Analyzing the applicability of the Virtual IP stack in FreeBSD as a platform for a virtualized IP router.

M.Sc. Project.

Kristen Nielsen
kristen@diku.dk, krn@krn.dk

August 31, 2010
Final Version 1.0

Department of Computer Science
University of Copenhagen
Denmark

Contents

List of Figures

Acknowledgments

At the last day of the present project we would like to make a few comments on the process leading up to the start of this project and thank the people that helped us during the period we worked at the project

Originally the project was started during a period where we was waiting to defend of our master thesis project in beginning of 2008, when the initial discussions was taken with Professor Eric Jul at DIKU. From that time until 2010 where we continued working on the project nothing really happened except the FreeBSD virtual network environment projects called IMAGE and VNET was developing and finally released in winter 2010 with the release of the FreeBSD 8.0-Rel. This introduced the first official release of the FreeBSD virtual network environment that we use in this project. This of-cause had the dis-advantage of a very limited access to updated documentation but great help and information was found at the Internet especially status reports from the FreeBSD developer summits in the last 2-3 years, including the one held at BSDcan may 2010 presenting the released version of the virtual network environment.

We would like to thank the people at DIKU helping us with all aspects of this project including project guidance, passing all kinds of administrative obstacles. Special and dedicated thanks goes to: Associate professor Klaus Hansen for mentoring during the project and Andrzej Filinski for helping especially with study administrative aspects during the project.

Most of the project is written at my home in Valby, but nevertheless the DIKU Kantine and all folks and friends there was the place where the social parts of the project took place. There hasn't been a single day where the social fellowship of the DIKU Kantine and its services let me down.

Thanks to every one for being there.

Kristen Nielsen
Valby, Denamrk
30. August 2010.

This project report and the referenced documents are available in electronic form from my website: http://krn.dk/xorp-virt.

Referenced documents will be removed if copyrights are violated.

Chapter 1

Introduction

The idea of Packet switched networking systems has been around since early 1960s, where the concept was explored independently by two recherches: Donald Davies at the National Physical Laboratory in the UK and Paul Baran at the RAND Corporation[1] in Los Angeles, USA. Leonard Kleinrock, University California Los Angeles (UCLA) contributed the early research in queuing theory that is an essential part of packet switching.

Since the beginning of the packet switched network area, packet routers have been the central part of this network type. Packet switching is described in the article: *packet switching* [1] at Wikipedia as:

> Packet switching is a digital networking communications method that groups all transmitted data - regardless of content, type, or structure - into suitably-sized blocks, called packets. Packet switching features delivery of variable-bit-rate data streams (sequences of packets) over a shared network. When traversing network adapters, switches, routers and other network nodes, packets are buffered and queued, resulting in variable delay and throughput depending on the traffic load in the network.

One of the early and widely known implementations of a packet switching gateway (today known as routers) was the IMP [2] (Interface Message Processor) that was used to connect participant networks to the ARPANET from late 1960s to 1989 where routers were generally commercially available. The Internet as we know it today is a successor of the ARPANET where the basic packet switching protocols were developed, deployed and tested in a wider scale.

1.0.1 Adressing

From the very beginning of packet switching networks – the concept of addresses assigned to source and destination hosts has been an essential property of this type of network. The Internet Protocol[2] (IP) has since 1983 been used as the protocol at the ARPANET, and since then – enabling the Internet to develop and expand to the Internet we know today. Packets of the TCP/IP protocol contains addresses of the source host (sending host) and of the destination host (the host intended to receive the packet) as the two most important

[1] Project RAND began after World War II as a special initiative within the Douglas Aircraft Company of Santa Monica, California. The purpose was to continue in peacetime the advances in knowledge that civilian research scientists had been recruited to develop during the war. In 1948, Project RAND separated from Douglas Aircraft and became the RAND Corporation, an independent, nonprofit organization dedicated to promoting scientific, educational, and charitable purposes for the public welfare. See webpage `http://rand.org`

[2] The Internet Protocol (IP) was was developed during the 1970s and replaced the ARPANET NCP (Network Control Program) during 1980 - 1983. January 1st. 1983 the ARPANET changed from the NCP to the TCP/IP protocol suite and since then IP addresses has been the base transport and addressing system used at the ARPANET/Internet.

addressing fields of the protocol header. See figure1.1 for a principle IP packet. Every host connected to the Internet has been assigned an unique IP address at which it can be contacted and use as sending and receiving address when communicating with other hosts on the Internet.

When a host wants to communicate with another host connected to the Internet, it writes the address of the destination host and its own address in the packet header. This header data stays unchanged when the packet travels from the sending host to the receiving host.

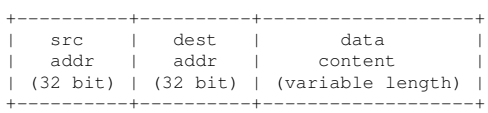

Figure 1.1: The picture shows a simplified IP packet, with the source address field (src addr) and the destination address field (dest addr) and the data to be transported from the source address to the destination address.

An IP address range is assigned to each Internet access interface (Access Interface to the Internet network) and hosts connected via this line must have assigned an address within the assigned IP address range of the access interface. Each Internet access interface is connected to a router. Routers exchange information about which IP address ranges they have assigned to their interfaces. The *routing table* inside each router is created from these informations. The routing tables is used by the router to decide where to send an incoming packet. The router looks up the destination address in the routing table, and sends the packet off in the direction that the route table entry is indicating. This process is called *packet routing* or in daily speak just *routing*.

Figure 1.2 shows a packet switched network consisting of a number of packet routers. Two access lines is depicted, each has been assigned a unique IP address range. At each access line one host is connected which has been assigned a unique IP address from the IP address range assigned to the access line. Assigning of addresses is performed by the network administration of the network, which normally has been assigned a number from a larger IP address blocks to be used in the network. Typically IP addresses is provided by the IP network service provider who gets them from the regional office that administers IP address ranges on behalf of The Internet Assigned Numbers Authority, (IANA)[3] . In Europe the regional office is RIPE and is located in Amsterdam, NL.

1.1 IP packet routing

Routing of IP packets is the task that IP routers perform. When a IP packet arrives at the router, it looks at the destination IP address of the packet and decides which interface the packet is forwarded to. Each router plays its part of "the network wide routing task" where the common goal of all routers is getting each packet from its sending host to its destination host in the most efficient and direct way. Each router can base its routing decision on a number of criterias, which all are stored in the *routing table* of the router. A number of details that are part of the routing decision are summarized in one parameter in the *routing table* normally called *cost* or *metric*. In this text we will use the term *cost* and we describe the *cost* parameter and how it is created in more detail in the following paragraph. The router always selects the most precise route, for a given IP address being looked up, which means the route to the smallest IP address range, in the *routing table*, matching the destination address of the IP packet being routed. If more route candidates appear then the route candidate with the lowest *cost* or *metric* value is chosen.

[3]IANA is available at http://iana.org.

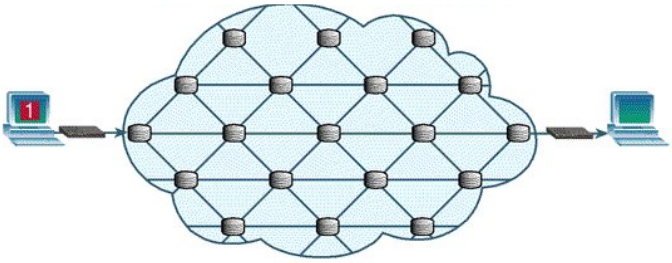

Figure 1.2: Packet switched network. The figure shows a packet switched network with two hosts connected at either side (left and right host). When the left host communicates with the right host, it sends the communication divided into packets, one after the other, addressed to the destination host (here the right host). Each packet travels from the left host into the network. Each network node (router) has an internal *routing table* with routes for any address known to the node, which is used to decide which direction to send an incoming packet to reach its destination address. In this example each node in the network looks up the destination address in each packet and sends it off in the direction that the routing table is indicating, which is in the direction towards the right host.

1.1.1 What is an IP route

An IP route is a set of data that describes in which direction to send IP packets for the network range that the IP route is valid for. Routes are stored in the *routing table* which is located inside each router in the network. Typically a route can be valid for a contiguous range of existing IP address ranges located somewhere in the network. Several neighbor IP address ranges that is physically located close to each other in the network, may share a route when they can be reached via the same *next hop* router. In the *routing table* the route is represented with an IP address and a subnet mask, exactly like an IP subnet definition. Here it means that all IP addresses within the IP address range that the route (IP subnet) describes can be reached by using this route.

The *routing table* contains all known routes seen from the router where it resides. This means that the *routing table* is the routers own view of the network topology of the network it is part of – seen from its location in the network. The *route table* is normally located in memory inside each router, see figure 1.3 for an small example of a *route table*. The *routing table* knows which interface of a router to send a packet out of, and to which IP address the packet should be sent to, to travel in the direction towards the destination host.

When an IP address is being looked up in the *routing table* it matches the smallest IP subnet present in the routing table where the IP address being looked up is within the address range of the route. To clarify this we give a small example: If the *routing table* contains 2 entries `10.0.0.0/16`[4] and `10.200.18.0/24` and the packet that currently is being processed in the router have a destination address of `10.200.18.18`, then both the routing table entries `10.200.18.0/24` and `10.200.0.0/16` will match but when looking for the most specific route (i.e. the most specific route, which is the router with the highest network prefix value) then only the `10.200.18.0/24` will match this criteria,

[4]Subnet masks have the form: `255.255.0.0` and network prefixes have the `/16` form. Network prefixes are a shorthand version of subnet masks. Subnet masks informs the IP protocol code which bits of an host IP address is holding the network address and what bits of the IP address that holds the host address at the network. When subnet masks are written in binary form they contains a number of contiguous one-bits followed by a number of contiguous zero-bits, in all 32 bits together e.g. `11111111.11111111.00000000.00000000` . The network prefix is the number of one-bits in the subnet mask. The higher prefix number the smaller subnet (IP address range), and the more specific are the route being described.

and the route that is contained in this *routing table* entry is used for sending the packet off to the *next hop* router.

Routing tables consists of all routes known to the router. Each entry in the routing table consists of the route and the associated network mask that defines the IP subnet range for which the route is valid for and the interface to send the packets out of to reach the *next hop* router or the destination host itself. Each entry in the routing table also contains a *cost* or *metric* parameter. Cost is a typeless numerical value that is attached to each *routing table* entry. The cost value sums up all details of each routing table entry into one numerical value. An example: routes announced via the BGP (Boarder Gateway Protocol) [3] should have their original cost inserted into the *routing table* but routes received by the OSPF (Open Shortest Path First) [4] routing protocol should have their cost value added with the constant 10. This simple method makes BGP routes take precedence over OSPF routes in the given router. The cost calculation algorithm is decided by the network administrators when configuring the routers. The cost value only has local significance to the router. Cost can be calculated by many different parameters, e.g. link speed or price of the link the route is using in the next hop direction. Cost doesn't need to have anything to do with price but can also be used to select e.g. faster links over slower links. Having different cost values for the same route table entry, is an often used method to prioritize primary links over backup links or cheaper IP WAN links over more expensive. Many aspects of cost calculation are configured by the network administrators and the cost value consists of a combination of technical and business political aspects

Address	Mask	Next Hop	Interface	Protocol	Age	Metric
0.0.0.0	0.0.0.0	80.124.1.1	2	Default	0	1
5.0.0.0	255.0.0.0	90.124.100.100	1	Static	0	1
73.2.3.0	255.255.255.252	80.124.10.240	2	Static	0	1
73.6.1.0	255.255.255.248	80.124.10.240	2	Static	0	1

Figure 1.3: An example of a routing table in a traditional router. The route is described by the *Address* and the *Mask* (subnet mask) columns. This is at the same time the IP address range for which the `Next Hop` router is the node to route IP packets to, when routing IP packets with destination address within this IP address range. (matching the route table entry) The `Interface` field tells which Interface to send the packet out of to reach the `Next Hop` IP address. The `Protocol` field indicate where the route originates from i.e `static` means configured manually by the network administrators. `Age` shows how old this route entry are and `Metric` tells how many hops (number of intermediate routers) the Next Hop router is away from this router. `Metric` has the same function as the *cost* value described in the text above.

1.1.2 Queues

Routers apply queues at their interfaces. Queues are of the First In - First Out (FIFO) type. Queues can be applied at the incoming side of an interface (receive queue) or at the outgoing direction of an interface (transmit queue).

Transmit queues are used to queue packets waiting to be sent out of an interface. An interface can have one or more transmit queues, if more queues exists the queues connected to each interface is prioritized among each other, so packets in the highest priority queue is sent before packets in the next highest prioritized queue and so on. Each time a packet is to be selected for transmission the queues are scanned from the highest priority queue and towards the lowest priority queue, the first queue with a packet ready for transmission is selected and the first packet in the queue is picked.

Receiving queues located between the interfaces and the packet routing process, are used to queue packets waiting to be routed and if we use different traffic priorities queues are used to sort incoming packets in different priorities. Each queue has an assigned priority to it. Each time a new packet is to be routed, the routing process takes the first packet from the highest priority queue at the receiving interface and processes it. The result is that high priority packets are being routed before lower priority packets.

1.2 Virtual Private Networks

Newer types of routers are able to handle Virtual Private Networks (VPN) which is logical closed networks that is used within e.g. companies to connect different physical sites in a closed network. VPN networks also need routers to perform the packet routing for the packets traversing them. This can be done by individual routers but is often done by Virtual router instances of the same routers that routes the Internet within an administrative domain. An administrative domain is a group of routers administered together, e.g. an ISP (Internet Service Provider) or company network. VPN routers are normally deployed within an administrative domain and VPN peering between administrative domains is done by agreement between different administration networks – if at all.

Each packet entering a VPN router have to be identified to belong to either the Internet or to a specific VPN. This is often done with a label inserted in front of the IP header (OSI layer 3[5] e.g. MPLS (Multi Protocol Label Switching) but it can be done in various ways e.g. at layer 2 where VLAN (Virtual Local Area Network) is a possible method.

In the following section we will look more into the aspects of VPN networks and the technologies used to implement these.

1.3 IP domains

IP domain or IP realm are two words we use indistinguishably to describe an IP world[6] Two different IP domains are two total independently IP worlds that doesn't share anything between them, and that doesn't know the existence of the other part. Routers that are able to handle more IP domains or IP realms are able to distinguish an IP address in one IP domain from the IP address with the same numerical value in the other IP domain. Routers with the ability to handle addresses in more IP domains makes the router able to perform routing in more IP domains, as it can distinguish the IP addresses that enters the router from one IP domain from the IP addresses that enters the router from another IP domain. IP packets from different IP domains normally enters via different channels or interfaces, or by other means are marked so the router knows to what IP domain each packet belongs to. This is what we during this project calls a Virtual Router.

When networks are created by virtualized routers the routers must have the ability to distinguish each network from the other networks handled. Figure 1.4 shows this with an example of 3 customer networks handled by virtualized routers.

[5]Network protocols are divided divided into layers. ISO International Standards Organization) have defined the OSI (Open Systems Interconnect) 7 layered network protocol model that is a framework for understanding, describing and developing network protocols. Layer 1 is the lowest layer and closest to transmission interface and layer 7 is the highest layer is the Application layer that provides services directly to user applications. See appendix A for a short description of the OSI model.

[6]An IP world, IP domain or IP realm is an independent IP address space, no addresses are used before the administrators place them into the IP domain. IP domains can be used to several things, separation of networks, test networks, VPN networks etc. Often IP domains have some sort of connection with the Real IP world (the Internet address space) maybe through a firewall or similar security device, controlling access to and from the IP world. Traffic that flows into an IP world must be able to see some part of the Internet address space (i.e. via routing or (Network Address Translation (NAT) [5] to be able to address hosts in the other IP domain. The uses of these features are widely deployed.

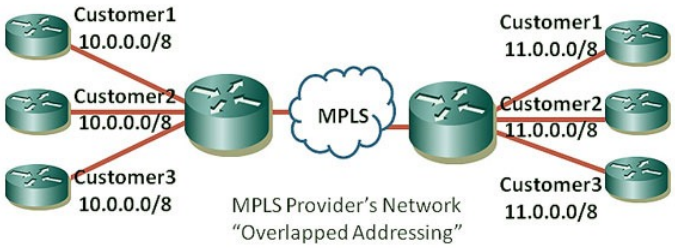

Figure 1.4: 3 customer networks are handled by one physical router with 3 virtual routing instances. Each customer network uses exact the same IP address range which are complete separated from each other when in transport from the left location of the customer to the right location of the same customer. The network cloud between the two routers is operating as a Multi Protocol Label Switch (MPLS) network which enables separation between virtual private networks (VPN) during transport in the MPLS cloud. Picture done by: Shafagh Zandi [6]

Routers in each IP domain must communicate with other routers in the same IP domain. When a router capable of running virtual routing instances is designed, the separation has to be created so each instances only handles its dedicated network. This can be designed in many ways, but if we look at the task at a functional level two main properties is obvious: Separation at the IP packet forwarding level and at the routing process / routing table level. At the IP packet forwarding level the separation is about isolation of the individual IP packets from different IP domains. At the routing table level the task is to be able to handle more route tables typically one for each virtual router instance in the box.

There are many ways to implement this. During this project we will look through the existing XORP project code base and search for the optimal way to implement virtual routers in the existing XORP project, and with help from features provided by the new virtual network environment in FreeBSD 8-Rel.

1.3.1 MPLS - Multi Protocol Label Switching

Multi Protocol Label Switching [7] (MPLS) is a technique that makes it easy to create virtual links between nodes in the same network. MPLS is protocol independent and packet forwarding is based alone on the label attached to the beginning of each packet. Labels are attached to each packet at the first node, where data packets enters the MPLS network and are popped of when the packet leaves the last MPLS node in the network. Before and after the MPLS part of the network, normal IP address based routing is used.

With packet forwarding decisions solely based on the contents of the attached label, (and the rest of the data packet is not needed to be analyzed for routing purposes) makes it easy to create virtual end-to-end circuits for any type of protocols. MPLS network is an overlay network, which means that a new level of network or network layer is introduced in the router. MPLS is existing at the upper half part of layer 2 of the OSI protocol stack, just below layer 3 and is sometimes referred to as a layer 2,5 protocol.

MPLS also has its force in its ability to transport any protocol over a variety of underlaying transport systems and protocols, it supports variable length packets size and not having the large overhead of the small cell size of ATM[7] networks.

[7] ATM networks is a layer 2 transport network with a fixed packet size of 58 byte, compared to Ethernet that has a packet size of around 1500 bytes.

We will go into further details of MPLS later in this report, mainly with the specific aim of presenting a suggestion of the MPLS protocol into the XORP virtual router project.

1.3.2 VRF - Virtual Routing and Forwarding

Virtual Routing and Forwarding (VRF) [8] is a technology that allows multiple instances of a routing table to coexist within the same router. Because the routing instances are separate, overlapping IP addresses can be used without conflicting.

VRF is a key functionality in creating IP Virtual Private Networks (IPVPN) where the VRF function handles the routing function of each VPN instance. The VRF is a local (to the router) entity, creating IPVPNs over distant network nodes the separation of IPVPN traffic from each other during transport, has to be provided by other technologies e.g. MPLS.

1.4 Virtualized IP routers today and in the future.

With the increasing number of IP Virtual Private Networks (VPN) in e.g. Multi Protocol Label Switching (MPLS) networks being offered by IP service providers – the number of routers needed to route the traffic in each network is growing quickly.

When looking at the available routers they are quickly diverted into two main groups: Routers that support Virtual Routing and Forwarding function (VRF) and routers that does not. VRF functionality is a technology that allows multiple virtual instances of a router to exist within the same physical router at the same time. This way a single router contains multiple VRFs typically one for each virtualized IP network.

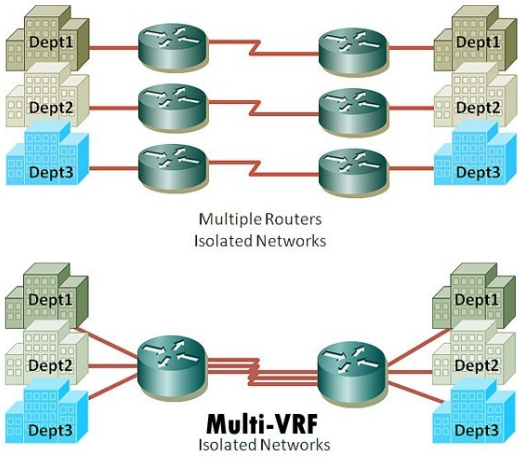

Figure 1.5: When networks are virtualized one router per physical location can handle all the networks in virtual instances. The network separation is maintained as if it is individual routers. The upper picture shows 3 private company networks created with dedicated hardware for each network. The lower picture shows the same 3 private company networks being created or serviced by virtualized routers. The network functionality in both networks are the same, but with shared hardware.

When looking deeper into the topic it is quickly observed that the group of routers that supports virtual routing functions only contains members from major router vendors

like Cisco[8], Extreme Networks[9] and Juniper Networks[10], and the group of routers without support for VRF functionality is containing a large number of commercial routers including the open source based routers available. This has been the situation at the router scene until now.

One of the first aspects to notice when evaluating the reasons behind this is that a suitable Open Source IP stack is not available. The problem with the general available Open Source IP subsystems are that they do not have support for multiple IP domains (or IP realms) which is a basic requirement for virtual routing functions. In the following paragraphs we will summarize the requirements for a single virtual router instance and for multiple virtual router instances.

Routers that supports multiple VRFs are able to emulate multiple router instances in the same box and hereby enables the hardware to perform routing functions for multiple IP networks simultaneously.

The most important functions of a single virtual router instance are: 1) A *routing table* which keeps track of which direction to send IP packets so they reach their intended destination. 2) The support for one or more routing protocols that can exchange route information with peer routers in the (private) network. 3) Handle virtual and/or physical interfaces that connects the router to the (private) IP network. 4) The Ability to route IP packets between its interfaces according to the routing table.

When we add VRFs into the same hardware a few extra requirements for the system appears: 4) The ability to handle IP addresses from/between different IP realms. 5) Ability to connect different IP domains to their appropriate networks. 6) Having a separate routing table for each virtual routing instance (IP domain). 7) The ability to handle routing protocols within each IP domain that exchange route information with peer routers.

Recently support for multiple IP stacks has been proposed for the FreeBSD operating system which then removes one of the obstacles toward the realization of an open source based router that has support for multiple router functions. The proposed change implements support for virtual kernel instances (including network environment)called *Virtual Images* is suggested by Marko Zec, a networking researcher from University of Zagreb, Croatia, and is now generally available in FreeBSD 8-Rel. This sets the scene from where our project starts.

1.5 Method chosen

We will analyze the conformability between the requirements for the proposed virtualized XORP architecture and the capabilities provided by the Virtual Images system in FreeBSD. We will apply a number of measurement methods to quantify the conformability. The methods we will use are: 1) Count the number of extra functions needed to glue the two architectures together. 2). Count the number of needed functionality needed by XORP to handle virtual instances. 3). Count the number of needed workarounds to the FreeBSD Images to be able to handle XORP in a virtualized setup. 4). Other relevant quantitative measurements that express the missing parts needed to make XORP virtualization with FreeBSD Images.

As we are not implementing the project these measurements will be on a higher abstraction level and we are aware of the risk of this is an method that may give an imprecise result, and that a quantitative approach will still be based on a higher level abstraction level

[8]Cisco is the old IP router company, from the beginning IP routing was their core business. Today Cisco are the largest company in the IP routing and switching market. See more at the Cisco homepage http://cisco.com

[9]Extreme Networks is a newer fast growing IP routing and switching company with products for the IP core network, The Data center and service provider market segments. http://extremenetworks.com

[10]Juniper Networks is a newer fast growing company challenging Cisco at its core IP network marked segment, but are also operating in the Switching and Firewalls area. See Juniper Networks home page: http://www.juniper.net

than C++ functions and counts of lines of C++ code. But we still believe we can get some qualitative results in this way.

1.6 Motivation

Today open source routers has no support for virtual instances, which is one of the obstacles that has to be passed before open source based routers will have a realistic position in modern IP backbone core networks. We would like to use our project work to help with passing this obstacle, which is the main reason to choose exactly this topic as our project. Besides this we have a deep interest in IP networks and IP routing, which of cause also is an important reason. Within the time frame available it is not possible also to implement the project as a demonstration project.

Today open source routers has most of the needed functionality required in IP backbone core networks except support for virtual instances and to some extent physical interface types, but this is mostly a hardware and a capacity issue of standard PC hardware, but also to some extent an open source driver (or access to hardware specifications) issue, where support for virtual routing instances is solely a software issue.

1.7 Thesis

Today XORP only supports a single router instance which means it only supports one routing table and one IP network domain (IP realm). We want XORP to have support for virtual routing instances and will examine how well the proposed virtual images for FreeBSD supports the requirements of a virtualized XORP architecture. The result of this work is hoped to be useful for implementing virtual routing instances in XORP at a later point in time.

We will create a proposal for an XORP architecture that supports virtualized routing functions and is based on the FreeBSD Release 8, *Virtual images* for the FreeBSD system and analyze how well the two fits together. This work can be viewed as (or will bee) both an analysis of the FreeBSD virtual network environment support for routers such as XORP with multiple IP domain support, and as a planning phase for extending the XORP project architecture to support multiple routing instances and multiple independent IP network domains (realms).

1.8 Approach

We intend to analyze the facilities provided by Marko Zecs virtualized IP subsystem for FreeBSD (called virtual network environment) and create a proposal for how XORP can get support for virtualized routing instances. In the end we will analyze how well these thing fits together. One of the methods we would use in the evaluation of the conformability is to analyze the set of extra functionality and functions necessary to get these two systems to work together, and analyze what type of functionality these functions has.

1.9 Proposal of a solution

The immediate proposal of a solution is to make use of the high modularity that the XORP project already have in its non virtualized form. All components are separate UNIX processes with a suitable XORP interface called eXternal Resource Locators (XRL) for communicating with other XORP modules. A firsthand solution is to place each IP domain (IP realm) in separate FreeBSD virtual network environment, and startup the necessary XORP processes according to the XORP configuration for the IP realm. We anticipate that the

master instance of the XORP Router Manager Process *rtrmgr* creates and configures virtual images and establishes proper networking setup in each Virtual image according to its configuration file at startup.

Quite some analysis work and design considerations has to be done before choosing the way the XORP Router manager process (*rtrmgr*) communicates with XORP processes in other virtual images. XORP uses eXternal Resource Locators (XRL) for this today, but there is currently no existing IPC channels between Virtual images. Here we can suggest implementing support for UNIX domain pipes or named FIFOs with an endpoint in each virtual images. XRL communication is in principle a full duplex communication, like a remote procedure call (RPC). A XRL function call is made to a XORP module with parameters supplied, and parameters are either returned immediately or asynchronously at a later time via a call back event from the called function. As a first impression it seems that every module that has the possibility to communicate with each other via a seemingly direct channel between the modules. It is currently not obvious to what extent these needs to be between sibling or parent-child virtual images. If XRL communication only will be between parent and child virtual images we do not foresee major problems with implementing this. Extra IPC functionality between Virtual images might be needed to achieve full feature functionality in a virtualized version of XORP.

1.10 Description of work done

During the project we have done the following work:

- Described and analyzed (evaluated) the virtual network environment extensions to FreeBSD, suggested by Marko Zec and implemented in FreeBSD 8-Release.
- Described and evaluated the current XORP project with regards to adding support for virtual images and support for more than one IP realm (IP domain).
- Proposed changes to the XORP architecture design to support virtual routing instances.
- Proposed changes in the XORP configuration language to support configuration of the virtualized version.
- Evaluated the usability (conformability) between our suggested virtualized XORP architecture and the virtual network environment implementation of FreeBSD.

1.11 Evaluation method

The evaluation method we plan to use is based of the evaluation of how much extra functionality that needs to be added to the implementation for the two systems to fit together (or work together). We can use this evaluation technique on several issues (items):

- Counting/evaluate the number of functions needed.
- Counting/evaluate the number of topics/areas where “glue code” is needed.
- Counting/evaluate the line of code needed in every topic/area, and use this as an expression for the misfit.
- Finally suggest which improvements we can suggest or implement at the virtual network environment system to remove some of the identified obstacles.

1.12 Expected results

We expect to be able to validate that there is good conformability between the FreeBSD virtual network environment system and the proposed virtualized architecture for XORP.

1.13 Conclusion

After having analyzed the network properties of virtual images, designed a virtualized XORP architecture and evaluated the conformability of the two, we can conclude that there is a conformability with a grade between good and perfect, on a line from poor to perfect.

The area of least support is Inter Process Communication (IPC) features between XORP processes residing in different virtual images of FreeBSD.

See the full conclusion in chapter 7.

Chapter 2

The FreeBSD virtual network stack

The virtualizing of the FreeBSD kernel started with the *jail* concept described by Poul-Henning Kamp and Robert Watson in the paper: ”Confining the omnipotent root” [9] concept in FreeBSD 4-release marts 2000. Jails was originally a user process environment virtualization.

A few years later Marco Zec wrote a paper ”Implementing a cloneable Network Stack in the FreeBSD Kernel” [10] that discussed the remaining parts of FreeBSD system that still needed to be virtualized. This paper started the process of virtualizing the rest of the FreeBSD kernel.

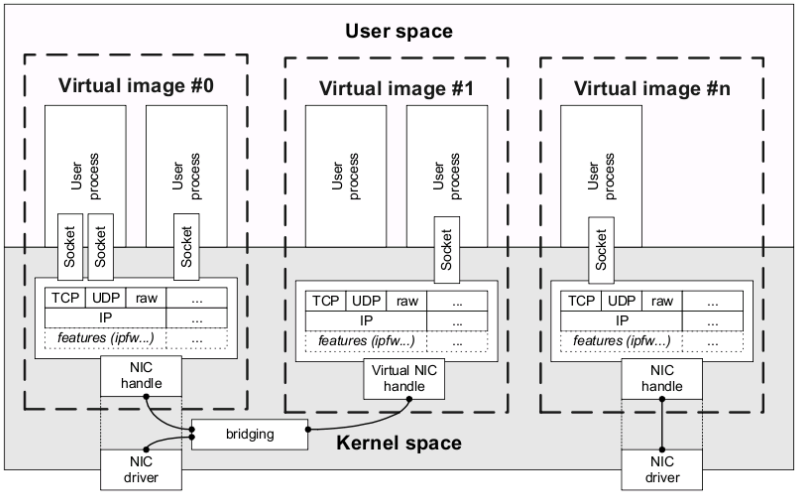

Figure 2.1: Operating system partitioned into virtual images (jails). 3 virtual images exists in the kernel. User processes executing inside these can create a network socket to the network subsystem of the virtual image. Communication between virtual images is via network connections e.g. bridges implemented by the Netgraph [11] system. Virtual images can either connect directly to a NIC or to a virtual interface e.g. bridging interface. The figure is from Marco Zecs paper “Implementing a Cloneable Network Stack in the FreeBSD Kernel” [10]

The first step in virtualizing the FreeBSD system was taken in marts 2000, with FreeBSD 4-Release, which contained the `jail(2)` FreeBSD system call implementation. With *jails* processes could be isolated in separate containers called *jails*. *Jails* provided a virtualized process environment abstraction to the user space, enabling virtualization of the user space environment. This enabled virtualization of the environment for applications running in user space as long as no special kernel abstraction except the IP address, a process tables and a few other kernel administrative objects is needed. At this stage the rest of the kernel did not support virtualization.

Around 2003 the virtualization of the kernel and kernel subsystems came up as a task in the FreeBSD project. To begin with Marco Zec worked on a separate project named Virtual (kernel) Image (VIMAGE), that should demonstrate the concept. In 2008, when the virtualization project was about to be implemented into the FreeBSD project, the VIMAGE project was divided up into two smaller projects, virtualization of the network stack named VNET and virtualization of the kernel subsystems named IMAGE. The VNET task is concerned about providing a virtualized network stack (Ethernet, IP, TCP, ...) and the IMAGE task is concerned about virtualizing all other parts of the kernel (kernel globals, Management Information Base options (MIB), subsystems, etc.)

Implementation of the virtualized kernel subsystem and network stack added virtualization of the process and network environment to the FreeBSD system. First the virtual network stack (VNET) was established and then the virtualization of the kernel subsystems (IMAGE) was implemented, both appeared in the FreeBSD 8-Release in January 2010.

Figure 2.1 above shows a virtualized FreeBSD kernel with VIMAGE functionality. The figure is divided up into a kernel space (the grey shaded area) and the user space area (the light grey shaded area). The boundaries of the virtual kernel images are drawn with broken lines. Each virtual kernel image is like an object instantiation in C++ where each instance have a local dataset for the variables and shares the program code between them. The white boxes marked “User process” shows the virtualized user space where applications are executed. The lowest part of the gray area of the figure, shows how the virtual kernel instances can be connected to the outside world via various network options, e.g. bridges.

Between jails, processes are able to communicate network wise via Unix domain sockets in shared file system space, or via external (or virtualized) networking infrastructure.

Figure 2.1 illustrates the concept of Virtual images (jails) and the separation of the network stack. Each virtual image is completely transparent to user land programs compared to a non virtualized version of the FreeBSD kernel, as all the functionality of the kernel virtualization (virtual images) is hidden in the kernel. The Virtual Images system has as a design goal that ensures that ”user land” binary programs can be moved from a non virtualized FreeBSD kernel into a virtualized version without recompilation. This implicates that the system call interface to the kernel is unaltered when implementing the virtualization code. An administration tool named `jail(8)` controls and administers the creation and deletion of virtual images, including their network interfaces and startup of processes inside virtual images through a new system call created for this. Seen from the outside network and process environment view, each virtual image appears as an independent system however resources such as RAM, disks and other I/O hardware is shared between the virtual instances as in a traditionally single image UNIX operating system. This ensures a better resource utilization but has the disadvantage compared to e.g. IBM-VM that it is not possible to run different flavors of Operating Systems inside virtual images. It is only possible to run the same version as the kernel itself, as it is the kernel that is being virtualized.

2.1 The FreeBSD 8-Release

The FreeBSD 8-Release implementation, is the first production release of the FreeBSD system with extended jail virtualization. (jail + VNET + IMAGE). The new virtual kernel features are not fully production quality yet. The most important issues are the destruc-

tion of virtual kernel elements (object destructor's etc.). This release provides a set of building blocks that can be used to create the environment needed for almost any given purpose. Building blocks are jails, virtual network stacks, routing tables, network elements e.g. Ethernet interface pairs, virtual bridges etc. The use of these building blocks enables the administrator to create virtualized process environments, with or without a dedicated network stack and routing table. From this release of FreeBSD a virtual kernel instance is a jail with an optional independent network stack instance.

The kernel can handle a high number[1] of virtualized network stacks, which can be attached to or shared by one or more jails. This means that individual IP stacks can be chosen per jail. Every process, socket and network interface present in the system are always attached to one, and only one, virtual network stack instance (VNET).

In this project we will use the terms *jail* and *virtual network environment* as a general term when we speaks of the virtualized kernel (IMAGE + VNET) as a whole, and the *VNET* and *IMAGE* terms when referring specific to either of these sub entities.

2.2 Properties of jails.

Jails have 2 major variants: A user land virtualization only, just like an original *jail* and a user land plus kernel virtualization version. The user land only virtualization is useful for running applications in virtual hosts, and the user land with kernel virtualization is useful when the requirements specifies network separation, different routing options, individual firewalls, tunneling, TCP/IP options tweaking etc.

The operation and management of the new functionality is done with parameters added to the `jail(8)` command. The binding of interfaces functionality is integrated into the `ifconfig(8)` utility. The `setfib(8)` command controls what routing table processes in a jail or a set of jails, will be associated with. A small number of additional utilities supports the jails, among these are: `pgrep(1)` which searches and lists processes at the system or in a jail. `jexec(8)` which executes a command inside a jail. `jls(8)` which lists existing jails, `epair(4)` which creates a pair of virtual Ethernet interfaces with a cross-over cable between them. The `if_bridge(4)` is an Ethernet bridge device that bridges Ethernet connections together inside the kernel, just like a hardware Ethernet Bridge, and the `netgraph(4)` [11] subsystem, is a collection of network element building blocks that can be combined into more complex setups.

In FreeBSD 8-Release, the IMAGE and VNET functionality is accessed via extensions to the `jail(8)` command. Jails can be created with various level of virtualized support. The existence of the `vnet` keyword on the `jail(8)` command line decides whether a jail gets an individual network stack instance or shares the network stack with the creating process of the new jail.

Almost every knob and bolt of the virtual kernel instance can be tweaked via the `sysctl(8)` MIB interface.

The user land virtualization provides a simulated virtual machine based on an abstraction made primarily at the `syscall(2)` interface. The user land application runs in a protected environment from other user land processes, and is only offered a simple kernel virtualization with own IPv4/6 IP address for binding sockets, a hostname and a few other properties.

Jails with user land only virtualization[2] shares the following properties:

- Applications that runs inside the virtualized process environment are unable to touch anything outside the jail. Individual jails protects its applications from reaching out

[1] The jail virtualization is a cheap (resource wise) way of virtualizing, the intent is that thousands of virtual hosts can exists within a normal server-grade hardware.

[2] User land only jails are jails without a virtualized IP network stack instance. The `vnet` keyword was not given in the creating `jail (2)` command.

into the host OS jails (from child to parent), or into other sibling jails. Jails each has a separate set of Process Control Blocks (PCB)[3] used by the kernel (jail) to administer and group processes that runs inside the jail.

- Jails have a file system subtree associated, which the root user of the jail can manage with its root privileges. This directory tree are defined at jail-creation time, either as a shared directory tree among several jails or given as a dedicated file system sub tree for each jail to have its own dedicated sub tree. This starting directory of the sub tree is the root directory of the jail its attached to. The directory tree must have the usual UNIX file system disk layout with files existing needed by applications to be run within the jail e.g. shared objects, start up command scrips, password and group files etc. Just like the base OS is needing its file system – so does every jail. The configuration requirement for a jail specifies whether a file sub tree can be shared or not. The applications that runs inside a jail make use of these files. All known normal user and group rights are maintained inside the jail according to the standard `/etc/passwd(5)` and `/etc/group(5)` in the file system sub tree assigned to the jail.

- Jails can have a number of IPv4 or IPv6 IP addresses for binding network interfaces, the first IP address of the supplied IP address list in each family is the primary IP address of that family for this jail, and is used as source address of connections to and from the loopback address (127.0.0.1 for IPv4 and ::1 for IPv6) will be changed to be to or from the primary address of the jail for the given address family. Jails can either inherit the IP addresses of the host system, or be given a list of IP addresses that can be used by the jail.

- Jails can be allowed to create child jails. The number of descendant of this jail can be restricted by the `children.max` parameter. Parent jails can access the child jails, as they are the creator of these. Parent jails can start and stop processes inside child jails, change the parameters of child jails and even remove child jails from the system.

- Jails are identified by a unique Jail-ID *jid* and may have a name set by the creator, which can be used at the same places as the jid can be used in commands managing jails.

- The `persist` keyword at the creating jail command, enables the jail to exist even without any processes running inside it. If no `persist` keyword was given when creating the jail the jails exits or remove itself when the last process in the jail has terminated.

- Jails can be restricted to run on a limited or dedicated cpuset on a multicore system.

- Option `cpu.max` controls each jails use of CPU resources. Jails can be allocated a maximum allowed percentage of the total CPU time to protect the system from runaway processes starving other processes for CPU resources.

Jails created with the `vnet` keyword have in addition to the features listed above for user land virtualization jails, their own virtual instance of the kernel subsystems and share these additional virtualized properties. Local copies of: global kernel variables, callout timers, event handlers, dedicated `sysctl(3)` MIBs, handling of startup and shutdown of each jail and debugging support.

Instances of the virtual network stack (jails with the `vnet` keyword can be shared between several jails. Each network stack instance have a local copy of the following properties: loopback interface (IPv4: 127.0.0.1 and IPv6: ::1), virtual interface of the

[3]The Process Control Block (PCB) is a kernel data structure that holds pointers to processes inside each jail.

`netgraph(4)` subsystem for access to the network stack, Hardware resources (e.g. NIC cards) can be dedicated to a specific virtual network stack, own statistics (counters etc.), own firewalls, IPsec, raw sockets, routing tables, routing sockets, IPv4 and IPv6 udp, tcp and sctp subsystem properties and counters, traffic shapers, IpSec module, Ipfw firewall module, netgraph nodes, divert sockets, interfaces (real and virtual). The aim is to instantiate the complete network stack enabling the complete virtualization of the kernel including network stack and all network subsystems.

Generally the whole virtualization concept of FreeBSD is created with a high level of modularity and granularity, so only the parts that really needs to be virtualized in at given application configuration are virtualized and the parts that doesn't need to be virtualized can be left unvirtualized. This way the usage of resources are optimized and the virtualization overhead of the full system is minimized.

2.3 Way of operation

When a FreeBSD system with jails is booted only the *default* jail is existing. System administrators can dynamically create jails and associate real or pseudo network interfaces and execute user processes in the virtual environments. User processes executing in a jail will be able to interact only with their own network stack instance. Jails are logically organized in a hierarchical tree structure. When a jail creates a new jail it becomes a "child" instance (similar to UNIX processes, or file system directories). Parent jails are allowed to spawn a new process into its child jails e.g. for starting up the programs to be executed or for pure management purposes. The child jail processes are prohibited from manipulating its parent and sibling jails.

Despite the isolation between the jails are an overall design idea it is necessary that jails should be able to communicate if desired by the administrator. This can be done through either bridged Ethernet interfaces or through virtual point-to-point channels constructed via the `netgraph(4)` (see the netgraph project [11]) framework or via `epair(4)` Ethernet devices. It has to be analyzed to what extent shared memory or other IPC (Inter Process Communication) methods can be used for this too.

2.4 Virtual Network Stack examples.

To illustrate the use of jails we have created a simple example shown in figure 2.2 with two virtual hosts connected by a cross-over Ethernet cable, implemented with a `epair(4)` device. IPv4 and IPv6 ping packets is sent from host "left" to host "right" and ping replies are received back at host "left". The command script that generates the setup with FreeBSD jails and the output from the ping test is shown in figure 2.3.

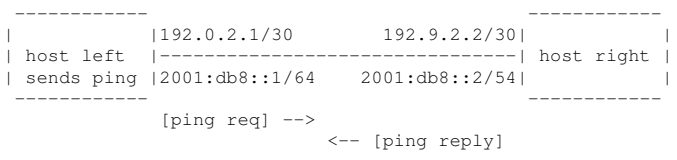

Figure 2.2: The setup consisting of 2 hosts (left and right) each with own dedicated network stack instance. Each host are assigned an IPv4 and an IPv6 IP-address enabling communication over both protocols.

```
001: # Start left and right jails.
002: ljid=`jail -i -c -n left  host.hostname=left.ping.net vnet persist`
003: rjid=`jail -i -c -n right host.hostname=right.ping.net vnet persist`
004:
005: #Create epairs and connect to jails
006: ep=`ifconfig epair create`
007: lmep=`expr ${ep} : '\(.*\).'`  # remove last char of epair dev name
008: ifconfig ${lmep}a vnet ${ljid}
009: ifconfig ${lmep}b vnet ${rjid}
010:
011: # left ljid
012: jexec ${ljid} ifconfig ${lmep}a inet  192.0.2.1/30 up
013: jexec ${ljid} ifconfig ${lmep}a inet6 2001:db8::1/64 alias
014:
015: # right rjid
016: jexec ${rjid} ifconfig ${lmep}b inet  192.0.2.2/30 up
017: jexec ${rjid} ifconfig ${lmep}b inet6 2001:db8::2/64 alias
018:
019: # ping
020: jexec ${ljid} ping  -c 5 192.0.2.2
021: jexec ${ljid} ping6 -c 5 2001:db8::2
022:
023: Output from program line 1-21.
024: PING 192.0.2.2 (192.0.2.2): 56 data bytes
025: 64 bytes from 192.0.2.2: icmp_seq=0 ttl=64 time=0.069 ms
026: 64 bytes from 192.0.2.2: icmp_seq=1 ttl=64 time=0.041 ms
027: (3 lines removed here)
028:
029: --- 192.0.2.2 ping statistics ---
030: 5 packets transmitted, 5 packets received, 0.0% packet loss
031: round-trip min/avg/max/stddev = 0.028/0.044/0.069/0.014 ms
032:
033:
034: PING6(56=40+8+8 bytes) 2001:db8::1 --> 2001:db8::2
035: 16 bytes from 2001:db8::2, icmp_seq=0 hlim=64 time=0.137 ms
036: 16 bytes from 2001:db8::2, icmp_seq=1 hlim=64 time=0.086 ms
037: (3 lines removed here)
038:
039: --- 2001:db8::2 ping6 statistics ---
040: 5 packets transmitted, 5 packets received, 0.0% packet loss
041: round-trip min/avg/max/std-dev = 0.078/0.093/0.137/0.022 ms
```

Figure 2.3: The example creates 2 jails with own dedicated network stack instance in line 2-3, connects these with a cross over Ethernet cable (lines 6-9). Configures an IPv4 and IPv6 network in each jail (line 12-13 and 14-15). Sends IPv4 and IPv6 ping commands test the connectivity (lines: 20-21). Output from ping commands showing packets are sent from host one to host two and reply packets returns back to host one (lines: 24-41).

2.5 Summary

In this introduction to the FreeBSD virtual network stack (jails) we have provided a small overview of the dynamics and possibilities in the design of FreeBSD virtualization setup.

Beginning with jails supporting application hosting in virtual kernel instances that mainly provides application level separation, to the introduction of the full kernel virtualization. The full kernel virtualization features either separate or shared network stack instances and separate or shared routing table instances. Virtual kernel instances features almost any feature of the FreeBSD kernel to be tweaked and manipulated to fit the needs of the task to be solved.

Modularity has been an important part of this project, as various application uses have very different requirements of what parts of the system needs to be virtualized. The ability to only use resources for virtualizing the needed parts of the system for a given application use, saves resources and optimizes the uses of the hardware the system operates on.

The goal for the FreeBSD kernel virtualization project is that thousands of virtual hosts can be running at a normal server grade hardware platform.

The FreeBSD project names the virtual kernel instance for IMAGE and the virtual network stack for VNET, we will use the term *virtual network environment* for both parts combined throughout this report or when we in general refers to the virtual kernel features.

This concludes our introduction to the FreeBSD virtual network stack and analyzing the features it provides.

Chapter 3

Introduction the XORP project

The test in this chapter is an updated and scaled down version from master thesis project from January 2008 [5].

This chapter introduces the eXtensible Open Router Platform (xorp) to the reader unfamiliar with the xorp project and the overall design of xorp. We provide a short introduction to the xorp project goals and to each of the modules which is a part of xorp. We present a short overview of the XORP top-level configuration clauses as we are using these in the redesign of the configuration language for the XORP virtual router support.

This chapter is inspired by the xorp document "Xorp Design Overview" [12] and the "Xorp Software status page" [13].

3.1 Xorp overview

The xorp homepage[1] says it like this:

> "Xorp is the eXtensible Open Router Platform. Our goal is to develop an open source software router platform that is stable and fully featured enough for production use, and flexible and extensible enough to enable network research. Currently xorp implements routing protocols for IPv4 and IPv6 and a unified means to configure them. In future, we would also like to support custom hardware and software forwarding architectures."

Xorp has been around since January 2003, where version 0.1 was released, the xorp project has on a regular basis released versions with enhanced functionality. Today (July 2010), we have release 1.6 of xorp. Xorp have a good overall design documentation which will be referenced throughout this report, but to introduce the reader to the overall design of xorp we have included a (not so) short introduction to the xorp design and modules.

Xorp is running on top of a host operating system (Host OS), which natively is FreeBSD but Linux (more distributions), Mac OS X and even MS Windows 2003 Server are current possible host operating systems. Some xorp features might not yet be implemented at some of the host OS'es.

Generally xorp has modularity as the overall design concept. Every module of the router is a separate Unix process at the host operating system. To facilitate future hardware with more processors or dedicated packet forwarding subsystems all communication between xorp modules is done via a unified communication system that facilitates modules can be executed at different hardware which are connected with some kind of communication path e.g. an IP network, or a high-speed internal backplane bus between the individual processors. In xorp communication between modules is done with eXternal Resource Locators (XRL).

[1] http://www.xorp.org

XRLs looks much like URLs known from the World Wide Web. Figure 3.1 shows a example of an XRL and explains the various parts. The Protocol Family, the Interface specification and the arguments.

```
finder://fea/fti/0.1/add_route?net:ipv4net=10.0.0.1/8&gateway:ipv4=192.150.187.1
+-----   +-- +-- +-- +-------- +-------------------------------------------------
|        |   |   |   |         |                      ======= ==== =============
|        |   |   |   Method    Arguments                 ^       ^          ^
|        |   |   |                          Parameter name -|       |          |
|        |   |   Interface version                                 |          |
|        |   |                              Parameter type --------|          |
|        |   Interface Name                                                    |
|        |                                  Parameter value -----------------|
|        Protocol Parameters
|
Protocol Family
```

Figure 3.1: Example of an xorp external Resource Locater (XRL). The figure define the various parts of an XRL. Arguments starts with the question mark "?" character and are separated with the ampersand character "&". The base example is from the Xorp Inter-Process Communication Library Overview document [14].

Xorp is deliberately not a multi-threaded architecture which reduces the design requirements and complexity of the individual xorp modules and eliminates the limits that a multi-threaded architecture adds to the programming rules.

3.2 Xorp modules

Xorp consists of a number of modules. Every xorp module is implemented as a separate operating system (Unix) process. Throughout this chapter we use the term module but process would have been an equal correct term to use here. The modules are divided into 3 main groups: Management modules, Unicast modules, Multicast modules and the few modules that don't fit into these classes are grouped under the "Other modules" group. The individual xorp modules/processes are depicted in figure 3.2 and described in the remaining parts of this section:

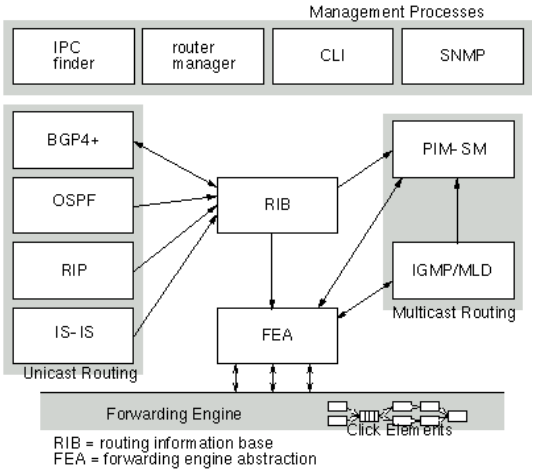

Figure 3.2: Xorp processes overview. The figure is from the xorp Design Overview document [12].

Management modules

Inter-Process Communication (IPC) Finder

The IPC finder is needed by the XRL communication method used among all xorp components. Each of the xorp components registers with the IPC finder. The finder assists the XRL communications by knowing the location of each XRL target, therefore a xorp process does not need to know explicitly the location of all other process, or how to communicate with them. The router manager process *(rtrmgr)* incorporates a finder, so a separate finder process is only needed if the *rtrmgr* is not being used such as during testing. For more information about the IPC finder and XRLs see these xorp documents: "Xorp Inter-Process Communication Library" [14] and "Xorp XRL Interfaces: Specification and tools" [15].

Xorp Router Manager *(rtrmgr)*

The *rtrmgr* is the xorp process responsible for starting all components of the router, to configure each of them, and to monitor and restart any failing process. It also provides the interface for the CLI to change the router configuration. For more information about the *rtrmgr* see "Xorp Router Manager Process" [16].

Command Line Interface (CLI) *xorpsh*

The CLI can be used by a user to access the router, view its internal state, or to configure it on-the-fly. Its functionality is closely related to the *rtrmgr*. However, because the robustness of the *rtrmgr* itself is extremely important, all functionality that can be run as a separate CLI process are separated from the *rtrmgr*. The process implementing this CLI functionality is called *xorpsh*. For more information about the CLI and the *xorpsh* process see "Xorp User Manual" [17].

Simple Network Management Process (SNMP) module

This is the SNMP management process of Xorp. It is used for SNMP access to the router. For Example, it can be used to translate SNMP requests into XRL requests. Internally, SNMP will communicate with the other processes using XRLs. For more information about the Xorp SNMP module see the document "Xorp SNMP Agent" [18].

Routing Policy module (Policy) module

This is the Xorp routing policies coordination process[2] The policy module basically works by checking the routes or other data that are policed by the module ensuring that the global policies are respected. As an example: if import of routes from OSPF into BGP are policed by the module, each of the involved modules can ask if a given route is blocked or allowed by the current policy. The policy module interacts with all routing protocols and the Routing Information Base (RIB) and instructs them how to handle the routes flowing to or from the system, export rules from one protocol to another, modify or remove routes as they flow through the system, etc. Currently only unicast policies are supported by the policy module. Policy is a newly added feature of xorp and the number of modules that implements policy functions is growing as this is currently being implemented. For more information about the xorp policy module see the paper: "Decoupling Policy from Protocols: Implementation Issues in Extensible IP Router Software" [19].

[2]The Routing Policies module is not shown on the figure 3.2.

Unicast routing modules

Border Gateway Protocol version 4+ (BGP4+) module

This is the BGP routing daemon. It implements IPv4 and IPv6 unicast routing in a single process, as well as Multicast BGP4 (MBGP4) for both IPv4 and IPv6 multicast RIB for multicast routing purpose. For more information about the BGP see "Xorp BGP Routing Daemon" [20] and rfc-4271 [3].

Open Shortest Path First (OSPF) module

This is the OSPF routing daemon. There are separate IPv4 and IPv6 daemons, because unlike BGP there is no real need to tie them together. For more information about the OSPF module see the xorp website, the source code and rfc-2328 [4]. No separate xorp document about the OSPF implementation has yet been released from the xorp project.

Routing Information Protocol (RIP) module

This is the RIP routing daemon. Similarly to OSPF, the IPv4 and IPv6 daemons are separate. For more information about RIP see "RFC-2453. RIP Version 2." rfc-2453 [21].

Intermediate System - Intermediate System (IS-IS) module

This is the IS-IS routing daemon. The is-is routing protocol derives from the OSI network specification. The xorp IS-IS project is not yet published.

Multicast routing modules

The overall design of the multicast routing system is described in the xorp design document "Xorp Multicast Routing Design Architecture" [22].

Protocol Independent Multicast – Sparse mode (PIM-SM) module

This is the PIM-SM module. For more information see the "PIM-SM Routing Daemon" document [23].

Internet Group Management Protocol / Multicast Listener Discovery (IGMP/MLD) module

This is the MLD/IGMP handler. It implements the router-side part of MLD and IGMP protocol. Its main purpose is to discover local multicast members and propagate this information to multicast routing daemons such as PIM-SM. The IGMP (IPv4) and MLD (IPv6) daemons are separate implemented modules. For more information about the xorp MLD/IGMP implementation see "Xorp MLD/IGMP Daemon" [24].

Other modules

Forwarding Entity Abstraction (FEA) module

The Forwarding Entity Abstraction (FEA) is a common interface towards the packet forwarding hardware. The FEA controls all network interface cards at the hosting OS (e.g. issuing proper `ifconfig` commands etc.), installs multicast support, and is also responsible for the communication with the Click modular router system if installed.

By definition xorp network interfaces has a two level hierarchy. The real (physical) interfaces are called `interfaces` and virtual interfaces are called `vif`. All IP setup is

done at the `vif` level. If an physical interface like an Ethernet NIC is capable of handling virtual LANs (VLAN) these are defined as a sub interface of the physical interface as a `vif` for each VLAN. If no virtual layer is existing at an real interface, a vif must still be configured to define the IP-interface for this interface card. The names for interfaces are usually the names used by the hosting OS for a given interface. `Vif` names are usually the same with an added decimal number to identify the VLAN.

The multicast-related functionalities are logically separated from the unicast-based functionalities in the Multicast Forwarding Engine Abstraction (MFEA), though the MFEA is part of the FEA process.

The FEA is described in more detail in the xorp document: "Xorp Forwarding Engine Abstraction" [25], and the MFEA is described in the xorp document "Xorp Multicast Forwarding Abstraction" [26].

Routing information Base (RIB) module

The RIB holds a user-space copy of the entire routing/forwarding table, complete with information about where each route originated from (e.g. which protocol, and when). For more information about the RIB see " xorp Routing Information Base (RIB) Process" [27].

Forwarding Engine (FE) subsystem

The forwarding engine is the underlaying IP subsystem that handles the actual IP packet stream that passes through the xorp host. It is usually the default IP subsystem of the xorp host, but it can also be a Click modular router (kernel) module that handles the packets being routed. Theoretically it can also be custom made IP subsystem, e.g. implemented at a multi port Network Interface Card (NIC) which is capable of routing IP packets between its local ports - and this way off-loading the IP subsystem of the xorp host.

3.3 Xorp operation overview

The xorp system has an overall manager module, called The xorp route manager *(rtrmgr)*, its main task is to read the xorp configuration file, interpret it and communicate the contents to the rest of the modules in the xorp router. To implement the setup described in the configuration file, the router manager builds an in-memory configuration tree where each configuration clause is stored in its own node. The configuration tree holds the running configuration of the router. Via the xorp shell *(xorpsh)*, users can interactively change the content and hence the running configuration of the router. As an example if an IP address of an interface is changed in the in-memory configuration tree, the xorp *rtrmgr* communicates this to the FEA which issues a proper command to the underlaying hardware to change the IP address of the interface.

Templates driven *xorpsh* and *rtrmgr*

The *rtrmgr* and the *xorpsh* uses templates that describes functionality for each module included in the xorp project.

Templates control the syntax and parameter types of each node in the in-memory configuration tree, allowed value ranges for each parameter and what actions to be executed when configuration parameters are added, deleted or changed. The template also describes where in the configuration tree parameters for each XRL call to a module are stored.

Templates also defines the structure of the configuration tree for each node and the names and types of the parameters available at each node. Templates contains a header and a number of sections. The configuration tree structure defined is used both for building

the in-memory configuration tree (holds the current configuration) and at the same time it defines the configuration language and what values and actions are valid for each node.

The header defines which other modules the current template depends on. This is used by the *rtrmgr* to startup the needed modules only.

One section of the template file defines the structure of the configuration tree.

Another section contains the information about what actions to emit towards the various xorp modules to implement the functionality expressed in a given in-memory configuration node. Actions can typically be: establishing, changing or deletion of functionality contained in a given node and what XRL calls to emit for each node, to implement the system function the node represents.

Yet another section of the template file contains help texts (short and long versions) for the interactive user using the *xorpsh* CLI interface program. Help texts can be displayed by user commands for each node in the configuration tree.

Xorp Shell *(xorpsh)*

The user interface for xorp is the xorp shell which is implemented in the program called *xorpsh*. The xorp shell connects to the *rtrmgr* process to get access to the in-memory configuration tree which holds the current configuration of the xorp system. The xorp shell uses the templates which describes each module in order to know what actions each node in the configuration tree supports. The xorp shell implements the CLI through which the users controls and configure the xorp system.

When a node or a parameter is crated, edited or deleted the xorp shell notifies the *rtrmgr*, which then executes the changes in the in-memory configuration tree and invokes the relevant actions as described in the template to deploy these to the system.

3.4 Xorp configuration language

The current Xorp configuration language will have to be extended to be able to express virtual router instances. To facilitate the discussion about this We will give a short overview of the existing configuration language from Xorp version 1.6. Below we provide a short description of the xorp configuration language.

The Xorp router has a configuration language, structured as a tree with branches, and leafs. Each branch can contain either an unnamed node or a number of named nodes and a number of leafs. Leafs consists of typed data. e.g. an IP version 4 address (type ipv4) consisting of 4 8-bit numbers. The configuration tree can be edited with the `xsorpsh` utility, or loaded into Xorp from a file at startup or when initiated by a user command. Below we provide a list of the configuration nodes at the top level of the xorp router version 1.6 configuration language.

```
fea          Configuration subtree for the Forwarding Engine
             Abstraction
firewall     Configuration subtree for the firewall
interfaces   Configuration subtree for network interfaces
plumbing     Configuration subtree for plumbing modules
policy       Configuration subtree for routing policies
protocols    Configuration subtree for routing protocols
rtrmgr       Configuration subtree for the Router Manager
```

Figure 3.3: XORP vers. 1.6 configuration language top nodes.

The configuration syntax consists of the following top-level nodes: *interfaces, protocols, fea, plumbing, policy, firewall and rtrmgr* which has the following uses:

interfaces node Describes the configuration of the physical and virtual interfaces that is controlled by the router.

protocols node Configures the routing protocols and all aspects of route exchange with peer routers. Sub elements can be one or more elements from this list: static, rip, ospf4, ospf6, bgp, igmp, mld, pimsm4, pimsm6, fib2mrib, snmp.

fea node: Configures the Forwarding Engine Abstraction (FEA), which controls the unicast packet forwarding setup of the router. The forwarding of packet is configurable in ipv4 and ipv6 address family.

plumbing node: This configuration clause controls the multicast forwarding of packets and plumbing together of packet forwarding functionality. The multicast configuration must be a part of this grouping.

policy node: Policies are logic operations / decisions that should be run on routes. The policy statements defines the logic operations and what routes they are valid for. The policies are used in e.g. protocol statements of a routing protocol to be activated.

firewall node: The firewall support in xorp enables packet inspection of packets sent through the Xorp router. This enables filtering rules based on content of each packet sent through the router.

rtrmgr node: The *rtrmgr* configuration clause is used to configure the xorp rtrmgr to use external helper programs when retrieving and saving configuration files. A few other configuration parameters can be configured here too. Examples are: `config-directory`, `load-tftp-command`, `save-tftp-command`, `load-ftp-command` and `load-ftp-command`.

With this short introduction to the Xorp router architecture and the configuration language, we have the basic understanding of the Xorp system to understand the changes imposed to Xorp when designing the support for virtual router instances.

Chapter 4

Analysis of XORP with virtual router instances

4.1 Introduction

This chapter focuses on analyzing how XORP can be remodeled into supporting virtual routing functions by using the FreeBSD 8-Release virtual network stack environment. For readers unfamiliar with the XORP system architecture we have provided an introduction to this in chapter 3 that vi recommend reading to get a better understanding of the XORP analysis and discussions in this chapter.

We introduce the analysis with a list of identified issues from our reading of the XORP project documentation and the FreeBSD virtual network stack environment documentation available, and from working with, and testing both systems. In the following we will work our way through the list and analyze the issues and drill down into the details topic by topic.

We will throughout the analysis and design phase prefer design solutions where the XORP code and modules fits into the FreeBSD 8-Rel virtual network environment concept with the smallest changes needed in the XORP code. Some times we may not choose the best and most obvious general solution to the problem, however we hope that this initial version of XORP supporting virtual router instances can be some sort of a base for a possible later general version of XORP supporting virtual router instances. We will try to mark the areas where a better general and long lasting solution may be available e.g. with an possible extra cost of rewriting or redesigning parts of a few XORP modules

4.2 Questions for further analysis.

This section lists relevant issues from the meeting of the current XORP single instance architecture and the FreeBSD virtual network environment, which we want to analyze further during the process of creating a system design with support for multiple virtual router instances. The list of open issues and questions needed to be analyzed further before we are able to propose an virtualized XORP architecture model. We have sorted the list so questions regarding the same XORP module or topic are listed next to each other:

- *General xorp design:* What XORP processes will be needed in a virtual router instance, to create a working setup for a virtual routing function? What modules will be relevant to have in the main XORP instance? (Further discussed in section 4.3.5.)

- *General xorp design:* What information besides configuration parameters must be exchanged between the main XORP instance (virtual network environment), con-

taining the *rtrmgr* and the virtual router instances servicing each network realm the router is servicing? (Further discussed in section 4.3.5.)

- *FEA:* Should / must every virtual router instance have its own FEA module talking to the hardware? (Further discussed in section 4.3.1.)
- *FEA:* From what XORP instance is the physical / virtual network interfaces configured by the FEA, and will there be an FEA process in each virtual instance? (Further discussed in section 4.3.1.)
- *FEA (Configuration language):* Where in the configuration language will we configure the XORP physical / virtual interfaces? (Further discussed in section 4.3.1.)
- *FEA:* The file *route_socket.cc*: Opens a AF_ROUTE / PF_ROUTE socket to the kernel routing table. How is this fitting into in the virtualized network environment? (Further discussed in section 4.3.1.)
- *RIB:* Will there be a RIB instance in every virtual router instance or will a central *RIB* process common for all virtual router instances be a better solution? (Further discussed in section 4.3.2.)
- *Rtrmgr:* How is the *rtrmgr* communicating with the virtual XORP router instances? Are commands sent from the *rtrmgr* in the main XORP instance using e.g. `jexec(8)` command calls be sufficient to do the job, or will more connection oriented RPC channels be needed e.g. for transport of XRL command call outs and the asynchronously call-backs with results? (Further discussed in section 4.3.3.)
- *Rtrmgr:* How will the login (user and group) identities be handled between virtual router instances? (Further discussed in section 4.3.3.)
- *Rtrmgr:* To what extent should the main XORP router instance have configuration right and control over the virtualized router instances, the operating system and the physical hardware devices? (Further discussed in section 4.3.3.)
- *Rtrmgr:* How do we ensure the configuration file can be saved when the rtrmgr processes are distributed into several virtual network instances? (Further discussed in section 4.3.3.)
- *Bridging:* How do we implement and configure *epair(4)* interfaces and *bridge(4)* devices? (Further discussed in section 4.3.7.)
- *Bridging:* How is the mapping of virtual router instances to VLANs done? (Further discussed in section 4.3.7.)
- *MPLS:* If MPLS is implemented at a later state, how would a suitable design model look like? (Further discussed in section 5.2.)
- *Inter VRF:* How do we leak addresses between different virtual router instances? A: How do we configure this? B: How does the traffic flow between these? (Further discussed in section 4.3.7.)
- *Inter VRF:* How is route information being exchanged between virtual router instances? (must they be exchanged?), or is standard routing protocol peering the way to do this? (Further discussed in section 4.3.7.)
- *SNMP:* How will SNMP communication to the router happen? (To a central SNMP agent or possible to SNMP agents in each virtual router instance?) (Further discussed in section 4.3.9.)

- *SNMP:* How will the SNMP info and alarms (polls and triggers) be routed internal between virtual router instances? (Further discussed in section 4.3.9.)

This concludes the list of issues we have noted during the initial considerations about this project. In the following sections we will analyze the issues listed and from there conclude which path we will take towards a design solution.

4.3 Analyzing the listed design issues.

Implementing a virtualized version of XORP rises a number of questions listed above that we will analyze further in this section.

During the analysis we will try to get the listed issues sorted out and create a XORP system design that we believe is functional and not too awkward to understand, maintain and operate. As stated earlier we will prefer using as many parts unchanged from the XORP single instance router to simplify the changes needed in this first version of a virtualized XORP router.

We will start analyzing the questions for the part of the XORP system that we believe will have the most effect on the total design and then next try to work through / analyze the questions with (possible) lesser overall design implications. The parts we will analyze first are the *FEA*, the *RIB* and the *rtrmgr*. We start with the *FEA* because it controls the interface to the hardware and the *rib* because it communicates with all the router protocol modules and might need to be changed to hold some virtual-instance (Virtual private network) informations for each route. Furthermore the *rtrmgr* is responsible for the overall internal router management and it will of cause needs to be changed to enable support for virtual router instances.

4.3.1 FEA analysis

In the single instance router the Forwarding Engine Abstraction *(FEA)* controls both levels of XORP interfaces, the physical network interfaces which in XORP is called *Interfaces*, and the logical interfaces which in XORP is called virtual interfaces or *vifs*. In our FreeBSD setup with virtual network instances physical interfaces can both be a real physical interface such as a network interface card (NIC) or a software version of the same inside a virtual network environment, in this case we will use the `epair(4)` device as the software network interface. Additional the *FEA* also maintains the communication to the *FEs* (Forwarding Engines) that the router utilizes. In FreeBSD this happens via a socket to the `route(4)` device.

FEA: From what XORP instance is the physical / virtual XORP interfaces configured?, and will there be a FEA process in each virtual instance?

We search for a solution to how the many combinations of physical interfaces and virtual XORP router instances shares the task of configuring the limited number of hardware a given router have. The picture might also be further complicated by the fact that we are introducing two new devices to be able to handle higher numbers of virtual router instances namely the `epair` and `bridge` devices. The configuration of these new devices are discussed below in section 4.3.7.

After considering this we face either a general rewrite of this part of the FEA or to reuse it as-is, with the added complexity by having the router administrator to only configure each physical device from one place in each router configuration common for both the main and virtual instances.

We are deciding to reuse the existing FEA module as it is, and implicitly this will be a decision that also implies that there will be a FEA module located in the main XORP router

instance and in each virtual XORP router instances that are responsible for the configuration of the dedicated physical interfaces and virtual interfaces *vifs* existing and used (configured) in each XORP router instance.

FEA: Should/must every virtual router instance have its own FEA module talking to the hardware?

Yes, implicitly decided above. In the next question to be analyzed, we discuss the configuration language aspect of the same question to ensure it fits here too.

FEA (Configuration language): Where in the configuration language will we configure the XORP physical / virtual interfaces?

There are two overall design paths that can be taken from here. One which leaves the original configuration syntax and semantic as it is, and another where the physical configuration of interfaces is moved to a new section of the configuration file e.g. a global-setup clause, that holds global configuration section and is only interpreted by the *rtrmgr* in the main XORP instance.

As we don't really get any interesting new functionality out of moving the configuration despite a bit more clean system setup we believe this (for now) will only result in a lot of work without any real new outcome.

Based on this we are deciding to keep the existing semantic with the following comment:

The existing *FEA* can only configure interfaces that are existing (available) inside the virtual router instance (jail). This means that interfaces must be created or/and moved to the relevant jails before the *FEA* starts configuring these. This removes the problem of which router instance should/can configure interfaces.

The FEA opens a AF_ROUTE / PF_ROUTE socket to the kernel. How is this fitting into the virtualized network environment?

The *FEA* process opens a routing socket to the kernel forwarding table[1] that contains the active routes that IP packets are routed by the FreeBSD network stack. The kernel forwarding table stores the routes sent from the *RIB*. When changes are received by the *RIB* from e.g. a routing protocol - these are sent via the `route(4)` device to the FreeBSD kernel forwarding table by the *FEA*.

The virtual network environment `vnet` in FreeBSD 8-Release contains a dedicated instance of the routing infrastructure, where communication from the kernel routing socket is sent to. If all XORP processes that handles routing informations for a specific IP realm stays within the same network environment, then calls to the kernel will be serviced by the same network environment instance of the kernel and it all looks exactly the same as in the non virtualized kernel (and XORP router). Then the XORP modules from the single instance XORP router can then be reused unchanged in each virtual XORP router instance.

If the kernel routing table exists in each FreeBSD virtual network instance, this task is reduced to basic reuse of functionality from the single instance XORP router code. We believe this makes the two previous decisions about having a *FEA* process in the main and in each virtual XORP router instance even more correct.

Summary of the FEA analysis

We have analyzed 4 aspects of the *FEA* servicing virtual XORP router instances. All 4 aspects was solved by reusing the *FEA* module from the single instance XORP router, including its configuration syntax language. Some of the aspects would have been much

[1] In FreeBSD the kernel forwarding table is accessed via the `route(4)` device.

more complicated by not having the *FEA* module in the main and in each virtual router instance.

We also get much clearer about the question of what router instance should configure interfaces as these has to be put in place in the correct router instance (jail) prior to the XORP router instance is started.

4.3.2 RIB analysis

RIB: Will there be an RIB instance in every virtual router instance or will a central RIB process common for all virtual router instances be a better solution?

The *RIB* (Routing Information Base) contains all the routes known to the router by configuration and via the routing protocols, and stores details of information about where the routes originates from and a few other details of each route. The *RIB* selects the best route to each of (all) the networks it knows of and send these to the kernel forwarding table via the *FEA*. Regarding the *RIB* as a component in a virtualized router system, we have found the following details from the XORP documentation. The XORP rib design paper [27] (p2) states:

> "We do not currently support multiple RIBs for other purposes, such as VPN support, but the RIB architecture will permit such extensions."

and in the XORP design overview document [12] (p5) it states:

> "On a router with multiple FEs, the RIB is responsible for splitting up the routing table amongst the FEs and for figuring out how to forward between FEs."

In this context "We" are not a router with multiple FEs (forwarding engines) but a router with (possible) multiple virtual router instances. When 2 virtual instances of a router are connected together by IP links the (virtual) routers operates in the same IP realm and does normal exchange of routing information via routing protocols, just as if they were non virtualized dedicated routers. In all situations the standard RIB operation fully fulfills the purpose of holding route information for a single IP realm exactly as it does in the single instance router. Within the scope of this project we don't really need any additional information regarding external information about routes inside each virtual router instance. This is because all routes in the *RIB* and in the current network environment context belongs to the (same) current IP realm.

The picture might change if we add MPLS to the XORP system, as MPLS labels are local to each IP-MPLS link, and the labels identify which virtual network the packet belongs to. But even this might be easy to solve by a local label table in the MPLS layer. See further MPLS discussions in the MPLS section in chapter 5.

Summary og the RIB analysis.

Based on this we decide that each virtual router instance has its own RIB process. The existing RIB module, from the single instance XORP router can be used as-is without changes in each of the virtual instances of the router.

4.3.3 Router manager (rtrmgr) analysis

The *rtrmgr* is the central router administrator process in the XORP router system. At startup it reads the XORP configuration file and builds an in-memory configuration tree from it. Next the *rtrmgr* starts up and initializes the individual XORP processes to the state specified by the in-memory configuration tree. The Command Line Interface (CLI)

program *xorpsh* communicates with the *rtrmgr* and distributes configuration changes from the user into the in-memory configuration tree. After each *commit* of a set of changes to the configuration-tree, the *rtrmgr* implements the changes into the relevant processes of the XORP router. We will cover the *xorpsh* part of the *rtrmgr* in the next section (section 4.3.4).

In a virtualized version of the XORP router the *rtrmgr* is still going to be the central process but some changes needs to be implemented to be able to manage virtual instances, and understand new configuration elements e.g. the virtual router instances, `bridge(4)` and `epair(4)` devices (both covered later). We are considering the following architectural changes to the *rtrmgr* enabling it to support virtual instances:

- The *rtrmgr* is prepared for controlling a router with virtual instances.
- The configuration language is extended with support for virtualized router instances, each in a separate FreeBSD child `jail` with a dedicated instance of the virtual network environment instance.
- A *rtrmgr* instance is located in each virtual router instance, and should be responsible for managing the local XORP processes inside each virtual router instance. To be able to distinguish between the *rtrmgr* process in virtual router instances and the *rtrmgr* instance in the main XORP process. We will rename the *rtrmgr* process residing in the virtual instances to *rtrmgr-vi*, and keep the *rtrmgr* name for the version living in the main XORP router.
- The *rtrmgr* should create and destroy virtual router instances as needed.
- The *rtrmgr* should be able to control the *rtrmgr-vi* in the virtual router instances via some suitable form of communication channels.
- The user and rights management will have to be identical in each virtual XORP router instances.

Rtrmgr: How is the rtrmgr communicating with the virtual router instances?

Inside each instance of the XORP router a lot of communication is happening between the modules. In this project we are trying to implement several instances of each modules, each instance in a separate virtual XORP router instance.

There are no problems with establishing communication paths from the virtual network environment, where the main XORP router instance is living and into the jails where the virtual XORP router instances are residing, as these are child jail instances of the main XORP router virtual network instance.

The other direction is not as easy as it is forbidden for child jail instances to establish communication to their parent jails. They actually don't know anything about their existence at all (at the jail level). So if XRL callbacks is needed between the child jail instances and the main XORP router jail instance we need to create this possibility.

This problem seems to require a bit more analyzing. We believe this is a problem thats should be treated under the IPC finder analysis which can be found further down this section where we will continue this discussion.

Rtrmgr: To what extent should the main XORP router instance have configuration right and control over the virtualized router instances, the operating system and the physical hardware devices?

The main XORP instance will create each virtual XORP network instances and will be the parent network instance to these. This also means they have all possible administrative

rights over the virtual network instances which includes: changing, destroying, assigning and removing of devices, start up root processes etc.

The main XORP router need to have configuration rights to the physical network instances that is a part of the configuration. This is normally provided by starting the rtrmgr from a root user, either at system startup or manually by the root user. This means that the user or process that starts up XORP must have the administrative rights of the network devices where XORP is started.

In a jailed environment the root user have all administrative rights over the interfaces available in (e.g. assigned to) the virtual network environment, but the interfaces must be present inside the virtual network environment or be able to create these. The root privilege can create interfaces, but it is not able to move these to or from the parent virtual network instance (jail). This means the interfaces has to be created or moved from the main router instances as it is the parent of all virtual router instances.

Rtrmgr: How will the login (user and group) identities be handled between virtual router instances?

The XORP router user authorization, decides which user has the right to perform administration of the router. In the single instance XORP router it has been implemented this way: The Unix group membership of files in the file system, from the standard Unix system is used here. User data is contained in the Unix system `/etc/passwd(5)` file and the list of users with administrative rights is held in the Unix system group file `/etc/group(5)`. The group named *xorp* defined in the `/etc/group(5)` file of the Unix host which the XORP software is installed at, is used to assign administrator rights to users. Any Unix user that is a member of the *xorp* group has granted router administrative rights and can enter the configuration mode of the *xorpsh/rtrmgr* and are able to change the configuration of the XORP router. Any other users on the Unix host only have read only rights to the configuration of the XORP router.

To ensure that all virtual XORP instances will share the same administrative user rights, we need to ensure that the two files `/etc/passwd(5)` and `/etc/group(5)` have identical setups regarding the XORP users and the xorp administrators group.

As all our virtual XORP instances will share the whole file system with the main XORP instance as each virtual network environment `(jail)` will have the same *root* directory as the main XORP virtual network environment. This way we have ensured the user credentials will be the same for all main and virtual instances of the XORP router.

Also the *rtrmgr* communicates via file based methods with the *xorpsh* when validating administrator rights for the user starting instances of the *xorpsh*, by writing a file in the `/tmp` directory, that the *xorpsh* reads. This method is also supported by the chosen solution. If a fixed file name is used for the file in the `/tmp` directory the then we must ensure the these files does not clash with the same file names form other virtual router instances. The solution can be either to use the process-id which is a globally unique numerical value in the host, as a part of the file name or alternatively to mount a dedicated `/tmp` file system for each virtual router instance.

Rtrmgr: How do we ensure the configuration file can be saved when the rtrmgr processes are distributed into several virtual network instances?

In the single instance XORP router there is one configuration file that is read by the *rtrmgr*. When we distribute the *rtrmgr* into more virtual network instances we need a way of maintaining the router configuration in a single configuration file. So how do we distribute the configuration file to the various XORP *rtrmgr* modules that each need a part of the file content, and how do we re-assemble the distributed configuration into one file again.

Here we would benefit by having one central in-memory configuration tree residing in the main XORP instance as it then could be saved to disk by the *rtrmgr* living here. In the

first version of the virtualized XORP router we reuse the original *rtrmgr* as-is inside the virtual XORP instances, and therefore we need to make the parts fit together. By choosing a distributed version of the in-memory configuration tree, where each virtual XORP instance has its own local sub-tree to be responsible of, then we need a method to interface to the main XORP *rtrmgr* instance in a way it knows of from the single instance router. Two methods comes into mind when we consider this:

- File based configuration interface.
- Shell command based interface. E.g. where xorp commands are streamed from the main virtual router instance (parent jail) to the xorpsh in the virtual router instances (child jails). See the XORP user manual section 1.8 [17].

As we (in section 4.3.4 "Analyzing of the *xorpsh* administration utility" below) have decided to start off with a administrator model where the router administrator starts an instance of the *xorpsh* in each virtual network instance where he would like to administer XORP, we can pass this problem at startup by splitting the configuration file for the XORP router (with support for virtual instances) into a number of configuration files, one for each virtual XORP instance that matches the single instance XORP router syntax. By designing the configuration language so that the syntax used for the virtual XORP instances is exactly identical to the standard single XORP instance configuration syntax ensures it will match the syntax of the reused *rtrmgr* module from the single instance XORP router.

The *rtrmgr* in the main XORP instance will have to create a configuration file for each virtual XORP instance, from the main XORP configuration file. The name of the virtual XORP instances will be a part of the configuration filenames for the virtual XORP instances. When the virtual instances starts up, they will each read the file that belongs to their virtual XORP router. This is initiated by the *rtrmgr* in the main XORP instance when it starts the virtual XORP instances with an option `-b <config-file>` that points to the correct configuration file in the shared file system.

The task of saving a configuration file from the running contexts of all (main and virtual) XORP instances is a bit more complicated. The main XORP instance must direct a command to each virtual XORP instances to save the current running configuration to the same file as the router was started from, then afterwards the *rtrmgr* of the main XORP router will have to assemble these into a global configuration file with the correct syntax.

This is of cause a hack and the optimal solution to this problem will be to rewrite parts of the *rtrmgr* to work in a distributed configuration that have a suitable protocol between them to coordinate the in-memory configurations and other work tasks e.g. *xorpsh* command proxy/routing and reading and writing of configuration files to or from disk or via the network.

Summary of the rtrmgr analysis.

During the analysis of the *rtrmgr* we have analyzed various aspects of its operation. We have found solutions and workarounds for the problems we have analyzed, but it seems clear that the overall virtual XORP router would benefit greatly by a rewriting the *rtrmgr* to a distributed model.

First we encountered that the two way communication channel between the main XORP instances and the virtual XORP instances has issues regarding communication from the virtual XORP instances and back to the main XORP instances, and further have a module naming issue that will create problems for the sending/routing of XRL messages between modules with the same *rtrmgr* name that lives in different virtual network instances. These aspects will be analyzed further in the section 4.3.6 "IPC finder analysis" below. We have analyzed the user and administrator group rights between the virtual network instances, which we have argued will not be a problem with the implementation we propose. At last we have found a workaround for the task of the loading and saving of the configuration

in the distributed setup we have proposed. It is clearly a workaround and an rewrite of the *rtrmgr* to become a distributed module will provide the needed extra communication between instances of the *rtrmgr* that is needed to solve this.

4.3.4 Analysis of the *xorpsh* administration utility

In the single instance XORP router the *xorpsh* program connects to the *rtrmgr* process. In the virtualized XORP router there is one main *rtrmgr* process in the main router instance, and a *rtrmgr-vi* process managing each virtual router instance.

We will need a way to select which virtual instance of the router is affected by user commands when we communicate with the router from the *xorpsh*. Normally the *xorpsh* commands are manipulating nodes in the in-memory configuration tree. Each location in the configuration tree affects exactly one instance of the XORP router, which is the one the related command should be sent to.

A central issue is how the *xorpsh* program communicate with the *rtrmgr* and *rtrmgr-vi* processes and still keep some sort of usable administrative interface to the complete router system, and maintain the feeling of working with one router with several virtual instances. There are more ways to accomplish this.

One obvious and general solution is to let the *xorpsh* client program connect to the *rtrmgr* in the main XORP instance, and then let the *rtrmgr* distribute the commands to the *rtrmgr-vi* processes each responsible for a virtual router instances. A virtual XORP router instance is living in a child jail with `vnet` virtual network environment enabled. This would require the specification of a new protocol and introduce a new XRL interface instrumented to handle communication between the *rtrmgr* and the *rtrmgr-vi* modules. This way we would still maintain the look and feel of communicating with a single router, that have several virtual instances that are seamless manipulated by changes committed into the in-memory configuration tree.

Another solution to this might be making the *xorpsh* responsible for contact to the relevant *rtrmgr* or *rtrmgr-vi* processes according to the commands issued by the user.

But a much easier solution to implement would be to let the user connect the *xorpsh* process directly to the relevant *rtrmgr* or *rtrmgr-vi* process. It is by any means not a perfect solution in the long run, but for simplifying the changes needed it is a much easier solution. The *xorpsh* utility does not have a option for specifying which *rtrmgr* (or process) to connect to, so the quick solution would be to install a version of the *xorpsh* utility in each virtual XORP router instance (virtual network environment), that the user can start up with a suitable `jexec(1)` command from the virtual network environment where the main XORP instance is living.

Summary of the xorpsh analysis.

We are choosing the last and easiest solution where the user stats up a instance of the *xorpsh* in each virtual XORP router instance he wants to administrate. This decision is argued by minimizing the amount of changes needed. The cost is a somehow more inefficient user interface and an not so perfect user experience. The user pays the extra cost of having to start a new *xorpsh* instance in each virtual XORP router being administered.

When the *rtrmgr* at a later state is going to be redesigned this will be one of the topics to take into account then. Introducing an interface to handle communication between the *rtrmgr* in the main XORP router instance and its peer *rtrmgr-vi* processes residing in each of the virtual router instances seems to be the ideal solution to go for. This will maintain the look and feel of communicating with one router with several virtual instances, and also take care of the workarounds from sending configuration data to the virtual XORP instances and retrieving these again when the configuration are saved into one file.

4.3.5 General XORP design analysis

The general design of the virtualized XORP router has many aspects. The fact that we have this subject somehow late in the analysis work is that we needed to have a good understanding of other central aspects such as the *FEA* and the *rtrmgr*, as we believe the way we can realize a virtual XORP router with these modules in place, will have a significant influence on the rest of the design choices we will take.

From the list of issues in section 4.2 we have two issues regarding the *general design* category.

General XORP design: What XORP processes will be needed in a virtual image?

To get a natural flow of the reading of the report we will answer this question as a part of the presentation of the proposed architecture in section 4.6.

General XORP design: What information besides configuration parameters must be exchanged between main and virtual XORP router instances?

The dependencies between the main XORP router instance and the virtual XORP router instances are entirely of router administrative type, and should only consist of *XRL* (eXternal Resource Locater) type communication.

All communication between XORP modules is done by *XRLs* and consists of asynchronously communication between XORP modules. Modules can call other XORP modules and send *XRL* commands (e.g. configuration commands or routing updates) from one module to another. The response can either be returned right away or can asynchronously be returned from the called module, depending of the request issued. Asynchronous responses can arrive any time in the future when the data is available. In asynchronous call scenarios, the initial *XRL* call to the module returns immediately only with a small return status that acknowledges the transmission to the other XORP module.

At this point the first communication channel is closed (similar to when a function call has returned) and the called function will have to, use a callback method to initiate a new communication channel (or function call) to send back the response to the exact same object instance that initiated the XRL call. The XRL system has features to provide this inside a normal UNIX environment.

Here we have a problem as a process in a FreeBSD child virtual network instance (*jail*) (e.g. processes in a virtual XORP router instance) are not allowed to call out of the *jail* and back into the parent *jail*. The only allowed direction is from parent *jails* into child *jails*.

This problem can only be solved by using some form of communication that can be opened from the parent *jail* and stays open and wait for the responses to return via this channel. A lot of possible ways can be thought of here. IPC, FIFO or shared memory between 2 *jails*.

We consider searching a solution in the area of the *IPC finder* process as it already is responsible of the task of communicate with the XORP modules. We guess that the finder process can be instrumented to establish channels between the finder processes in each *jail* that is a part of the virtualized XORP router (jails with main and virtual XORP router instances)

We will continue the analysis and search for a solution to this issue in the section “IPC finder analysis” see section 4.3.6 below.

The remaining part of the question about what information that must be exchanged between xorp router instances, highly depends on the version of e.g. the *rtrmgr* we chose. So we will postpone the more precise answer to the section where we present the proposed solution for a XORP router architecture that supports virtual router instances. See more in section 4.6.

Summary of the general XORP design analysis.

All the communication between XORP modules are entirely based on *XRLs*, and we need a way to establish these between router instances and take a closer look into this in during the analysis of the IPC finger in 4.3.6 below.

4.3.6 IPC finder analysis.

The XORP *IPC finder* module is normal integrated with the same executable as the *rtrmgr* which means that it isn't necessary to start up an individual *IPC finder* instance.

A look into the documentation of *IPC finder* module tells us that the *IPC finder* uses module name and version number to identify each XORP module.

In our virtual router design we are working with multiple instances of the same XORP modules (each instance of the same module has identical module name and version number) but only one module with each name will exist within each virtual network instance (jail). We may use an unique extra id tag for each virtual XORP router instance to differentiate instances of the same XORP module from each other. Either the *jail-id* or the configured name for each virtual XORP instance can be used here. If the configured name for a XORP virtual router instance is also configured as the FreeBSD *jail-name* when the *jails* are created then both parameters will have the feature of being unique among all *jails* in a FreeBSD system, and then usable as a routing-id between the existing XORP router instances (main or virtual) in the *IPC finder* module.

From section 3 in the document "XORP Inter-Process Communication Library overview" [14], we know that each XORP module registers to the *IPC finder* at startup. During the registration process all XRL interfaces in a module registers their full list of supported XRL interfaces one at a time, and when the registration has taken place the modules interfaces will be published towards other XORP modules via the finder.

When a XORP process wishes to dispatch an XRL call for the first time it sends the XRL lookup request to the finder that resolves the XRL and replaces the transport and protocol specific parameters found for the XRL being resolved, and returns the XRL to the requesting module. Next the XRL can be sent directly to the requested XORP module. The resolved XRL can be saved in a client cache for later use, so the *IPC finder* doesn't have to be contacted for resolving all similar XRL calls to be sent.

Each XRL target has an associated class name and an instance name. Class names indicate the functionality that the target implements and there may be multiple targets in the IPC system with the same class name. Instance names are unique identifiers for each target in the IPC system. XRLs to be resolved by the finder may address a target by class name or by instance name. The first XRL target class that registers with the *finder* in each class will be the default XRL for that class that future requests for the same class resolves to.

For this project it only seems necessary for the main XORP instance to query names for virtual XORP instances and from the virtual XORP instances the main use will be querying XRL targets in the main XORP instance. Communication between virtual XORP instances will be limited and most relevant for future requirements.

To achieve the desired functionality where multiple instances of XORP modules living in different virtual XORP instances can be identified we need to add the name of the virtual XORP router instance to the module name being queried. In normal operation inside a virtual XORP instance this is not specifically needed but when communicating from the main XORP router instance it is important that communication can be routed to modules in the right XORP instance.

The long term design we aim towards is the following: The *IPC Finder* will be able to identify each XORP module directly by a name where the name of the XORP instance has a part of the module name in its name. This way we establish a "global" resolver function that can resolve all queries for modules registered in any XORP instance.

A possible way of doing this would be to have a local and a global naming scheme. Where the local naming scheme is the one we have from the single instance *XORP router*, and the global scheme can be invoked by adding a global name part which can be the name of the non local XORP router instance.

The local name scheme resolves always to local (inside the same virtual instance) modules, and the global name scheme resolves to XRL targets inside the named (remote) instance. We might consider to add a global *any-instance* or wild-card name if global finder lookup/resolver operation is wanted. This can be done by picking a global instance name as a reserved name or keyword meaning *all* instances. The selected global search keyword can not be used as the name of at XORP router instance as these are the valid routes to existing virtual XORP router instances.

IPC finder: XRL communication transport between virtual XORP instances?

Even though we have provided name resolution for XRL targets globally, we still need to create at transport between the virtual instances.

We still need to establish some form of transport channel for the XRL communication between virtual XORP instances, that we discovered during the analysis of the *rtrmgr*: "Communication from the virtual network instances back to the main XORP instances isn't possible to establish". To resolve this we consider to create a distributed *XRL router* module that has an instance running in each *XORP router* instance that are responsible for maintaining a channel between router instances. The *XRL router* in the main XORP instance connects to the *XRL router* modules in the virtual XORP router instances and uses this connection for the XRL transport both ways. The *XRL router* module knows all other XRL modules in the XORP router and can route XRL messages between these. To have a bit of flexibility some coordination (or communication) between the IPC Finder global XRL target lookups and the *XRL router* could create the routing addresses in the XRL router when needs for communication between XRL modules in different router instances is required e.g. when XRL lookups are performed by the *IPC finder*. This way we doesn't waste resources on creating a any-to-any routing scheme when it is only a small subset of the routes between XORP instances that are actually used .

Summary of the IPC Finder analysis

We have analyzed the IPC finder for ways to enable a global XRL lookup scheme and proposed a solution that supports the current *IPC finder* instance operation as-is and extends the feature set with a global XRL name search that resolves XRL requests to XRL targets residing in other named or unnamed router instances.

We also proposed a solution for the XRL transport problem that we initially believed could be solved by the *IPC finder* alone, but as this module mainly provides a XRL name-to-transport resolver service we could not solve the XRL transport problem within the scope of the *IPC finder* module. This lead to the proposal of an *XRL router* module that resides in all XORP router instances and establish XRL communication between instances of the XORP router.

The *XRL router* module isn't needed for the first version of the virtualized XORP router, as we only have a very limited need for communication between the main and the virtual XORP instances, which can be handled by `jexec(1)` commands from the main router instance. But when a redesign of the *rtrmgr* into a distributed model comes up the *XRL router* will be needed.

4.3.7 Bridging - Inter connecting virtual routers

Bridging: How do we implement and configure `epair(4)` interfaces and `bridge(4)` devices?.

The `epair(4)` and `bridge(4)`[2] are both FreeBSD devices that are created and managed from the `ifconfig(8)` utility. Like any other network entity of FreeBSD the devices can be created inside any existing FreeBSD virtual network environment. The two device types are needed to connect FreeBSD virtual network instances with each other or the outside world, if they aren't connected to the outside world directly via a dedicated NIC. Both these devices are unsupported by XORP version 1.6. So we need to provide some basic XORP support for these.

The `epair` is a set of 2 Ethernet interfaces with a logical cross-over Ethernet cable connecting them back to back. Just like most physical Ethernet Network interface cards they can be configured with IP addresses and moved between *jail* devices with the `ifconfig(8)` command. The two *epair* interfaces in a set are named `epairNa` and `epairNb` where N is a unique numerical value among all virtual network environments.

The *bridge* device is an Ethernet bridge device that creates a logical link between 2 or more Ethernet network interfaces available in the same virtual network environment as the bridge. It connects Ethernet interfaces but is not an Ethernet interface by itself. So to connect an virtual network device to an bridge device a set of *epair* interfaces are needed. The *bridge* device is named `bridgeN`, where N is a numerical value. The *bridge* takes any Ethernet interface available in the virtual network instance where it resides as a *bridge* member.

In our virtual XORP router setup we will not need more than one level of child jail instances, for the virtual XORP router instances, so configuration of *bridges* and *epair* devices will only be needed in the main XORP router instance. This enables us to connect any number of the virtual XORP router instances with either of the device types defined in the main XORP router instance, and *epairs* are used to connect virtual network instances with *bridges* defined in the main XORP instance.

The configuration will be much like the normal XORP Ethernet interfaces. The devices are named with their actual FreeBSD device names in the XORP configuration, and available via the *fea* interface XRL methods and configuration language.

Before we can start using any of these devices they must be created, and a common name must be agreed on in the XORP configuration. We will define the *epair* and *bridge* in a new XORP `global-setup` configuration clause that are created to hold all new configuration statements that is only needed at the main XORP network instance. The `global-setup` configuration clause will be placed as the top-level clause (node) in the configuration tree. The idea behind this is to keep the original single instance XORP configuration language intact, in all the virtual XORP instances. The configuration language and syntax for the *epair* and *bridge* devices is presented in section: 4.5.

Bridging: How is the mapping of virtual router instances to VLANs done?

A virtual router instance need to have an Ethernet interface assigned before a VLAN can be configured. (VLAN is an Ethernet multiplex technology.) Either a physical or a virtual interface e.g. the *epair* interface, can be configured in the virtual router instance. XORP supports configuring VLANs directly on its *vif* devices (virtual interfaces). When configuring a VLAN on an Ethernet interface the interface adds VLAN-tags to each Ethernet packet being sent out. A small VLAN configuration example is provided in figure 4.1 below.

[2]The `if_bridge(4)` is the name of the bridge kernel module in FreeBSD, the interfaces that this module provides area named *bridge(4)* in FreeBSD. We prefer to use the *bridge(4)* name in this report as most of our use is related to the provided bridge interface.

```
interfaces {
  interface dc0 {
    description: "Ethernet interface with a VLAN"
    vif dc0 {
      address 10.10.10.10 {
        prefix-length: 24
      }
    }
    vif vlan1 {
      vlan {
        vlan-id: 18
      }
      address 10.10.20.20 {
        prefix-length: 24
      }
    }
  }
}
```

Figure 4.1: The example shows a XORP configuration of an Ethernet network card which shows up as device: dc0 in the FreeBSD device list. In XORP the device name is available as an *interface* in the configuration. The *vif* with the same name can be configures with IP-addresses and vlan-id. In this example we configure the `dc0` interface in our XORP instance to send and receive Ethernet packets tagged with the VLAN-id: 18. All other VLAN tags received are ignored by this interface.The example is from the "XORP User Manual" [17], section 3.2.2.

Summary of the bridging analysis.

We have analyzed the bridging and interconnect requirements and can fulfull the possible setups by providing support for *epair* and *bridge* devices in the main XORP instance. We have shown how VLAN can be configured with existing XORP functionality. We have created a separate top-level configuration clause (node) to hold the configuration language constructs of *epairs* and *bridge* devices. The configuration language constructs are presented in section: 4.5.

4.3.8 Analysis of inter VRF related issues.

Due to the limited project time we have been forced to skip this part of the analysis to a future occasion. The inter VRF routing questions are about how we can insert routes in a virtual router instances that point to IP addresses in other virtual router instances, and the solution is having both a packet transport side as well as a route protocol/RIB aspect.

We have been looking forward to solve this issue but it will have to wait till some other time.

4.3.9 SNMP functionality analysis

SNMP: How will SNMP communication to the router happen? (To a central SNMP agent or possible to SNMP agents in each virtual router instance?)

For the first version of the virtual XORP router we will need to talk SNMP with the SNMP instance in the main and in each virtual XORP router instance. This is of-cause a hack and not a long term solution, but as SNMP is for most cases a management system interface mostly used in larger production setups, we doesn't believe this will make a big difference in our first version of XORP with virtual support. Also we don't believe that this version will be used for production before it has undergone the next phase of clean up various aspects of this initial design.

SNMP: How will the SNMP (info and alarms (polls and triggers)) be routed internal between virtual router instances?

There will not be any internal SNMP routing between virtual router instances, in the first version of the virtualized XORP router. The SNMP agent in XORP is by design a SNMP-to-XRL gateway, so all internal routing of SNMP will already be performed by using the XRL protocol. SNMP triggers from the SNMP module in the virtual XORP instances sent to SNMP module in the main XORP instance, requires the functionality from the *XRL router* module we specified above, in the *IPC finder* analysis section. So until this is in place SNMP will only work isolated inside each XORP instance.

Summary of the SNMP analysis.

To summarize the above analysis, SNMP from the single instance XORP version will be available in each of the XORP router instances, main and virtual, and the SNMP management station will have to get access into each and every of the virtual networks being serviced by the router, to access the router. Any centralized solution SNMP will depend on the *XRL router* functionality.

4.4 Analysis of XORP virtual router configuration language

When changing the XORP router system from a single instance router to a multi instance router some changes in the configuration language will be needed to express the configuration of the new functionality available.

We will try to reuse as many parts of the existing configuration language and structure as possible, which in some ways may lead to a less perfect general design than if a more general approach of redesigning the XORP configuration language was taken. Again we hope that the resulting solution can be a step stone to a more genuine router configuration language with support for virtual router instances.

To change the current configuration language to support virtual router instances, we need to be able to express the XORP virtual router instances in the configuration syntax.

As we are planning with reusing the current single instance XORP router functionality as-is, inside each XORP virtual router instance. We initially propose a container format that have a full copy of the existing configuration language configuration clause for each virtual router instance. This virtual router container can be multiplied as as many times as needed to configure each virtual router instance. Each XORP router instance have a unique name defined in the configuration.

Some parts of the existing configuration language might give less meaning to have inside the virtual container, but if we as a start leaves it as-is then a possible later clean up of the language subset for the virtual router instance can take this into account.

Below we list issues that we have discovered during the work with XORP and FreeBSD that is related to the configuration language.

- How do we configure/represent virtual instances?
- Where in the configuration tree is Network interface cards configured?
- Where is the `bridge(4)` and `epair(4)` configured in the language? These device types are unsupported by XORP today, and must be used to connect each XORP router instance to other router XORP instances or network devices.
- A name for each virtual router instance must be provided. Either we can use the *jail-id* or *jail name*, which is an obvious way of acquiring a unique id for each virtual

instance. This can be either manually configured or automatically assigned (e.g. by using the *jid* created by the jail create command.) Here we will use the name in the `virtual-router` configuration clause.

- The language changes need to express the parts of the XORP router that resides in each `jail`

The questions has all been answered during the analysis done previously. So we will fast forward to presenting the proposed XORP configuration language for virtual router instances.

4.5 The proposed XORP virtual router configuration language

Extending the XORP router with virtual routing instances impacts the configuration language of the router. As a first hand solution we will try to extend the existing configuration language syntax in the following way: We create a new configuration clause for the top-level language syntax and use an unchanged single instance router configuration syntax in each configured virtual instance.

The XORP single instance configuration syntax consists of a number of named configuration blocks, at the global level. To be able to use the proposed format for an extended configuration syntax we need to create a super level instance to contain each XORP single instances and provide a name to each of these.

In figure 4.2 is a list of existing configuration elements at the top level of the single instance router (left), and a proposed top-level syntax design for a multiple instance XORP router configuration language syntax (right).

```
-- Single instance configuration --  -- Multi instance configuration --
   interfaces { ... }                    global-setup { ... }
   protocols { ... }                     main-router <name> { <sic> }
   fea { ... }                           virtual-router <name> { <sic> }
   plumbing { ... }                      virtual-router <name> { <sic> }
   policy { ... }                        virtual-router <name> { <sic> }
   firewall { ... }                      virtual-router <name> { <sic> }
   rtrmgr { ... }
               sic means single instance configuration.
```

Figure 4.2: Overview of XORP single instance configuration language (left part) and the proposed new configuration language (right part). A new top-level configuration clause `global-setup` has been added to hold the new features added to the main XORP instance, during this project. Each XORP instance is configured in a named `virtual-router` configuration clause holds the full syntax of the single instance XORP router configuration language. The `main-router` clause configures the main xorp router instance.

The top level configuration elements of the XORP single instance router are moved from the flat top-level scope into the `virtual-router` and `main-router` clause, where the configuration of each router instance reside. The configuration of the main XORP router instance is contained in the `main-router` clause, and the `virtual-router` clause contains the configuration of each of the virtual routers defined. The `name` parameter contains the name of the XORP router instances and is also used as the name for the virtual network instances *jail* of FreeBSD where the virtual router instances are residing. The interface configuration of each virtual instance is allowed to specify both physical and virtual interfaces. A new top-level block the `global-setup` clause is designed to hold the

configuration statements of the new functionality added to the virtual router. We have defined a configuration language syntax for the `bridge(4)` device and for the `epair(4)` interface.

```
xorp-configuration          ::= <global-setup-statement>
                              | <main-router-statement>
                              | [<virtual-router-statement>] ...

main-router-statement       ::= main-router  <main-r-name> { <xorp-sic> }

virtual-router-statement    ::= virtual-router <virtual-r-name> { <xorp-sic> }

main-r-name                 ::= <text string>

virtual-r-name              ::= <text string>

xorp-sic                    ::= <A xorp single instance configuration statement>

global-setup-statement      ::= global-setup { <global-statement>
                                             [ <global-statement> ] ... }

global-statement            ::= <epair-statement>
                              | <bridge-statement>
                              | <route-import-statement>

epair-statement             ::= epair <epair-id> {
                                  node-a-in: <main-r-name>|<virtual-r-name>
                                  node-b-in: <main-r-name>|<virtual-r-name> }

epair-id                    ::= <numerical value>

bridge-statement            ::= <bridge> <bridge-id> { <bridge-cmd-list> }

bridge-id                   ::= <numerical value>

bridge-cmd-list             ::= <bridge-cmd> [<bridge-cmd> ... ]

bridge-cmd                  ::= bridge-cmd<num. value 0 .. 15>: <text string>

# The route-import-statement is not fully developed but is for now
# placed here.

route-import-statement      ::= <specification not done yet>.
```

Figure 4.3: The XORP multi instance configuration language. The language extend the existing XORP language with `main-router` and `virtual-router` instances and a `global-setup` part that holds the new additions to the language.

In figure 4.4 we provide a small example of a virtual XORP configuration with 2 virtual instances named `router-1` and `router-2` and a main XORP instance named `main` all 3 router instances are connected with a *bridge* device. The example is created to show the configuration language for the language parts we have created. A full XORP router configuration example would easily be several pages long and only show existing XORP language constructs besides what we show in this example.

Summary of the language configuration

We have analyzed and created a configuration language that is able to configure virtual router instances and the main XORP router instance with the exact same syntax as the single instance XORP router. The new added functionality is kept inside a new top-level configuration clause named `global-setup` which contains the configuration of the two added devices, the *epair* and *bridge* devices and additionally we have reserved a language construct for the configuration of inter VRF routes that is routes between virtual router instances. Due to limited time for this project we did not crate this part of the language.

The *bridge* device has an theoretically unlimited number of ports we have provided

```
global-setup {
  epair 0 {
    node-a-in: main
    node-b-in: main
  }

  epair 1 {
    node-a-in: router-1
    node-b-in: main
  }

  epair 2 {
    node-a-in: router-2
    node-b-in: main
  }

  bridge 0 {
    bridge-cmd1: addm epair0b
    bridge-cmd2: addm epair1b addm epair2b
    bridge-cmd3: addm bge0
  }

  main-router main {
    <Instance main: XORP single instance configuration with interface epair0a>
  }

  virtual-router router-1 {
    <Instance router-1: XORP single instance configuration with interface epair1a >
  }

  virtual-router router-2 {
    <Instance router-2: XORP single instance configuration with interface epair2a >
  }
```

Figure 4.4: Example of the XORP virtual instance router configuration language. The example shows 3 router instances named: main, router-1 and router-2, that is connected to a bridge in the main instance by 3 sets of epair devices, *epair0*, *epair1* and *epair2*, each having an “a” and a “b” interface. The “a” interfaces are connected to each router instance and the “b” interfaces are connected to the bridge device. Notice the bridge-cmd2 line containing 2 bridge commands, which is introduced to be able to create bridges with a high number of ports.

15 lines of `bridge-cmdN:` in the specification. Each of these can hold several bridge commands, only limited of the capability of the `ifconfig(8)` program as the lines are fed directly to the argument line like this: `ifconfig bridgeN <bridge-cmdN>`, each line at a time.

We are confident that the language is general enough to be able to express the router instances that are needed in network setups as we have no restrictions in the language except the one mentioned in the bridge-statement mentioned.

4.6 The proposed XORP virtual router architecture

After having analyzed the XORP architecture and the properties of virtual images we have actually two versions that we propose: A "Quick" version with a high degree of reuse which can be used as a first test version and a "General" design version that have a more genuine infrastructure in place for future developments, mainly featuring communication facilities between different XORP router instances and a better integrated management interface where the router administrator connects to the main router instance and from there controls all instances of the router.

The virtual XORP router implementation utilizes the FreeBSD virtual network environment with a virtual network instance (jail) for each virtual instance of the XORP router. Both proposed solutions of the XORP virtual router architecture uses jails with the `vnet` and the `persist` property set and a separate routing table associated.

The initial "Quick" version of the XORP virtual router architecture we propose are depicted in figure 4.5 and explained in section 4.6.1 and the "General" design is depicted in fig. 4.6 and explained in section 4.6.3 below.

4.6.1 The proposed "Quick" XORP virtualized router architecture.

The "Quick" XORP virtual router architecture consists of a main XORP router instance that is residing in the virtual network environment where the XORP router starts up. Whether this is the default FreeBSD kernel instance or a virtual FreeBSD kernel instance (*jail*) is not important as long as the devices needed (configured) by the XORP router is available in the network environment where XORP is started.

During our analysis have created an solution with a high degree of module reuse from the single instance XORP router. We have have introduced a new module named *xorp-boot* that is responsible for the initial setup of the XORP virtual environment and start up of the virtualized XORP router instances. We describe the *xorp-boot* boot process further in section 4.6.2 below.

The new *bridge* and *epair* devices is controlled by the *xorp-boot* module. The *xorp-boot* module initializes the *bridge* and the *epair* instances in the main XORP router instance and relocate *epair* interfaces to the relevant virtual XORP router network environments, according to the configuration file.

After the *xorp-boot* module has initialized the virtual XORP router environment, the XORP virtual router instances are started in each prepared virtual netowork instance. In each virtual XORP router instance, we reuse the complete XORP single user module setup, so the *rtrmgr* in each XORP instance starts up as usual, builds the in-memory configuration tree that controls the router, and so on.

4.6.2 Booting of XORP with virtual instances with xorp-boot.

The *xorp-boot* module is responsible for starting up our initial version of XORP with virtual instances. It is necessary as we have reused the *rtrmgr* module from the single instance XORP router as-is for the normal XORP router operation in the main and virtual XORP

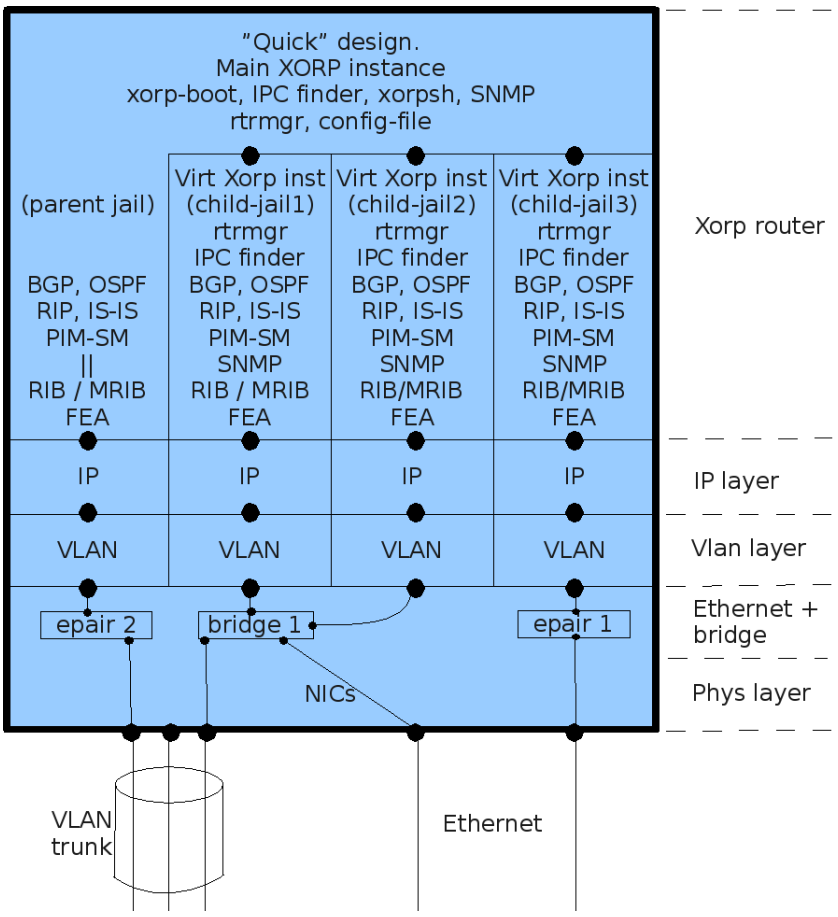

Figure 4.5: The proposed "Quick" XORP virtualized router architecture. The XORP parent instance creates and controls the virtual instances according to the configuration file. Each virtual router instance resides in a dedicated FreeBSD jail and have a dedicated instance of the IP stack. The `bridge` and `epair` devices resides in the main router instance (the parent jail). `Epair` interfaces and `bridges` are used to connect the virtual interfaces in each jail to the NICs. The `bridge` device depicted is connected to the virtual router instances with *epair* interfaces (not depicted).

router instances. We believe that the *xorp-boot* module can be implemented as a shell or Perl script for the initial version.

The *xorp-boot* module are responsible for the following tasks:

- The *xorp-boot* module reads the global XORP configuration file and identifies names of main and virtual router instances.
- The *xorp-boot* module searches the configuration file and identifies what interfaces and network devices are used by each virtual router instance configured. This information is used to prepare the virtual network instances with the correct configuration of devices before starting the XORP router.
- The *xorp-boot* module splits the global configuration file into a file for each XORP router instances into files with filenames identified by having a part of the router instance-names included, e.g. the configuration for the virtual-router named *router-1* is named `xorp-virtual-router-1` and is located in the same directory as the global configuration file.
- The *xorp-boot* module creates a virtual network environment *jail* instance for each virtual XORP router instances identified. The main XORP router instance will reside in the current network environment.
- The *xorp-boot* module creates *bridge* and *epari* devices as specified in the `global-setup` configuration clause and relocates the *epair* interfaces and physical network interface card interfaces NICs into the virtual network instances where they are specified by the configuration file.
- The *xorp-boot* module starts up an instance of the XORP single instance *rtrmgr* in the main and in each virtual XORP router instance with an argument of the relevant configuration file that configures each router instance.
- The *xorp-boot* module merges the individual configuration files back into the global configuration file on a given signal from the administrator.
- When the XORP router system is going to be shut down, the *xorp-boot* module stops the XORP processes, moves interfaces back to the main XORP instance, removes virtual network environment to shutdown the virtual XORP instances and restore the host to the state it had when before staring XORP.

We believe that the *xorp-boot* module will provide the glue that creates the first version of the XORP router with support for virtual router instances. The administration of the virtual XORP router and the virtual instances suffer greatly from this by forcing the user to administrate each router instance individually from a *xorp shell* in each instance, but this will be established with the redesign of the *rtrmgr* as a distributed module and the *XRL router* described earlier.

4.6.3 The proposed "General" virtual XORP routing architecture

The "General" design depicted in figure 4.6 differs with having a changed *rtrmgr* and *IPC finder* module, and a new *XRL router* module introduced. The *rtrmgr* is responsible for boot of the virtualized XORP router instances and the *xorp-boot* module is not used in this version.

The functional aspects in the "General" architecture differs in the following areas: The *XRL router* module establishes communication channels between the main and the virtual XORP router instances enabling bi-directional XRL communication between all virtual router instances and the main router instances. The XRL routing functionality enables the

communication between all XORP instances via the *XRL router* node in the main XORP instance.

The enhanced *IPC finder* functionality enables XORP modules to resolve XRL targets in remote router instances by applying either the name of the instance or a *wild-card* name for searching registered XRL targets in either a specific virtual XORP instance or globally in all XORP router instances.

Finally an extension to the *rtrmgr* module that enables it to be a distributed module, with instances in all XORP instances will open up for having a global in-memory configuration tree experience. Whether it will be implemented as a true global in-memory configuration tree living entirely in the main XORP instance or it will be implemented as a truly distributed in-memory configuration tree where each subtree lives in the distributed *rtrmgr* instance in each router instance. We have not analyzed this part thoroughly enough to give a final answer of what solution would be the most beneficial for the design, but we tend to prefer the distributed version of the in-memory configuration tree to the non distributed solution, but MPLS or portability reasons may change the picture when analyzed thoroughly.

4.6.4 Summary of the design proposals

We have presented 2 slightly proposals for the virtual XORP router architecture, the “Quick” design and the “General” design. They differ in the way that the “Quick” design reuse all parts from the single instance router and we introduce a new *xorp-boot* module with the responsibility of initializing the virtual router setup according to the configuration file. and the “General” design proposal where a few central modules have been redesigned with some added functionality which provides a general solution with a functioning router with a well structured user interface and integrated infrastructure for the internal support and administration of the virtual router instances.

We believe that the “Quick” design may be used to as an intermediate step for evaluation of the design requirements for a like the “General” design.

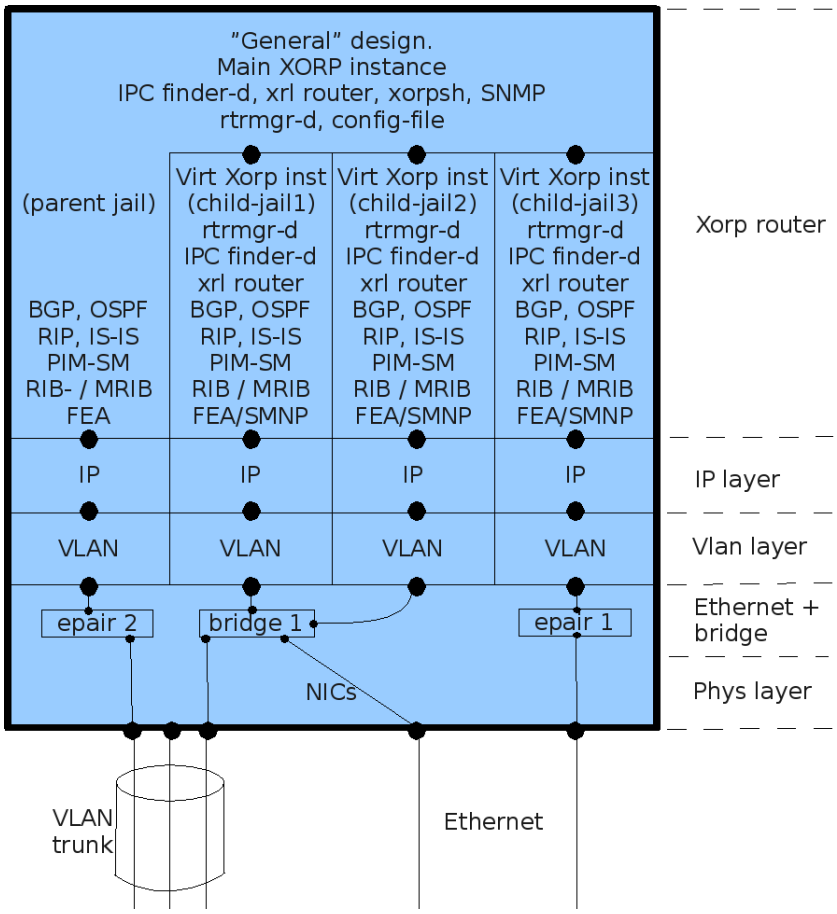

Figure 4.6: The "General" XORP virtualized router architecture. The *rtrmgr* in the main XORP instance create and controls the virtual instances according to the configuration. Each virtual router instance resides in a dedicated FreeBSD jail with a dedicated instance of the virtual network environment. Epair connectors and bridges are used to connect the virtual interfaces in each jail to the NICs. The bridge device depicted is connected to the virtual router instances with *epair* interfaces (not depicted.)

Chapter 5

MPLS in the XORP architecture

5.1 Introduction.

Someday MPLS (Multi Protocol Label Switching) will be considered being a part of the XORP feature set. The way we provide virtual router instance support in this project are not close to the traditionally solutions provided in routers, which is a reason to take a short glimpse of how MPLS does fit into the architecture of this project.

MPLS is a set of technologies that provide a set of tools to manipulate IP traffic, one of the more used applications of MPLS is to create MPLS based virtual private networks. Se introduction to VPN in section 1.2 and to MPLS in section 1.3.1.

MPLS operates with insertion of labels in front of each IP packet, that is used for routing of the packet, instead of the destination IP address. This makes MPLS operate just below the IP layer of the network protocol. Some people calls MPLS for a layer 2,5 protocol, as it operates between layer 2 and layer 3 in the ISO protocol specification.

When the MPLS labeled packet reaches its destination router the label is "popped" of and normal IP routing is used for the last part of the way to the destination host. Usually the MPLS part is only inside one administrative network domain, e.g. an IP network provider or inside a larger company network.

5.2 MPLS analysis

MPLS enabled network is typically a network which is managed inside an administrative network domain. MPLS labels has only link-local significance and labels are typically switched at each MPLS switching/forwarding point, often with hardware support from the forwarding engine (FE) part of the router.

When MPLS is going into the XORP design, we will have to be able to consider the layering of MPLS and VLAN function blocks. (see figure 5.1). XORP and FreeBSD today only knows about VLANs and we believe[1] that an MPLS link can be transmitted over a VLAN link, so we need to have this option in the design. If VLAN should be transmitted over an MPLS link it will have to be implemented with help of an encapsulation protocol.

We see two main categories of MPLS packets entering our router via links connected to it. MPLS packets that have to be relabeled and forwarded into another line, and MPLS packets that has to be delivered to an interface or non-MPLS network connected to the router.

Packets that have to be relabeled and forwarded in the MPLS domain, are re-labeled with a new label from the label distribution protocol (LDP) system, no IP routing tables are

[1] This is based on the fact that MPLS is layered above the VLAN layer in the ISO protocol. We haven't investigated this further in this project.

required here. Packets that have to exit (or enter) the MPLS domain has to be delivered to an Interface at the router. This interface has to reside in the IP realm that the IP address plan belongs to - e.g. the IP interface needs to be connected to a virtual network stack configured to hold RIB and the (kernel) IP routing table for the relevant IP realm.

Packets that are re-labeled and forwarded can be processed in a MPLS kernel module. LDP support will likely be implemented as a xorp module. The general use case will have one MPLS forwarding instance, but as depicted in 5.1 this possibility is a feature of all virtual router instances.

With the above in mind the XORP architecture design will be quite different from the ”xorp-without-MPLS” design, as there is not a direct relation between virtual kernel images and IP-network (IP realms) configuration in the MPLS world. Despite we need to have MPLS inside some router instance, and MPLS is situated between the IP-layer and the Ethernet VLAN layer it will need to be implemented by a kernel module in FreeBSD, that can provide the forwarding functionality (labeling and relabeling of MPLS packets that enters and leaves the router), and user land processes that provide the management support of the MPLS labels.

We have provided this figure where MPLS is inserted as a layer between the IP and the VLAN layer, and is existing in each virtual router instance. The MPLS layer depicted is the one responsible for the labeling and relabeling of MPLS packets, the MPLS administrative modules are not depicted on the figure. It resides as a user land processes in the main or in the virtual XORP router instances.

5.3 Summary of the MPLS analysis

We have provided a short analysis of MPLS in the XORP architecture. It shows the architectural location of MPLS in the router model and added a few comments to the solution. It have to be further analyzed how the implementation in the FreeBSD kernel would be most beneficial and effective done.

MPLS is an important functionality to have in routers which are a part of larger modern IP networks, and if XORP is going to have its place as a research platform for routers and routing protocols it will be an important issue for XORP to be able to communicate with MPLS network infrastructure.

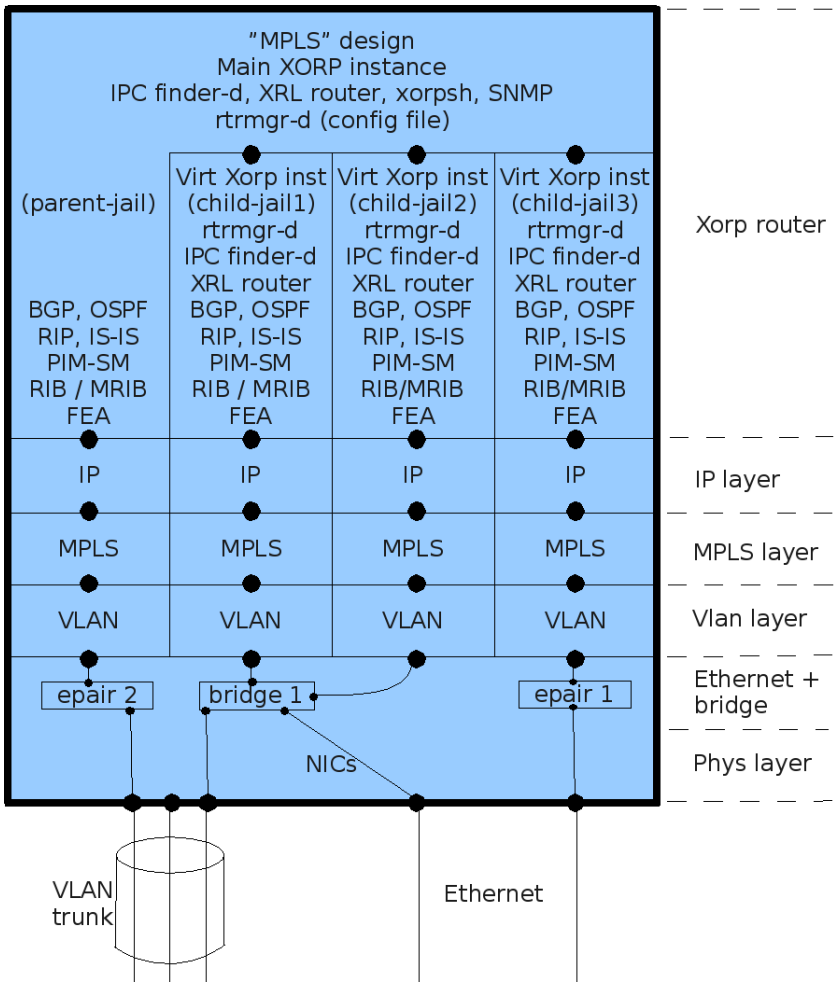

Figure 5.1: MPLS in the XORP Virtualized architecture. The MPLS layer between the IP and the VLAN layer is responsible for labeling and relabeling of MPLS packets passing the router. MPLS packets in transit only need relabeling before sent off to the next link. MPLS packets entering or leaving the MPLS domain needs a IP routing instance that is provided by a virtual XORP router instance in the IP realm they belong to. The `epair` and `bridge` devices are residing in the main XORP router instance. The bridge device depicted is connected to the virtual router instances with *epair* interfaces (not depicted).

Chapter 6

Conformability between the FreeBSD IP stack and the suggested XORP architecture

After having worked with the FreeBSD virtual network stack (VNET) and XORP during the project we will now try to evaluate the conformability of the two. Based on this is an entirely theoretical project so we can not use quantitative facts as lines of C++ code or number of C++ functions required in the implementation. We will instead try to list issues we have found and worked around during the project and give an evaluation based on this.

6.1 FreeBSD related issues.

The FreeBSD virtual network environment is an almost complete virtualization of the FreeBSD kernel making the FreeBSD kernel able to invoke virtual kernel instances much like instantiating a C++ Class as an object instance.

The only thing vi did not find direct in the FreeBSD virtual network environment system is the ability to create communication from a child virtual network instance and back to the parent network instance. This is a feature created by design, and not a flaw of the system. This "missing" functionality is the only reason for implementing the *XRL router* module. That is responsible for creating this missing feature.

6.2 XORP system related issues.

On the XORP side we of cause had to create solutions for supporting the Virtual XORP router instances which XORP did not have before. We will list the XORP issues found here:

- Xorp needed infrastructure to communicate between virtual router instances where we, at the same module, implemented the solution to work around of the restriction of not being able to create communication from a child network instance and back to the parent network instance.
- Xorp also needed infrastructure to distribute the global router configuration to the child / virtual router instances, which we solved in the "Quick" version of our design by creating individual files in the shared file system of the XORP router instances, and in the "General" design solution we solved this by redesigning the central XORP administrative process the *rtrmgr* into a distributed process with an peer module in

each XORP router instance, responsible of control and communication with local XORP modules in each router instance and for communicating with the central *rtrmgr* instance in the main XORP router instance.

- XORP needed support for two FreeBSD network devices the *bridge* and the *epair* to be able to connect virtual router instances with each other and the outside world.
- The XORP "Quick" design a new module in the main XORP router instance to administer the startup and shutdown of the XORP virtual router instances according to the configuration file and a few other router infrastructure administrative tasks. In the "General" design this functionality has been build into the distributed version of the *rtrmgr* module

All the XORP issues listed except the `epair` and `bridge` devices is derived from the new virtual functionality which we have brought into the XORP router. This is hard to blame. FreeBSD for.

6.3 Summary of conformability.

To summarize the conformability study we have had way fewer obstacles than expected prior to the start of the project. We had worked our way through the FreeBSD network environment and the XORP system architecture and been able to create a "quick" solution with only a handful of changes to the original XORP, enabling a fast prototyping track, that can provide a better understanding for the "general" solution that required a redesign of a few modules. All the issues that have to be redesigned here is connected with enabling of communication between the various instances of the XORP router.

The high modularity and general design of the XORP code especially with respect to the XORP Inter Process Communication (IPC) feature based on generic eXternal Resource Locators (XRL) resulted in a analysis that only revealed few issues.

Finally based on the argumentation above we can state that the conformability between the FreeBSD virtual network environment and the XORP router system is somewhere between good and perfect on a line that goes from poor to perfect. We know this isn't an exact metric, but with the samples we have stated above we believe we had provided a good impression of the overall really good conformance between the two systems.

Chapter 7

Conclusion

During this project we have analyzed the features of the FreeBSD virtual network stack and crafted two versions of architectural designs for virtualizing the XORP router. From this process we have achieved a good understanding of the requirements that the XORP router with support for virtualized router instances has to the network stack and to the host operating system.

We have proposed an enhanced version of the XORP router configuration language supporting the configuration of virtual router instances including the *bridge* and *epair* network devices. The full configuration language from the single instance XORP router is reused in each instance of the virtual XORP router, and new functionality is located in a new `global-setup` configuration language construct. In the "Quick" design we have proposed a interpretor for this, which is located in the `xorp-boot` module. The `xorp-boot` module is also responsible for the initialization of the XORP router and virtual instances during startup and for decommissioning the virtual instances and network devices and re-establishing the host to its initial state when the router is shut down.

We have designed two versions of the architecture a "Quick" and a "General" design solution. Both implementing the new configuration language. The two solutions differs in two places. The user interface experience and the internal communication between the main and the virtual XORP router instances.

The "Quick" version of the design reuses the complete set of modules from the single instance XORP router in each virtual router instance. The global configuration file is split into several files, one for each XORP instance being started, each containing the configuration for an instance of the router, in the single instance XORP configuration language syntax. Communication from the *xorp-boot* module in the main xorp instance to the other xorp instances is done via the configuration file and by executing `jexec(1)` commands from the main XORP router instance into the *rtrmgr* or *xorpsh* modules in the virtual (child) router instances.

The "General" version of the design also reuses a large number of the single instance XORP modules unchanged, except the *rtrmgr* and *IPC finder* which has added extra functionality and a extra new module named *XRL router* that has been introduced. These changes enables the bi-directional XRL communication between all router instances and the *IPC finder* to resolve XRL targets in foreign router instances. These features enables the bi-directional user communication to all XORP virtual router instances from a single instance of the xorp shell *xorpsh* and the exchange of configuration data between modules via IPC channels, instead of using the simple file based method we use in the "Quick" version of the design. Furthermore it establishes infrastructure to the centrally located SNMP management agent and future implementations of features that needs communication between the virtual router instances. e.g. import/export of routers between IP realms.

Al together we believe we have presented a strong "General" design proposal, that utilizes the features of both XORP and the FreeBSD virtual networking environment in a

efficient way, and a proposal of a usable "Quick" solution to create a working setup for evaluating the details of specific design details before implementing the next step of the evolution in XORP with support for virtual router instances.

After we proposed the design solutions we have analyzed the conformability between the FreeBSD virtual network environment and the XORP router design with support for virtual router instances and fund that these two entities have a degree of conformability with each other between good and perfect on a line from poor to perfect.

We have provided a short analysis and design proposal for how MPLS would fit into our proposed architecture for the parts of MPLS closest to the forwarding engine, which we find are the most interesting question in respect to this project.

As an introduction to the project we have written a short introduction to the FreeBSD virtual network environment in chapter 2 explaining the basic functionality of the XORP router, and written a more detailed section of selected commands related to the FreeBSD virtual network environment which is available in appendix B. We have also provided an introduction to the XORP project in chapter 3 which is an updated and a cut-down quick summary of a similar section from our master thesis from marts 2008 [5].

It is our hope that this work will be used as an input for the discussion of virtualizing the XORP router when this comes up in the XORP project.

Kristen Nielsen
Valby, Denmark.
30. August 2010.

Appendix A

The OSI model

The International Standards Organisation (ISO) have in the 1980s defined the OSI (Open Systems Interconnect) 7-layer model, that describes and categorizes network protocols.

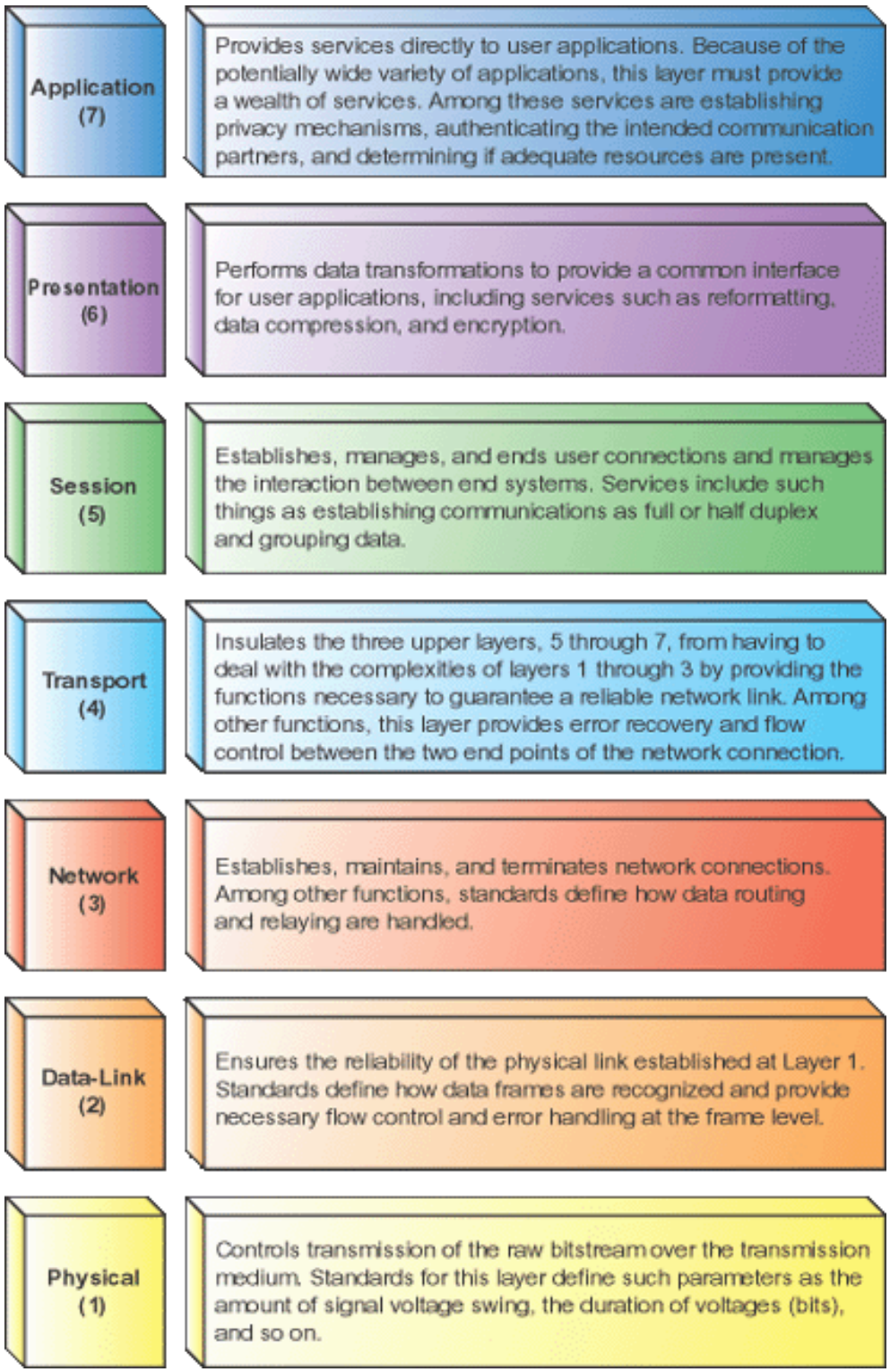

Figure A.1: Global and Local NAT definitions.

Appendix B

Selected FreeBSD utilities

B.1 The `jail(8)` utility

The (VNET) features has been integrated in the `jail(8)` utility. This enables the *jail* utility to handle the new features at the same time as a *jail* process environment is created.

The `jail(8)` command is used for the creation `-c`, modification `-m` and removing `-r` of jails.

Recently functionality for managing vnet instances has been added. The `jail(8)` command identifies jails by `jid` (jail-id.) which all jails are identified by. The jid is automatically set and can be specified as a parameter. Jails can be given an name, that acts like an alias for the jid. If no name is specified for a jail the jid is assumed.

A number of system variables can be specified at the `jail(8)` command line or can be allowed to be set from inside the jail itself using `allow.*` MIB options at the `jail` command line. The list includes system settings like `hostname`, `domainname`, `hostid` and `hostuuid`. Jails can be hierarchical allowing jails to be created inside jails. This feature can be controlled by the number of child jails allowed.

Jails can be restricted to a specified range op IP addresses by the `ip4.addr` and `ip4.addr` parameters, or can be set up so it can bind to any IPv4 or IPv6 family IP addresses available (configured) on the system. The first address of each family will be the default binding address for processes not specifying any other address.

More information of the `jail(8)` utility can be found in the FreeBSD manpage [28].

B.2 The `ifconfig(8)` utility

The `ifconfig(8)` utility handles network interfaces. Ifconfig have an extensive list of options and uses, we will just mention the options related to jails and to the examples shown here.

The basic operation of `ifconfig(8)` is to create network devices or interfaces, and instrument these with suitable parameters such as IP addresses for an Ethernet interface. Other uses is to create `epair(4)` nodes and `if_bridge(4)` devices. Ifconfig can also create and associate (move) an interface with/into a jail, this is particular interesting when moving one end of an `epair` interfaces into another jail with the purpose of connecting these together.

- The `vnet <jail-id>` option moves an interface from the current nework environment into the jail identified by the jail-id parameter if the jail has a virtual network stack. The interface will disappear from the current network environment and be available in the jail.

- The parameter `-vnet <jail-id>` option for ifconfig removes the interface from the jail identified by the jail-id and places it in the current network environment.
- The `epair` parameter creates a set of logical Ethernet interfaces with a cross-over network cable between them. The epair interfaces can be used to crate internal connections between network entities such as jails and bridges, and can even connect interfaces in different jails. This is done by movint one end of the `epair` interfaces to another jail instance with the `ifconfig` command.
- the `bridge` option is used to create a bridge(4) device. Bridges behaves like real Ethernet bridges and are controlled by options to the `ifconfig(8)` utility by the following options: `addm <interface>` adds an interface to a bridge, `deletem` removing an interface from a bridge.

More information of the `ifconfig(8)` utility can be found in the FreeBSD manpage [29].

B.3 setfib(1) utility

The `setfib(1)` utility is used to associate a process with a non-standard routing table. When a process is associated to a given routing table all descendants to the process is associated with the same process table. Routing tables are named from 1 and to the number of supported routing tables in the system. The supported number can be set either with a `option ROUTETABLES=<number>` kernel configuration file option, or configured at boot time with the `net.fibs=<number>` line in the `/boot/loader.conf` file. More information of the `setfib(1)` utility can be found in the FreeBSD manpage [30].

B.4 The `vimage(8)` utility

The functionality of the `vimage(8)` utility has been fully integrated into the `jail(8)` utility. The `vimage(8)` utility is now moved to the `/usr/src/tools/tools/vimage` and can be installed if needed for backward compatibility.

The options to the `vimage(8)` utility are:

- The default operation of `vimage(8)` is to print the current virtual image name (of the shell executing `vimage(8)`.
- `-c vi_name [options]` Creation and deletion of new virtual images. Options can be e.g. `chroot <path>` to make processes in the new virtual image `chrooted(2)` to the `path`. Virtual images are not `chrooted(2)` by default, and lives in the same file system as the parent virtual image.
- `-i vi_name interface [target_interface]` Network interfaces can be moved from one virtual image into another virtual image.
- `vi_name [command]` Processes can move themselves from their current virtual image to any child virtual images. Processes can not move themselves to their parent or sibling processes. (The design is still being discussed so changes may appear).
- `-l vi_name` Lists the existing virtual images, options and other information of each virtual image instance.
- `-m vi_name` Modifies the parameters of a virtual image with the name `vi_name`, using the same syntax as the `-c` form of the command.

- `-d vi_name` Deletes a virtual image. Virtual images can be deleted when no processes exists in the virtual image to be deleted. All interfaces except loopback interfaces will be reassigned to the parent virtual image.
- The security model of the API created to maintain the vimage, has been designed to only allow the root process user, to perform any task at the system call interface, even the read only operations.

The `vimage(8)` utility manages and operates the Virtual Network Stack of the FreeBSD kernel. The utility is capable of creating, listing and deleting of virtual images (vimages), start up commands inside virtual images, move the console to child vimages, move interfaces from the parent vimage into virtual images and back again and change various options and parameters connected to each vimage. Options include limitation of CPU usage, maximum number of child images and existing processes and also a possibility to isolate a virtual image into a `chroot(2)` environment (The root file system of the child virtual image is locked into a subdirectory of the parents file system).

A short example of using the `vimage(8)` utility is given here as an example of basic use of the program:

```
usbbsd(1)# vimage -l
"default":
84 processes, load averages: nan, nan, nan
CPU usage: (0.00% user, 0.00% nice, 0.00% sys, 0.00% intr)
CPU limits: min 0.00%, max 100.00%, weight 0, no intr limit
No proc limit
Sockets (current/max): 91/0
4 network interfaces
usbbsd(2)# vimage -c h0
usbbsd(3)# vimage -l
"default": [rest of "Default" is deleted]
"h0":      [rest of "h0" is deleted]
usbbsd(4)# vimage h0
Switched to vimage h0
(5)# vimage -l
"h0":
1 processes, load averages: nan, nan, nan
CPU usage: (0.00% user, 0.00% nice, 0.00% sys, 0.00% intr)
CPU limits: min 0.00%, max 100.00%, weight 0, no intr limit
No proc limit
Sockets (current/max): 1/0
2 network interfaces, parent vimage: "default"
(6)# exit
exit
usbbsd(7)# vimage -l
"default": [rest of "Default" is deleted]
"h1":      [rest of "Default" is deleted]
```

Figure B.1: Vimage(8) intro tour. The commands are numbered from (1) to (7). Notice that the command number (4) `vimage h0` starts up a new sub shell in the virtual image "h0" (notice the change of the prompt) and when exiting this sub shell in command (6) `exit` we return back into the parent image named "default".

More detailed information about the `vimage(8)` utility is available at the `vimage(8)` manual page [31].

B.5 The `jexec(8)` utility

The `jexec(8)` utility executes a command within a jail. `Jexec(8)` takes an argument of a jail-id (jid) and an optional user name (from host environment or the jailed environment and runs a command with that users id in the given jail. More information of the `jexec(8)` utility can be found in the FreeBSD manpage [32].

B.6 The `jls(8)` utility

The `jls(8)` command lists the existing jails visible for the environment where it is executed. If the `jls(8)` command is executed at the host environment, all jails at the host is listed, if the `jls(8)` command is executed inside a jail, the jails visible (e.g. descendants of this jail) is listed. More information of the `jls(8)` utility can be found in the FreeBSD manpage [33].

B.7 The `epair(4)` device

The `epair` is a pair of Ethernet-like software interfaces, which are connected back-to-back with a virtual cross-over cable. The two interesting property with `epair` interfaces is that they can be assigned IP and MAC addresses like other interfaces, and the interfaces can be moved to other network environments, making `epairs` an important utility to connect different network environments together.

`Epairs` are a set of software cross-over Ethernet cables between interfaces in the host. The basic intend is to provide connectivity between two virtual network stack instances. As the epair are implemented as two Ethernet interfaces these needs to have a network address configured. A locally administered address by default, this is only guaranteed to be unique within one network stack. Care should be taken when connecting `epairs` between different virtual network stacks.

`Epairs` are assigned a number at creation time, and each end of an epair are identified by the letter "a" or "b". The first created `epair` will then have the two interfaces (ends) named `epari0a` and `epari0b`.

Epairs are created by the ifconfig command which creates a set of epairs.

```
ifconfig epair create

darkron# ifconfig epair create
epair0a

darkron# ifconfig
epair0a: flags=8842<BROADCAST,RUNNING,SIMPLEX,MULTICAST> metric 0 mtu 1500
ether 02:c0:a4:00:06:0a
epair0b: flags=8842<BROADCAST,RUNNING,SIMPLEX,MULTICAST> metric 0 mtu 1500
ether 02:c0:a4:00:07:0b
```

More information of the `epair(4)` utility can be found in the FreeBSD manpage [34].

B.8 The `if_bridge(4)` device

The `if_bridge(4)` device is an software Ethernet bridge, capable of bridging Ethernet network interfaces together. The *if_bridge(4)* bridge module device provides the device named: *bridge(4)* to the kernel.

The `bridge(4)` device is created with the `ifconfig(8)` utility. The `bridge(4)` devices is confined entirely within a (single) network environment. The `if_bridge(4)` connects standard network devices such as `epair(4)`, Network Interface Card devices (NICs) and host network interfaces, tunneling devices etc existing in the same network environment as the *bridge* itself.

The `if_bridge(4)` device also supports the Spanning Tree Protocol (STP) and can be used as a endpoint for tunnel interfaces.

More information of the `bridge(4)` utility can be found in the FreeBSD manpage [35].

Glossary

ATM	Asynchronous Transfer Mode
BGP	Border Gateway Protocol
cost	Cost of a route. A typeless numeric value used to differentiate routes when more possible routes is available. The lower cost the better is the route.
DLCI	Data Link Circuit Identifier
FEC	Forwarding Equivalence Class
FIFO-Queue	First-In - First-Out queue. A queue type where the first element inserted into the queue will be the first element apreaing at the output. Elements will be queued if more elements are written to the queue than are read from the queue.
FTN	FEC to NHLFE Map
IANA	Internet Assigned Numbers Authority. http://iana.org
IGP	Interior Gateway Protocol
ILM	Incoming Label Map
IP	Internet Protocol
LAN	Local Area Network.
layer 2	The protocol layer under layer 3 (which therefore offers the services used by layer 3). Forwarding, when done by the swapping of short fixed length labels, occurs at layer 2 regardless of whether the label being examined is an ATM VPI/VCI, a frame relay DLCI, or an MPLS label.
layer 3	The protocol layer at which IP and its associated routing protocols operate link layer synonymous with layer 2
LDP	Label Distribution Protocol
L2	Layer 2 L3 Layer 3
LSP	Label Switched Path
LSR	Label Switching Router
metric	same as cost. The word indicates a distance to a route, but metric is used in exactly the same way as cost. Normally a router either uses the term cost or the term metric.
MPLS	MultiProtocol Label Switching
MPLS domain	A contiguous set of nodes which operate MPLS routing and forwarding and which are also in one Routing or Administrative Domain

MPLS label	A label which is carried in a packet header, and which represents the packet’s FEC
MultiProtocol Label Switching	An IETF working group and the effort associated with the working group
NHLFE	Next Hop Label Forwarding Entry
OSPF	Open Shortest Path First. A link state routing protocol.
RADIUS	Remote Authentication Dial In User Service
SVC	Switched Virtual Circuit
SVP	Switched Virtual Path
TACACS	Terminal Access Controller Access-Control System.
TTL	Time-To-Live
UUID	Universally Unique Identifier
VC	Virtual Circuit
VCI	Virtual Circuit Identifier
VLAN	Virtual Local Area Network. A technology than enables more 802.3 (Ethernet) Local Area Networks at the same switch or cable.
VLAN Trunk	VLAN Trunk is an interface with several VLANs configured. Traffic flows in and of the interface in decidated channels. Each af these channels are called a VLAN.
VP	Virtual Path
VPI	Virtual Path Identifier
VPI/VCI	A label used in ATM networks to identify circuits
virtual circuit	A circuit used by a connection-oriented layer 2 technology such as ATM or Frame Relay, requiring the maintenance of state information in layer 2 switches.
WAN	Wide Area Network.

Bibliography

[1] Wikipedia. Packet switching, description and history of. Technical report, Wikipedia, 2007, 2010. Available at: http://en.wikipedia.org/wiki/Packet_switching (last seen Sat. 26. june 2010).

[2] Steve Crocker. Rfc 1, host software, a summary of the imp software. Technical report, UCLA, April 1969. Available at: http://tools.ietf.org/html/rfc1 (last seen Sat. 26. june 2010).

[3] S. Hares Y. Rekhter, T. Li. "a border gateway protocol 4 (bgp-4)" rfc 4271. Technical report, IETF, January 2006. Available at: ftp://ftp.rfc-editor.org/in-notes/rfc4271.txt.

[4] Rekhter, Y., Moskowitz, B., Karrenberg, D., and G. de Groot. "ospf version 2", rfc 2328. Technical report, J. Moy Ascend Communications, Inc., April 1998. Available at: ftp://ftp.rfc-editor.org/in-notes/rfc2328.txt.

[5] Kristen Nielsen. "implementing nat support into the xorp project.". Technical report, Department of Computer Science, University of Copenhagen, Denmark., January 2008. Available at: http://krn.dk/thesis Last seen aug 2010.

[6] Shafagh Zandi. Vrf-lite in the "how internetworks work". Technical report, Shafagh Zandis internetworking blog, 07 2010. Available at: http://www.shafagh.net/2009/10/vrf-lite.html (last seen Fri. 3. july 2010).

[7] Wikipedia. Multiprotocol label switching. Technical report, Wikipedia, 06 2010. Availagle at: http://en.wikipedia.org/wiki/Multiprotocol_Label_Switching (last seen Sun. 27. juny 2010).

[8] Wikipedia. Virtual routing and forwarding (vrf). Technical report, Wikipedia, 06 2010. Available at: http://en.wikipedia.org/wiki/VRF (last seen Sun. 27. june 2010).

[9] Poul-Henning Kamp and Robert N. M. Watson. Jails: Confining the omnipotent root. In *Proc. 2nd Intl. SANE Conference*, 2000. Available ps version at: http://docs.freebsd.org/44doc/papers/jail/jail.ps.gz and pdf version at: http://phk.freebsd.dk/pubs/sane2000-jail.pdf.

[10] *"Implementing a Clonable Network Stack in the FreeBSD Kernel"*, San Antonio, Texas, USA, June 2003. USENIX 2003 Annual Technical Conference, FREENIX Track. Available at: http://www.usenix.org/event/usenix03/tech/freenix03/full_papers/zec/zec.pdf.

[11] Archie Cobbs. All about netgraph. Technical report, The FreeBSD Project, Marts 2000. Available at: http://people.freebsd.org/~julian/netgraph.html.

[12] The XORP Project. Xorp design overview. Technical report, The XORP Project, January 2009. Available at: http://www.xorp.org/releases/1.6/docs/design_arch/design_arch.pdf.

[13] The XORP Project. The software status page. Technical report, The XORP Project, April 2004. Available at: http://www.xorp.org/releases/0.5/docs/status.html.

[14] The XORP Project. Xorp inter-process communication library overview. Technical report, The XORP Project, January 2009. XORP technical document available at: http://xorp.org/releases/1.6/docs/libxipc/libxipc_overview.pdf.

[15] The XORP Project. Xorp xrl interfaces: Specifications and tools. Technical report, The XORP Project, January 2009. XORP technical document, available at: http://xorp.org/releases/1.6/docs/libxipc/xrl_interfaces.pdf.

[16] The XORP Project. Xorp router manager process (rtrmgr). Technical report, The XORP Project, January 2009. XORP technical document, avaialable at: http://xorp.org/releases/1.6/docs/rtrmgr/rtrmgr.pdf.

[17] The XORP Project. Xorp user manual version 1.6. Technical report, The XORP Project, January 2009. XORP User Manual, available at: http://xorp.org/releases/1.6/docs/user_manual/user_manual.pdf.

[18] The XORP Project. Xorp snmp agent. Technical report, The XORP Project, January 2009. XORP technical document, available at: http://xorp.org/releases/1.6/docs/snmp/snmp_overview.pdf.

[19] Mark Handley Andrea Bittau. Decoupling policy from protocols. Technical report, Andrea Bittau, Mark Handley, February 2006. A draft paper which discusses the XORP policy framework, the architectural problems we faced, and the solution deployed in XORP. XORP paper, available at http://www.xorp.org/papers/policy.pdf.

[20] The XORP Project. Xorp bgp routing daemon. Technical report, The XORP Project, January 2009. XORP technical document, available at: http://xorp.org/releases/1.6/docs/bgp/bgp.pdf.

[21] G. Malkin Bay Networks. “rip version 2”, rfc 2453. Technical report, November 1998. Availale at: ftp://ftp.rfc-editor.org/in-notes/rfc2453.txt.

[22] The XORP Project. Xorp multicast routing design architecture. Technical report, The XORP Project, January 2009. XORP technical document available at: http://xorp.org/releases/1.6/docs/multicast/multicast_arch.pdf.

[23] The XORP Project. Xorp pim-sm routing daemon. Technical report, The XORP Project, January 2009. XORP technical document, available at: http://xorp.org/releases/1.6/docs/pim/pim_arch.pdf.

[24] The XORP Project. Xorp mld/igmp daemon. Technical report, The XORP Project, January 2009. XORP technical document available at: http://xorp.org/releases/1.6/docs/mld6igmp/mld6igmp_arch.pdf.

[25] The XORP Project. Xorp forwarding engine abstraction. Technical report, The XORP Project, January 2009. XORP technical document available at: http://xorp.org/releases/1.6/docs/fea/fea.pdf.

[26] The XORP Project. Xorp multicast forwarding engine abstraction. Technical report, The XORP Project, January 2009. XORP technical document available at: http://xorp.org/releases/1.6/docs/mfea/mfea_arch.pdf.

[27] The XORP Project. Xorp route information base (rib). Technical report, The XORP Project, January 2009. XORP technical document, available at: http://xorp.org/releases/1.6/docs/rib/rib.pdf.

[28] Bjoern A. Zebb. Poul-Henning Kamp, Robert Watson. jail(8) manpage. Technical report, The FreeBSD Project, July 2009. Available at: project-cd as file:/references/ref-28-freebsd-jail-8-manpage.txt.

[29] The FreeBSD Project. ifconfig(8) manpage. Technical report, The FreeBSD Project, July 2009. Available at: project-cd as file:/references/ref-29-freebsd-ifconfig-8-manpage.txt.

[30] The FreeBSD Project. setfib(1) manpage. Technical report, The FreeBSD Project, April 2008. Available at: project-cd as file:/references/ref-30-freebsd-setfib-1-manpage.txt.

[31] Marco Zec. vimage(8) manpage. Technical report, The FreeBSD Project, August 2009. Available at: project-cd as file:/references/ref-31-freebsd-vimage-8-manpage.txt.

[32] The FreeBSD Project. jexec(8) manpage. Technical report, The FreeBSD Project, May 2009. Available at: project-cd as file:/references/ref-32-freebsd-jexec-8-manpage.txt.

[33] The FreeBSD Project. jls(8) manpage. Technical report, The FreeBSD Project, July 2009. Available at: project-cd as file:/references/ref-33-freebsd-jsl-8-manpage.txt.

[34] Bjoern A. Zebb. epair(4) manpage. Technical report, The FreeBSD Project, July 2009. Available at: project-cd as file:/references/ref-34-freebsd-epair-4-manpage.txt.

[35] Andrew Thompson Jason L. Wright, Jason R. Thorpe. af_bridge(4) manpage. Technical report, The FreeBSD Project, June 2009. Available at: project-cd as file:/references/ref-35-freebsd-bridge-4-manpage.txt.