface eth1 {
                    address 172.16.20.1 {
                    }
                }
            }
            export: ”EXPORT_CONNECTED”
        }
    }
```

- **Router 2:**

```
    policy {
        policy-statement ”EXPORT_CONNECTED” {
            term1 {
                from{
                    protocol: ”connected”
                }
                then {
                    action: ”accept”
                }
            }
        }
    }
    protocols {
        ospf4 {
```

```
            router-id: 10.0.2.2
            area 0.0.0.1 {
                interface eth1 {
                    address 10.9.1.2 {
                    }
                }
            }           area 0.0.0.2 {
                interface eth0 {
                    address 172.16.20.2 {
                    }
                }
            }
            export: ”EXPORT_CONNECTED”
        }
    }
```

- **Router 3:**

```
    policy {
        policy-statement ”EXPORT_CONNECTED” {
            term1 {
                from{
                    protocol: ”connected”
                }
                then {
                    action: ”accept”
                }
            }
        }
    }
    protocols {
        ospf4 {
            router-id: 10.0.3.3
            area 0.0.0.0 {
                interface eth0 {
                    address 172.16.30.3 {
                    }
                }
            }           area 0.0.0.1 {
                interface eth1 {
                    address 10.9.1.3 {
                    }
                }
                interface eth2 {
                    address 192.168.1.3 {
                    }
                }
```

```
            }
            export: "EXPORT_CONNECTED"
        }
    }
```

A.3.6 iBGP

- **Router 1:**

```
protocols {
    bgp {
        bgp-id: 10.0.1.1
        local-as: 65000
        peer "10.0.2.2" {
            local-ip: "10.0.1.1"
            local-as: 65000
            next-hop: 10.0.1.1
        }
        peer "10.0.3.3" {
            local-ip: "10.0.1.1"
            local-as: 65000
            next-hop: 10.0.1.1
        }
    }
}
```

- **Router 2:**

```
protocols {
    bgp {
        bgp-id: 10.0.2.2
        local-as: 65000
        peer "10.0.1.1" {
            local-ip: "10.0.2.2"
            local-as: 65000
            next-hop: 10.0.2.2
        }
        peer "10.0.3.3" {
            local-ip: "10.0.2.2"
            local-as: 65000
            next-hop: 10.0.2.2
        }
    }
}
```

- **Router 3:**

```
protocols {
    bgp {
        bgp-id: 10.0.3.3
        local-as: 65000
        peer ”10.0.1.1” {
            local-ip: ”10.0.3.3”
            local-as: 65000
            next-hop: 10.0.3.3
        }
        peer ”10.0.2.2” {
            local-ip: ”10.0.3.3”
            local-as: 65000
            next-hop: 10.0.3.3
        }
    }
}
```

A.3.7 eBGP

- **Router 1:**

```
protocols {
    bgp {
        bgp-id: 10.0.1.1
        local-as: 65000
        peer ”172.16.20.2” {
            local-ip: ”172.16.20.1”
            local-as: 65001
            next-hop: 172.16.20.1
        }
        peer ”172.16.30.3” {
            local-ip: ”172.16.30.1”
            local-as: 65002
            next-hop: 172.16.30.1
        }
    }
}
```

- **Router 2:**

```
protocols {
    bgp {
        bgp-id: 10.0.2.2
        local-as: 65001
        peer ”172.16.20.1” {
            local-ip: ”172.16.20.2”
            local-as: 65000
```

```
                next-hop: 172.16.20.2
            }
            peer ″10.9.1.3″ {
                local-ip: ″10.9.1.2″
                local-as: 65002
                next-hop: 10.9.1.2
            }
        }
    }
```

- **Router 3:**

```
protocols {
    bgp {
        bgp-id: 10.0.3.3
        local-as: 65002
        peer ″172.16.30.1″ {
            local-ip: ″172.16.30.3″
            local-as: 65000
            next-hop: 172.16.30.3
        }
        peer ″10.9.1.2″ {
            local-ip: ″10.9.1.3″
            local-as: 65001
            next-hop: 10.9.1.3
        }
    }
}
```

Appendix B

Step by Step: Building a Linux Router

1. Everything starts with downloading a LiveCD from Vyataa official website. *http://www.vyatta.com/download/index.php*
2. When you finishing downloading an ISO image, it is a time to turn it into a LiveCD. Creating a LiveCD is easy and it can be accomplished on either the machines running Microsoft Windows or the machines running Linux.
    - Windows-based options - There are many tools availabe including Roxio, Nero and so on.

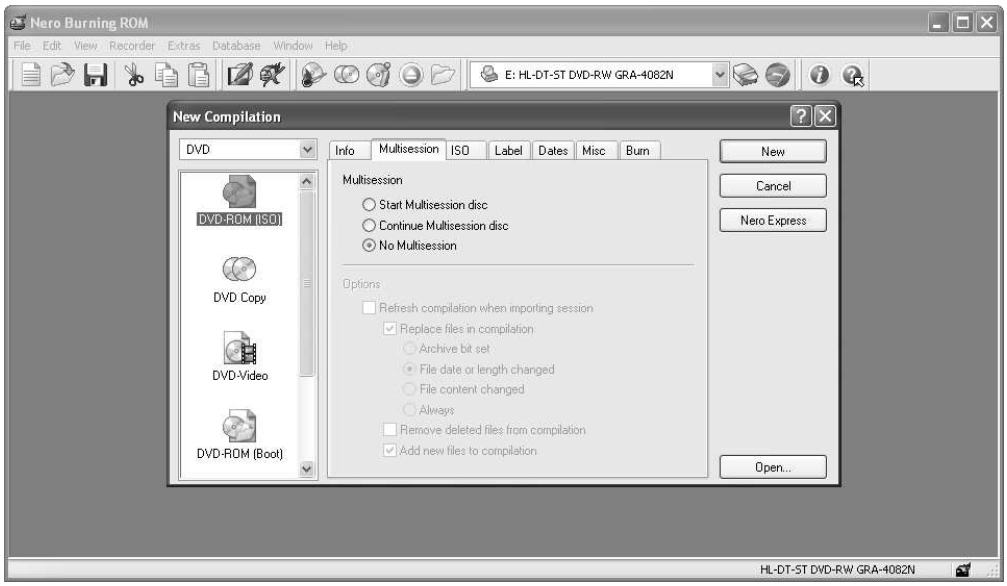

Figure B.1: An example for Windows-based software, Nero Burning Roms

    - Linux-based options - You can should to use GUI-based CD the authoring software or a CLI command such as ”cdrecord”

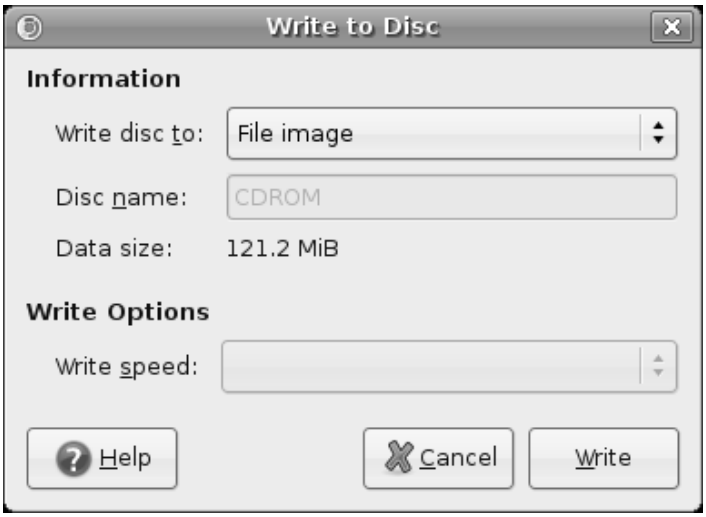

Figure B.2: An example for Linux-based software, CD/DVD creator

```
root@userPC:/home/user# cdrecord -vv livecd-vyatta-2.2.iso
cdrecord: No write mode specified.cdrecord: Assuming -tao mode.
cdrecord: Future versions of cdrecord may have different drive dependent defaults.
cdrecord: Continuing in 5 seconds...
...
...
...
...
Fixating...
Fixating time:    0.234s
cdrecord: fifo had 1901 puts and 1901 gets.
cdrecord: fifo was 0 times empty and 1821 times full, min fill was 93%.
root@userPC:/home/user#
```

Figure B.3: An example for creating LiveCD using Linux CLI

3. Now we have a LiveCD. What we have to do next is to boot from the LiveCD and the first step to use the LiveCD is to verify and set boot order in BIOS by selecting CD-ROM first then save, exit, insert the LiveCD into the CD-ROM drive and reboot. See Figure B.4.

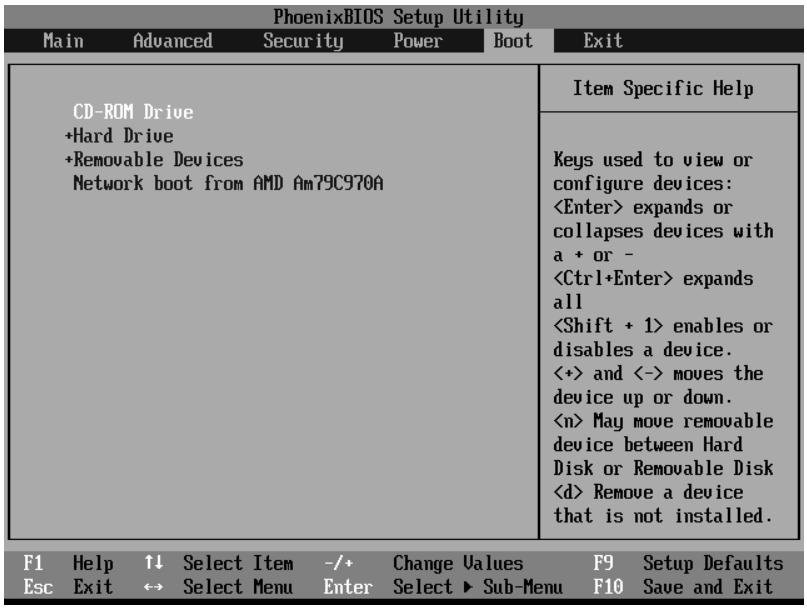

Figure B.4: Booting off order in BIOS

4. When the device finished from booting from the LiveCD, log in as user ”root” with a default password ”vyatta” See Figure B.5.

```
Welcome to Vyatta - vyatta tty1
vyatta login: root
Password:
Last login: Thu Apr 10 21:46:11 2008 on tty1
Linux vyatta 2.6.20 #1 SMP Wed Oct 24 01:04:49 PDT 2007 i686
Welcome to Vyatta.
This system is open-source software. The exact distribution terms for
each module comprising the full system are described in the individual
files in /usr/share/doc/*/copyright.
vyatta:~# _
```

Figure B.5: The first login as root with a Vyatta default password

5. The next step for installation to the hard drive is to run the installation script. At the root prompt type ″install-system″ and hit the [Enter] key. The installation script will ask you a series of questions as configuring software to suit your individual different kind of hardware situations. Eash question are shown in ″[ ]″. See Figure B.6.

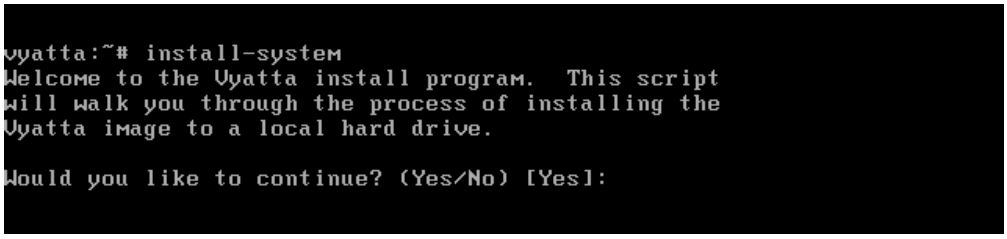

Figure B.6: To install to the hard drive by a command ″install-system″

6. The next step is to select the partitioning scheme. Vyatta suggested picking up ″Auto″ option. Automatic partioning creates two partitions on the disk. One is for root partition which holds majority of files including the system binaries and log files. The second partitions holds configuration informations. It is a good idea to separate routing configuration from system partition because it helps protect the configurations from being deleted during system upgrade. We selected ″Auto″. See Figure B.7.

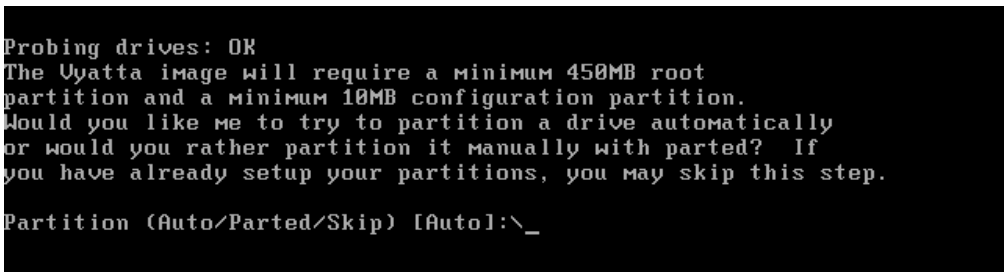

Figure B.7: To select the partitioning scheme

7. The next step is to choose the drive where the software will be installed. The default location is the first drive in the chain. Typical names are ″sda″ or ″hda″. As we have only one drive. We have no other ways except to select ″sda″ by default. See Figure B.8.

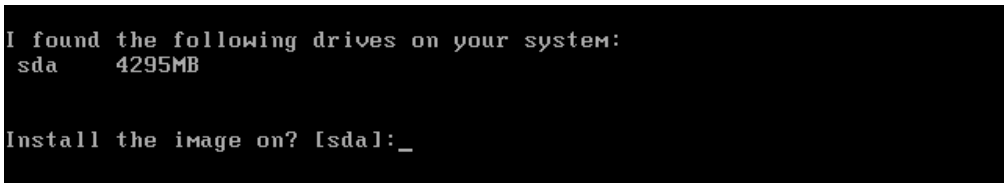

Figure B.8: To choose the drive where the Vyatta software to be installed

8. Then we need to select the size for root and configuration partition. Vyatta strongly suggested to select minimum 450MiB for root partition to give a plenty of space for additonal updates and log files. The installation script will automatically subtract 10MiB from the total disk size to reserve space for a configuration partition. You can choose more space for the configuration partition by choosing a lower value here. We used the default value passing by the step by pressing only [Enter] key. See Figure B.9.

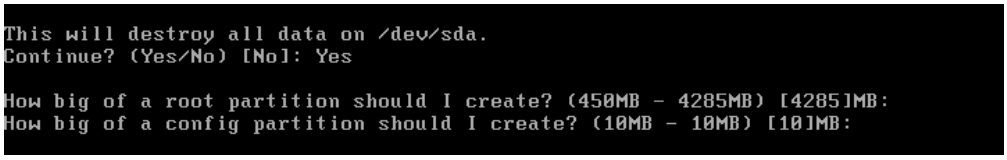

Figure B.9: To select the size for root and configuration partitions

9. The system would ask you where to copy the configuration file. As we see in this example, a normal Vyatta installation suggested to use this path */opt/vyatta/etc/config/config.boot*. We pressed [Enter] key to accept the default "config.boot" file. See Figure B.10.

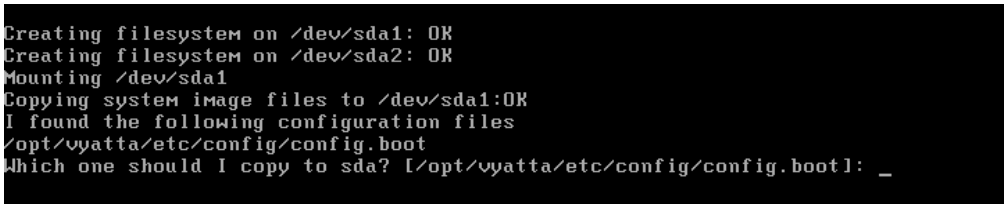

Figure B.10: Default path to copy the configuration file

10. The next step is to install a linux bootloader. Briefly, the bootloader is the first software program that run when the computer starts. It is responsible for loading and then transfering control to an operating system kernel software. The bootloader for Linux is called GRUB which stands for Grand Universal Boot Loader. Once the installation completed, the root prompt return. See Figure B.11.

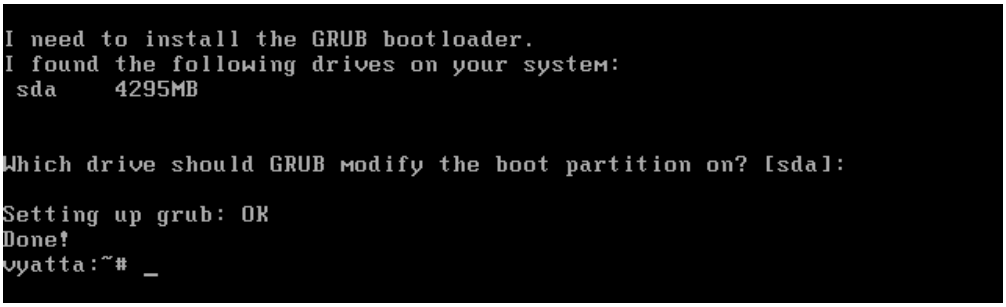

Figure B.11: To install the GRUB bootloader

Appendix C

Glossary

AS: see Autonomous System.

Autonomous System: a routing domain that is under one administrative authority, and which implements its own routing policies. Key concept in BGP.

BGP: Border Gateway Protocol. See chapter 2.1.3.

Cfengine: a policy-based configuration management system. See chapter 3.2.

Dynamic Route: A route learned from another router via a routing protocol such as RIP or BGP.

EGP: see Exterior Gateway Protocol.

Exterior Gateway Protocol: a routing protocol used to route between Autonomous Systems. The main example is BGP.

IGP: see Interior Gateway Protocol.

Interior Gateway Protocol: a routing protocol used to route within an Autonomous System. Examples include RIP, OSPF and IS-IS.

Live CD: A CD-ROM that is bootable. In the context of Vyatta, the Live CD can be used to produce a low-cost router without needing to install any software.

OSPF: See Open Shortest Path First. See chapter 2.1.2.

Open Shortest Path First: an IGP routing protocol based on a link-state algorithm. Used to route within medium to large networks.

Promise theory: a theory to describe and model an autonomous system. See chapter 3.1.

RIB: See Routing Information Base.

RIP: Routing Information Protocol. See chapter 2.1.1.

Routing Information Base: the collection of routes learned from all the dynamic routing protocols running on the router which can be subdivided into a Unicast RIB for unicast routes and a Multicast RIB.

Static Route: A route that has been manually configured on the router.