



Facultés Universitaires Notre-Dame de la Paix - University of Namur
Institut d'Informatique - Computer Science Institute

Dynamic Configuration of Label Switched Path

Julien Bisconti

Supervised by:

Laurent Schumacher

Mémoire présenté en vue de l'obtention du grade de Maître en
Informatique

Année académique 2006-2007

ABSTRACT

This report deals with the analysis of dynamic configuration of label switched paths (LSPs) with help of the project: **mpls-linux**. This analysis was made over a virtual testbed which was created with **Netkit** and User-Mode Linux (UML) under Linux. Different scripts were developed in order to configure the LSPs and allow a datastream to switch from one LSP to another.

This paper starts with explaining the main MultiProtocol Label Switching (MPLS) concepts. The reader that already masters these concepts may safely skip this chapter. Then, the virtual testbed is described as well as the underlying concepts of virtual machines. Afterwards, the use of an implementation of the Label Distribution Protocol (LDP) is described to create a LSP based on an IP-prefix FEC (i.e. destination-based FEC). In the following chapter, the explicit routing and dynamic rerouting of LSP is outlined. Finally, pointers to further research will be exposed.

RÉSUMÉ

Ce rapport décrit l'analyse de configuration dynamique de "chemins d'échange de label" (LSPs) avec l'aide du projet: **mpls-linux**. Cette analyse a été effectuée sous Linux, grâce à un testbed virtuel créé à l'aide de **Netkit** et de User-Mode Linux (UML). Différents scripts ont été développés pour configurer les LSPs et permettre à un flux de données de passer d'un LSP à un autre.

Ce mémoire s'ouvre sur la présentation des principaux concepts du multi-protocol d'échange de label (MPLS). Le lecteur maîtrisant déjà ces concepts peut passer ce chapitre. Ensuite, une description du testbed virtuel sera donnée ainsi que des concepts sous-jacents aux machines virtuelles. Suite à cela viendra une description de l'utilisation de l'implémentation du protocole de distribution de labels qui servira à créer un LSP bas sur un FEC à préfixe IP. Dans le chapitre suivant, le routage explicite et reroutage dynamique de LSP sera décrit. Finalement, des pointeurs vers de futures recherches seront donnés.

PREFACE

This report is mainly the result of a contribution to a European project called AROMA, carried out in Fall 2006 in the Universitat Politècnica de Catalunya (Barcelona, Spain) and continued in the University of Namur (Belgium) in spring 2007.

The structure of the report does not exactly reflect the way this project was conducted. This AROMA project aims to guarantee the end-to-end QoS in the context of an all-IP heterogeneous network. The documentation on the AROMA testbed and MPLS came first. Then, the validation of the MPLS and LDP implementation was done. Afterwards, MPLS was installed on the AROMA testbed after upgrading it in order to be compliant with its requirements. Afterwards, the set up of a virtual testbed designed the same way as the AROMA testbed was necessary to continue the tests. Finally, simulations concerning dynamic LSP configuration were executed and the results are exposed in this document.

I would like to thank my supervisor, Laurent Schumacher, for his guidance during the project and for his help in writing this report. I would like to thank my Spanish supervisors, Dr. Fernando Casadevall and Anna Umbert, for accepting me during five months on the AROMA project.

ABBREVIATIONS

ARIS	Aggregate Route-based IP Switching
ARP	Address Resolution Protocol
ATM	Asynchronous Transfer Mode
BGP	Border Gateway Protocol
CIDR	Classless Inter Domain Routing
CLI	Command Line Interface
CR-LDP	Constraint-based Routing Label Distribution Protocol
DoD	downstream on-demand label distribution (mode)
DU	downstream unsolicited label distribution (mode)
FEC	Forwarding Equivalency Classes
FTN	FEC-to-NHLFE map
ILM	Incoming Label Map
IPv4	IP version 4
IPv6	IP version 6
ISP	Internet Service Provider
LAN	Local Area Network
LDP	Label Distribution Protocol
LER	Label Edge Router
LSP	Label Switched Path
LSR	Label Switched Router
MIB	Management Information Base
MPLS	MultiProtocol Label Switching
NHLFE	Next Hop Label Forwarding Entry
NSP	Network Service Provider
PHP	Penultimate Hop Popping
PVC	Permanent/Private Virtual Circuit

QoS	Quality of Service
RFC	Request for Comments
RSVP	Resource Reservation Protocol
TE	Traffic Engineering
UML	User Mode Linux
VPN	Virtual Private Network

TABLE OF CONTENTS

1	Multi-Protocol Label Switching	1
1.1	Introduction	2
1.1.1	Protocol-Independent Forwarding	2
1.1.2	Forwarding granularity	3
1.1.3	Traffic Engineering	3
1.2	Routing versus Switching	4
1.2.1	Example	5
1.2.2	Comparison of Approaches	6
1.3	Architecture	9
1.3.1	Key Concepts	9
1.3.2	Forwarding Equivalency Classes	14
1.3.3	Hierarchy and Label Stacking	16
1.3.4	Label Stack Encoding	18
1.3.5	Loop Detection	20
1.4	Label Distribution Protocol	21
1.4.1	LDP	21
1.4.2	CR-LDP	24
1.4.3	RSVP-TE	26
1.4.4	Comparison	30
2	Virtuals Testbeds	33
2.1	Introduction	33
2.1.1	Definitions	33
2.2	Netkit	34
2.2.1	Introduction	34
2.2.2	Usability and Readiness	36

2.2.3	The Zebra Routing Software Suite	42
2.2.4	Conclusion on Netkit	44
2.3	AROMA testbed	44
2.3.1	Description	44
2.3.2	The Real AROMA testbed	46
2.3.3	Deployment Study Case	48
3	Label Distribution on AROMA	51
3.1	Introduction	51
3.1.1	Hypothesis on Experiment	52
3.2	Implementation of LDP	56
3.3	Experiment	57
3.3.1	Starting the Testbed	57
3.3.2	Configuring a virtual machine	57
3.3.3	Setting up LSPs	60
3.4	Benchmarking	61
3.4.1	Benchmark Environment	61
3.4.2	Results	62
3.5	Conclusion	65
4	Constraint Routing on AROMA	67
4.1	Rerouting LSPs method from RFC 3214	67
4.2	Description of the LSPs	69
4.2.1	Main LSP	70
4.2.2	Backup LSP	70
4.3	Hypothesis on experiment	71
4.4	The mpls command	71
4.4.1	Ingress Router Command	72
4.4.2	Core Router Command	72
4.4.3	Egress Router Command	73
4.5	Rerouting LSPs	74
4.6	Conclusion	75
5	Future Work	77
5.1	Conclusion	77
5.2	MPLS versus IPv6	78
5.3	Fast Reroute	78
5.4	Preemption	78
	Appendices	83

A	Certificate AROMA	85
B	Enlarge Netkit Filesystem	87
B.1	Checking Filesystem Consistency (optional)	87
B.2	Resizing Filesystem	88
B.3	Install new software	89
C	Quagga Configuration File	91
D	MPLS Command Syntax	93
E	AROMA Starting Script	95
F	Rerouting Script	107
F.1	Setup Script	107

LIST OF FIGURES

1.1	Architecture of a normal IP router [22]	4
1.2	Basic Operations of MPLS	6
1.3	Architecture of a label switch [22]	7
1.4	A simple MPLS network	10
1.5	Label-forwarding table	11
1.6	Nested LSP	16
1.7	Label stack entry format	19
1.8	CR-LDP LSP setup. [22]	25
1.9	RSVP-TE explicit LSP setup. [22]	28
2.1	UML and host system	41
2.2	UML-MPLS Linux Kernel.	42
2.3	Zebra/Quagga and ldpd	43
2.4	Traffic Engineering: The Fish Problem. [21]	48
2.5	The Virtual AROMA Testbed.	48
3.1	4 different routes for LSPs.	52
3.2	Screenshot of a started virtual machine	58
3.3	Screenshot of LDP message capture.	59
3.4	LSP setup by LDP.	61
3.5	MPLS vs IPv4 TCP/IP benchmark.	62
4.1	2-way explicit MPLS route.	69
4.2	Graphic of packets from LSPs switch (simple ping).	74
4.3	Switch of LSP.	75
A.1	Certificate from UPC for the AROMA testbed.	86

LIST OF TABLES

1.1	Routing versus Switching [22]	8
1.2	New Objects in RSVP-TE	27
1.3	Comparison of CR-LDP and RSVP-TE	31
2.1	IP configuration of the virtual AROMA testbed.	49
3.1	TOS field value for iptables.	54
3.2	Layers involved in the LDP implementation.	56
3.3	MPLS vs IPv4 - values.	63
B.1	Checking filesystem consistency	87
B.2	Resizing filesystem.	88
B.3	Install new software.	89

MULTI-PROTOCOL LABEL SWITCHING

Multiprotocol Label Switching represents the convergence of two fundamentally different approaches in networking: datagram and virtual circuit. The datagram model¹ use routing protocols to precalculate the paths to all destination networks by exchanging routing information, and each packet is forwarded independently based on its destination address. On the other hand, a virtual circuit must be set up explicitly by a signaling protocol before packets can be transmitted into the network. MPLS is designed to allow a virtual circuit to be set up in a datagram network.

Label switching uses a short, fixed-length label inserted in the packet header to forward packets. A *label-switched router* (LSR) uses the label in the packet header as a index to find the next hop and the corresponding new label. The packet is sent to its next hop after the existing label is swapped with the new one assigned for the next hop. The path that the packet traverses through a network is defined by the transition in label values. Such a path is called a *label-switched path* (LSP). Since the mapping between labels is fixed at each LSR, an LSP is determined by the initial label value at the first LSR of the LSP. The purpose of label switching is not to replace IP routing but rather to enhance the services provided in the IP networks by offering scope for traffic engineering, guaranteed QoS, and virtual private network (VPNs).

In this chapter, the architecture and protocols of MPLS will be discussed. The first section will introduce few routing concepts and network management. The second section explains the differences between the two the fundamental networking concepts, routing versus switching. The third section exposes the architecture of MPLS. Finally, the last section compares two extensions of label distribution protocols.

¹Which the traditional IP forwarding in the Internet is based on

The motivations of MPLS will not be discussed here. The reader who is interested in the historical reasons and motivations of MPLS can refer to [26] or [10].

This chapter cites material from the book of Zheng Wang, [26], from the article [10] and from [22].

1.1 Introduction

“In this section, three concepts will be described in order to let the reader understand what is at stake with the MPLS technology. MPLS is not a silver bullet to cure existing or forthcoming problems, but rather an enabling technology which addresses some of these scaling issues. It does this by replacing the standard destination-based hop-by-hop forwarding paradigm with a label-swapping forwarding paradigm. This has the benefit of simplifying the packet-forwarding engine and enabling easy scaling.

1.1.1 Protocol-Independent Forwarding

Routing and forwarding are tightly coupled in current IP architecture. Any changes in routing architecture will affect the forwarding path. For example, IP forwarding used to be based on three classes with network prefixes 8, 16, and 24 bits long. The address exhaustion problem promoted the development of *Classless Inter Domain Routing* (CIDR). However, implementing CIDR requires a change in the forwarding algorithm of all IP routers to use longest-prefix lookup. The forwarding algorithm is implemented in hardware or fine-tuned software to ensure performance; making changes to it can be expensive.

Label switching decouples forwarding from routing. An LSP may be set up in a variety of ways, and once the LSP is established, packet forwarding is always the same. Thus new routing architectures can be implemented without any changes to the forwarding path. Take multicast as an example: unicast packet forwarding is typically based on the destination address. However, multicast forwarding may require a lookup based on both the source address and the destination address. Modifying the unicast forwarding engine to accommodate multicast requires substantial changes or a completely separate forwarding engine for multicast. With label switching, a label for a multicast stream will be associated with the source and destination addresses at the setup phase. The label lookup during multicast forwarding remains unchanged for multicast.

1.1.2 Forwarding granularity

Forwarding granularity refers to the level of aggregation in the decision making of routing protocols. For example, forwarding granularity in current IP routing is destination based: all packets with the same network number are grouped and treated the same way. Although destination-based forwarding is highly scalable, different forwarding granularities are sometimes desirable. For example, an ISP may want specific customers to receive different forwarding treatments. Implementation of such a scheme requires routers to know from which customer a packet originates; this may in turn require packet filtering based on the source or destination address in routers throughout the network.

1.1.3 Traffic Engineering

Label switching was initially driven by the need for seamless IP/ATM integration and to simplify IP forwarding. However, rapidly changing technologies have made these considerations less important. Instead traffic engineering has emerged as the key application of label switching.

Traffic engineering refers to the process of optimizing the utilization of network resources through careful distribution of traffic across the network. In today's datagram routing, traffic engineering is difficult to achieve. IP routing is destination based, and traffic tends to distribute unevenly across the backbone. Although some links are heavily congested, others may see very little traffic. The result of such unbalanced traffic distribution is that resource utilization is typically poor. Some of traffic engineering can be done by manipulating the link metrics. For example, when a link is congested, its cost metric can be increased in order to move traffic to other links. However, it is typically a trial-and-error process and becomes impractical for large networks.

Most Internet backbones today use ATM or FR to interconnect IP routers. The PVCs² in ATM and FR allow engineers to manually configure multiple PVCs and adjust the routes to equalize the load of the traffic across the network. This is an important functionality currently missing in IP routing. Label switching uses similar connection-oriented approaches and can easily replace the PVC functionality in ATM and FR. With label switching, all traffic flows between an ingress node and an egress node can be individually identified and measured. LSPs can also be set up with explicitly specified routes, or explicit routes. The entire path can be computed based on sophisticated algorithms that optimize resource utilization.

²Private Virtual Circuit

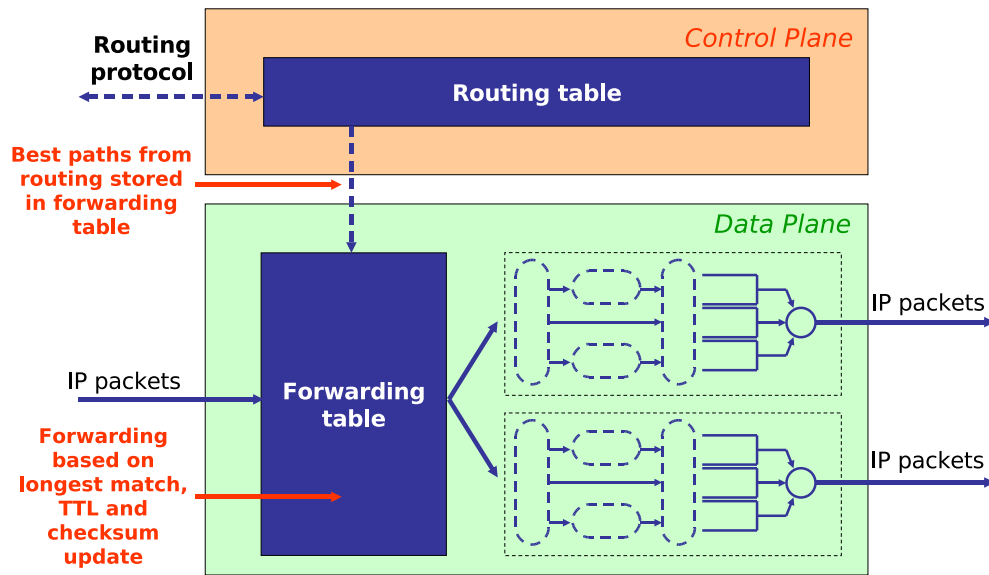


Figure 1.1: Architecture of a normal IP router [22]

1.2 Routing versus Switching

To understand label switching, let us first look at how packets are forwarded in IP routers (see Figure 1.1). The operation of an IP router can be partitioned into two basic threads: a **data path** and a **control path**. The *data path* is the "executive branch," which performs the actual forwarding of packets from ingress ports to their appropriate egress ports. When a packet arrives at the router, the data path uses a forwarding table and the information contained in the packet header to determine where the packet should be forwarded. The control path, on the other hand, is responsible for making forwarding decisions. In the *control path*, routing protocols exchange updates among routers and calculate the route to each network prefix based on the routing metrics. The forwarding table consists of the network prefix and the corresponding next-hop information produced by the routing protocols.

In typical layer-3 IP routers, unicast forwarding is done by taking the destination address of the incoming packet and performing a longest match against the entries in the forwarding table. The lookup involves a longest match on the source address and a fixed-length match with the destination address.

In contrast to layer-3 routers, layer-2 switches such as ATM, FR, or Eth-

ernet switches all use a short, fixed-length label for route lookup. Typically a label does not encode any information from the packet headers and has local significance; it can be viewed simply as an index in the forwarding table. With this approach, the data path becomes very straightforward: it involves a direct lookup to find the outgoing interface and the label for the next hop. The simplification in the data path, however, comes at a price. The control path has to set up the labels across the path that packets traverse. This is done in ATM and FR by their own signaling protocols, which set up the connections.

These two different approaches are often referred to as *routing* versus *switching*. The main difference is that in the routing approach, the router has to look at the fields of the packet header in the data path and match the entries in the forwarding table; in the switching approach, the information in the packet header is examined in the control path and the result is associated with an index, which is used in the forwarding. Routing versus switching is once again an open debate; it is a continuation of the two classic approaches to networking, datagram versus virtual circuit.

1.2.1 Example

MPLS uses the switching approach. In many aspects it is very similar to ATM or FR. Let us look at the basic operations of MPLS. This example explains the concept of switching, technical details concerning the architecture and protocols are discussed later in this chapter.

Figure 1.2 shows a simple MPLS backbone network connecting multiple customer sites. Two LSPs are established: one LSP connects customer 1 and customer 3 using labels 23 and 42 over path $A \rightarrow C \rightarrow E$, and the other connects customer 2 to customer 4 using labels 12, 96, and 24 through path $A \rightarrow B \rightarrow D \rightarrow E$. Suppose that customer 1 wants to send packets to customer 3. Node A attaches label 23 to the packets from customer 1. When node C receives the labeled packets from node A it looks them up in the label-forwarding table for the corresponding outgoing label, which is 42. Node C then replaces label 23 with 42 and sends the packet to node E . Node E realizes that it is the end of an LSP and removes the label, sending the packet on to customer 3.

In a broad sense MPLS is a general framework for the switching approach (Figure 1.3). There are two components: a signaling protocol, which can be IP, ATM, FR, and so on and sets up an LSP, and a data plane that forwards packets based on the labels in the packets. This framework can

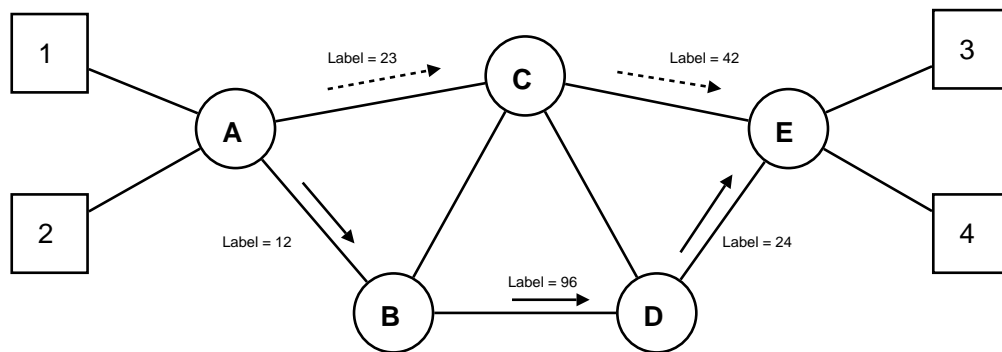


Figure 1.2: Basic Operations of MPLS

be mapped to a specific set of protocols in many different ways. For example, MPLS has been mapped to ATM by using IP control protocols to manage ATM switches. However, an alternative proposal could be to use ATM control protocols to manage label-switched routers.

In practice, however, the control plane in MPLS is usually IP based. This is because IP control protocols, particularly routing protocols, have proved to be more mature and scalable than possible alternatives. An IP-based control plane in MPLS enables seamless integration between IP and MPLS, avoiding many of the problems in running IP over ATM.

MPLS, as standardized by the IETF, has an IP-based control plane that manages different types of label-based link layers. The MPLS signaling protocols may be used to set up LSPs for IP routers, ATM switches, and FR switches. MPLS has even been proposed to manage optical transport networks that do not perform label-based forwarding at all. The data plane may be ATM cells, FR frames, or IP packets with a shim header (described in Section 1.3.4).

A unified control plane tightly integrated with the IP control protocols that MPLS offers has considerable attraction. It brings simplicity to the management of large IP networks and provides the necessary mechanisms for performing traffic engineering and performance optimization, which the current IP network lacks.

1.2.2 Comparison of Approaches

One of the key issues in label switching is the way the LSPs are established. The many different approaches can be divided into two basic groups: data driven and control driven. In a control-driven approach, the setup of

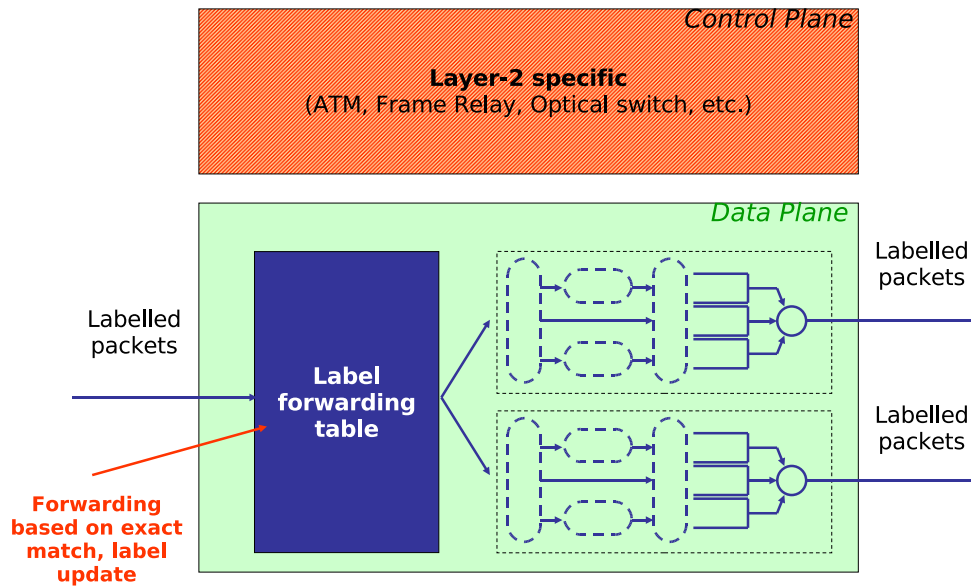


Figure 1.3: Architecture of a label switch [22]

LSPs is initiated by control messages such as routing updates and RSVP³ messages. This can be implemented in two ways. One way is to piggyback the label information in the control messages. For example, a label can be carried in the routing updates to pass on the label that a neighbor should use. This approach is simple but requires modifications to the existing protocols; for example, extensions to current routing protocols are needed in order to carry labels in routing updates. An alternative is to use a separate label distribution protocol for setting up LSPs. The label distribution protocol may be driven by the control protocols.

Label setup is decoupled from the data path in the sense that all LSPs are preestablished during the processing of control messages. Thus when the data packets arrive, the LSPs are already established. The control-driven approach allows flexible control over the way the LSPs are set up. For example, it is possible to set up an LSP for all traffic that exits from a specific egress router of a network or an explicitly specified route, as we will discuss later in this chapter (Section 1.3.1).

The control-driven approach has its downside too. LSPs may be confined to one control domain. For example, routing-based schemes, such as

³Resource Reservation Protocol

Routing	Switching
Conventional datagram routing	Label switching
Connectionless	Connection-oriented
Packet-oriented	Circuit-oriented Virtual Circuit
Scalability	MAXimise resource commitments
Resistance to failures	MAXimise network utilization
Shortest path	Explicit routing

Table 1.1: Routing versus Switching [22]

tag and ARIS⁴, can establish LSPs only within a routing domain. Additional mechanisms such as label stacking are needed in order to cross the control domain borders. For a large network, pre-establishing all LSPs may lead to scalability issues; the number of LSPs that have to be set up can easily run into the tens of thousands. The messaging overheads during routing changes may cause congestion and overloading in control processors.

The mechanisms used in a control-driven approach are inevitably specific to the control protocols on which the mechanisms are based. For example, both tag switching and ARIS use route-based mechanisms for setting up switched paths along the forwarding table produced by the routing protocol and use RSVP-based mechanisms for setting up switched paths for RSVP flows. [...]

In the data-driven approach, the setup of an LSP is triggered by data packets. The LSPs are set up on-the-fly while the data packets are arriving. Obviously the first few packets must be processed at the IP level until the corresponding LSP is set up. In the Ipsilon scheme⁵, for example, the setup of a switched path starts when a switch detects that a flow is long lasting (either by matching a well-known port number or by waiting for a threshold number of packets to be forwarded). Thus the data-driven approach is less deterministic since it depends on the traffic patterns in the network. When there are many short-lived connections, the performance of label switch-

⁴Aggregate Route-based IP Switching

⁵**Ipsilon Networks** was a computer networking company which specialized in IP switching.

ing tends to deteriorate since setting up LSPs for short-lived connections is much more costly in terms of overhead.

The data-driven approach is also less flexible than the control-driven approach. Note that in the data-driven approach, an LSP works like the "cache" of a path: it somewhat passively reflects the path that packets traverse. As such, it cannot be used to control the setup of the LSP. For example, it would be difficult to implement explicit routes with the data-driven approach." [26]

1.3 Architecture

In this section we describe the basic concepts, architecture, and protocols in MPLS.

1.3.1 Key Concepts

"Figure 1.4 shows a simple MPLS network with four LSRs and three LSPs ($A \rightarrow B \rightarrow C$, $A \rightarrow B \rightarrow D$, and $C \rightarrow B \rightarrow D$). The first and last LSRs over an LSP are called the *ingress* and *egress*, respectively. For LSP 1 in Figure 1.4, LSR *A* is the ingress and LSR *C* is the egress. The operation of ingress and egress LSRs is different from that of an intermediate LSR in many aspects. LSPs are directional. For any pair of LSRs, the LSR that transmits packets with respect to the direction of data flow is said to be **upstream**, whereas the LSR that receives packets is **downstream**. For example, for LSP 1 in Figure 1.4, LSR *A* is upstream of LSR *B*, and LSR *B* is downstream of LSR *A*.

Label

As described in section 1.2, a label is a short, fixed-length, locally significant identifier that is used for label switching. A packet is called a labeled packet if a label has been encoded into the packet. The label may be encoded into a packet in many ways. In some cases the label may be mapped to some field in an existing data link or network layer protocol. For example, when we use ATM switches as LSRs, the VCI/VPI⁶ fields are used as the MPLS label. However, this approach is not always feasible. A simple MPLS encapsulation has been standardized that will carry the label and some additional information. This adds another thin layer between the data link layer and the IP layer specifically for MPLS processing.

⁶Virtual Channel Identifier/Virtual Path Identifier

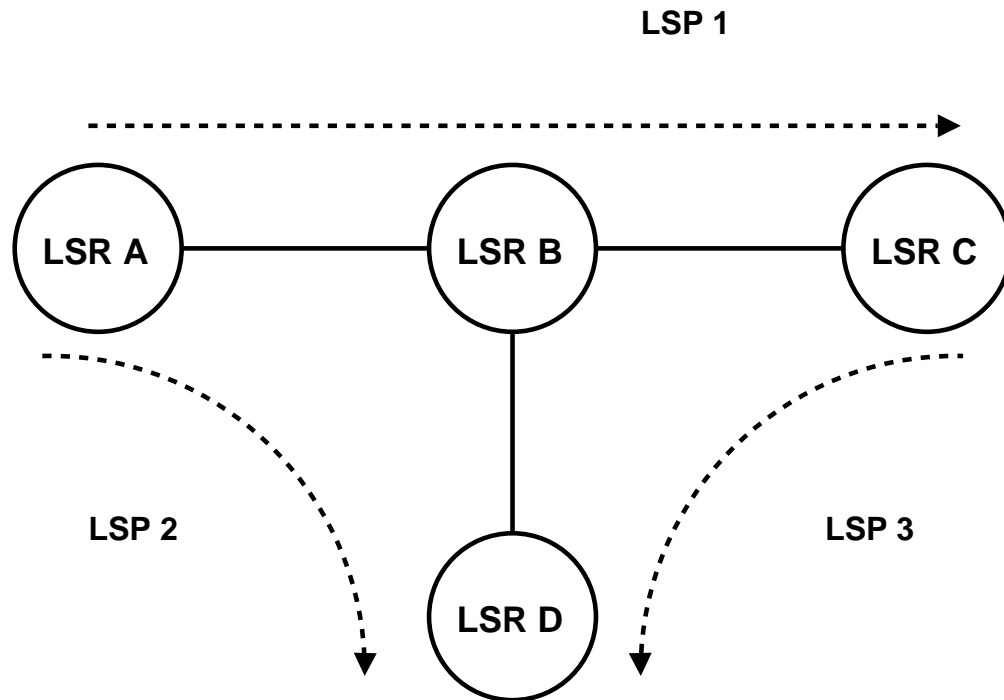


Figure 1.4: A simple MPLS network

Because the label is the only identifier that is used for packet forwarding, an LSR must be able to associate the incoming label with an LSP. For LSRs that are connected by point-to-point connections, the label needs to be unique only to the point-to-point interface. In other words, the interface can use the entire label space, and the same label value may be used over different interfaces. When there are multiaccess interfaces, an LSR may not be able to determine, solely based on the label, which neighbor sent the packet. Thus the label must be allocated uniquely across all multiaccess interfaces. In actual implementations **label space** may be unique across the entire LSR. In this case label space is partitioned among all interfaces, so the usable space for each interface is much smaller.

Each label is associated with a FEC⁷. An FEC defines a group of IP packets that are forwarded over the same LSP with the same treatment. It can be described as a set of classification rules that determine whether a packet belongs to a particular FEC or not. Different types of FEC will be discussed later in this section. It is important that the binding from a label to an FEC be one to one. If multiple FECs are associated with a label, a downstream LSR will not be able to distinguish packets of different FECs

⁷Forwarding Equivalency Classes. See section 1.3.2

by simply looking at the label.

Hierarchical Label Stack

MPLS allows more than one label to be encoded in a packet. This is referred to as a label stack since the labels are organized as a last-in, first-out stack. A label stack is used to support nested tunnels. An LSP may have another nested LSP between an LSR and can push in a new label on top of the current label so that the packet will follow the tunnel pointed out by the new label on the top of the stack. When the packet reaches the end of the tunnel, the LSR at the end of the tunnel discards the top label and the one below pops up. Label stacking is discussed in depth in Section 1.3.3.

Label-Switching Table

The label-switching table, also called an *incoming label map* (ILM), maintains the mappings between an incoming label to the outgoing interface and the outgoing label (Figure 1.5). Its functions are similar to those of the packet-forwarding table in IP routers. The entry that the incoming label points to is called the *next-hop label-forwarding entry* (NHLFE). Each incoming label typically points to one NHLFE. However, in the case of load sharing, there may be multiple NHLFEs for an incoming label. The method for splitting traffic in the case of multiple NHLFEs is not specified in the standard.

Typically the NHLFE contains the next hop and the outgoing label for that next hop. If an LSR is the ingress or egress of an LSP, the NHLFE also specifies the actions for manipulating the label stack. NHLFEs may also contain additional state information related to the LSP; for example, hop count and data link encapsulation to use when transmitting the packet.

Incoming Label	Outgoing Label	Next-hop address	Per-label state
----------------	----------------	------------------	-----------------

Figure 1.5: Label-forwarding table

LSRs use the label-switching table for forwarding labeled packets. When a packet arrives, an LSR finds the corresponding NHLFE for the incoming label by performing a lookup in the label-switching table. The LSR then replaces the incoming label with the outgoing label and forwards the packet to the interface specified in the corresponding NHLFE.

Label Distribution Protocols

Before LSPs can be used, the label-switching table at each LSR must be populated with the mappings from any incoming label to the outgoing interface and the outgoing label. This process is called *LSP setup* or *label distribution*.

A label distribution protocol is a set of procedures by which two LSRs learn each other's MPLS capabilities and exchange label-mapping information. Label distribution protocols set up the state for LSPs in the network. Since protocols with similar functions are often called signaling protocols in ATM or circuit-based networks, the process of label distribution is sometimes called **signaling**, and label distribution protocols are called signaling protocols for the MPLS networks.

The MPLS architecture does not assume that there is only a single label distribution protocol. In fact, it specifically allows for multiple protocols for use in different scenarios. The IETF MPLS Working Group has specified LDP as a protocol for hop-by-hop label distribution based on IP routing information. For explicitly routed LSPs or LSPs that require QoS guarantees, CR-LDP⁸ and RSVP-TE⁹ are two protocols that support such functions. We will cover label distribution protocols in Section 1.4.

Label Assignment and Distribution

In MPLS the decision to bind a particular label to a particular FEC is always made by the downstream LSR with respect to the flow of the packets. The downstream LSR then informs the upstream LSR of the binding. Thus the data traffic and control traffic flow in opposite directions. For LSP 1 in Figure 1.4, packets flow from LSR A to LSR B, whereas label assignment between A and B is determined by LSR B and distributed to LSR A.

Although the upstream LSR can assign labels and inform the downstream LSR, downstream label distribution was chosen for good reasons. Consider LSR A and B in Figure 1.4. Suppose that the label for LSP 1 between A and B is F. The upstream LSR A only has to put label F in a packet on LSP 1 before it sends to LSR B. When LSR B receives the packet from LSR A, it has to perform a lookup-based label F. Implementation is much easier when the downstream LSR B, gets to choose the label so that the labels are assigned only from specific ranges and the lookup table can be made more compact. Label merging, which we will discuss next, also requires downstream label distribution.

⁸Constraint-based Routing Label Distribution Protocol

⁹Reservation Protocol for Traffic Engineering

There are two different modes of downstream label distribution: **downstream on demand** and **unsolicited downstream**. With the downstream-on-demand mode, an LSR explicitly requests a neighbor for a label binding for a particular FEC. The unsolicited downstream mode, on the other hand, allows an LSR to distribute label bindings to its neighbors that have not explicitly requested them. Depending on the characteristics of interfaces, actual implementations may provide only one of them or both. However, both of these label distribution techniques may be used in the same network at the same time. On any given label distribution adjacency, the upstream LSR and the downstream LSR must agree on which mode is to be used.

Label Merging

In MPLS, two or more LSPs may be merged into one. Take LSP 2 and LSP 3 in Figure 1.4 as an example again. Note that LSR B receives packets from LSR A for LSP 2 and packets from LSR C for LSP 3. However, all these packets go from LSR B to LSR D. Thus it is possible for LSR B to use the same label between LSR B and LSR D for all packets from LSP 2 and LSP 3. In essence the two LSPs are merged into one at LSR B and form a label-switched tree. In general, when an LSR has bound multiple incoming labels to a particular FEC, an LSR may have a single outgoing label to all packets in the same FEC. Once the packets are forwarded with the same outgoing label, the information that they arrived from different interfaces and/or with different incoming labels is lost.

Label merging may substantially reduce the requirement on label space. With label merging, the number of outgoing labels per FEC need only be one; without label merging, the number could be very large. Let us look at a practical example. Suppose that we would like to set up MPLS LSPs between all edge nodes of a network and there are N edge nodes. The worst-case label requirement without label merging, namely, the maximum number of labels required on a single link in one direction, is approximately $N^2/4$. However, if we merge labels of packets destined to the same destination node, the worst-case number is merely N .

Not all LSRs may be able to support label merging. For example, LSRs that are based on ATM cannot perform label merging because of cell interleaving. In such cases different labels should be used even though an LSR has packets from different interfaces with the same FEC. This issue will be discussed in Section 1.3.4.

Route Selection and Explicit Routing

During the label distribution process, an LSR needs to determine which is the next hop for the LSP that it tries to establish. There are two basic approaches to determine this: **hop-by-hop routing** and **explicit routing**. The hop-by-hop approach relies on IP routing information to set up LSPs. The MPLS control module will, at each hop, call the routing module to get the next hop for a particular LSP. The routing module at each LSR independently chooses the next hop based on IP routing or other routing methods. The other approach is explicit routing. In this mode a single LSR, generally the ingress or the egress of the LSP, specifies the entire route for the LSP. The routes for the LSPs can be computed by routing algorithms designed to achieve certain prespecified objectives. Such routing algorithms are often referred to as constraint-based routing.

If the entire route for the LSP is specified, the LSP is “strictly” explicitly routed. If only part of the route for an LSP is specified, the LSP is “loosely” explicitly routed. This is very similar to the concept of strict source routing and loose source routing in IP. Once a loosely explicitly routed LSP is established, it may change or it can be pinned so that it always uses the same route.

The explicitly routed LSP, or explicit route, has emerged as one of the most important features in MPLS. It provides a mechanism for overriding the routes established by IP routing. This can be used to route traffic around congested hot spots and optimize resource utilization across the network. Without the explicit route mechanism, such features cannot be easily implemented in current IP networks.

1.3.2 Forwarding Equivalency Classes

IP routers currently use a small number of fields in a packet header to make forwarding decisions. In destination-based routing, only the network number part of the destination address is used to select the next hop. All packets that have the same destination network number follow the same path and receive the same treatment. Thus the forwarding process can be viewed as one that partitions packets into a finite number of sets. Within the same set, all packets are treated the same way. We call a set of packets that are treated identically in the forwarding process a FEC.

An FEC can be expressed as a set of classification rules that determine if a packet belongs to the FEC. For example, a set of packets with the same destination network number is an FEC for destination-based IP routing these packets receive identical treatment in the forwarding process.

IP-based networks currently do not support many different types of FECs since this would require classification of packets during packet processing to match the packets to FECs. MPLS can easily support many different types of FECs. In MPLS the classification of packets is moved from the data plane to the control plane. Once an LSP is set up, only the ingress LSR needs to classify packets to FECs.

FEC is closely related to the concept of forwarding granularity discussed early in Section 1.1.2. The types of FECs supported by a network in fact determine the forwarding granularity. For example, suppose that a network supports a FEC that classifies packets based on their source and destination addresses. This will result in a finer forwarding granularity than current destination-based forwarding. A coarse forwarding granularity is essential to scale to large networks, whereas a fine granularity allows maximal control over the forwarding of packets inside the network. MPLS allows multiple types of granularity to coexist over the same forwarding path. The common types of FECs that MPLS supports include :

- **IP prefix.** Packets that match an IP destination prefix in the routing table are considered as one FEC. This is a direct mapping from the routing table to the label-switching table, enabling MPLS to support the destination-based forwarding in current IP routing. One advantage of such FECs is that the label distribution may be closely coupled with IP routing and driven by the events in routing protocols. It is also feasible to piggyback on the routing protocols so that the message overheads for label distribution are minimized.
- **Egress router.** In most backbone networks, packets come in from the ingress node and go out from the egress node. A useful FEC includes all the packets that go out on the same egress node. Such granularity is very hard to support in a datagram model. With MPLS, however, one can set up LSPs to a particular egress LSR based on the information from the BGP¹⁰ Next Hop in a BGP update message, from the OSPF Router ID in the OSPF advertisement, or directly via MPLS label distribution. This represents the coarsest granularity currently available and can scale to large networks. The ability to identify streams between ingress and egress node pairs is also useful when it comes to supporting traffic engineering within a backbone network¹¹.
- **Application flow.** This type of FEC results in the finest granularity since each application flow is one FEC. It is the least scalable of all

¹⁰Border Gateway Protocol

¹¹RFC 2702 [5] and RFC 3272 [4]

granularity types. The advantage of application flow is, however, that it provides end-to-end switching and allows maximum control of the traffic flows in the network. Application flow is best suited for handling special purposes.

There is a clear trade-off between scalability and controllability. The ability of MPLS to support multiple FECs with different types of forwarding granularity gives a lot of flexibility in accommodating different requirements and combining different forwarding granularities in the same network.

1.3.3 Hierarchy and Label Stacking

MPLS allows multiple labels to be encoded into a packet to form a label stack. Label stacking is used to construct nested LSPs, similar to the capability of IP-in-IP tunneling or loose source routing in IP routing. Such nested LSPs can create a multilevel hierarchy where multiple LSPs can be aggregated into one LSP tunnel.

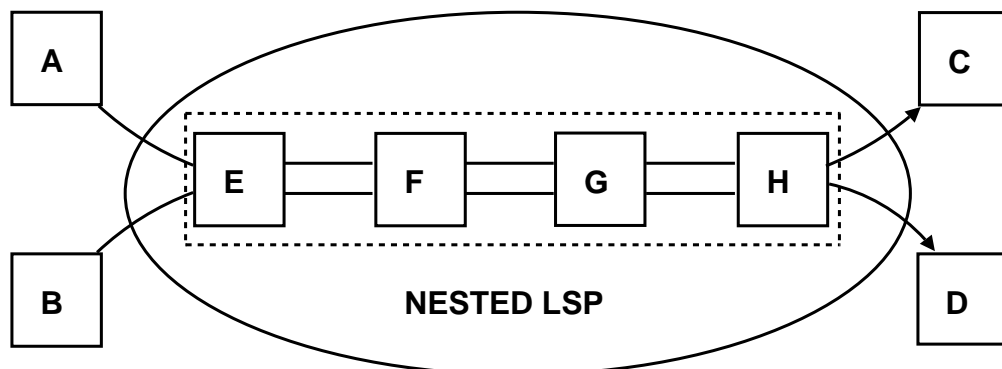


Figure 1.6: Nested LSP

Consider the backbone network shown in Figure 1.6, which connects many networks. Suppose that LSR *A* and *B* want to set up two LSPs to LSR *C* and *D*, respectively. We can set up two LSPs as $A \rightarrow E \rightarrow F \rightarrow G \rightarrow H \rightarrow C$ and $B \rightarrow E \rightarrow F \rightarrow G \rightarrow H \rightarrow D$. With label stacking, we can first set up an LSP tunnel as $E \rightarrow F \rightarrow G \rightarrow H$ and LSPs from *A* to *C* and *B* to *D* through this tunnel: $A \rightarrow E \rightarrow H \rightarrow C$, $B \rightarrow E \rightarrow H \rightarrow D$. Note that only LSR *E* and *H*, the ingress and egress of the LSP tunnel, appear in the LSPs from *A* to *C* and *B* to *D*.

The benefit of label stacking is that we can aggregate multiple LSPs into a single LSP tunnel. For example, thousands of LSPs may go through $E \rightarrow F \rightarrow G \rightarrow H$. Without the hierarchy, all backbone nodes (E , F , G , H) have to be involved in the setup of these LSPs. By creating an LSP tunnel, the information about these LSPs becomes invisible to the interior nodes of the backbone (nodes F and G). The interior nodes of the backbone are not affected by any changes of the LSPs going through the tunnel.

Let us look at processing when a packet travels from A to C via the LSR tunnel $E \rightarrow F \rightarrow G \rightarrow H$. When the packet P travels from A to E , it has a label stack of depth 1. Based on the incoming label, node E determines that the packet must enter the tunnel. Node E first replaces the incoming label with a label that it has agreed on with H , the egress of the tunnel, and then pushes a new label onto the label stack. This level-2 label is used for label switching within the LSP tunnel. Nodes F and G switch the packet using only the level-2 label. When node H receives the packet, it realizes that it is the end of the tunnel. So node H pops up the top-level label and switches the packet with the level-1 label to C .

MPLS also supports a mode called **penultimate hop popping**, where the top-level label may pop up at the penultimate LSR of the LSP rather than the egress of the LSP. Note that in normal operation, the egress of an LSP tunnel must perform two lookups. For example, node H must determine that it is the egress of the tunnel. It then pops the top-level label and switches the packet to node C with the next-level label.

When penultimate hop popping is used, the penultimate node G looks up the top-level label and decides that it is the penultimate node of the tunnel and node H is the egress node of the LSP. The penultimate node then pops up the top-level label and forwards to the egress node H . When H receives the packet, the label used for the LSP tunnel is already gone. Thus node H can simply forward the packet based on the current label to C . Penultimate hop popping can also be used when there is only a single label. In this case the penultimate node removes the label header and sends an unlabeled packet to the egress. To illustrate this, let us examine how node H processes the packet from G . When node H receives the packet from G , the packet has only one label left in the stack. Node H performs a lookup with the label and finds that it is the penultimate node of LSP from A to C and the next hop is C . If node H operates in the penultimate hop-popping mode, it removes the label header and sends the unlabeled packet to node C . In this example there are two labels in the stack, and both are popped out by the penultimate nodes of their respective LSPs.

MPLS supports two types of peering for exchanging stack labels: **ex-**

explicit and **implicit**. Explicit peering is similar to MPLS neighbor peering for [Label Distribution Protocol] LDP; the only difference is that the peering is between remote LSRs. In explicit peering, LDP connections are set up between remote LDP peers, exactly like the local LDP peers. This is most useful when the number of remote LDP peers is small or the number of higher-level label mappings is large.

Implicit peering does not have an LDP connection to a remote LDP peer. Instead the stack labels are piggybacked onto the LDP messages when the lower-level LSP is set up between the implicit-peering LSRs. The intermediate LDP peers of the lower-level LSP propagate the stack labels as attributes of the lower-level labels. This way the ingress nodes of the lower-level LSP receive the stack label from the egress LSR. The advantage of this peering scheme is that it does not require the N-square peering mesh, as in explicit peering especially when the number of remote peers is very large. However, this requires that the intermediate LSR maintain the label stack information even when it is not in use.

1.3.4 Label Stack Encoding

MPLS works over many different link-layer technologies. The exact encoding of an MPLS label stack depends on the type of link-layer technologies. For packet-based technologies such as Packet over SONET¹² (POS) and Ethernet, the MPLS header is inserted between the link-layer and the IP layer and is used for label switching. [...] The MPLS frame consists of the original IP packet and the MPLS header.

For ATM and FR, which are inherently label switching, the top entry of the MPLS label stack is mapped to certain fields in the ATM cell header or FR frame header. Thus label switching is actually performed with the native header of the link-layer protocols. For example, when MPLS is used over ATM, the top-level label may be mapped to the VPI/VCI space in the cell header. The MPLS stack, however, is still carried in the payload. [...] This section discuss the possible label values and their meaning.

Label Stack Header

The MPLS label stack header is also called the MPLS shim header. The top of the label stack appears first in the packet, and the bottom appears last. The network layer packet (e.g., IP packet) follows the last entry of the label stack.

¹²Synchronous Optical Network

The label stack header consists of a sequence of label stack entries. Each entry is 32 bits long and has the format shown in Figure 1.7.

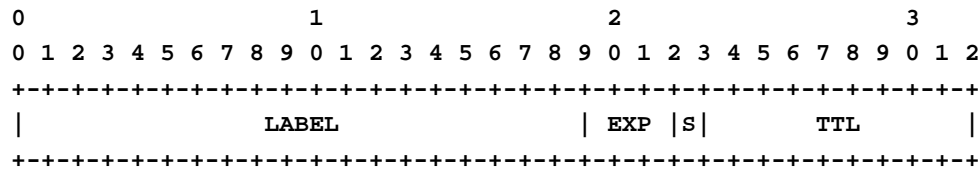


Figure 1.7: Label stack entry format

Each label stack entry has the following fields:

- **Label value.** The label field is 20 bits long. When a labeled packet is received, an LSR uses the label at the top of the stack to find the information associated with the label, including the next hop to which the packet is to be forwarded; the operation to be performed on the label stack, such as label swapping or stack pop-up; and other information such as network layer encapsulation and bandwidth allocation.
- **Experimental use.** The 3-bit field is reserved for experimental use. One possible use is to set drop priorities for packets in a way similar to that in Differentiated Services.
- **Bottom of stack (S).** The 5 bit is used to indicate the bottom of the label stack. The bit is set to 1 for the last entry in the label stack and to 0 for all other entries.
- **Time to live (TTL).** The 8-bit field is used to encode a time-to-live value for detecting loops in LSPs.

Several reserved label values have special meanings:

- **Label value 0.** This value represents the IPv4 Explicit NULL label. This label value is legal only when it is the sole label stack entry. It indicates that the label stack must be popped and the resulting packet should be forwarded based on the IPv4 header.
- **Label value 1.** This value is used to indicate Router Alert, similar to the Router Alert option in IP packets. This Router Alert label can appear anywhere in the stack except at the bottom. When a packet is received with this label on the top of the stack, the packet should be delivered to the local controller for processing. The forwarding of the

packet is determined by the label beneath the Router Alert label. The Router Alert label should be pushed in if the packet is to be forwarded further. The Router Alert label can be used to indicate that a packet contains control information that needs to be processed at each hop by the local control processor.

- **Label value 2.** This value represents the IPv6 Explicit NULL label. This label value is similar to label value 0 except that it is reserved for IPv6.
- **Label value 3.** This label value represents the Implicit NULL label. Label value 3 may be assigned and distributed but should never appear in the label stack. When an LSR would otherwise replace the label at the top of the stack with a new label but the new label is the Implicit NULL label, the LSR will pop the stack instead of doing the replacement.
- **Label values 4 to 15.** These values are reserved.

Determining the Network Layer Protocol

The label stack header does not have a field that explicitly identifies the network layer protocol for processing the packet at the bottom of the label stack. This information should be associated with the label at the bottom of the stack during the label distribution process. Thus when an LSR pops the last label off a packet, it can determine which network layer protocol should be used to process the packet.

With this approach labeled packets from multiple network layer protocols can coexist. Under normal conditions only egress routers pop off the last label and process the packet inside. However, when there are errors, for example, undeliverable packets, it becomes necessary for an intermediate LSR to generate error messages specifically to the network layer protocol. Therefore the information about the network layer protocol should be associated with the entire LSP rather than just the egress node.

1.3.5 Loop Detection

Loops in LSPs can cause severe damage to an MPLS network; traffic in a loop remains in that loop for as long as the LSP exists. IP routing protocols routinely form transient routing loops while routing convergence is taking place. Since MPLS may use IP routing information for setting up LSPs, loops could be formed as a result of IP routing inconsistency. Configuration errors and software bugs may also create loops in LSPs.

In IP routing, the damage from routing loops is mitigated by the use of a TTL field within the packet header. This field decrements by 1 at each forwarding hop. If the value decrements to zero, the packet is discarded. The label stack header also has a TTL field for this purpose. When an IP packet is labeled at the ingress node, the TTL field in the label stack header is set to the TTL value of the IP header. When the last label is popped off the stack, the TTL value of the label stack is copied back to the TTL field of the IP header.

MPLS packets forwarded on ATM labels, however, have no such mechanism since the ATM header does not have a TTL field. The solution to this problem requires loop detection during the setup phase of LSPs. [...] [26]

The reader interested in a more detailed explanation, may be interested in [26].

1.4 Label Distribution Protocol

“The IETF MPLS working group initially considered only one *label distribution protocol* (LDP). LDP was largely based on Tag Switching and ARIS proposals, which were designed to support hop-by-hop routing. The support for explicit routing became critical after it became apparent that traffic engineering is a key application of MPLS. Two different proposals were put forward. One proposal, “*Constraint-based LSP Setup using LDP*”, adds a set of extensions to LDP to support explicit routing.

The other proposal, “*Extensions to RSVP for LSP Tunnels*,” extends RSVP protocols to perform label distribution.

The two competing proposals have caused some heated debates in the MPLS working group, but a consensus could not be reached to pick one of them. In the end the working group decided that both proposals would be standardized. The two protocols are often referred to as CR-LDP (constraint routing label distribution protocol) and RSVP-TE (RSVP with traffic engineering extension). CR-LDP and RSVP-TE can also perform hop-by-hop LSP setup.

In this section we first describe LDP and then compare the similarities of and differences between CR-LDP and RSVP-TE.

1.4.1 LDP

LDP is the first label distribution protocol standardized by the MPLS working group. The protocol is designed to support hop-by-hop routing. Two LSRs that use LDP to exchange label/FEC mapping information are known as LDP peers.

1. Discovery messages for announcing and maintaining the presence of an LSR in a network
2. Session messages for establishing, maintaining, or terminating sessions between LDP peers
3. Advertisement messages for creating, changing, or deleting label mappings for FECs
4. Notification messages for distributing advisory information and error information

Discovery messages allow LSRs to indicate their presence in a network by sending the Hello message periodically. This message is transmitted as a UDP packet to the LDP port at the all-routers-on-this-subnet group multicast address. Once a session is established between two peers, all subsequent messages are exchanged over TCP.

Mapping FEC to LSP

When to request a label or advertise a label mapping to a peer is largely a local decision made by an LSR. In general, the LSR requests a label mapping from a neighboring LSR when it needs one and advertises a label mapping to a neighboring LSR when it wants the neighbor to use a label.

LDP specifies the FEC that is mapped to an LSP. Currently only two types of FECs are defined: address prefix and host address. In order to avoid loops, the following set of rules is used by an ingress LSR to map a particular packet to a particular LSP:

1. If the destination address of the packet matches the host address FEC of at one LSP, the packet is mapped to the LSP. If multiple LSPs have the same matched FECs, the packet may be mapped to any one of these LSPs.
2. If the destination of the packet matches the prefix FEC of one LSP, the packet is mapped to that LSP. If there are multiple matched LSPs, the packet is mapped to the one with the longest prefix match.
3. If a packet must traverse a particular egress router (e.g., from the BGP routing information) and an LSP has an address prefix FEC element that is an address of that router, the packet is mapped to that LSP.

LDP Identifiers

Since each interface of an LSR may use the entire label space, it is important that an LSR identifies each label space within the LSR in any message

exchanges with its peers. The LDP identifier is used to identify an LSR label space. An LDP identifier is 6 bytes long. The first 4 bytes encode an IP address assigned to the LSR, and the last two octets identify a specific label space within the LSR. The last two octets of LDP identifiers for platform-wide label spaces are always set to zero.

LDP Discovery

LDP allows an LSR to automatically detect its LDP peers. There are two mechanisms, one for discovering LSR neighbors that are directly connected and the other for detecting LSR neighbors that are remotely connected. The basic discovery mechanism sends out LDP Link Hello messages on each interface. The messages are sent as UDP packets addressed to the LDP discovery port with the all-routers-on-this-subnet group multicast address. To detect LDP neighbors that are remotely connected, an LSR can send Targeted Hello messages to a specific IP address at the LDP discovery port. An LSR that receives Hello messages may choose to reply with Hello messages if it wants to establish a peer relationship. The exchanges of Hello messages establish the adjacency.

LDP Session Management

The exchange of Hello messages between any two LSRs establishes the communication channel between them and the label space that the LSRs will use in their peer relationship. After that the two LSRs can establish a session for the specified label space by setting up transport connections and starting the initialization process. Initialization includes negotiation of protocol version, label distribution method and timer values. [...] An LSR can accept only initialization messages from LSRs that it has exchanged a Hello message with.

LSRs maintain their peer and session relationship by sending Hello and Keepalive messages periodically to each other. Timers are set when these messages are received. An LSR considers the peer or the session down if the corresponding timer expires before new messages are received.

Label Distribution and Management

LDP supports both **downstream on demand** and **downstream unsolicited** label distribution. Both of these label distribution techniques may be used in the same network at the same time. However, for any given LDP session, only one should be used.

LSPs may be set up independently between all LSRs along the path or in order from egress to ingress. An LSR may support both types of control as a configurable option. When using an independent approach, each LSR may advertise label mappings to its neighbors at any time it desires. Note that in this case an upstream label can be advertised before a downstream label is received. When setting up LSP with the orderly approach, an LSR may send a label mapping only for a FEC for which it has a label mapping for the FEC next hop or for which the LSR is the egress. For each FEC for which the LSR is not the egress and no mapping exists, the LSR must wait until a label from a downstream LSR is received before mapping the FEC and passing corresponding labels to upstream LSRs.

1.4.2 CR-LDP

CR-LDP is a label distribution protocol specifically designed to support traffic engineering. It is largely based on the LDP specification with a set of extensions for carrying explicit routes and resource reservations. The new features introduced in CR-LDP include

- Explicit routing
- Resource reservation and classes
- Route pinning
- Path preemption
- Handling failures
- LSP ID

Setup of Explicit Routes

In CR-LDP an explicit route is also referred to as a constraint-based route or CR-LSP. CR-LDP supports both the strict and loose modes of explicit routes. An explicit route is represented in a Label Request message as a list of nodes or groups of nodes along the explicit route. Each CR-LSP is identified by an LSP ID, a unique identifier within an MPLS network. An LSP ID is composed of the ingress LSR Router ID and a locally unique CR-LSP ID to that LSR. An LSP ID is used when the parameters of an existing LSP need to be modified.

In the strict mode each hop of the explicit route is uniquely identified by an IP address. In the loose mode there is more flexibility in the construction of the route. A loose explicit route may contain some so-called abstract nodes. An abstract node represents a set of nodes. The following

types of abstract nodes are defined in CR-LDP: IPv4 prefix, IPv6 prefix, autonomous system (AS) number, and LSP ID. With an abstract node the exact path within the set of nodes represented by the abstract node is determined locally rather than by the explicit route itself. For example, a loose explicit path may specify a list of AS numbers that the explicit routes must follow. The exact route within each AS is not specified and is decided based on routing information and policies within the AS. This adds a different level of abstraction and allows LSPs to be specified in such a way that the effect of changes in individual links may be isolated within the AS.

The basic flow for an LSP setup with CR-LDP is shown in Figure 1.8. The ingress node, LSR A, initiates the setup of LSP from LSR A to LSR C. LSR A determines that the LSP should follow an explicit route from LSR B. It then sends a Label Request message to LSR B with an explicit route (B, C). LSR B receives the message and forwards it to LSR C after modifying the explicit route. LSR C determines that it is the egress of the LSP. It sends a Label Mapping message backward to LSR B with allocated label 15 for the LSP. LSR B uses the LSP ID in the Label Mapping message to match the original Label Request message. It then sends a Label Mapping message to LSR A with label 12. LSR B also populates the label-switching table with incoming label 12 pointing to outgoing label 15.

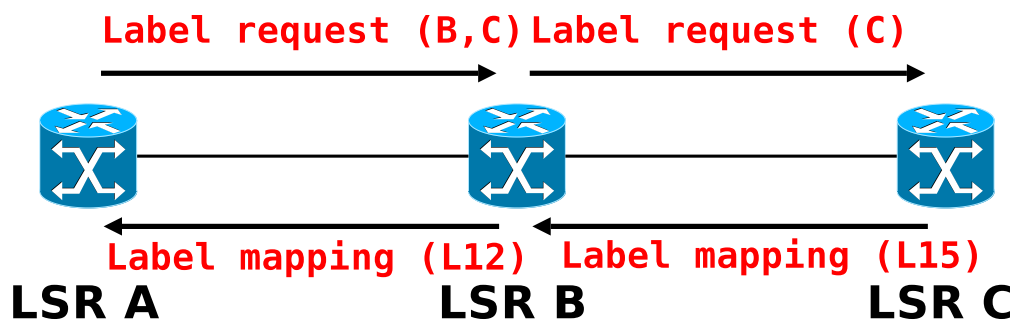


Figure 1.8: CR-LDP LSP setup. [22]

Resource Reservation and Class

CR-LDP allows sources to be reserved for explicit routes. The characteristics of a path can be described in terms of peak data rate (PDR), committed data rate (CDR), peak burst size (PBS), committed burst size (CBS), weight, and service granularity. The peak and committed rates describe the bandwidth constraints of a path, and the service granularity specifies the

granularity at which the data rates are calculated. The weight determines the relative share of excess bandwidth about the committed rate. These parameters are very similar to those used for traffic policing in Differentiated Services. An option also exists to indicate that the resource requirement can be negotiable: an LSR may specify a smaller value for a particular parameter if it cannot be satisfied with existing resources. Network resources can also be classified into resource classes or colors so that NSPs¹³ can specify which class an explicit route must draw resources from.

Path Preemption and Priorities

If an LSP requires a certain resource reservation and sufficient resources are not available, the LSP may preempt existing LSPs. Two parameters are associated with an LSP for this purpose: **setup priority** and **holding priority**. The setup priority and holding priority reflect the preference for adding a new LSP and holding an existing LSP. A new LSP can preempt an existing LSP if the setup priority of the new LSP is higher than the holding priority of the existing LSP. The setup and holding priority values range from 0 to 7, where 0 is the priority assigned to the most important path, or the highest priority.

Path Reoptimization and Route Pinning

For a loose explicit route the exact route within an abstract node is not specified. Thus the segment of the route with an abstract node may adapt when traffic patterns change. CR-LDP can reoptimize an LSP, and an LSP ID can be used to avoid double booking during optimization. Under some circumstances route changes may not be desirable. CR-LDP has a route pinning option. When the route pinning option is used, an LSP cannot change its route once it is set up.

1.4.3 RSVP-TE

As we know, RSVP was initially designed as a protocol for setting up resource reservation in IP networks. The RSVP-TE protocol extends the original RSVP protocol to perform label distribution and support explicit routing. The new features added to the original RSVP include :

- Label distribution
- Explicit routing

¹³Network Service Provider

- Bandwidth reservation for LSPs
- Rerouting of LSPs after failures
- Tracking of the actual route of an LSP
- The concept of nodal abstraction
- Preemption options

RSVP-TE introduces five new objects, defined in this section (Table 1.2).

Table 1.2: New Objects in RSVP-TE

Object name	Applicable RSVP messages
LABEL_REQUEST	PATH
LABEL	RESV
EXPLICIT_ROUTE	PATH
RECORD_ROUTE	PATH, RESV
SESSION_ATTRIBUTE	PATH

LSP Tunnel

Although the original RSVP protocol was designed to set up reserved paths across IP networks, there is an important difference between a reserved path set up by the original RSVP protocol and an LSP. In original RSVP a reserved path is always associated with a particular destination and transport-layer protocol, and the intermediate nodes forward packets based on the IP header. In contrast, with an LSP set up by RSVP-TE, the ingress node of the LSP can determine which packets can be sent over the LSP and the packets are opaque to the intermediate nodes along the LSP. To reflect this difference, an LSP in the RSVP-TE specification is referred to as an LSP tunnel.

Figure 1.9 shows the flow in setting up an explicit route with RSVP-TE. To create an LSP tunnel, the ingress node LSR A first creates a PATH message with a session type LSP-TUNNEL. The PATH message includes a LABEL_REQUEST object, which indicates that a label binding for this path is requested and also provides an indication of the network layer protocol that is to be carried over this path.

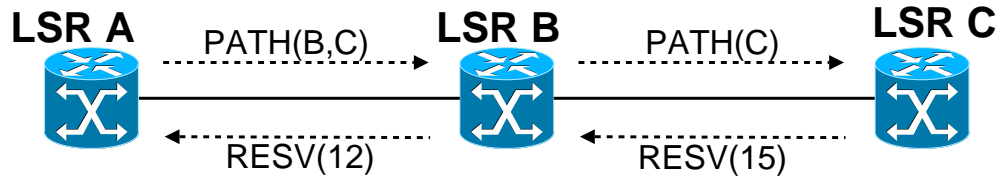


Figure 1.9: RSVP-TE explicit LSP setup. [22]

To set up an explicit route, LSR A needs to specify the route in an `EXPLICIT_ROUTE` object and adds it to the `PATH` message. LSR A may add a `RECORD_ROUTE` object to the `PATH` message so that the actual route is recorded and returns to the sender. The `RECORD_ROUTE` object may also be used to request notification from the network about changes of the actual route or to detect loops in the route.

Additional control information such as preemption, priority, local protection, and diagnostics may be included by adding a `SESSION_ATTRIBUTE` object to the `PATH` message.

Once the `PATH` message is constructed, LSR A sends it to the next hop as indicated by the `EXPLICIT_ROUTE` object. If no `EXPLICIT_ROUTE` object is present, the next hop is provided by the hop-by-hop routing.

Intermediate node LSR B modifies the `EXPLICIT_ROUTE` object and forwards to the egress node LSR C. LSR C allocates a new label, includes it in a `LABEL` object, and inserts into the `RESV` message. LSR C then sends the `RESV` message backward to the sender, following the path state created by the `PATH` message, in reverse order.

When the intermediate node LSR B receives an `RESV` message, it retrieves the label in the `LABEL` object and uses it as the outgoing label for the LSP. It also allocates a new label and places that label in the corresponding `LABEL` object of the `RESV` message, which it sends upstream to the *previous hop* (PHOP). LSR B then adds this new pair of labels to the label switching table. When the `RESV` message propagates upstream to the ingress node LSR A, an LSP is established.

Reservation Styles

For each RSVP session, the egress node has to select a reservation style. The original RSVP protocol has three styles: fixed filter (FF), wild filter (WF), and shared explicit (SE). In RSVP-TE only FF and SE reservation styles are supported.

The FF reservation style creates a distinct reservation for traffic from

each sender that is not shared by other senders. With RSVP-TE this will create a point-to-point LSP for each ingress and egress pair. Most LSPs are expected to be set up using this filter.

The [FF] style allows a receiver to specify explicitly the senders to be included in a reservation. There is a single reservation on a link for all the senders listed. This in essence creates a multi-point-to-point tree to the egress. With RSVP-TE, because each sender is explicitly listed in the RESV message, different labels may be assigned to different senders, thereby creating separate LSPs. The [SE] style is particularly useful for backup LSPs, which are used only if their corresponding active LSPs have failed. Thus these backup LSPs can share bandwidth among themselves and with active LSPs during normal operation.

Rerouting LSP Tunnels

Under many circumstances it may be desirable to reroute existing LSPs. For example, an LSP tunnel may be rerouted in order to optimize the resource utilization in the network or to restore connectivity after network failures.

RSVP-TE uses a technique called 'make before break' to minimize the disruption of traffic flows during such rerouting. To reroute an existing LSP tunnel, a replacement LSP tunnel is first set up, then the traffic switches over, and finally the old LSP tunnel tears down.

During the transition period the old and new LSP tunnels may coexist and so compete with each other for resources on network segments that they have in common. This may lead to a racing condition where the new LSP tunnel cannot be established because the old LSP tunnel has not released resources, yet the old LSP tunnel cannot release the resources before the new LSP tunnel is established. To resolve this problem, it is necessary to make sure that the resource reservation is not counted twice for both the old and new LSP tunnels. This can be achieved in RSVP-TE by using SE reservation style. The basic idea is that the old and new LSP tunnels share resources along links that they have in common.

To make this scheme work, the LSP_TUNNEL object is used to narrow the scope of the RSVP session to the particular tunnel in question. The combination of the tunnel egress IP address, a tunnel ID, and the tunnel ingress IP address is used as a unique identifier for an LSP tunnel. During the reroute operation the tunnel ingress needs to appear as two different senders to the RSVP session. A new LSP ID is used in the SENDER_TEMPLATE and FILTER_SPEC objects for the new LSP tunnel.

The ingress node of the LSP tunnel initiates rerouting by sending a new PATH message using the original SESSION object with a new SEN-

DER_TEMPLATE, a new EXPLICIT_ROUTE object, and a new LSP ID. This new PATH message is treated as a conventional new LSP tunnel setup. However, on links that are common to the old and new LSP tunnels, the SE reservation style ensures that the old and new tunnel share the same reservation. Once the ingress receives an RESV message for the new LSP, it can switch traffic to the new LSP tunnel and tear down the old LSP tunnel.

1.4.4 Comparison

As we mentioned at the beginning of Section 1.4, the fact that we have two competing label distribution protocols was more a result of the compromise by the MPLS working group than a conscious technical decision. It is not clear whether the two protocols will both be supported in the long run or whether one of them will emerge as the winner in the marketplace. Although CR-LDP and RSVP-TE share many similarities (Table 4.2), there are also some key differences. We will discuss these differences and their implications in the rest of this section.

Transport Protocol

CR-LDP and RSVP-TE are based on different transport mechanisms for communicating between peers. CR-LDP uses TCP and UDP, whereas RSVP-TE uses raw IP and requires Router Alert option support. This difference has a number of implications that must be considered in selecting one or the other:

- Although most operating systems support full TCP/IP stack, TCP may not be available in some embedded systems. [Alternatively] on some platforms raw IP and the Router Alert option may not be supported.
- Since raw IP does not provide any reliable transport, RSVP-TE must implement mechanisms for detecting and retranslating lost packets within its own protocol. CR-LDP can assume orderly and reliable delivery of packets provided by TCP.
- CR-LDP may use the standard security mechanisms available to TCP/IP such as IPSEC or TCP MD5 authentication. Because the messages in RSVP-TE are addressed to the egress of the LSP rather than the next-hop intermediate node, RSVP-TE must use its own security mechanisms.
- The need for high availability often necessitates the implementation of redundant network controllers. When the active controller fails,

the backup one can take over and continue the operations. Because RSVP-TE is running over connectionless raw IP and handles packet losses within its protocol, it is easier to implement a smooth failover to the backup system. For TCP, a smooth failover is not impossible, but it is known to be a difficult problem because of the connection-oriented nature of and complex internal state keeping in this system.

Table 1.3: Comparison of CR-LDP and RSVP-TE

Feature	CR-LDP	RSVP-TE
Transport	TCP and UDP	Raw IP
Security	IPSEC	RSVP Authentication
Multipoint to point	✓	✓
LSP merging	✓	✓
LSP state	Hard	Soft
LSP refresh	Not needed	Periodic, hop by hop
Redundancy	Hard	Easy
Rerouting	✓	✓
Explicit Routing	Strict and loose	Strict and loose
Route pinning	✓	By recording path
LSP preemption	Priority based	Priority based
LSP protection	✓	✓
Shared reservation	✗	✓
Traffic control	Forward path	Reverse path
Policy control	Implicit	Explicit
Layer-3 protocol ID	✗	✓

State Keeping

In network protocol design, the issue of soft state versus hard state often causes much debate. With the soft-state approach, each state has an associated time-out value. Once the time-out period expires, the state is automatically deleted. To keep the state alive, it is necessary to refresh it before it expires. In contrast, in a hard-state system, once a state is installed, it remains there until it is explicitly removed. RSVP-TE is based on the soft-state approach. Thus it is necessary for RSVP-TE to periodically refresh the state for each LSP in order to keep it alive. In a large network with a

substantial number of LSPs, the refreshing may pose significant messaging and processing overheads. Because of this, concerns have arisen about the scalability of RSVP-TE to large networks. To address this issue, the IETF has adopted a proposal to add the refresh reduction extensions to the RSVP-TE protocols.

CR-LDP uses the hard-state approach, so it has fewer messaging and CPU overheads compared with RSVP-TE. However, as a hard-state-based system, all error scenarios must be examined and handled properly. Since any state will remain in the system in a hard-state system unless explicitly removed, some LSPs may be left in limbo as a result of unforeseeable errors in the system. In a soft-state system this will not happen because the state for the LSPs is removed after the time-out period expires.

Summary

MPLS uses a technique called label switching. With label switching, packets are forwarded based on a short, fixed-length label. The connection-oriented nature of label switching offers IP-based networks a number of important capabilities that are currently unavailable. MPLS has been used to ease the integration of IP over ATM and simplify packet forwarding, and its support for explicit routing provides a critical mechanism for implementing traffic engineering in Internet backbones.

Before packets can be transmitted in an MPLS network, an LSP must be set up. There are two basic approaches: control driven and data driven. In the control-driven approach the setup of LSPs is initiated by control messages such as routing updates. In the data-driven approach the LSPs are triggered by data packets and set up on the fly while the data packets are arriving. The MPLS standards use the control-driven approach, in which the LSPs are set up by label distribution protocols that are driven by IP routing or explicit routes from the network management systems.

MPLS supports variable forwarding granularity through multiple types of FECs. Edge LSRs have the responsibility for mapping packets onto FECs. MPLS allows multiple labels to be encoded into a packet to form a label stack. An MPLS label stack consists of 32-bit entries, and each entry contains a 20-bit field for the label value. The MPLS label stack header is inserted between the IP packet and the link-layer header. In ATM and FR the top-level label is mapped to fields in the ATM cell header or FR header.

Three label distribution protocols, LDP, CR-LDP, and RSVP-TE, have been standardized. LDP is primarily used for supporting hop-by-hop” [26]

VIRTUALS TESTBEDS

This chapter intends to outline the concepts of a virtual testbed. At the beginning, vocabulary and theoretical concepts are explained without getting into detailed technical issue or code example. Then, a concrete implementation of a virtual machine is presented (Netkit) and used to set up MPLS networks. Finally, the virtual testbed upon which dynamic configuration of LSP will be analyzed in the following chapters, is described at the end of this chapter.

2.1 Introduction

2.1.1 Definitions

In order to explain the basics of a virtual testbed, let us start by defining a few concepts:

“A **testbed** is a platform for experimentation. Testbeds allow to test in a transparent and replicable way, software and/or specific network cases. The platform is usually composed by few interconnected computers running the same operating system and the same programs. These computers are used to simulate a specific behavior.” [27]

A **virtual testbed** is a software that emulate the behavior of a real testbed. It runs on a single computer, called the host, and the instance of the running software are called the virtual machines.

“A **sandbox** is a security mechanism for safely running programs. It is often used to execute untested code, or programs from unverified third-parties, suppliers and untrusted users.” [27]

These definitions are not rigorously complete. The reader must regard them as an explanation of a concept in order to understand the context in which they are used for. For instance, a virtual testbed is used to emulate a real testbed. A sandbox is used to

simulate the execution of a program as if it were on the real system.

At the end of this chapter, the real testbed upon which tests were done will be presented. This testbed is *emulated* with a tool described in Section 2.2, hence enabling the investigation of dynamic configuration of LSPs in Chapter 3 and 4

2.2 Netkit

The beginning of this section comes from the article describing Netkit [18]. This article is cited several times in this chapter because it is really resourceful and explain with the most rigorous way the features and tools used in Netkit.

“An emulator is a software or hardware environment that is capable of closely reproducing the functionalities of a real world system. Emulators, especially if implemented in software, are very useful for performing experiments that might compromise the operation of the target system or simply when the real system itself is not available. This is true in particular for computer networks, where configurations of network devices often need to be tested before being deployed. In [Section 2.2], the network emulation systems will be described as well as Netkit, a lightweight emulator based on User-Mode Linux. Besides being an effective instrument to support teaching of computer networks, Netkit has also proven itself to be helpful in testing the configuration of large scale real world networks. Netkit provides tools for a straightforward setup of complex network scenarios that can be easily distributed and comes with a set of ready to use experiences that permit to immediately experiment with specific case studies. Netkit also fully installs and runs in user space and provides users with a familiar environment consisting of a Debian based Linux box and well known routing software and networking tools.

2.2.1 Introduction

Testing configurations is a common need both for network administrators and for computer scientists interested in networking. The former can take advantage of a testing phase for checking that a particular configuration works as expected before deploying it, while the latter can exploit test results in order to validate theoretical models with practical experimentation. Ideally, testing should take place under the very same conditions in which the configuration is to be eventually deployed. However, this often means injecting artificially generated, potentially harmful traffic into a live

network, which may cause damage to it. An effective alternative to live testing consists in implementing the network configuration of interest inside a safe, isolated software environment which closely reproduces the real target setting. Such environments are usually available in two flavours:

- **Simulation** environments allow the user to predict the outcome of running a set of network devices on a complex network by using an internal model that is specific to the simulator. With the network as an input and the outcome (possibly a network state) as an output, simulators do not necessarily reproduce the same sequence of events that would take place in the real system, but rather apply an internal set of transformation routines that brings the network to a final state that is as close as possible to the one the real system would evolve to. As this approach can be optimized in performance, the simulated network can typically scale well in size. The drawback is that the simulated devices may have limited functionalities and their behaviour may not closely resemble that of real world devices.
- **Emulation** environments aim at closely reproducing the features and behaviour of real world devices. For this reason, they often consist of a software or hardware platform that allows to run the same pieces of software that would be used on real devices. Differently from simulation systems, in an emulator the network being tested undergoes the same packet exchanges and state changes that would occur in real world. The real advantage of the emulation approach comes out when the emulator itself is a software piece, as this allows much higher flexibility in carrying out network tests.

Since emulation makes use of real routing software, every aspect of the network can be influenced and monitored like it could be in a real network. While this ensures very high accuracy, the computational resources needed to run an emulated device are typically higher than those available in the device itself. Hence, the performance of an emulated device is, in general, lower than that of the real one, and this often poses limits on the scalability of the size of the emulated network.”¹ [18]

To make a clear distinction between an emulated device and the real machine it is running on, in the following we label Netkit virtual machines and the software they run as *guest*, and we refer to the real machine and the software it runs as *host*.

¹More information about Netkit and emulation environment can be found in [18]

2.2.2 Usability and Readiness

This section describes how to set up and use a virtual testbed with Netkit. Within the Netkit environment each network device is implemented by a virtual machine, and interconnection links are emulated by using virtual collision domains which can be seen as **virtual hubs**.

In other words, each virtual machine that is connected to a collision domain will receive all the packets sent on that collision domain. This note is important because it explains a part of the results in this chapter. For example, on Figure 2.5 page 48, the network domain 192.168.80.0/24 and 192.168.50.0/24 are two collision domains.

Each virtual machine can be configured to have an arbitrary number of virtual network interfaces. Virtual machines can also be configured to have no interface at all, in which case they can serve as standalone emulated hosts. However, this is not the application Netkit has been thought for. Netkit is not a host emulator: it is a network emulator.

Netkit is really a great tool when it comes to save time while building a virtual testbed. Three things are needed to make Netkit works :

1. Scripts developed to configure, launch and halt virtual machines;
2. A filesystem image;
3. A Linux kernel compiled for the User-Mode Linux architecture.

Scripts

Netkit virtual machines are based on the User-Mode Linux (UML) kernel. Starting a virtual machine means starting a UML instance, which often requires dealing with somewhat complex command line arguments. For this reason Netkit supports straight-forward configuration and management of virtual machines by means of an intuitive interface consisting of several scripts.

Virtual machines are UML instances that directly run on the host kernel and are managed by a set of commands. Netkit provides two alternative interfaces to start and configure virtual machines.

1. A set of v-prefixed commands: vclean, vconfig, vcrash, vhalt, vlist, vstart, which allow to start and manage single virtual machines while providing fine grained control on their configuration;
2. a set of l-prefixed commands (lclean, lcrash, lhalt, linfo, lrestart, lstart, ltest), which ease setting up preconfigured network laboratories consisting of several virtual machines.

Before going any further, the attention of the reader should be drawn to the fact that there exists two different ways to preconfigure a lab or a testbed. On the one hand, the l-prefixed commands are used, and on the other hand a script can be generated by providing an XML Schema of the testbed to the NetML parser².

XML language can be used to describe networks and hence be used to specify scenarios. The Netkit group also developed NetML, a parser that takes into input the XML schema of the network and returns a script allowing to start the emulated network thanks to Netkit. As both tools were developed by the same group, there is strong integration between them. NetML also contains a set of tools which transform vendor independent description of a network into configuration statements for specific routing software (Cisco, Juniper, Zebra).

Network descriptions are written according to the NetML grammar (i.e., an XML-Schema instance). XSLT (eXtensible Stylesheet Language Transformations) is then used to generate the configuration files for different types of routers and firewalls.

With huge and complex networks, it is better to use enterprise-oriented methods to describe networks as NDL.³ NDL (Network Description Language) is a language based on the Resource Description Framework⁴ from W3 and a bunch of tools making network descriptions easier. As a matter of fact, NetML is efficient for describing from the tiniest to average size networks but for large networks NDL is more appropriate. For instance, NDL takes into account the geographical location of network nodes. These methods are way too strong for the virtual testbed that is going to be used. NetML is enough to suit our requirements.

In this case, only the v-prefixed commands are used in the script generated by NetML. The reason is that the l-prefixed commands are well suited when the goal is to set up complex networks but are difficult to adapt to specific needs. For instance, the script generated by NetML was modified in order to set up LSPs automatically. This would be very difficult and time consuming to implement with the l-prefixed commands. Writing the script that launches and configures all the virtuals machines was the most time consuming part anyway.

Filesystem

Let us move to a description of the Netkit filesystem. It will appear that it is not suited to our needs, which motivated a few changes. Then, the answer to the question “why is

²<http://www.dia.uniroma3.it/compunet/netml>

³<http://www.science.uva.nl/research/sne/ndl>

⁴<http://www.w3.org/RDF>

it useful to modify the Netkit filesystem?” will be exposed.

“Each virtual machine in Netkit has its own filesystem. Hence, if [the user creates, alters or deletes] a file inside a virtual machine, the change will only involve that machine. File systems are preserved across reboots of the virtual machines, so that [the user can save his/her] configuration data and retrieve[s] it when the machine is later restarted. When a virtual machine is started for the first time its filesystem is not empty, but it contains boot scripts, basic configuration settings and several tools and utilities. That is, file systems are first built on the basis of a common “model”.

A virtual machine filesystem is nothing but a special file on the host machine. However, maintaining a full copy of the filesystem for each virtual machine would be too space consuming. [For instance,] a virtual machine filesystem is hundreds of megabytes large. For this reason, there is only a single “master” filesystem, which is called model (or backing) filesystem. [...]

[Upon first boot, every virtual machine sees the contents of the model filesystem.] When a virtual machine writes [data] to its own filesystem, a special file called COW file is created. This file contains the differences between the model filesystem and the current filesystem status. The strategy of “just saving the changes” is also called **“Copy On Write”** strategy (COW) [...]. That is, whenever data has to be written to the model filesystem, a copy of the affected portion is made and the changes are saved to an external file (the COW file). This saves a great amount of space. COW files are automatically created when a virtual machine is first started. [...]

COW files and model files cannot be mixed. That is, if a COW file for a virtual machine has been created on the basis of a certain model filesystem, then every virtual machine using that COW file must use the same model filesystem as well. In other words, COW files and model files cannot be arbitrarily coupled. [If the model filesystem is changed, the COW files are useless.]”[17]

In order to work properly, virtual machines require a filesystem containing, at least, a Linux installation. With Netkit, a full-fledged Debian distribution tuned according to specific UML use is actually installed on the filesystem. Other relevant softwares are installed such as routing daemons, servers, firewall, diagnostic tools, among other kinds of programs. When a virtual machine system is configured, it can run as a real network device (e.g. router).

A virtual machine can act as a network device if the guest operating system (i.e. Linux) is configured accordingly. The Netkit filesystem is preconfigured and works perfectly. However:

1. It is a bit old, and starts to become **obsolete**. The versions of the installed software are sometimes years behind the most up-to-date version of that very same program.
2. If the host system is not the same as the one installed on the virtual machine, i.e. the libraries installed are not the same, it might be worth to have a **C compiler** and the corresponding libraries installed in order to update the system. But the Netkit filesystem does not contain a C compiler. It must therefore be installed which might require more space than available on the genuine Netkit filesystem.
3. Compiling programs is one thing, installing them is another one. In order to compile and install programs, few **tools** such as make, autoconf, are needed. These programs must be installed too. Fortunately, Netkit is debian-based and these packages can be installed as binary packages (which do not require a compiler).

The modernity (up-to-date) is not the only matter. The addition of packages requires much more space than available. In order to enlarge the Netkit filesystem, a complete method was provided by the author of this thesis on the Netkit developing community⁵. The complete method is described in the Appendix B. It should soon appear in the man page of the next Netkit release.

After enlarging the filesystem, updating the packages, installing the C compiler, the tools and the required libraries to compile the programs from mpls-linux, the necessary programs to use MPLS and set up LSPs must be compiled and installed. These programs and the way to use them will be discussed in Chapter 3 and Chapter 4.

Finally, the most difficult and complex task was to build a filesystem that include all the necessary tools to activate the MPLS features (i.e. iproute2, iptables and Quagga).

UML Kernel

Virtual machines make use of a variant of the standard Linux kernel which is compiled to be run as a userspace process. This variant is known as User Mode Linux kernel. More information about UML can be found at the User-Mode Linux Kernel Page⁶.

⁵<http://list.dia.uniroma3.it/mailman/private/netkit.users/2006-December/000220.html>

⁶<http://user-mode-linux.sourceforge.net>

“In an emulation environment virtual machines have nearly the same characteristics of a real host, including their own kernel. Netkit exploits User-Mode Linux as kernel for the virtual machines. User-Mode Linux is widely used by kernel hackers, who are doing filesystem and memory management development and debugging, as well as by hardware developers, who are prototyping new types of device in software. It also meets the interests of the security community, as it fits well the creation of jails and honeypots, and is often employed by the hosting industry to run virtual servers. The fundamentals of UML are illustrated by its designer Jeff Dike in his book [9].

User-Mode Linux is a port of the standard Linux kernel which is designed to run as a userspace process. Being a kernel in itself, UML comes with its own kernel subsystems, including scheduler, memory manager, filesystem, network, and devices. In this sense an instance of UML provides a virtualized environment in which everything (processes, memory, filesystem, etc.) is controlled by itself instead of the host kernel.

In practice, UML appears as a userspace process on the hosting machine and acts as a kernel for its own processes. Actually, a kernel for a Netkit virtual machine is nothing but a special version of a kernel that has been compiled as a User-Mode Linux kernel. An UML kernel takes care of mapping system calls generated inside the virtual machine to the proper functions of the real host kernel.” [18]

Every virtual machine can be started with its own UML kernel. Figure 2.1 shows the interaction between User-Mode Linux and the other components of the host system. In principle, nothing forces to use particular combinations of kernels and file systems. The current version of Netkit also provides support for loadable kernel modules. Modules are kernel components that can be attached or removed on the fly from a running kernel. This is very useful because all MPLS features from the mpls-linux project can be compiled as a module. This is also a cause of many mistakes, as the tester sometimes forgets to load the MPLS module.

It is also possible to build a customized kernel for the virtual machines. It is not needed to have the genuine Netkit kernel which starts to get old. Furthermore, the Netkit kernel is not MPLS compliant, as it does not contain the necessary modules to create and use LSPs. The kernel from Netkit version 2.2 includes a 2.6.11.7 (UML) kernel.

By the time of writing this thesis, the latest stable kernel version is 2.6.21.1. Note that

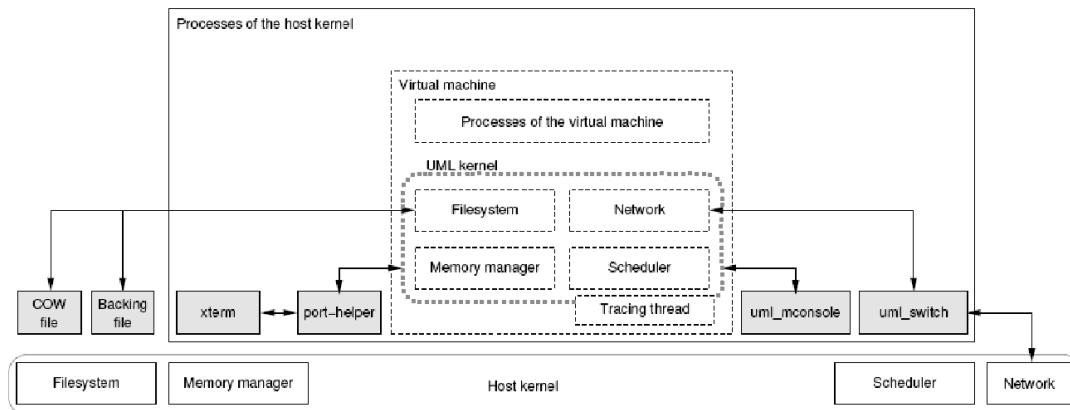


Figure 2.1: Relationships between User-Mode Linux and the other components of the host system. Boxes represent kernel entities (subsystems, interfaces, or processes). Dashed boxes represent virtualized resources, while gray filled ones [named UML kernel] are instantiations of kernel entities (processes or files).[18]

UML is included in the standard Linux kernel but not necessarily evolves at the same cycles.

Figure 2.2 shows the representation of an “UML-MPLS linux” kernel with the following legend.

— . . . — . . . —	Symbolic representation of the Linux kernel.
— —	Section of the Linux kernel.
—————	Features of the Linux kernel.

Besides the MPLS modules, the rest comes from a vanilla kernel⁷. Note that some drivers might be loaded as a module but this will not be debated here. Concerning MPLS-patched kernel, it can be turned into a UML kernel⁸ and its version is 2.6.20. Compiling a UML kernel is much more easier than compiling a Linux kernel for a real computer. With the UML kernel, there is no need to specify the device drivers or to know which chipset the motherboard is using. It is just needed to specify few options for UML, for MPLS and the network tools (iptables, iproute).

⁷These kernels are released by Linus Torvalds. See <http://www.kernel.org>

⁸See this url for more information about compiling UML:
<http://user-mode-linux.sourceforge.net/UserModeLinux-HOWTO-2.html>

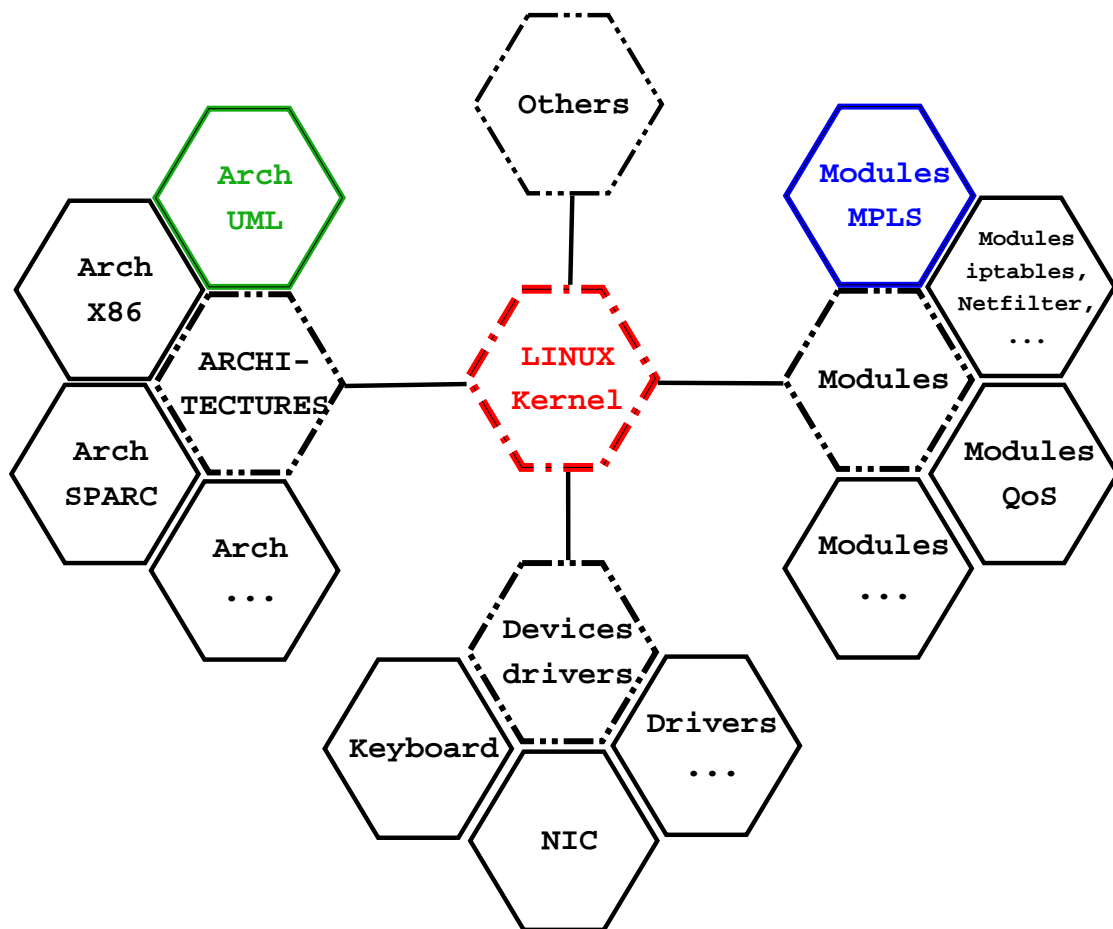


Figure 2.2: UML-MPLS Linux Kernel.

2.2.3 The Zebra Routing Software Suite

This section is devoted to the Zebra routing software.

“In order to experiment with routing protocols, Netkit comes with an installed release of the Zebra routing software. Zebra is a suite of daemons that provide support for several routing protocols, including RIP, OSPF, and BGP. Routing protocols take care of spreading information about available destinations on a network in order to automatically update the routing tables of each device.

[Figure 2.3] describes an abstraction of the architecture of Zebra⁹ and of the way in which it injects information in the kernel routing table. Each

⁹<http://www.zebra.org>

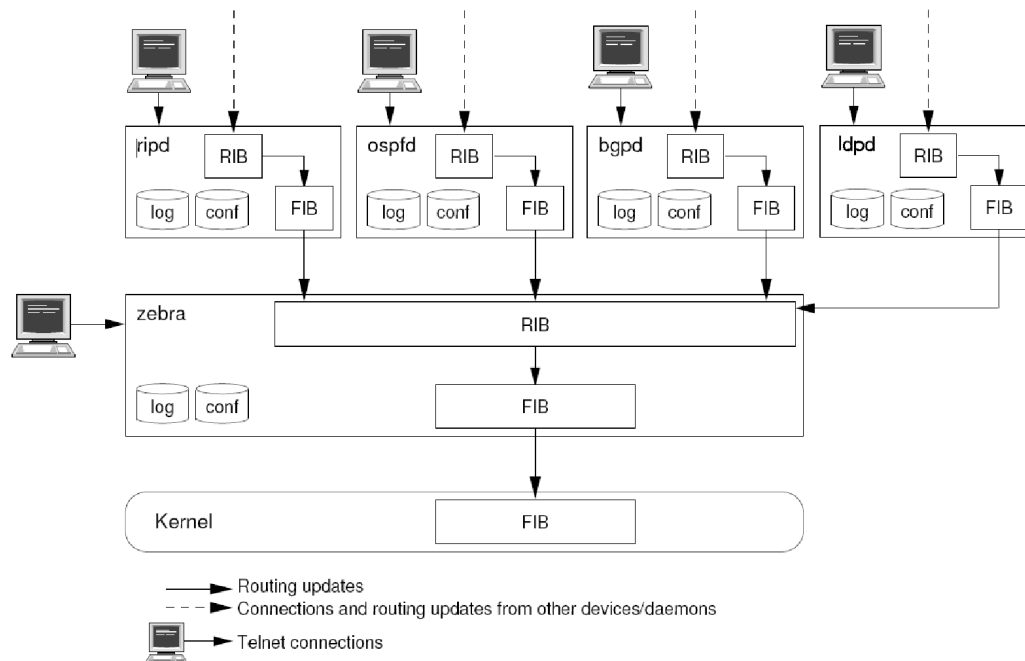


Figure 2.3: An abstraction of the architecture of the Zebra routing software. This image is a modified version of the original one from [18] to include the `ldpd` daemon.

Zebra routing daemon manages a specific routing protocol, has its own configuration file, and writes to its own log. They listen on different TCP ports, so that messages of a particular routing protocol can be sent to the appropriate daemon.

For each routing protocol, a Routing Information Base (`RIB`) and a Forwarding Information Base (`FIB`) are maintained. The `RIB` is the set of all destinations known to that protocol, together with the path to reach them and some additional reachability information. The `FIB` contains, for each possible destination on the network, only the alternative that is considered the best one to reach it. Zebra in itself is a routing daemon: it receives information from the `FIBs` of the other daemons and, for each destination, selects the best alternative among those made available by different routing protocols.

Zebra's best routes are finally injected into the routing table of the kernel, which is used to actually forward packets. All the routing daemons, including Zebra, can be contacted via telnet on a dedicated TCP port to check the status of routing protocols and perform "on the fly" configuration.

The daemons provide a Command Line Interface (`CLI`) which closely

resembles that of Cisco routers. Most of the commands available on real devices can be used, but each daemon only provides those commands that are specifically oriented to the routing protocol it manages.

For example, `ripd` does not provide the `show ip bgp` command, and `bgpd` must be contacted in order to be able to issue it. However, the Zebra suite also comes with `vttysh`, an integrated shell that provides a unique interface to all the daemons. Unfortunately, Zebra development is somewhat slow. For this reason, and also in order to create a community that does not rely on a centralized model, the Quagga project has been started.

Quagga¹⁰ is essentially a fork of Zebra in which proposals from a community of users are usually discussed and more quickly acknowledged. As a result, Quagga provides bug fixes and functionalities that are missing in Zebra, sometimes at the expense of stability (both stable and unstable releases of Quagga are available).

At present, Netkit does not provide the Quagga routing suite. However, it can be easily installed in case. It is needed, and there are plans to include it in future releases” [18]

Since `ldpd`, the implementation of LDP protocol as daemon discussed in the next chapter, requires Quagga, Zebra must be uninstalled to avoid unexpected conflicts. Quagga is described in detail in Chapter 3.

2.2.4 Conclusion on Netkit

Netkit is definitely a great tool, very customizable, a lightweight emulator and very easy to install. Its developer community is still small but the support provided by the Netkit team is really outstanding. Furthermore, this tool takes more time to download than to install and configure. Everything is simple and well documented. The man pages are really helpful and the usability is very good.

2.3 AROMA testbed

2.3.1 Description

AROMA stands for **A**dvanced **R**esource Management Solutions for Future All IP Heterogeneous **M**obile **R**adio Environments. It is the name of an European project, let us

¹⁰<http://www.quagga.net>

see what is it all about:

“The objective of the AROMA project is to devise and assess a set of specific resource management strategies and algorithms for both the access and core network part that guarantee the end-to-end QoS in the context of an all-IP heterogeneous network. [...]

AROMA project aims not only to assess and maximize the potential benefits coming from the medium-term evolution of the considered radio-access technologies (e.g. HSDPA/HSUPA; MBMS) but in parallel also to promote and investigate potential benefits coming from a long-term evolution towards an all IP heterogeneous mobile and wireless network architecture. In that context, the RAN [(Radio Access Network)] architecture should also evolve to accommodate future IP-based networks, which allow a common transport even in different access networks, simple resource management, and easy heterogeneous inter-working.

On the other hand, in order to support end-to-end QoS in a heterogeneous wired and wireless mobile environment, an appropriate interaction between the QoS management entities of the core network (CN) and the Common Radio Resource Management (CRRM) in the radio part is crucial. These kinds of issues are extensively covered in the project.

Last but not least, it is also prime important to carry out economic evaluation on the impacts of the novel architecture solutions considered by the project.

In summary AROMA aims at providing tangible contributions, in terms of resource management, for the future all IP heterogeneous wireless systems, which will take into account 2G/2.5/3G (e.g. GERAN, UTRAN) and 3.5G networks (e.g. HSDPA), including the newly emerging RAN technologies (e.g. WLAN , WIMAX) and services, for the 2010-2015 time frame.

In order to accomplish these objectives, the project evolves around two main activities:

1. Algorithmic development and simulation by means of advanced simulation tools,
2. Demonstration of the technology by means of implementing real-time testbeds for proof of concepts.

[...] Results obtained in AROMA are expected to be of significant momentum, the beneficiaries to which are service-providers, operators, manufacturers and end-users.”[24]

2.3.2 The Real AROMA testbed

The testbed built for the AROMA project is located at the **Universitat Politècnica de Catalunya** (Barcelona, Spain), at the department of Signal Theory and Communication. During five months on his traineeship, the author of this thesis was in charge to make few changes on the testbed in order to fulfill the AROMA requirements. The Appendix A shows the certificate thereof.

The virtual tested studied in Chapters 3 and 4 is based on the real AROMA testbed. Figure 2.5 shows the virtual testbed AROMA. The real (physical) AROMA contains more computers such as a computer dedicated to generate traffic (Traffic Generator) according to stochastic models, User Equipment, and so forth.

At the beginning, the testbed name was EVEREST. Further studies showed that this testbed was missing three core routers to provide accurate and multicase tests. The EVEREST testbed with the three new core routers is now called AROMA. Three core routers were needed for special case testing purpose. Currently, AROMA can reproduce the Fish problem.

Figure 2.4 illustrates the well-known Traffic Engineering Fish problem. The AROMA testbed has the same configuration besides collision domains¹¹, see Figure 2.5 and Figure 2.4.

In order to be rigorous, the virtual AROMA testbed was designed with the exact same interface names and IP addresses than the real ones. By definition of a collision domain, each virtual machine that is connected to a given collision domain will receive all the packets sent on that domain. Hence, the behavior is not exactly the same since the real AROMA testbed has all its nodes connected by switches whereas the virtual AROMA testbed uses collision domains (or virtual hubs). This impacts the behavior of the subnetworks 192.168.80.0/24 and 192.168.50.0/24. The result is obvious when all virtual machines have a routing daemon running up. All virtual machines are reachable from every other virtual machine, thus, a ping from SOURCE to DESTINATION through these two collision domains will be double acknowledged¹². The packets are

¹¹See Section 2.2.2, page 36, for more information about collision domains.

¹²Not clear whether duplicates occurred on the way to DESTINATION when hopping from CR1 to CR2/CR3 or on the way back between ER and CR2/CR4.

duplicated and thanks to the routing table, every virtual machine is able to route them across the network.

```

PING 192.168.70.2 (192.168.70.2) 56(84) bytes of data.
64 bytes from 192.168.70.2: icmp_seq=1 ttl=60 time=4.07 ms
64 bytes from 192.168.70.2: icmp_seq=1 ttl=60 time=4.10 ms (DUP!)
64 bytes from 192.168.70.2: icmp_seq=2 ttl=60 time=2.12 ms
64 bytes from 192.168.70.2: icmp_seq=2 ttl=60 time=2.95 ms (DUP!)
64 bytes from 192.168.70.2: icmp_seq=3 ttl=60 time=1.99 ms
64 bytes from 192.168.70.2: icmp_seq=3 ttl=60 time=3.61 ms (DUP!)
64 bytes from 192.168.70.2: icmp_seq=4 ttl=60 time=1.99 ms
64 bytes from 192.168.70.2: icmp_seq=4 ttl=60 time=3.69 ms (DUP!)

```

However, there are two ways to avoid these packet replicas :

1. First, by creating more interface in order to connect the nodes by a direct-single network interface.
2. Second, by installing a virtual machine acting like a switch between the virtual machines.

This is a real dilemma because the first solution can be easily done but the virtual testbed can no longer be called AROMA since the configuration changed. The second solution is harder to set up and requires more resources. The packet processing would take more time and the results might be affected.

But how these replicas could impact the results anyway? As this scenario occurs only with an IP network, a MPLS network will not have the packet replicas problem. The path and the labels are globally defined at the contrary of an IP network which take local routing decision with the hop-by-hop routing.

The Fish problem reveals why hop-by-hop routing (current IP routing) is not accurate enough for the traffic engineering. Obviously, all the traffic from $R1$ to $R5$ would automatically be routed through $R2 \rightarrow R3 \rightarrow R4$ because it is the shortest path.

This might cause extremely unbalanced traffic distribution. For instance, what about the routers $R6$ and $R7$ being unused while $R3$ is a bottleneck?

The reasons are [22] :

Destination based forwarding: For all the data streams having the same destination, eventually converge to the same path. All the packets containing the same destination IP, arriving at the router, will have the same next hop which may flood the latter.

Local optimization of routes: Every router make a local decision, according to the network information it receives it has to select (compute) the best path.

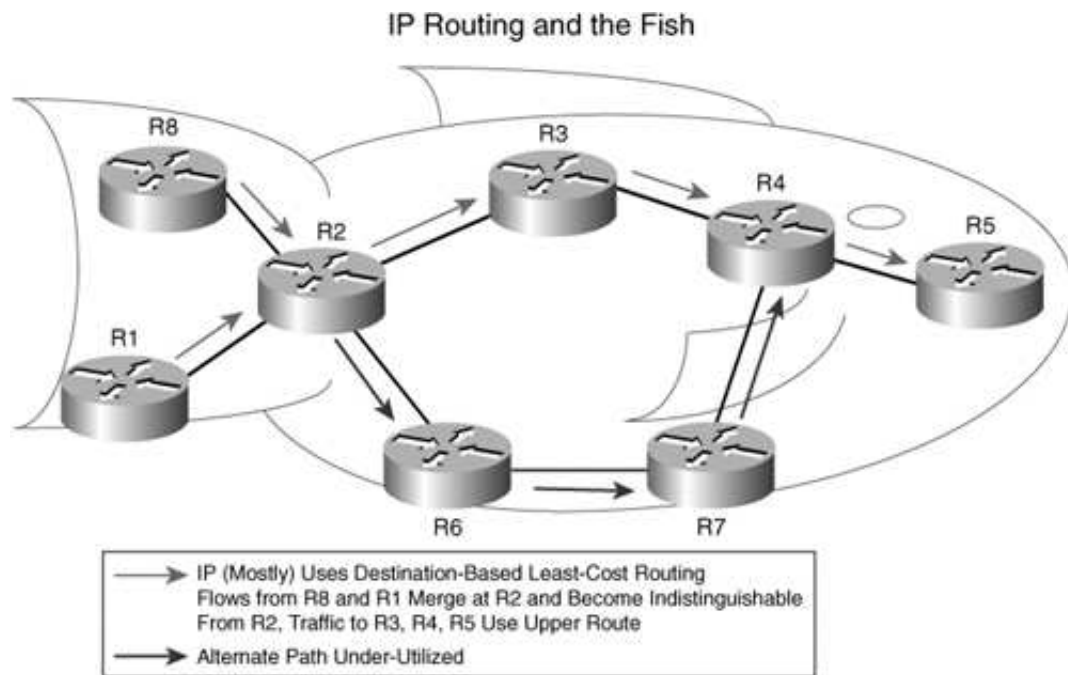


Figure 2.4: Traffic Engineering: The Fish Problem. [21]

Hence, in order to have control over packet routing, MPLS (mpls-linux) seems to be appropriate since it is able to set up explicit routes. Chapter 4 covers this part.

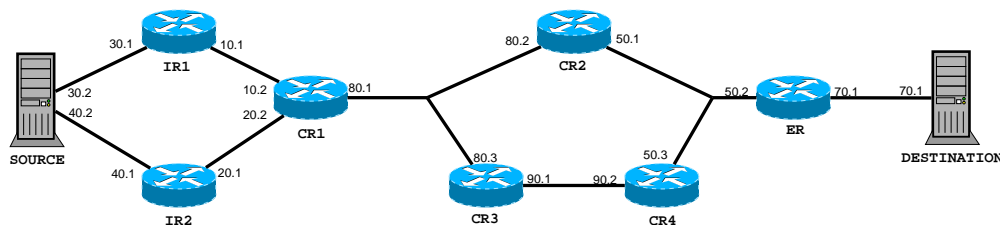


Figure 2.5: The Virtual AROMA Testbed.

2.3.3 Deployment Study Case

During the deployment of AROMA and its the three new core routers, a few problems arose. They are all common problems in relation to a IT management project. They will be explained in chronological order. This was definitely a real-scale study case of project management which enriched the experience of the author.

SRC:	eth0	192.168.30.2
	eth1	192.168.40.2
IR1:	eth0	192.168.10.1
	eth1	192.168.30.1
IR2:	eth0	192.168.20.1
	eth1	192.168.40.1
CR1:	eth0	192.168.10.2
	eth1	192.168.20.2
	eth2	192.168.80.1
CR2:	eth1	192.168.50.1
	eth2	192.168.80.2
CR3:	eth1	192.168.90.1
	eth2	192.168.80.3
CR4:	eth1	192.168.90.2
	eth2	192.168.50.3
ER:	eth0	192.168.50.2
	eth1	192.168.70.1
DST:	eth0	192.168.70.2

Table 2.1: IP configuration of the virtual AROMA testbed.

First of all, the three new nodes arrived more than a month late. Once arrived, they were so slim and long that the rack into which they were supposed to be placed was too short. There were no other way than waiting for another adapted rack before installing the new nodes on the testbed. When the new rack arrived a week later, they needed to be installed and configured according to the AROMA requirements. At the same time, each node needed three network interface cards, but they only contained two, which means that the testbed could not be deployed without a third card on each node. As the three nodes had special dimensions, a special network card had to be ordered.

Meanwhile the three nodes were installed but still not configured because of the missing network card. While waiting for the missing network card, the three nodes were running but not connected to the other nodes from EVEREST.¹³ Once turned on, the power supply fans from the new computers were so noisy that those new machines were switched with other computers from the department. They were less powerful but they made a tolerable ambient noise. Nobody could have worked in the same room that these computers. Most of these problems - insufficient number of network cards, too large computers - are related to a discrepancy between AROMA requirements and specification of the supplied hardware. Better communication should have been considered at the first place.

¹³The previous AROMA testbed.

Then the three new “switched” computers were installed and configured. The network configuration was a very time-consuming task because the physical network interfaces and their name in Linux (eth0, eth1, eth2) were mixed up. In other words, the only way to know how a given network interface card matched a given name under Linux was to ping every single interface and to see which (physical) interface was blinking on the back of the computer.

Once all the interfaces matched their name under Linux, they had to be connected to the Cisco switch and to the other computers. Unfortunately, a problem occurred in the documentation of the Cisco switch configuration **and / or** the target configuration of the AROMA testbed which led to a **documentation problem**. The error was difficult to find out because the interface name of the new nodes did not match the names on the target documentation. Moreover, the route configuration of the older nodes had to be altered in order to take into account the new nodes. So, the whole testbed network configuration was checked and correctly set up. A good tip is to glue a piece of paper showing the name of the network interface on the back of the computer, on the cable, and have an accurate documentation of switches and routers.

Another problem which was **unpredictable** is a power supply breakdown of the testbed’s firewall — on computer running Linux — after a total power cut of the building. Without the firewall, the access to the testbed was impossible. The only way to have a power supply within the hour was to take off the power supply from another computer and substitute it to the broken one. In this case, improvising is the key concept. Security is a domain where it is difficult and highly dangerous to improvise. Fortunately, the whole network of the university was protected by another “main” firewall. To get access to the whole university network, a login/password, a certificate and a key is needed to be granted by the server and the firewall. The network of the testbed had even more restricted access. Only few people including me had access to it.

Besides the few problems aforementioned, taking part to the deployment of the AROMA testbed was a very interesting experience and it shows how hard it is to spread the good information at the right time. To facilitate the work of researchers by writing scripts and help them to solve their problems is really rewarding. Infrastructure management is a lot of improvising but in the end, everything was up and running and ready to use for MPLS.

LABEL DISTRIBUTION ON AROMA

LSPs might be set up either automatically, with the help of LDP, or manually. This chapter is dedicated to the LDP way whereas Chapter 4 will deal with the manual configuration of LSPs at every hop. Both ways to configure a LSP have advantages and drawbacks. They will be discussed in this chapter and the following one.

The easiest way to configure a LSP is to specify at the edge router that a path will be drawn to a given destination. This is an IP-prefix based FEC, as described in Section 1.3.2. In the present chapter, a validation of the LDP implementation `ldp_portable` will be presented for this scenario. The performance gap between nowadays IPv4 networks and MPLS networks will also be illustrated with a benchmark.

3.1 Introduction

As previously seen in Section 2.3, the AROMA testbed allows 4 different routes from SOURCE to DESTINATION to be used :

1. SOURCE \rightarrow *IR 1* \rightarrow *CR1* \rightarrow *CR 2* \rightarrow *ER* \rightarrow DESTINATION
2. SOURCE \rightarrow *IR 2* \rightarrow *CR1* \rightarrow *CR 2* \rightarrow *ER* \rightarrow DESTINATION
3. SOURCE \rightarrow *IR 1* \rightarrow *CR1* \rightarrow *CR 3* \rightarrow *CR4* \rightarrow *ER* \rightarrow DESTINATION
4. SOURCE \rightarrow *IR 2* \rightarrow *CR1* \rightarrow *CR 3* \rightarrow *CR4* \rightarrow *ER* \rightarrow DESTINATION

The graphic representation is shown on Figure 3.1. Only the routes going through *IR1* will be used in this chapter, there is no good reason to go through *IR2* right now.

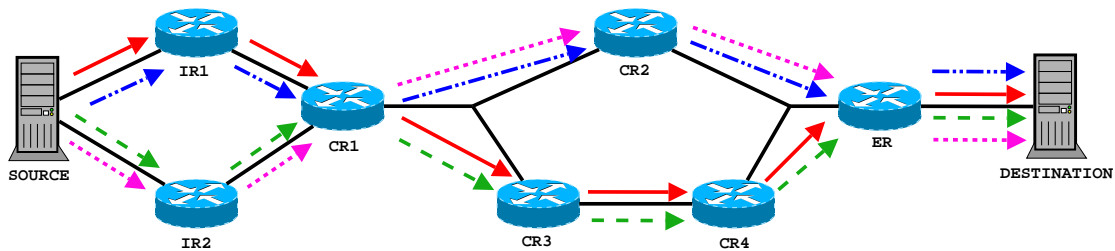


Figure 3.1: 4 different routes for LSPs.

3.1.1 Hypothesis on Experiment

To assure the reproducibility of the results and for the reader understanding, let us define precisely what the testing environment is. Then, the programs used to set up a MPLS network are described.

Environment

On the mailing list of the mpls-linux project, the most common mistake done by the testers is to forget to load the MPLS module. It may result in unexpected behavior of MPLS programs (iproute2, iptables, quagga). This command must be executed with root privilege.

```
$ /sbin/modprobe mpls4
```

Many testers wonder why MPLS programs return weird errors and do not work properly without the module loaded. The reason seems obvious.

Then, the IP forwarding of packets between interfaces must be enabled. On many systems, it is enabled by default but it is better to confirm its activation.

```
$ echo "1" > /proc/sys/net/ipv4/ip_forward
```

Finally, and the most important, the software required to create MPLS networks must be installed. They are presented hereafter in the following order: iproute2, iptables, quagga. They can be installed from the source code downloaded from the Perforce repository, see the mpls-linux website for more information. Alternatively they can be installed from the RPM's on SourceForge¹

¹http://sourceforge.net/project/showfiles.php?group_id=15443

Iproute2

Iproute2 is a collection of tools used for controlling TCP and UDP networking (basics) as well as QoS (advanced), in both IPv4 and IPv6 networks. It is designed to interact with the Linux kernel to provide quality of service and other routing features such as load balancing, shaping, policing and so on. For the reader interested in these features, the “Linux Advanced Routing & Traffic Control HOWTO” is a document providing theoretical and practical concepts of quality of service using the **tc**² tool. From queuing disciplines for bandwidth management to IPSEC, the documentation is a valuable resource to learn how to manage a network.

The mpls-linux project has extended the iproute2 software suite (**iproute2-mpls**) in order to have a front-end tool that interacts with MPLS kernel. This tool is the command *mpls*. It is further described in Section 4.4 because it is not used in the present chapter.

Iptables

Iptables is a tool used by administrators to create packet filtering rules. From the technical perspective, this tool just interacts with the kernel which controls the packet filtering and NAT (Network Address Translation) modules. The name Iptables may also refer to the complete software suite that contains:

- The user space tool called “*iptables*”,
- *Netfilter*, a framework providing a set of Linux kernel modules for capturing and modifying network packets,
- NAT module and
- *Connection tracking*.

The MPLS version of iptables will be used to intercept the traffic and turn it into MPLS traffic; in other words, it is used to match IP traffic and to push the relevant label in front of the IP header.

For instance, if we want to match the TOS field of an IPv4 packet and push a MPLS label on top of it, the command looks like this:

```
iptables -A INPUT -p tcp -m tos --tos 0x16 -j mpls --nhlfe KEY
```

²Traffic Control

The KEY argument is returned by the mpls command, see Section 4.4 for more details. The general syntax used to match TOS bits looks like:

```
-m tos --tos mnemonic [ other-args ] -j target
```

The mnemonics are listed in Table 3.1. The target, from the “-j target” argument, is in fact the same as the rest of the previous command : -j mpls --nhlfe KEY.

Mnemonic	Hexadecimal	Decimal
Normal-Service	0x00	0
Minimize-Cost	0x02	2
Maximize-Reliability	0x04	4
Maximize-Throughput	0x08	8
Minimize-Delay	0x10	16

Table 3.1: TOS field value for iptables.

It is also possible to match the port, the source or the destination address. These commands are very useful when the test includes many datastreams at the same time.

Quagga Routing Software Suite

This section is the continuation of Section 2.2.3 describing the Zebra routing software suite. These sections were split because:

- Quagga is used the mpls-linux project and GNU Zebra is not.
- Quagga is a fork of GNU Zebra which was developed originally by Kunihiro Ishiguro.
- There is a more active community around Quagga than the current centralized model of GNU Zebra.

Quagga is a set of open source routing daemons which implement the common routing protocols (RIP, OSPF, BGP). Each protocol runs as a separate daemon, and those daemons are all synchronized via a management daemon (zebra). The reader must pay attention to the differences between the program (daemon) called “zebra” and the software suite “GNU Zebra” which contains zebra. Figure 2.3, summarizes the Zebra architecture and also includes the ldpd daemon (LDP implementation).

“The Quagga architecture consists of a core daemon:

zebra Which acts as an abstraction layer to the underlying Unix kernel and presents the Zserv API over a Unix or TCP stream to Quagga clients. It is these Zserv clients which typically implement a routing protocol and communicate routing updates to the zebra daemon. Existing Zserv clients are:

ospfd: implementing OSPFv2

ripd: implementing RIP v1 and V2

ospf6d: implementing OSPFv3 (IPv6)

ripngd: implementing RIPng (IPv6)

bgpd: implementing BGPv4+ (including address family support for multicast and IPv6)

Additionally, the Quagga architecture has a rich development library to facilitate the implementation of protocol/client daemons, coherent in configuration and administrative behaviour.

Quagga daemons are each configurable via a network accessible CLI (called a 'vty'). The CLI follows a style similar to that of other routing software. There is an additional tool included with Quagga called 'vtysh', which acts as a single cohesive front-end to all the daemons, allowing one to administer nearly all aspects of the various Quagga daemons in one place.”[16]

Another daemon is added to the list, it is ldpd, the implementation of the LDP protocol. It is outlined in Section 3.2. Before entering into deeper technical aspects, let us explain what ldpd needs in order to make it work.

First, the **zebra** daemon must be up and running. To set up the zebra daemon, documentation can be found on the Quagga's website. An example of configuration file can be found in Appendix C.

Second, a layer 3 or above **routing protocol** must be up and running. Ldpd only advertises labels for routes which zebra learnt through an ip-routing protocol (OSPF, IS-IS, etc). In this case, the OSPF daemon (ospfd) is playing that role. Setting up an OSPF network is not so difficult but it requires some knowledge about Quagga configuration. See Appendix C for example of an ospfd configuration file.

User Interface	Protocol configuration Maintain configuration state Status queries Error notification
ldp-portable	Packet encode/decode Message processing
Porting Layer (quagga-mpls)	Socket activity: read/write, multicast Timers Memory allocation Routing interaction Interface interaction

Table 3.2: Layers involved in the LDP implementation.

3.2 Implementation of LDP

The project **ldp-portable**, also developed by James R. Leu³, is in fact a library implementing the LDP protocol. Some code has been added to Quagga in order to use that library as a separate daemon. The name of that modified Quagga is quagga-mpls.

Ldp-portable relies upon a layer called a "porting layer" to provide the low level primitive and infrastructure to build a working implementation of LDP. Table 3.2 shows a graphic of the layers involved and what they handle.

The API that needs to be filled in by the **porting layer** is defined in *ldp-portable/common/*_impl.h*. To port ldp-portable to another routing system than Quagga (e.g. XORP) these files would need to be re-implemented. These files were pointing out merely to give an overview of how ldp-portable relates to Quagga.

Ldp-portable provides an API to the user interface and the porting layer for some operating system driven actions, which is defined in *ldp-portable/ldp/ldp_cfg.c*. This is called the configuration API. It tries to mimic the LDP-MIB and the TE-MIB.⁴ Ldp-portable also contains all the data structures for the message processing and packet encoding/decoding. Nortel released a file containing all encoding/decoding functions⁵. Quagga-mpls is a just one porting layer that exists and by far the most mature.

Most of the actual **user interface** for Quagga's usage of ldp-portable is defined in *quagga-mpls/ldpd/ldp_vty.c*. The user interface is mainly the CLI from Quagga adapted

³<http://mpls-linux.sourceforge.net>

⁴MIB stands for Management Information Base, it is a virtual database where information about the network is stored. For more information, refer to the SNMP protocol or to the RFC 1156.

⁵For more information, see <http://www.nortel.com/products/announcements/mpls/source/disclaimer.html>

with `ldp-portable` command. It relies upon the utilities and structure implemented in `ldp_interface.c`, `ldp.c` and `ldp_remote_peer.c` to maintain the configuration state for `ldpd`. It makes calls to the configuration API to actually make `ldp-portable` work. In addition, the code in `ldp_zebra.c` is responsible for feeding routing and interface changes into `ldp-portable`. The user can dynamically change routing settings thanks to the user interface.

Currently, `ldp-portable` running on top of `quagga-mpls` provides :

- Handling of MPLS labels as nexthop
- Handling of MPLS labels as recursive nexthop.
- Implementation of CLI for static LSPs.
- Implementation of a porting layer.

There are also a few features `ldp-portable` does not provide yet such as penultimate hop popping and loose/explicit routing. These features would have been really helpful in this thesis.

3.3 Experiment

The testbed upon which the test of LDP will occur is described in Figure 2.5. There are two ways to configure `ldpd` : by the `vysh` CLI or by the `telnet` command. The use of `vysh` is definitely the most convenient, so the `telnet` way will not be described here.

3.3.1 Starting the Testbed

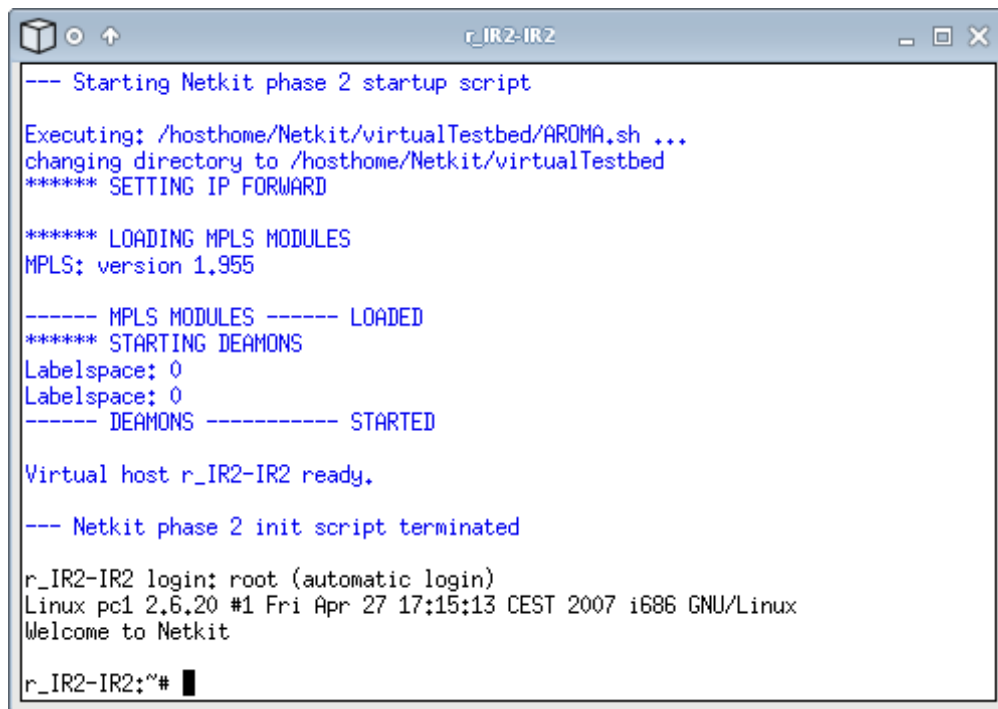
First things first, the virtual testbed must be started. The command to start the testbed can be executed by a simple user, `virtualTestbed` is the directory from which the AROMA virtual testbed will be launched. Just enter the following command :

```
virtualTestbed $ ./AROMA.sh start
```

It launches a terminal for each virtual machine that can be configured separately by the latter. The `AROMA.sh` script can be found in Appendix E.

3.3.2 Configuring a virtual machine

The `vysh` CLI is not really handy when it comes to configure the seven virtual MPLS routers of Figure 2.5 at the same time. As the configuration file makes LDP daemon



```

--- Starting Netkit phase 2 startup script

Executing: /hosthome/Netkit/virtualTestbed/AROMA.sh ...
changing directory to /hosthome/Netkit/virtualTestbed
***** SETTING IP FORWARD

***** LOADING MPLS MODULES
MPLS: version 1.955

----- MPLS MODULES ----- LOADED
***** STARTING DEAMONS
Labelspace: 0
Labelspace: 0
----- DEAMONS ----- STARTED

Virtual host r_IR2-IR2 ready.

--- Netkit phase 2 init script terminated

r_IR2-IR2 login: root (automatic login)
Linux pc1 2.6.20 #1 Fri Apr 27 17:15:13 CEST 2007 i686 GNU/Linux
Welcome to Netkit
r_IR2-IR2:~#

```

Figure 3.2: Screenshot of a started virtual machine

(ldpd) crash, it cannot be configured in advance. Otherwise, the configuration of every single virtual machine would have been automatically done by the *AROMA.sh* script. Eventually, vtysh is the most appropriate tool to configure ldpd. To enter into vtysh, just execute this command :

```
$ vtysh
```

on a bash prompt of a virtual machine, let's say CR2, and here is what should be printed out :

```

r_CR2-CR2:~# vtysh

Hello, this is Quagga (version 0.99.6).
Copyright 1996-2005 Kunihiro Ishiguro, et al.

r_CR2-CR2:#

```

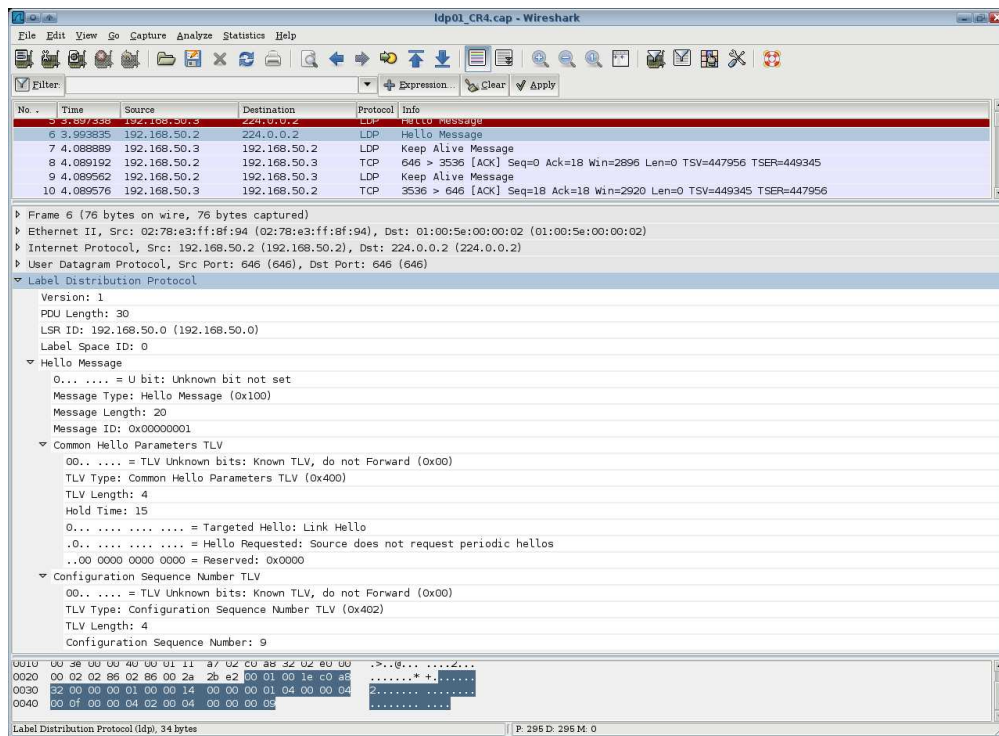


Figure 3.3: Screenshot of LDP message capture.

Note that the tilde (~) between the ':' and the '#' does not appear after the vttysh command was entered. This means that prompt is not bash anymore (or sh or zsh, whatever prompt used), it is the vttysh prompt. A screenshot of the present vttysh prompt is shown on Figure 3.2.

Afterwards, let us enter the ldpd related commands to start the LDP protocol. Here is for example the LDP configuration of the interfaces eth1 and eth2 of a virtual machine (CR2) with a per-platform labelspace:

# configure terminal	—— enter into the configuration mode
# (config) \$ mpls ldp	—— activate LDP
# (config-ldp) \$ exit	—— go back to configuration mode
# (config) \$ interface eth1	—— enter into interface mode of eth1
# (config-if) \$ mpls labelspace 0	—— set per-platform labelspace
# (config-if) \$ mpls ip	—— activate LDP for this interface
# (config-if-ldp) \$ exit	—— go back to interface mode
# (config-if) \$ exit	—— go back to configuration mode
# (config) \$ interface eth2	—— let's do the same for eth2
# (config-if) \$ mpls labelspace 0	—— set per-platform labelspace
# (config-if) \$ mpls ip	—— activate LDP for this interface
# (config-if-ldp) \$ exit	—— go back to previous mode
# (config) \$ end	—— end the configuration
#	—— top level

Note that if the “mpls ip” command is not entered on a network interface configuration mode, this network interface will not execute the LDP protocol!

3.3.3 Setting up LSPs

After all the routers have been configured, the virtual testbed is ready to set up a LSP by LDP. A simple way to see if the LDP daemon is actually working is to sniff the network during 10 seconds or more. A *Hello Message* should be captured. Figure 3.3,“ shows a capture of the so-called *Hello Message*. Note that the destination address is 224.0.0.2, which is a broadcasting address. This is meant to discover new neighbors by broadcasting a message over UDP. Just below the *Hello Message* there is a *Keep Alive Message* which is meant to be sure that the router is still working (alive).

The ingress router, in this case IR1, is able to setup a LSP by creating an entry into the NHLFE table and by mapping a FEC to the latter. Note that if the datastream is connection oriented, it should have an upstream LSP (i.e. set up an LSP at the egress router, ER). These commands are described in Section 4.4.

The commands relative to the LSP creation are explained in Chapter 4. Only the edge routers are allowed to create a LSP, it justifies that only the ingress router commands are needed in order to set up a LSP.

Figure 3.4 shows which path LDP use to set up a LSP from the source to the destination. It is using this path because it relies on zebra to provide the routes to advertise the labels accordingly. The routes provided by zebra are actually provided by ospfd to zebra. This is a way to abstract the fact that ldpd gets its route information from a

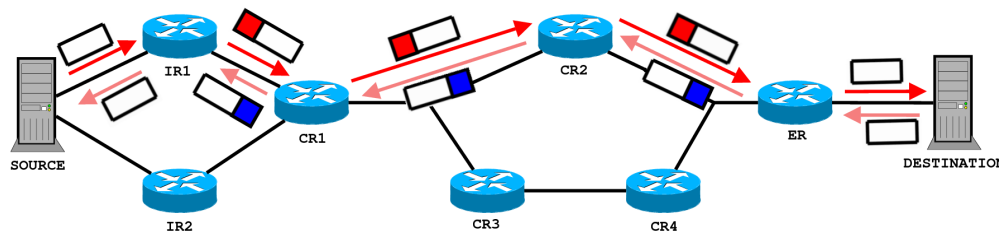


Figure 3.4: LSP setup by LDP.

layer-3 protocol as explained in the end of Section 3.1.1 (The second requirement for ldpd to work properly, page 55). As none is especially required, ospfd seemed to be a good candidate. The OSPF protocol is not the only one, IS-IS could have done the work but ospfd is older, hence more reliable and robust.

3.4 Benchmarking

MPLS standards has evolved since its creation, lots of drafts and RFCs are dedicated to it and the number is still growing. The benchmarking is a technique used to compare multiple object performances. In this section, the implementation of the MPLS protocol used all along this thesis, mpls-linux, is compared to the TCP/IP protocol. In other words, it shows the results of IPv4 versus MPLS.

3.4.1 Benchmark Environment

The AROMA virtual testbed will be used to benchmark the MPLS and the IPv4 network. As in the IPv4 network OSPF protocol is running in order to make every node reachable from every other node. The route selected will be the exact same route as in the LDP case (i.e. the shortest path based on the OSPF information).

The tool used to measure the network performances is Iperf. ⁶ Iperf is designed to measure maximum TCP bandwidth. It is not its only purpose but the delay jitter and datagram loss rate are not very relevant with the virtual testbed. With a virtual testbed, these parameters are limited by the CPU which emulates all virtual machines. That is the reason why this is a relative comparison, not an absolute performance measurement.

The tests conducted were the measurement of maximum TCP bandwidth during 30 seconds. The tests were repeated five times and the latest results are exposed hereafter. As the computer upon which the virtual testbed was emulated is a desktop computer, the results might have been altered by other running applications and/or daemons.

⁶<http://dast.nlanr.net/Projects/Iperf>

3.4.2 Results

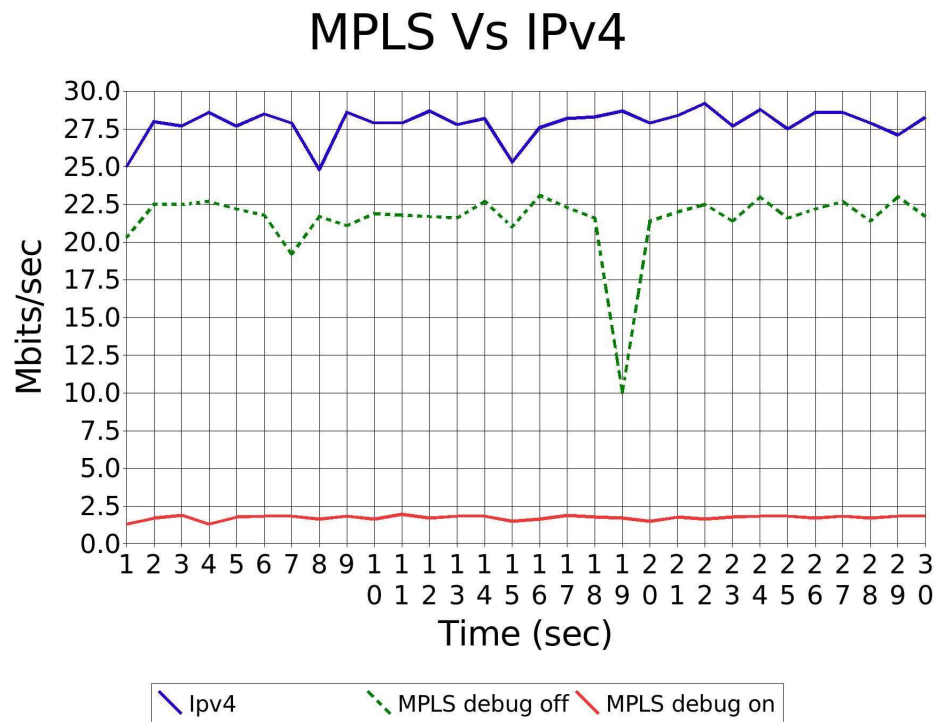


Figure 3.5: MPLS vs IPv4 TCP/IP benchmark.

Figure 3.5 shows the graphical comparison between a IPv4 network and a MPLS network. As we can see, the IPv4 traffic (higher line) is faster than the MPLS traffic (dotted line and lower line). Table 3.3 shows the numerical results from the comparison. In theory, the MPLS network should be faster than the IPv4 network. The results may seem incorrect because the IPv4 network is faster than the MPLS one but the reason is given by James R. Leu, developer of the mpls-linux project. Here is his verbatim comment:

“Why is MPLS Linux slower at forwarding packets then Linux’s IPv4 stack?

There are some misconceptions out [there] regarding the speed of MPLS vs IPv4 packet processing

Back in the mid 90’s the state-of-the-art in edge and core routing technology was processor based packet forwarding. At that same time the requirements for how per packet forwarding decisions were being made was

	Ipv4	MPLS debug off	MPLS debug on
	Mbits/sec		
1	25.0	20.3	1.31
2	28.0	22.5	1.70
3	27.7	22.5	1.90
4	28.6	22.7	1.31
5	27.7	22.2	1.77
6	28.5	21.8	1.84
7	27.9	19.2	1.84
8	24.8	21.7	1.64
9	28.6	21.1	1.84
10	27.9	21.9	1.64
11	27.9	21.8	1.97
12	28.7	21.7	1.70
13	27.8	21.6	1.84
14	28.2	22.7	1.84
15	25.3	21.0	1.51
16	27.6	23.1	1.64
17	28.2	22.3	1.90
18	28.3	21.6	1.77
19	28.7	10.0	1.70
20	27.9	21.4	1.51
21	28.4	22.0	1.77
22	29.2	22.5	1.64
23	27.7	21.4	1.77
24	28.8	23.0	1.84
25	27.5	21.6	1.84
26	28.6	22.2	1.70
27	28.6	22.7	1.84
28	27.9	21.4	1.70
29	27.1	23.0	1.84
30	28.3	21.7	1.84
AVG	27.9	21.5	1.73

Table 3.3: MPLS vs IPv4 - values.

getting more complicated. Edge and core routers were being asked to consider source and destination addresses, incoming and outgoing interfaces, as well as TCP/UDP port numbers. This forced router vendors to switch to some sort of "flow" or "hash" based look up to determine the forwarding treatment (next hop and/or queuing). As any [computer science] major knows both flow and hash based look up schemes can suffer from high amounts of "key collisions" when 1000s of packet flows per second are being considered. This in essence change the look up depth from 32 bits to something greater than 32 bits depending on the technique and the amount of "key collisions". So per packet decisions making was becoming a bottle neck in the core of the network. Along came various "IP Switching" techniques and "tag switching" all of which contributed to MPLS. One of the benefits of MPLS at that time was that the complex decision making for forwarding treatment was done once before at "LSP setup time" and per packet processing would be a consistent 20 bit look up. If the state-of-the-art in packet forwarding has stood still, then MPLS would have been the savior of core routing, but in the time it took for MPLS to become a standard the world of packet forwarding was revolutionized by ASICs and FPGAs. These hardware based packet look up engines could do the complex look up required by core and edge routers faster than the pipes could transport the packets.

So when people said "MPLS should be faster than IPv4 at packet processing" they were not referring to standard destination based IPv4 forwarding, they were talking about complex forwarding decision making. Theoretically standard IPv4 destination only processing has a worst case of 32 bits of look up and MPLS has a constant 20 bits of look up, not enough of a difference to show up in throughput tests. So if your comparison of MPLS Linux forwarding versus Linux IPv4 forward is only based on IPv4 destination look ups, you should not expect to see a performance benefit (in fact MPLS Linux forces all ILM keys into a 32 bit number, so it too is doing a 32 bit look up :-). That in combination with the fact that MPLS Linux has not under gone any sort of optimization and has enormous amount of debug/tracing code, while the Linux IPv4 stack has undergone years of optimization by some of the brightest minds in the world. I'm surprised that MPLS Linux has performed as well as it has in the tests results I've seen."⁷

Note that the performances are approximately ten times worse with the debug message activated. In order to have better performances, the debug message must be deac-

⁷<http://mpls-linux.sourceforge.net/mpls-faster-then-ipv4.html>

tivated. Here is the command to turn it off:

```
$ echo 0 > /sys/mpls/debug
```

3.5 Conclusion

The LDP protocol is good enough for simple LSP creation but is not designed to provide quality of service (QoS) or traffic engineering (TE). It is mainly aimed to facilitate the administration of a MPLS network, in other words, to maintain and manage the database of the LSRs. However, the validation of the LDP protocol implementation (ldp_portable) demonstrates that it works fine and enables to create LSP. Once configured, the virtual machines act as MPLS-LDP compatible routers.

Unfortunately, the routing information provided by another routing protocol might cause congestion. Hence, it is needed to provide traffic engineering in order to have more control about the route used and the QoS parameters. On the other hand, to conduct a test on a virtual testbed to measure QoS parameters is not relevant. Only a physical testbed could do this work.

Fortunately, an extension of the LDP protocol can provide TE and QoS. This is *Constraint-based Routing Label Distribution Protocol* (CR-LDP). This protocol is outlined in Section 1.4.2, page 24.

The benchmark showed that the mpls-linux implementation is slower than the current IPv4 forwarding process. The thesis of Pere Tuset [11]⁸ confirms this fact by testing mpls-linux on a real hardware testbed.

⁸English title: “Performance analysis of a mpls-linux network”

CONSTRAINT ROUTING ON AROMA

This chapter will cover the rerouting of one explicit route to another in a way aiming at compliancy with RFC 3214. Only the way of doing so and the results will be discussed here, since the tools and softwares used for this test have already been described in Chapter 3.

4.1 Rerouting LSPs method from RFC 3214

This section shows how the rerouting of LSPs was designed at the first place. The rerouting should actually be done by a daemon that implements the CR-LDP protocol. The LDP protocol is implemented as daemon thanks to the Quagga routing software. The LDP daemon could have been modified in order to provide the proper extensions to set up interactively and in an abstract way the LSPs but by lack of time and the high complexity of Quagga and the LDP daemon make the task too hard to achieve.

Here the verbatim method taken from the RFC 3214:

“LSP modification can also be used to reroute an existing LSP. Only modification requested by the ingress LSR of the LSP is considered in this document for CR-LSP. The Ingress LSR cannot modify an LSP before a previous modification procedure is completed.

Consider a CR-LSP L1 with LSPID L-id1. To modify the route of the LSP, the ingress LSR R1 sends a Label Request Message. In the message, the LSPID TLV indicates L-id1 and the Explicit Route TLV is specified with some different hops from the explicit route specified in the original Label

Request Message. The action indication flag has the value `_modify_`.

At this point, the ingress LSR R1 still has an entry in FTN as $\text{FEC1} \rightarrow \text{Label A}$. R1 is waiting to establish another entry for FEC1.

When an LSR R_i along the path of L1 receives the Label Request message, its behavior is the same as that of receiving a Label Request Message that modifies some other parameters of the LSP. R_i assigns a new label for the Label Request Message and forwards the message along the explicit route. It does not allocate any more resources except [in the specific cases] described in Section 3.1.¹

At another LSR R_j further along the path, the explicit route diverges from the previous route. R_j acts as R_i , but forwards the Label Request message along the new route. From this point onwards the Label Request Message is treated as setting up a new LSP by each LSR until the paths converge at later LSR R_k . The `_modify_` value of the action indication flag is ignored.

At R_k and subsequent LSRs, the Label Request Message is handled as at R_i .

On the return path, when the Label Mapping message is received, two sets of labels for the LSPID exist where the new route coincide with the old. Only one set of labels will exist at LSRs where the routes diverge.

When the Label Mapping message is received at the ingress LSR R1 it has two outgoing labels, A and B, associated with the same FEC, where B is the new outgoing label received for LSP L1. R1 can now activate the new entry in the FTN, $\text{FEC1} \rightarrow \text{Label B}$ and de-activate the old entry $\text{FEC1} \rightarrow \text{Label A}$. This means that R1 swaps traffic on L1 to the new label B. The packets are now sent with the new label B, on the new path.

The ingress LSR R1 then starts to release the original label A for LSP L1. The Label Release Message is sent by R1 towards the down stream LSRs following the original route. The Release message carries the LSPID of L-id1 and the Label TLV to indicate which label is to be released. At each LSR the old label is released - no further action is required to change the path of the data packets which are already following the new route programmed by the Label Mapping message.

¹This section of the RFC describes how to avoid double booking resources. It does not concern the scenario of this chapter because it is not planned to reserve resources.

At some LSRs, where the routes diverged, there is only one label for the LSPID. For example, between Rj and Rk, the Label Release Message will follow the old route. At LSRs between Rj and Rk only the labels from the original route will exist for LSPID L-id1. At these LSRs the LSPID TLV does not need to be examined to release the correct label, but it must still be updated and passed on to the next LSR as the Label Release message is propagated. In this way, at Rk where the routes converge, the downstream LSR will know which label to release and can continue to forward the Label Release Message along the old route.”[3]

4.2 Description of the LSPs

Two LSPs will be drawn across the virtual testbed: the main LSP and the back LSP. For both LSPs, packet processing be performed at edge routers: ingress (IR1) and egress router (ER).

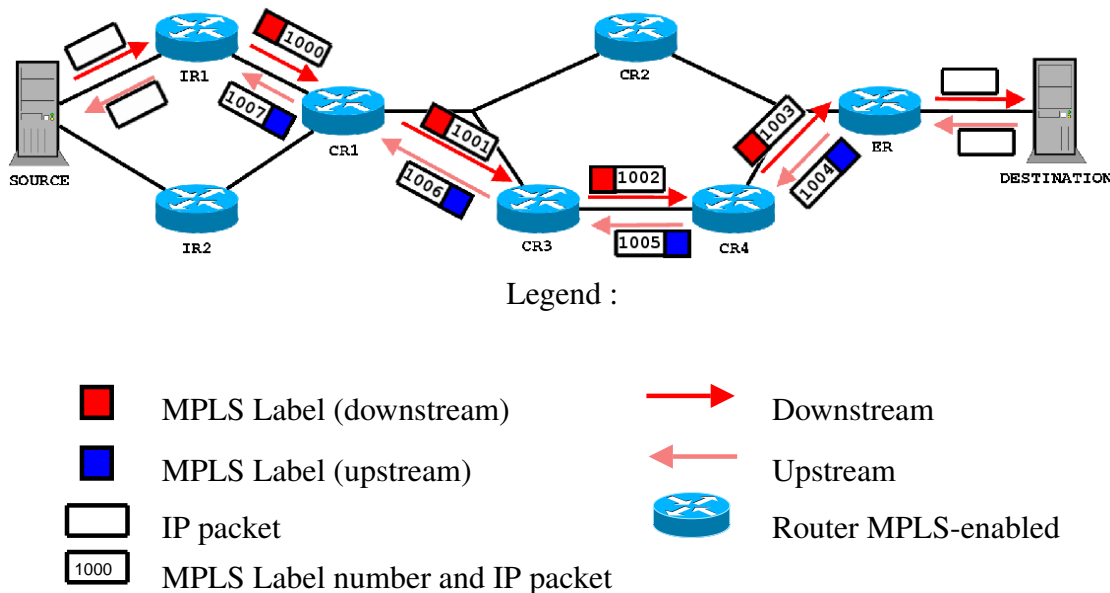


Figure 4.1: 2-way explicit MPLS route.

4.2.1 Main LSP

This LSP is an explicit route set up with the help of the **mpls** command which is part the *iproute2-mpls* program, as seen in Section 4.4. The LSP, let's call it **LSP-A**, is composed by two LSPs : one in downstream (LSP-A DOWN) and one in upstream (LSP-A UP).

The path is:

SOURCE \rightarrow IR1 \rightarrow CR1 \rightarrow CR 3 \rightarrow CR 4 \rightarrow ER \rightarrow DESTINATION

for the downstream and

SOURCE \leftarrow IR1 \leftarrow CR1 \leftarrow CR 3 \leftarrow CR 4 \leftarrow ER \leftarrow DESTINATION

for the upstream.

Figure 4.1 shows the main LSP that the datastream will switch from. The real and effective behavior might be a bit different. Figure 4.1 shows that the downstream between CR4 and ER has a label 1003 and the upstream between CR1 and IR1 has a label 1007. This is not actually true if the **penultimate hop popping**² mode is active. So, CR4 on downstream and CR1 on upstream would pop the label before sending the packets to the outermost MPLS router; ER and IR1 respectively.

Note that Figure 4.1 is different from Figure 3.4. The most obvious difference is the path used by the datastream but also whether the labels are defined by the user or not. In Figure 3.4, the labels are managed by the LDP protocol and are not defined by the user interface such as in this test. In Figure 4.1, the labels (numbers) are set on each and every node thanks to the **mpls** command.

Any hop-by-hop routing protocol would have oriented the traffic on CR2 because it is the shortest path. With explicit routing, it is possible to guide the traffic in order to balance it across the network and to reduce possible congestion spots. As seen in Chapter 3, LDP gets its routing information from the routing tables that were filled by the OSPF daemon. The route that would have been used if OSPF was running is the route passing through CR2 (if the metrics were not changed on purpose).

4.2.2 Backup LSP

The LSP, let's call it **LSP-B**, is composed by two LSPs : one in downstream (LSP-B DOWN) and one in upstream (LSP-B UP).

The path is:

SOURCE \rightarrow IR1 \rightarrow CR1 \rightarrow CR 2 \rightarrow ER \rightarrow DESTINATION

²See Section 1.3.3, page 17.

for the downstream and

$\text{SOURCE} \leftarrow \text{IR1} \leftarrow \text{CR1} \leftarrow \text{CR2} \leftarrow \text{ER} \leftarrow \text{DESTINATION}$

for the upstream.

In the real world, this LSP will be used only if the main LSP crashes. This scenario is a make-before-break method as described in Section 1.4, page 29.

When the switch of LSP will occur, instead of going through the CR3 and CR4 nodes, the datastream will go through CR2. Before switching, the backup LSP must be set up with unique labels (according to the labelspace), i.e. different from the labels already in use by the main LSP. Once again, if the penultimate hop popping mode is active the downstream between CR2 and ER will not be labeled.

4.3 Hypothesis on experiment

1. **Lack of accuracy for delay measurements** — virtual testbeds are not accurate for delay measurements because the packets are not “really” sent over a cable or by microwaves. So, measuring the time it takes to switch from one LSP to another is not relevant.
2. **Manual set up of LSPs** — LSPs are created with the help of a configuration script. Their properties are hardcoded. Hence, both the primary LSP and its backup will be configured in such a script.
3. **LSP switching at edge routers only** — only the ingress and the egress nodes are allowed to switch the traffic from one LSP to another. Core nodes have a different role to play and a limited control over the whole LSP that goes through.

4.4 The mpls command

The mpls command is actually a front-end utility to interact with the MPLS kernel. With this tool, it is possible to add, delete and even change the MPLS configuration. This section is definitely not an exhaustive documentation, it only explains the command that will be used in this case. It is, however, a good start to create LSPs. The syntax of the mpls command is in Appendix D.

As there are three kinds of routers — ingress, core, egress — it is normal to set up a LSP accordingly. First, the ingress router acts like a classifier, it does some traffic engineering to determine what traffic should go on which LSP.

It is assumed that the reader knows a little of shell scripting, the use of pipes and so on.

4.4.1 Ingress Router Command

These commands must be executed on the ingress router IR1. First, it creates a NHLFE entry to add label 1000 (PUSH) and forward the packets (nexthop) to 192.168.10.2 using outgoing interface eth0.

```
$ key_1='mpls nhlfe add key 0 instructions push gen 1000 nexthop eth0 \
ipv4 192.168.10.2 |grep key |cut -c 17-26'
```

The “key 0” means that it will return a new key if the command is successfully. The “gen” parameter means that we use Ethernet encapsulation. And the “ipv4” parameter means that it use the IPv4 layer 3 protocol.

The mpls command, then, returns a key in hexadecimal, the rest of the command (i.e. “|grep key |cut -c 17-26”) only takes the hexadecimal key and not the text around. That is really useful because key need to be stored (in a variable) for further use.

Then, it maps the FEC to the NHLFE created earlier.

```
$ /usr/sbin/ip route add 192.168.70.2/32 via 192.168.30.1 mpls $key_1
```

The \$key_1 is a variable that contains the key returned by the mpls command while it created the NHLFE entry. In this case, for the sake of simplicity, it only uses the destination IP to match³ the traffic to a FEC.

4.4.2 Core Router Command

These commands must be executed on the core router CR1. The other core routers must also execute this command but parameters (labels, interfaces) should be adapted accordingly.

First, this command tells the router on which interface the MPLS packets should be expected to arrive. The labelspace is used to determine if it is per platform or per interface labelspace. When the labelspace is equal to zero (labelspace 0), it means that it is a per platform labelspace.

³To “catch” the traffic would be more appropriate since no other traffic exists besides the one that the source is generating


```
$ mpls labelspace set dev eth0 labelspace 0
```

Then, an entry to the ILM table must be added in order to list an expected label.

```
$ mpls ilm add label gen 1000 labelspace 0
```

Afterwards, another NHLFE entry must be created to forward the MPLS packet.

```
$ key_1='mpls nhlfe add key 0 instructions push gen 1001 nexthop eth2 \  
ipv4 192.168.80.3 |grep key |cut -c 17-26'
```

Finally, this command does the switching by mapping the ILM entry to the NHLFE entry.

```
$ mpls xc add ilm_label gen 1000 ilm_labelspace 0 nhlfe_key $key_1
```

The “xc” parameter is used to specify that it switches from label 1000 to the label defined by the NHLFE entry (1001 in this case).

4.4.3 Egress Router Command

These commands have the same meaning than the commands for the core router commands:

```
$ mpls labelspace set dev eth0 labelspace 0
```

```
$ mpls ilm add label gen 1003 labelspace 0
```

```
$ key_1='mpls nhlfe add key 0 instructions nexthop eth1 \  
ipv4 192.168.70.2 |grep key |cut -c 17-26'
```

This last command is actually the same as the core router commands but the behavior of the egress router is quite different.

```
$ mpls xc add ilm_label gen 1003 ilm_labelspace 0 nhlfe_key $key_1
```

A mode called the *penultimate hop popping* avoids resource wastage. This mode is described in Section 1.3.3. This mode is used to avoid the egress router to make two lookups: one for the MPLS label, one for IP address. Hence, it reduces the load on the edge router. Unfortunately, this mode is not implemented neither in mpls-linux nor in ldp-portable.

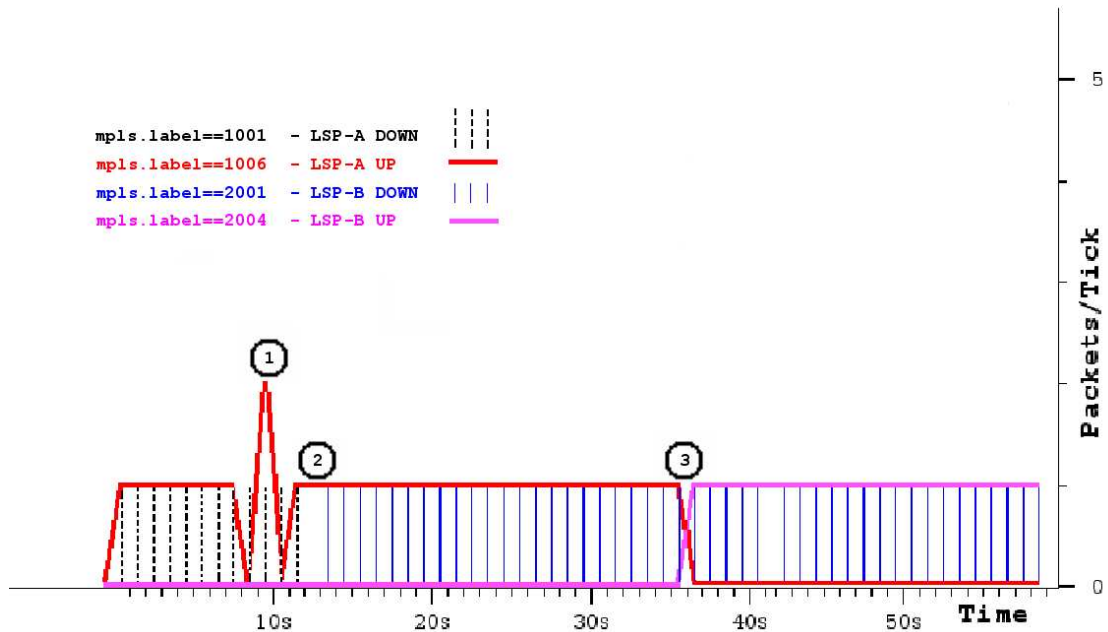


Figure 4.2: Graphic of packets from LSPs switch (simple ping).

4.5 Rerouting LSPs

This section intends to outline what happened when the traffic switches from LSP-A (main LSP) to LSP-B (backup LSP). Figure 4.2 was produced thanks to Wireshark⁴ and shows the result of the switch. All the packets were sniffed on the west interface of CR1. Note that there is no packet loss during this test because ping uses a TCP connection and the virtual testbed is not overloaded by packet processing of the ping requests and replies. Figure 4.3 shows the three s.pdf for rerouting a two-way LSP. The MPLS labels of LSP-A can be seen in Figure 4.1.

Figure 4.2 shows all the packets from the LSPs. The datastream used to test the LSPs is just a ping, executed on the source and pointing to the destination : “ping 192.168.70.2”. The spot ① is nothing but two ping replies received approximatively at the same time. Figure 4.3a describes the situation from the beginning of the transmission (time 0) to the spot ② (time 12) such as LSP-A is used. The spot ② is the moment where the downstream switches from LSP-A to LSP-B. The time frame between spot ② and ③ (time 36) is described by Figure 4.3b. Note that, at this moment, the ping reply (upstream) is still using LSP-A while the ping request (downstream) is now using LSP-B. At the spot ③, the upstream switches from LSP-A to LSP-B. Then, this case is described by Figure 4.3c.

⁴<http://www.wireshark.org>

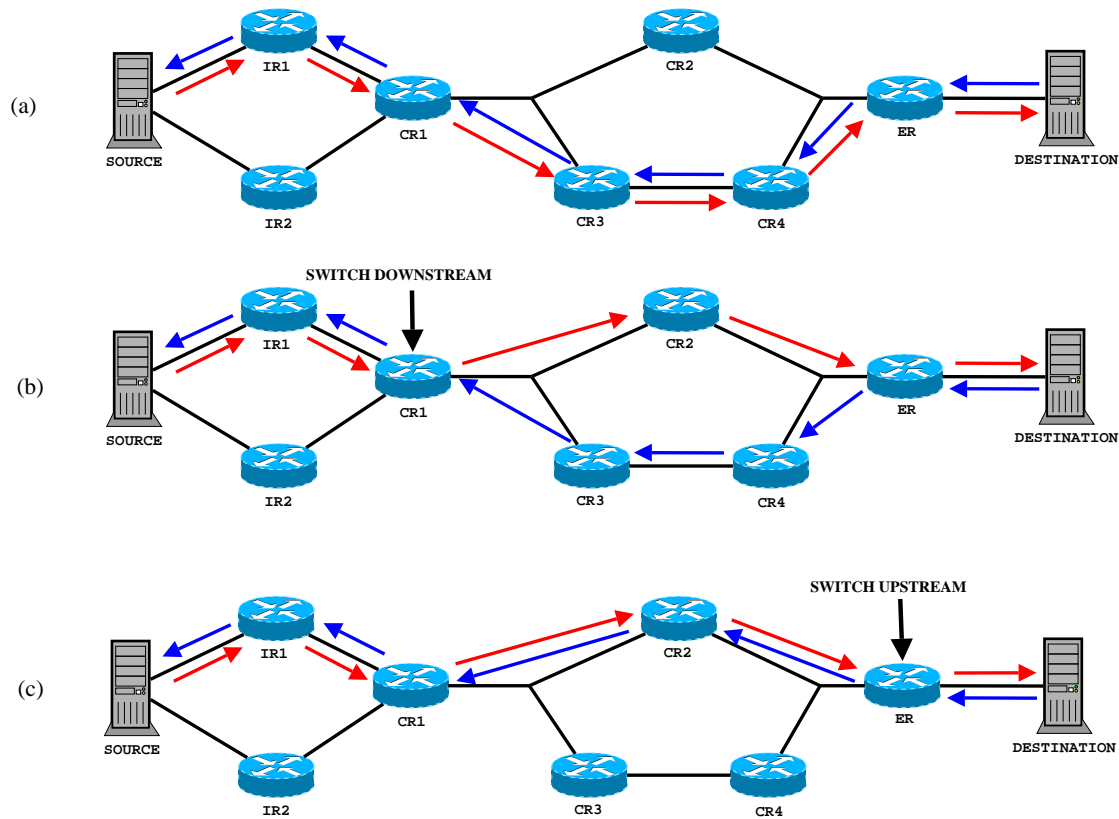


Figure 4.3: Switch of LSP.

4.6 Conclusion

This “switch LSP” method must be done by hand. To have it done automatically⁵, a full implementation of the CR-LDP⁶ protocol is needed. The advantage of a full implementation is that bandwidth reservation can be done and many other parameters can be tuned in order to have the best network performances. Hence, more tests could have been done and more scenarios could have tested with a full implementation of CR-LDP.

With the on-the-fly LSP rerouting, the protocol can detect a failure from a node and reroute (previously called “switch”) a datastream to another path (i.e. a backup LSP) without losing the guarantee of the quality of service. It means that for example, a Voice over IP communication can use more than one path along its session to reach its destination, it corresponds to some extends to the GSM hand-over.

⁵It means: on a single user interface and by a predefined command included into the software, not by a simple script gathering fews commands and using tricks or hacks instead of API.

⁶See Section 1.4.2, page 24

It is also possible to balance the load across the network by setting up different LSPs according to the requirements of the datastream. The more the parameters can be tuned, the more there are possibilities.

FUTURE WORK

Finally, this last chapter is intended to give some ideas for future developments after presenting the conclusion of this report. The first idea is a continuation of the benchmark done in Chapter 3 that compares MPLS and IPv4. The last two ideas are an application of the rerouting script executed in Chapter 4.

5.1 Conclusion

This report investigated the dynamic configuration of label switched path on a virtual testbed. The study was performed with the help of Netkit [17] and the mpls-linux project [15]. Some developments made this report possible.

Netkit has been used to emulate the virtual testbed while the mpls-linux project is designed to create a set of MPLS signaling protocols and an MPLS forwarding plane for the Linux operating system. Both projects are still on-going and some version conflicts have occurred but the community around them is active and help to solve these issues.

Netkit is definitely a great tool when it comes to perform networking experiments at low cost. Although it was necessary to modify the file system, once the script to start the testbed was written, everything was up and running. The mpls-linux project is very complex and requires higher level of technical skills than Netkit because it suffered from many version conflicts between kernels, modules and libraries.

On Linux, MPLS is a relatively new technology that can improve network performances. The on-the-fly rerouting feature that was previously described can be very helpful for in a context of setting up reliable QoS. In case one path is unavailable during a communication, another path can be used to preserve the QoS guarantee without interrupting the session. This feature would be interesting in many situations.

5.2 MPLS versus IPv6

In Chapter 3, a benchmark of MPLS versus IPv4 is described. Intuitively, as a MPLS label is coded on 20 bits, one would expect a MPLS lookup to be faster than an 32-bit IPv4 lookup. As James R. Leu explained in his comment, MPLS is slower because of the debugging and tracing code, the extension of the 20-bit MPLS label to 32 bits and the optimization of the IPv4 code.

Hence, the comparison of MPLS and IPv6 is relevant since IPv6 requires a 128-bit lookup. It would be interesting to compare the results with [11].

5.3 Fast Reroute

The fast reroute consists of three s.pdf:

1. A crash of a node used to forward traffic occurs ;
2. The crash is detected ;
3. A mechanism is activated to reroute the traffic to another route (e.g. backup LSP).

This technique was designed for RSVP at the first place. The RFC 4090, “Fast Reroute Extensions to RSVP-TE for LSP Tunnels” outlines the method thoroughly. In Chapter 4 we saw a script to manually execute on edge routers, allowing a LSP switch while traffic is still going through. This script can be improved in order to make a fast reroute scenario.

5.4 Preemption

By the time of writing this master thesis, *draft-deoliveira-diff-te-preemption-0X.txt* became the RFC 4829 — “Label Switched Path (LSP) Preemption Policies for MPLS Traffic Engineering”.

“When the establishment of a higher priority (Traffic Engineering Label Switched Path) TE LSP requires the preemption of a set of lower priority TE LSPs, a node has to make a local decision to select which E[xplicitly routed] LSPs will be preempted. The preempted LSPs are then rerouted by their respective Head-end Label Switch Router (LSR). [The RFC] presents a flexible policy that can be used to achieve different objectives: preempt the lowest priority LSPs; preempt the minimum number of LSPs; preempt the set of TE LSPs that provide the closest amount of bandwidth to the required bandwidth for the preempting TE LSPs (to minimize bandwidth wastage);

preempt the LSPs that will have the maximum chance to get rerouted. [...] A comparison among several different policies, with respect to preemption cascading, number of preempted LSPs, priority, wasted bandwidth and blocking probability is also included.” [8]

The main difference with the tests previously done is that resource reservation is needed. The resource reservation was not taken into account all along this thesis. Hence, using the virtual testbed to test preemption policies can provide a good start to obtain clear results.

This document can be resourceful for those who intend to test these new methods for MPLS on a virtual testbed.

BIBLIOGRAPHY

- [1] L. Andersson, P. Doolan, N. Feldman, A. Fredette, and B. Thomas. LDP Specification. RFC 3036 (Proposed Standard), January 2001.
- [2] J. Ash, M. Girish, E. Gray, B. Jamoussi, and G. Wright. Applicability Statement for CR-LDP. RFC 3213 (Informational), January 2002.
- [3] J. Ash, Y. Lee, P. Ashwood-Smith, B. Jamoussi, D. Fedyk, D. Skalecki, and L. Li. LSP Modification Using CR-LDP. RFC 3214 (Proposed Standard), January 2002.
- [4] D. Awduche, A. Chiu, A. Elwalid, I. Widjaja, and X. Xiao. Overview and Principles of Internet Traffic Engineering. RFC 3272 (Informational), May 2002.
- [5] D. Awduche, J. Malcolm, J. Agogbua, M. O'Dell, and J. McManus. Requirements for Traffic Engineering Over MPLS. RFC 2702 (Informational), September 1999.
- [6] Christophe Blaess. *Programmation Systeme en C sous Linux*. EYROLLES, second edition, January 2005.
- [7] C. Boscher, P. Cheval, L. Wu, and E. Gray. LDP State Machine. RFC 3215 (Informational), January 2002.
- [8] J. de Oliveira, JP. Vasseur, L. Chen, and C. Scoglio. Label Switched Path (LSP) Preemption Policies for MPLS Traffic Engineering. RFC 4829 (Informational), April 2007.
- [9] Jeff Dike. *User Mode Linux*. Prentice Hall, April 2006.
- [10] N. Feldman, A. Viswanathan, Z. Wang, and R. Callon. Evolution of multiprotocol label switching. *Communications Magazine, IEEE*, 36(5):165–173, May 1998.

- [11] Pere Tuset i Peiro. *Analisi de rendiment d'una xarxa mpls-linux*. Master's thesis, Escola Universitaria Politecnica de Mataro, 2007.
- [12] B. Jamoussi, L. Andersson, R. Callon, R. Dantu, L. Wu, P. Doolan, T. Worster, N. Feldman, A. Fredette, M. Girish, E. Gray, J. Heinanen, T. Kilty, and A. Malis. *Constraint-Based LSP Setup using LDP*. RFC 3212 (Proposed Standard), January 2002. Updated by RFC 3468.
- [13] Keith Kurose, James. Ross. *Computer Networking*. Pearson Education, third edition, 2005.
- [14] Dee-Ann. LeBlanc. *Linux System Administration — Black Book*. CoriolisOpen Press, 2000.
- [15] James R. Leu. <http://mpls-linux.sourceforge.net/>.
- [16] Quagga. <http://www.quagga.net>, 2007.
- [17] Massimo Rimondini. Netkit man page. <http://www.netkit.org>, 2005.
- [18] Massimo Rimondini. *Emulation of computer networks with netkit. Technical Report RT-DIA-113-2007*, January 2007. Roma Tre University, Jan 2007.
- [19] E. Rosen, D. Tappan, G. Fedorkow, Y. Rekhter, D. Farinacci, T. Li, and A. Conta. *MPLS Label Stack Encoding*. RFC 3032 (Proposed Standard), January 2001. Updated by RFCs 3443, 4182.
- [20] E. Rosen, A. Viswanathan, and R. Callon. *Multiprotocol Label Switching Architecture*. RFC 3031 (Proposed Standard), January 2001.
- [21] Monique. Sayeed, Azhar. Morrow. *Technology overview: Making the technology case for mpls and technology details*. <http://www.ciscopress.com/articles/article.asp?p=680839&seqNum=8>, January 2007.
- [22] Laurent Schumacher. *Teleinformatique matieres approfondies*. University of Namur (FUNDP), 2006. Lecture.
- [23] Andrew S. Tanenbaum. *Computer Networks*. Pearson Education, fourth edition, 2003.
- [24] Aroma Testbed. <http://www.aroma-ist.upc.edu>, 2006.
- [25] Yon Uriarte. *Zebra hacking howto*, Feb 2001. <http://www.quagga.net/zhh.html>.

- [26] Zheng Wang. *Internet Qos: Architectures and Mechanisms for Quality of Service*. Morgan Kaufmann Publishers, first edition, March 2001.
- [27] Wikipedia. <http://www.wikipedia.org>, 2007.

APPENDIX A

CERTIFICATE AROMA




Dr. Fernando Casadevall Palacio, Professor at Department of Signal Theory and Communication of UPC (Barcelona, Spain), as project manager of the AROMA (Advanced Resource management solutions for future all IP heterOgeneous Mobile rAudio environments) project, funded by European Union, Ref. IST-4-027567:

CERTIFIES THAT:

Julien Bisconti, student at the FUNDP (Namur, Belgium), has done five months traineeship at UPC, working with the AROMA project research team. He has **accomplished the following tasks:**

- Extend the testbed with three computers in order to fulfill the requirements of AROMA project ;
- Install MPLS (MultiProtocol Label Switching) on the AROMA testbed ;
- Install the needed programs in order to use LDP (Label Distribution Protocol) ;
- Automate the connection/login to all the testbed nodes ;
- Start to develop the code for CR-LDP (Constraint-base Routing – Label Distribution Protocol).

And in witness thereof, signs the present certificate,



Fernando Casadevall

Fernando Casadevall Palacio
Barcelona, January 10, 2007

ENLARGE NETKIT FILESYSTEM

This is the method to enlarge the Netkit filesystem. It might require this :

1. Every command must be executed on the host, not on the guest.
2. The user is able to mount a image file to a loop device (i.e. administrative rights are correctly set).
3. A backup of the image file should be done

B.1 Checking Filesystem Consistency (optional)

Attach the filesystem to a loop device. See “man netkit-filesystem” for more information
\$ losetup -o 16384 /dev/loop0 \$NETKIT_HOME/fs/netkit-fs-F2.2
Check if the filesystem is consistent.
\$ e2fsck -f /dev/loop0
Detach from loop device.
\$ losetup -d /dev/loop0

Table B.1: Checking filesystem consistency

B.2 Resizing Filesystem

Replace "newsize" by the size wanted (for example: 2G).

```
$ dd if=/dev/zero of=$NETKIT_HOME/fs/netkit-fs-F2.2 bs=1 count=1  
seek="newsize" conv=notrunc
```

Attach the filesystem to a loop device.

```
$ losetup -o 16384 /dev/loop0 $NETKIT_HOME/fs/netkit-fs-F2.2
```

Resize the filesystem.^a

```
$ resize2fs -p /dev/loop0
```

Check if the filesystem is consistent.

```
$ e2fsck -f /dev/loop0
```

Detach from loop device.

```
$ losetup -d /dev/loop0
```

Table B.2: Resizing filesystem.

^a<http://user-mode-linux.sourceforge.net/resize.html>

B.3 Install new software

Once the filesystem expanded, mount it.

```
$ mount -o loop,offset=16384 $NETKIT_HOME/fs/netkit-fs-F2.2 /mnt/loop
```

Mount the /proc filesystem.

```
$ mount -t proc proc /mnt/loop/proc
```

To access to the physical device from the chrooted environment, bind them.

```
$ mount -o bind /dev /mnt/loop/dev
```

To access to the Internet, copy the resolv.conf file

```
$ cp /etc/resolv.conf /mnt/loop/etc/resolv.conf
```

Then, chroot into the filesystem.

```
$ chroot /mnt/loop/ /bin/bash
```

Set the profile.

```
$ source /etc/profile && source /root/.bashrc
```

Modify the /etc/apt/sources.list to have the closest mirror

```
$ vim /etc/apt/sources.list
```

Update the package list and upgrade softwares

```
$ apt-get update && apt-get upgrade
```

Table B.3: Install new software.

QUAGGA CONFIGURATION FILE

Here is an example of the zebra configuration file for IR1:

```
!  
! zebra configuration file  
!  
hostname zebrad  
password root  
enable password root  
service advanced-vty  
!  
!  
log file /var/log/quagga/zebra.log  
debug zebra kernel  
debug zebra events  
!  
line vty  
exec-timeout 0 0  
!  
interface eth0  
mpls labelspace 0  
ip address 192.168.10.1/24  
!  
interface eth1  
mpls labelspace 0  
ip address 192.168.30.1/24  
!
```

Here is an example of the ospfd configuration file for IR1:

```
!  
hostname ospfd  
password root  
enable password root  
!  
log file /var/log/quagga/ospfd.log  
interface lo  
!  
interface eth0  
!  
interface eth1  
!  
router ospf  
network 192.168.10.1/24 area 0  
network 192.168.20.1/24 area 0  
network 192.168.30.1/24 area 0  
network 192.168.40.1/24 area 0  
network 192.168.50.1/24 area 0  
network 192.168.70.1/24 area 0  
network 192.168.80.1/24 area 0  
network 192.168.90.1/24 area 0  
!
```

MPLS COMMAND SYNTAX

```

Usage: mpls ilm CMD label LABEL labelspace NUMBER [proto PROTO | instructions INSTR]
      mpls nhlfe CMD key KEY [mtu MTU propagate_ttl | instructions INSTR]
      mpls xc CMD ilm_label LABEL ilm_labelspace NUMBER nhlfe_key KEY
      mpls labelspace set dev NAME labelspace NUMBER
      mpls labelspace set dev NAME labelspace -1
      mpls tunnel CMD dev NAME nhlfe KEY

      mpls ilm show [label LABEL labelspace NUMBER]
      mpls nhlfe show [key KEY]
      mpls xc show [ilm_label LABEL ilm_labelspace NUMBER]
      mpls labelspace show [dev NAME]
      mpls monitor ...

Where:
CMD      := add | del | change
NUMBER   := 0 .. 255
TYPE     := gen | atm | fr
VALUE    := 16 .. 1048575 | <VPI>/<VCI> | 16 .. 1023
LABEL    := TYPE VALUE
KEY       := 0 for add | previously returned key
NAME      := network device name (i.e. eth0)
PROTO    := ipv4 | ipv6
ADDR     := ipv6 or ipv4 address
NH        := nexthop NAME [none|packet|PROTO ADDR]
FWD       := forward KEY
PUSH      := push LABEL
INSTR     := NH | PUSH | pop | deliver | peek | FWD |
             set-dscp <DSCP> | set-exp <EXP> |
             set-tcindex <TCINDEX> | set-rx-if <NAME>
             forward <KEY> | expfwd <EXP> <KEY> ... |
             exp2tc <EXP> <TCINDEX> ... | exp2ds <EXP> <DSCP> ... |
             nffwd <MASK> [ <NFMARK> <KEY> ... ] |
             nf2exp <MASK> [ <NFMARK> <EXP> ... ] |
             tc2exp <MASK> [ <TCINDEX> <EXP> ... ] |
             ds2exp <MASK> [ <DSCP> <EXP> ... ] |
             dsfwd <MASK> [ <DSCP> <KEY> ... ]

```


AROMA STARTING SCRIPT

```
#!/bin/bash

#####
#
#   This script was generated by NetML
#   and adapt by me to fit my needs.
#   You can easily adapt this script.
#   Mail: julien.bisconti@student.fundp.ac.be
#
#####

SCRIPTNAME='echo $0 | awk -vFS='/' '{ print $NF}''
#EXEC=/hosthome${PWD#$/HOME}${0#'. '}'

allexecute ()
{
    echo "***** SETTING IP FORWARD"
    echo "1" > /proc/sys/net/ipv4/ip_forward
    echo
    echo "***** LOADING MPLS MODULES"
    /sbin/modprobe mpls          # main module
    /sbin/modprobe mpls4        # IPv4
    /sbin/modprobe mpls6        # IPv6
    /sbin/modprobe mplsbr       # bridge
    /sbin/modprobe ebt_mpls     # ebtables bridge
    /sbin/modprobe mpls_tunnel   # tunnel interface
    /sbin/modprobe ip6t_mpls     # iptables IPv6
    /sbin/modprobe ipt_mpls     # iptables IPv4
    echo
    echo "———— MPLS MODULES ———— LOADED"
    mkdir /var/log/quagga
}

start_daemon ()
{
    echo "***** STARTING DEAMONS"
    /usr/local/sbin/zebra -d -A 127.0.0.1 -f /usr/local/etc/zebra.conf
    /usr/local/sbin/ospfd -d -A 127.0.0.1 -f /usr/local/etc/ospfd.conf
    #/usr/local/sbin/ldpd -d -A 127.0.0.1 -f /usr/local/etc/ldpd.conf
}
```

```

    /usr/local/sbin/ldpd -d -A 127.0.0.1
    echo "_____ DEAMONS _____ STARTED"
}
help_exit()
{
    echo "Usage:$0 {start|crash}"
    echo "> lab start "
    echo "Starts all virtual machines for the lab "
    echo "> lab crash "
    echo "Crashes all virtual machines and delete .disk files "
    exit 1
}
#####
##      START / CRASH virtual machines      ##
#####
waitfinish()
{
    SEMAPHORE=$1.booting
    touch $SEMAPHORE
    while [ -f $SEMAPHORE ]; do
        sleep 2
    done
    #sleep 3
}

notifyfinish()
{
    rm $1.booting
}

startvm()
{
    #—append=ubdl=/home/umpls/Netkit/netkit2/fs/my-netkit-fs

X="—new —exec=/hosthome/${PWD#SHOME}/${SCRIPTNAME}"

if      [ -e r_IR1-IR1.disk ] ||
        [ -e r_IR2-IR2.disk ] ||
        [ -e r_CR1-CR1.disk ] ||
        [ -e r_CR2-CR2.disk ] ||
        [ -e r_CR3-CR3.disk ] ||
        [ -e r_CR4-CR4.disk ] ||
        [ -e r_ER-ER.disk ] ; then
    echo "$0: some .disk file exists in this directory!"
    echo " launch "$0 crash" first"
    exit 1
fi

vstart r_SRC-SRC —eth0=SRC1 —eth1=SRC2 $X
waitfinish r_SRC-SRC
vstart r_IR1-IR1 —eth0=OIR1 —eth1=SRC1 $X
waitfinish r_IR1-IR1
vstart r_IR2-IR2 —eth0=OIR2 —eth1=SRC2 $X
waitfinish r_IR2-IR2
vstart r_CR1-CR1 —eth0=OIR1 —eth1=OIR2 —eth2=ICN1 $X
waitfinish r_CR1-CR1
vstart r_CR2-CR2 —eth1=OER1 —eth2=ICN1 $X
waitfinish r_CR2-CR2
vstart r_CR3-CR3 —eth1=ICN2 —eth2=ICN1 $X
waitfinish r_CR3-CR3
vstart r_CR4-CR4 —eth1=ICN2 —eth2=OER1 $X
waitfinish r_CR4-CR4

```



```

vstart r_ER-ER —eth0=OER1 —eth1=DST1 $X
waitfinish r_ER-ER
vstart r_DST-DST —eth0=DST1 $X
echo ''
echo ' *** all machines started ***'
}

```

```

crashvm()
{
    vcrash r_SRC-SRC
    vcrash r_IR1-IR1
    vcrash r_IR2-IR2
    vcrash r_CR1-CR1
    vcrash r_CR2-CR2
    vcrash r_CR3-CR3
    vcrash r_CR4-CR4
    vcrash r_ER-ER
    vcrash r_DST-DST
}

```

```

#####
##### NETWORKING FILES #####
#####
make_ospfd_file()
{
    cat > /usr/local/etc/ospfd.conf << EOF
    !
    hostname ospfd
    password root
    enable password root
    !
    log file /var/log/quagga/ospfd.log
    EOF
}

make_zebra_file()
{
    cat > /usr/local/etc/zebra.conf << EOF
    ! —* zebra —*
    !
    ! zebra configuration file
    !
    hostname zebrad
    password root
    enable password root
    service advanced-vty
    !
    !
    log file /var/log/quagga/zebra.log
    debug zebra kernel
    debug zebra events
    !
    line vty
    exec-timeout 0 0
    !
    EOF
}

make_ldpd_file()
{

```

```

cat > /usr/local/etc/ldpd.conf <<EOF
!
hostname ldpd
password root
enable password root
!
log file /var/log/quagga/ldpd.log
EOF
}

make_quagga_file()
{
make_zebra_file
make_ospfd_file
make_ldpd_file
}

#####
#####          MAIN SCRIPT          #####
#####

if [ "$1" = "start" ]; then startvm
elif [ "$1" = "crash" ]; then
    crashvm
    rm -f *.disk *.booting
elif [ "$1" = "clean" ]; then
    rm -f *.disk *.booting *.log
    echo "All \"*.disk\" \"*.booting\" \"*.log\" files are deleted"
elif [ ! -z $1 ]; then help_exit
elif [ -z $1 ]; then
    if [ 'id -u' != "0" ]; then help_exit
    fi

SCRIPTFILE='cat /proc/cmdline | awk -v FS== -v RS=' ' '{if($1=="exec") print $2}''
SCRIPTDIR=${SCRIPTFILE%/${SCRIPTNAME}}
echo changing directory to $SCRIPTDIR
cd $SCRIPTDIR

case "$HOSTNAME" in

    r_SRC-SRC)
        ### SRC ###
        /sbin/ifconfig eth0 192.168.30.2 netmask 255.255.255.0 up
        /sbin/ifconfig eth1 192.168.40.2 netmask 255.255.255.0 up
        make_quagga_file
        notifyfinish r_SRC-SRC
        allexcute

    # ZEBRA
    echo "!" >> /usr/local/etc/zebra.conf
    echo "interface eth0" >> /usr/local/etc/zebra.conf
    echo "ip address 192.168.30.2/24" >> /usr/local/etc/zebra.conf
    echo "!" >> /usr/local/etc/zebra.conf
    echo "interface eth1" >> /usr/local/etc/zebra.conf
    echo "ip address 192.168.40.2/24" >> /usr/local/etc/zebra.conf
    echo "!" >> /usr/local/etc/zebra.conf

    # OSPFD
    echo "interface lo" >> /usr/local/etc/ospfd.conf
    echo "!" >> /usr/local/etc/ospfd.conf
    echo "interface eth0" >> /usr/local/etc/ospfd.conf

```

```

echo "!" >> /usr/local/etc/ospfd.conf
echo "interface eth1" >> /usr/local/etc/ospfd.conf
echo "!" >> /usr/local/etc/ospfd.conf
echo "router ospf" >> /usr/local/etc/ospfd.conf
echo "network 192.168.10.1/24 area 0" >> /usr/local/etc/ospfd.conf
echo "network 192.168.20.1/24 area 0" >> /usr/local/etc/ospfd.conf
echo "network 192.168.30.1/24 area 0" >> /usr/local/etc/ospfd.conf
echo "network 192.168.40.1/24 area 0" >> /usr/local/etc/ospfd.conf
echo "network 192.168.50.1/24 area 0" >> /usr/local/etc/ospfd.conf
echo "network 192.168.70.1/24 area 0" >> /usr/local/etc/ospfd.conf
echo "network 192.168.80.1/24 area 0" >> /usr/local/etc/ospfd.conf
echo "network 192.168.90.1/24 area 0" >> /usr/local/etc/ospfd.conf
echo "!" >> /usr/local/etc/ospfd.conf

/usr/local/sbin/zebra -d -A 127.0.0.1 -f /usr/local/etc/zebra.conf
/usr/local/sbin/ospfd -d -A 127.0.0.1 -f /usr/local/etc/ospfd.conf

;;

r_IR1-IR1)
### IR1 ###

/sbin/ifconfig eth0 192.168.10.1 netmask 255.255.255.0 up
/sbin/ifconfig eth1 192.168.30.1 netmask 255.255.255.0 up
make_quagga_file
notifyfinish r_IR1-IR1
allexecute

# ZEBRA
echo "!" >> /usr/local/etc/zebra.conf
echo "interface eth0" >> /usr/local/etc/zebra.conf
echo "mpls labelspace 0" >> /usr/local/etc/zebra.conf
echo "ip address 192.168.10.1/24" >> /usr/local/etc/zebra.conf
echo "!" >> /usr/local/etc/zebra.conf
echo "interface eth1" >> /usr/local/etc/zebra.conf
echo "mpls labelspace 0" >> /usr/local/etc/zebra.conf
echo "ip address 192.168.30.1/24" >> /usr/local/etc/zebra.conf
echo "!" >> /usr/local/etc/zebra.conf

# OSPFD
echo "interface lo" >> /usr/local/etc/ospfd.conf
echo "!" >> /usr/local/etc/ospfd.conf
echo "interface eth0" >> /usr/local/etc/ospfd.conf
echo "!" >> /usr/local/etc/ospfd.conf
echo "interface eth1" >> /usr/local/etc/ospfd.conf
echo "!" >> /usr/local/etc/ospfd.conf
echo "router ospf" >> /usr/local/etc/ospfd.conf
echo "network 192.168.10.1/24 area 0" >> /usr/local/etc/ospfd.conf
echo "network 192.168.20.1/24 area 0" >> /usr/local/etc/ospfd.conf
echo "network 192.168.30.1/24 area 0" >> /usr/local/etc/ospfd.conf
echo "network 192.168.40.1/24 area 0" >> /usr/local/etc/ospfd.conf
echo "network 192.168.50.1/24 area 0" >> /usr/local/etc/ospfd.conf
echo "network 192.168.70.1/24 area 0" >> /usr/local/etc/ospfd.conf
echo "network 192.168.80.1/24 area 0" >> /usr/local/etc/ospfd.conf
echo "network 192.168.90.1/24 area 0" >> /usr/local/etc/ospfd.conf
echo "!" >> /usr/local/etc/ospfd.conf

# LDPD
#echo "!" >> /usr/local/etc/ldpd.conf
#echo "interface lo" >> /usr/local/etc/ldpd.conf
#echo "!" >> /usr/local/etc/ldpd.conf

```

```
#echo "interface eth0" >> /usr/local/etc/ldpd.conf
#echo "mpls ip" >> /usr/local/etc/ldpd.conf
#echo "!" >> /usr/local/etc/ldpd.conf
#echo "interface eth1" >> /usr/local/etc/ldpd.conf
#echo "mpls ip" >> /usr/local/etc/ldpd.conf

start_deamon

;;

r_IR2-IR2)
### IR2 ###

/sbin/ifconfig eth0 192.168.20.1 netmask 255.255.255.0 up
/sbin/ifconfig eth1 192.168.40.1 netmask 255.255.255.0 up
make_quagga_file
notifyfinish r_IR2-IR2
allexecute

# ZEBRA
echo "!" >> /usr/local/etc/zebra.conf
echo "interface eth0" >> /usr/local/etc/zebra.conf
echo "mpls labelspace 0" >> /usr/local/etc/zebra.conf
echo "ip address 192.168.20.1/24" >> /usr/local/etc/zebra.conf
echo "!" >> /usr/local/etc/zebra.conf
echo "interface eth1" >> /usr/local/etc/zebra.conf
echo "mpls labelspace 0" >> /usr/local/etc/zebra.conf
echo "ip address 192.168.40.1/24" >> /usr/local/etc/zebra.conf
echo "!" >> /usr/local/etc/zebra.conf

# OSPFD
echo "interface lo" >> /usr/local/etc/ospfd.conf
echo "!" >> /usr/local/etc/ospfd.conf
echo "interface eth0" >> /usr/local/etc/ospfd.conf
echo "!" >> /usr/local/etc/ospfd.conf
echo "interface eth1" >> /usr/local/etc/ospfd.conf
echo "!" >> /usr/local/etc/ospfd.conf
echo "router ospf" >> /usr/local/etc/ospfd.conf
echo "network 192.168.10.1/24 area 0" >> /usr/local/etc/ospfd.conf
echo "network 192.168.20.1/24 area 0" >> /usr/local/etc/ospfd.conf
echo "network 192.168.30.1/24 area 0" >> /usr/local/etc/ospfd.conf
echo "network 192.168.40.1/24 area 0" >> /usr/local/etc/ospfd.conf
echo "network 192.168.50.1/24 area 0" >> /usr/local/etc/ospfd.conf
echo "network 192.168.70.1/24 area 0" >> /usr/local/etc/ospfd.conf
echo "network 192.168.80.1/24 area 0" >> /usr/local/etc/ospfd.conf
echo "network 192.168.90.1/24 area 0" >> /usr/local/etc/ospfd.conf
#echo "network 192.168.20.1/24 area 0" >> /usr/local/etc/ospfd.conf
#echo "network 192.168.40.1/24 area 0" >> /usr/local/etc/ospfd.conf
echo "!" >> /usr/local/etc/ospfd.conf

# LDPD
#echo "!" >> /usr/local/etc/ldpd.conf
#echo "interface lo" >> /usr/local/etc/ldpd.conf
#echo "!" >> /usr/local/etc/ldpd.conf
#echo "interface eth0" >> /usr/local/etc/ldpd.conf
#echo "mpls ip" >> /usr/local/etc/ldpd.conf
#echo "!" >> /usr/local/etc/ldpd.conf
#echo "interface eth1" >> /usr/local/etc/ldpd.conf
#echo "mpls ip" >> /usr/local/etc/ldpd.conf

start_deamon
```

```
;;

r_CR1-CR1)
### CR1 ###

/sbin/ifconfig eth0 192.168.10.2 netmask 255.255.255.0 up
/sbin/ifconfig eth1 192.168.20.2 netmask 255.255.255.0 up
/sbin/ifconfig eth2 192.168.80.1 netmask 255.255.255.0 up
make_quagga_file
notifyfinish r_CR1-CR1
allexecute

# ZEBRA
echo "!" >> /usr/local/etc/zebra.conf
echo "interface eth0" >> /usr/local/etc/zebra.conf
echo "mpls labelspace 0" >> /usr/local/etc/zebra.conf
echo "ip address 192.168.10.2/24" >> /usr/local/etc/zebra.conf
echo "!" >> /usr/local/etc/zebra.conf
echo "interface eth1" >> /usr/local/etc/zebra.conf
echo "mpls labelspace 0" >> /usr/local/etc/zebra.conf
echo "ip address 192.168.20.2/24" >> /usr/local/etc/zebra.conf
echo "!" >> /usr/local/etc/zebra.conf
echo "interface eth2" >> /usr/local/etc/zebra.conf
echo "mpls labelspace 0" >> /usr/local/etc/zebra.conf
echo "ip address 192.168.80.1/24" >> /usr/local/etc/zebra.conf

# OSPFD
echo "interface lo" >> /usr/local/etc/ospfd.conf
echo "!" >> /usr/local/etc/ospfd.conf
echo "interface eth0" >> /usr/local/etc/ospfd.conf
echo "!" >> /usr/local/etc/ospfd.conf
echo "interface eth1" >> /usr/local/etc/ospfd.conf
echo "!" >> /usr/local/etc/ospfd.conf
echo "interface eth2" >> /usr/local/etc/ospfd.conf
echo "!" >> /usr/local/etc/ospfd.conf
echo "router ospf" >> /usr/local/etc/ospfd.conf
echo "network 192.168.10.1/24 area 0" >> /usr/local/etc/ospfd.conf
echo "network 192.168.20.1/24 area 0" >> /usr/local/etc/ospfd.conf
echo "network 192.168.30.1/24 area 0" >> /usr/local/etc/ospfd.conf
echo "network 192.168.40.1/24 area 0" >> /usr/local/etc/ospfd.conf
echo "network 192.168.50.1/24 area 0" >> /usr/local/etc/ospfd.conf
echo "network 192.168.70.1/24 area 0" >> /usr/local/etc/ospfd.conf
echo "network 192.168.80.1/24 area 0" >> /usr/local/etc/ospfd.conf
echo "network 192.168.90.1/24 area 0" >> /usr/local/etc/ospfd.conf
#echo "network 192.168.10.2/24 area 0" >> /usr/local/etc/ospfd.conf
#echo "network 192.168.20.2/24 area 0" >> /usr/local/etc/ospfd.conf
#echo "network 192.168.80.1/24 area 0" >> /usr/local/etc/ospfd.conf
echo "!" >> /usr/local/etc/ospfd.conf

# LDPD
#echo "!" >> /usr/local/etc/ldpd.conf
#echo "interface lo" >> /usr/local/etc/ldpd.conf
#echo "!" >> /usr/local/etc/ldpd.conf
#echo "interface eth0" >> /usr/local/etc/ldpd.conf
#echo "mpls ip" >> /usr/local/etc/ldpd.conf
#echo "!" >> /usr/local/etc/ldpd.conf
#echo "interface eth1" >> /usr/local/etc/ldpd.conf
#echo "mpls ip" >> /usr/local/etc/ldpd.conf
#echo "!" >> /usr/local/etc/ldpd.conf
#echo "interface eth2" >> /usr/local/etc/ldpd.conf
#echo "mpls ip" >> /usr/local/etc/ldpd.conf
start_deamon
```

```

;;

r_CR2-CR2)
### CR2 ###

/sbin/ifconfig eth1 192.168.50.1 netmask 255.255.255.0 up
/sbin/ifconfig eth2 192.168.80.2 netmask 255.255.255.0 up
make_quagga_file
notifyfinish r_CR2-CR2
allexecute

# ZEBRA
echo "!" >> /usr/local/etc/zebra.conf
echo "interface eth1" >> /usr/local/etc/zebra.conf
echo "mpls labelspace 0" >> /usr/local/etc/zebra.conf
echo "ip address 192.168.50.1/24" >> /usr/local/etc/zebra.conf
echo "!" >> /usr/local/etc/zebra.conf
echo "interface eth2" >> /usr/local/etc/zebra.conf
echo "mpls labelspace 0" >> /usr/local/etc/zebra.conf
echo "ip address 192.168.80.2/24" >> /usr/local/etc/zebra.conf
echo "!" >> /usr/local/etc/zebra.conf

# OSPFD
echo "interface lo" >> /usr/local/etc/ospfd.conf
echo "!" >> /usr/local/etc/ospfd.conf
echo "interface eth0" >> /usr/local/etc/ospfd.conf
echo "!" >> /usr/local/etc/ospfd.conf
echo "interface eth2" >> /usr/local/etc/ospfd.conf
echo "!" >> /usr/local/etc/ospfd.conf
echo "router ospf" >> /usr/local/etc/ospfd.conf
echo "network 192.168.10.1/24 area 0" >> /usr/local/etc/ospfd.conf
echo "network 192.168.20.1/24 area 0" >> /usr/local/etc/ospfd.conf
echo "network 192.168.30.1/24 area 0" >> /usr/local/etc/ospfd.conf
echo "network 192.168.40.1/24 area 0" >> /usr/local/etc/ospfd.conf
echo "network 192.168.50.1/24 area 0" >> /usr/local/etc/ospfd.conf
echo "network 192.168.70.1/24 area 0" >> /usr/local/etc/ospfd.conf
echo "network 192.168.80.1/24 area 0" >> /usr/local/etc/ospfd.conf
echo "network 192.168.90.1/24 area 0" >> /usr/local/etc/ospfd.conf
#echo "network 192.168.50.1/24 area 0" >> /usr/local/etc/ospfd.conf
echo "network 192.168.80.2/24 area 0" >> /usr/local/etc/ospfd.conf
echo "!" >> /usr/local/etc/ospfd.conf

# LDPD
#echo "!" >> /usr/local/etc/ldpd.conf
#echo "interface lo" >> /usr/local/etc/ldpd.conf
#echo "!" >> /usr/local/etc/ldpd.conf
#echo "interface eth1" >> /usr/local/etc/ldpd.conf
#echo "mpls ip" >> /usr/local/etc/ldpd.conf
#echo "!" >> /usr/local/etc/ldpd.conf
#echo "interface eth2" >> /usr/local/etc/ldpd.conf
#echo "mpls ip" >> /usr/local/etc/ldpd.conf

start_daemon

;;

r_CR3-CR3)
### CR3 ###

/sbin/ifconfig eth1 192.168.90.1 netmask 255.255.255.0 up
/sbin/ifconfig eth2 192.168.80.3 netmask 255.255.255.0 up

```

```

make_quagga_file
notifyfinish r_CR3-CR3
allexecute

# ZEBRA
echo "!" >> /usr/local/etc/zebra.conf
echo "interface eth1" >> /usr/local/etc/zebra.conf
echo "mpls labelspace 0" >> /usr/local/etc/zebra.conf
echo "ip address 192.168.90.1/24" >> /usr/local/etc/zebra.conf
echo "!" >> /usr/local/etc/zebra.conf
echo "interface eth2" >> /usr/local/etc/zebra.conf
echo "mpls labelspace 0" >> /usr/local/etc/zebra.conf
echo "ip address 192.168.80.3/24" >> /usr/local/etc/zebra.conf
echo "!" >> /usr/local/etc/zebra.conf

# OSPFD
echo "interface lo" >> /usr/local/etc/ospfd.conf
echo "!" >> /usr/local/etc/ospfd.conf
echo "interface eth1" >> /usr/local/etc/ospfd.conf
echo "!" >> /usr/local/etc/ospfd.conf
echo "interface eth2" >> /usr/local/etc/ospfd.conf
echo "!" >> /usr/local/etc/ospfd.conf
echo "router ospf" >> /usr/local/etc/ospfd.conf
echo "network 192.168.10.1/24 area 0" >> /usr/local/etc/ospfd.conf
echo "network 192.168.20.1/24 area 0" >> /usr/local/etc/ospfd.conf
echo "network 192.168.30.1/24 area 0" >> /usr/local/etc/ospfd.conf
echo "network 192.168.40.1/24 area 0" >> /usr/local/etc/ospfd.conf
echo "network 192.168.50.1/24 area 0" >> /usr/local/etc/ospfd.conf
echo "network 192.168.70.1/24 area 0" >> /usr/local/etc/ospfd.conf
echo "network 192.168.80.1/24 area 0" >> /usr/local/etc/ospfd.conf
echo "network 192.168.90.1/24 area 0" >> /usr/local/etc/ospfd.conf
#echo "network 192.168.90.1/24 area 0" >> /usr/local/etc/ospfd.conf
#echo "network 192.168.80.3/24 area 0" >> /usr/local/etc/ospfd.conf
echo "!" >> /usr/local/etc/ospfd.conf

# LDPD
#echo "!" >> /usr/local/etc/ldpd.conf
#echo "interface lo" >> /usr/local/etc/ldpd.conf
#echo "!" >> /usr/local/etc/ldpd.conf
#echo "interface eth1" >> /usr/local/etc/ldpd.conf
#echo "mpls ip" >> /usr/local/etc/ldpd.conf
#echo "!" >> /usr/local/etc/ldpd.conf
#echo "interface eth2" >> /usr/local/etc/ldpd.conf
#echo "mpls ip" >> /usr/local/etc/ldpd.conf

start_daemon

;;

r_CR4-CR4)
### CR4 ###

/sbin/ifconfig eth1 192.168.90.2 netmask 255.255.255.0 up
/sbin/ifconfig eth2 192.168.50.3 netmask 255.255.255.0 up
make_quagga_file
notifyfinish r_CR4-CR4
allexecute

# ZEBRA
echo "!" >> /usr/local/etc/zebra.conf
echo "interface eth1" >> /usr/local/etc/zebra.conf

```

```

echo "mpls labelspace 0" >> /usr/local/etc/zebra.conf
echo "ip address 192.168.90.2/24" >> /usr/local/etc/zebra.conf
echo "!" >> /usr/local/etc/zebra.conf
echo "interface eth2" >> /usr/local/etc/zebra.conf
echo "mpls labelspace 0" >> /usr/local/etc/zebra.conf
echo "ip address 192.168.50.3/24" >> /usr/local/etc/zebra.conf
echo "!" >> /usr/local/etc/zebra.conf

# OSPFD
echo "interface lo" >> /usr/local/etc/ospfd.conf
echo "!" >> /usr/local/etc/ospfd.conf
echo "interface eth1" >> /usr/local/etc/ospfd.conf
echo "!" >> /usr/local/etc/ospfd.conf
echo "interface eth2" >> /usr/local/etc/ospfd.conf
echo "!" >> /usr/local/etc/ospfd.conf
echo "router ospf" >> /usr/local/etc/ospfd.conf
echo "network 192.168.10.1/24 area 0" >> /usr/local/etc/ospfd.conf
echo "network 192.168.20.1/24 area 0" >> /usr/local/etc/ospfd.conf
echo "network 192.168.30.1/24 area 0" >> /usr/local/etc/ospfd.conf
echo "network 192.168.40.1/24 area 0" >> /usr/local/etc/ospfd.conf
echo "network 192.168.50.1/24 area 0" >> /usr/local/etc/ospfd.conf
echo "network 192.168.70.1/24 area 0" >> /usr/local/etc/ospfd.conf
echo "network 192.168.80.1/24 area 0" >> /usr/local/etc/ospfd.conf
echo "network 192.168.90.1/24 area 0" >> /usr/local/etc/ospfd.conf
#echo "network 192.168.90.2/24 area 0" >> /usr/local/etc/ospfd.conf
#echo "network 192.168.50.3/24 area 0" >> /usr/local/etc/ospfd.conf
echo "!" >> /usr/local/etc/ospfd.conf

# LDPD
#echo "!" >> /usr/local/etc/ldpd.conf
#echo "interface lo" >> /usr/local/etc/ldpd.conf
#echo "!" >> /usr/local/etc/ldpd.conf
#echo "interface eth1" >> /usr/local/etc/ldpd.conf
#echo "mpls ip" >> /usr/local/etc/ldpd.conf
#echo "!" >> /usr/local/etc/ldpd.conf
#echo "interface eth2" >> /usr/local/etc/ldpd.conf
#echo "mpls ip" >> /usr/local/etc/ldpd.conf

start_deamon

;;

r_ER-ER)
### ER ###

/sbin/ifconfig eth0 192.168.50.2 netmask 255.255.255.0 up
/sbin/ifconfig eth1 192.168.70.1 netmask 255.255.255.0 up
make_quagga_file
notifyfinish r_ER-ER
allexecute

# ZEBRA
echo "!" >> /usr/local/etc/zebra.conf
echo "interface eth0" >> /usr/local/etc/zebra.conf
echo "mpls labelspace 0" >> /usr/local/etc/zebra.conf
echo "ip address 192.168.50.0/24" >> /usr/local/etc/zebra.conf
echo "!" >> /usr/local/etc/zebra.conf
echo "interface eth1" >> /usr/local/etc/zebra.conf
echo "mpls labelspace 0" >> /usr/local/etc/zebra.conf
echo "ip address 192.168.70.1/24" >> /usr/local/etc/zebra.conf
echo "!" >> /usr/local/etc/zebra.conf

```



```

# OSPFD
echo "interface lo" >> /usr/local/etc/ospfd.conf
echo "!" >> /usr/local/etc/ospfd.conf
echo "interface eth0" >> /usr/local/etc/ospfd.conf
echo "!" >> /usr/local/etc/ospfd.conf
echo "interface eth1" >> /usr/local/etc/ospfd.conf
echo "!" >> /usr/local/etc/ospfd.conf
echo "router ospf" >> /usr/local/etc/ospfd.conf
echo "network 192.168.10.1/24 area 0" >> /usr/local/etc/ospfd.conf
echo "network 192.168.20.1/24 area 0" >> /usr/local/etc/ospfd.conf
echo "network 192.168.30.1/24 area 0" >> /usr/local/etc/ospfd.conf
echo "network 192.168.40.1/24 area 0" >> /usr/local/etc/ospfd.conf
echo "network 192.168.50.1/24 area 0" >> /usr/local/etc/ospfd.conf
echo "network 192.168.70.1/24 area 0" >> /usr/local/etc/ospfd.conf
echo "network 192.168.80.1/24 area 0" >> /usr/local/etc/ospfd.conf
echo "network 192.168.90.1/24 area 0" >> /usr/local/etc/ospfd.conf
#echo "network 192.168.50.2/24 area 0" >> /usr/local/etc/ospfd.conf
#echo "network 192.168.70.1/24 area 0" >> /usr/local/etc/ospfd.conf
echo "!" >> /usr/local/etc/ospfd.conf

# LDPD
echo "!" >> /usr/local/etc/ldpd.conf
echo "interface lo" >> /usr/local/etc/ldpd.conf
echo "!" >> /usr/local/etc/ldpd.conf
echo "interface eth0" >> /usr/local/etc/ldpd.conf
echo "mpls ip" >> /usr/local/etc/ldpd.conf
echo "!" >> /usr/local/etc/ldpd.conf
echo "interface eth1" >> /usr/local/etc/ldpd.conf
echo "mpls ip" >> /usr/local/etc/ldpd.conf

start_daemon

;;

r_DST=DST)
### DST ###

/sbin/ifconfig eth0 192.168.70.2 netmask 255.255.255.0 up
notifyfinish r_DST=DST
make_quagga_file
allexecute

# ZEBRA
echo "!" >> /usr/local/etc/zebra.conf
echo "interface eth0" >> /usr/local/etc/zebra.conf
echo "ip address 192.168.70.2/24" >> /usr/local/etc/zebra.conf
echo "!" >> /usr/local/etc/zebra.conf

# OSPFD
echo "interface lo" >> /usr/local/etc/ospfd.conf
echo "!" >> /usr/local/etc/ospfd.conf
echo "interface eth0" >> /usr/local/etc/ospfd.conf
echo "!" >> /usr/local/etc/ospfd.conf
echo "interface eth1" >> /usr/local/etc/ospfd.conf
echo "!" >> /usr/local/etc/ospfd.conf
echo "router ospf" >> /usr/local/etc/ospfd.conf
echo "network 192.168.10.1/24 area 0" >> /usr/local/etc/ospfd.conf
echo "network 192.168.20.1/24 area 0" >> /usr/local/etc/ospfd.conf
echo "network 192.168.30.1/24 area 0" >> /usr/local/etc/ospfd.conf
echo "network 192.168.40.1/24 area 0" >> /usr/local/etc/ospfd.conf
echo "network 192.168.50.1/24 area 0" >> /usr/local/etc/ospfd.conf
echo "network 192.168.70.1/24 area 0" >> /usr/local/etc/ospfd.conf

```

```
echo "network 192.168.80.1/24 area 0" >> /usr/local/etc/ospfd.conf
echo "network 192.168.90.1/24 area 0" >> /usr/local/etc/ospfd.conf
#echo "network 192.168.70.1/24 area 0" >> /usr/local/etc/ospfd.conf
echo "!" >> /usr/local/etc/ospfd.conf
```

```
/usr/local/sbin/zebra -d -A 127.0.0.1 -f /usr/local/etc/zebra.conf
/usr/local/sbin/ospfd -d -A 127.0.0.1 -f /usr/local/etc/ospfd.conf
```

```
;;
```

```
*)
```

```
echo "error:don\'t know how to configure $HOSTNAME "
```

```
exit 1
```

```
esac
```

```
fi
```

REROUTING SCRIPT

F.1 Setup Script

```
#!/bin/bash

#####
#
#      AROMA TESTBED
#      PREEMPTION of LSP.
#      Mail: julien.bisconti@student.fundp.ac.be
#
#####

SCRIPTNAME='echo $0 | awk -vFS='/' '{ print $NF}''
#EXEC=/hosthome${PWD#HOME}${0#'. '}

allexecute()
{
    echo "***** SETTING IP FORWARD"
    echo "1" > /proc/sys/net/ipv4/ip_forward
    echo "***** TURNING OFF MPLS DEBUG"
    echo "0" > /sys/mpls/debug
    echo "***** LOADING MPLS MODULES"
    /sbin/modprobe mpls      # main module
    /sbin/modprobe mpls4     # IPv4
    /sbin/modprobe mpls6     # IPv6
    /sbin/modprobe mplsbr    # bridge
    /sbin/modprobe ebt_mpls  # ebtables bridge
    /sbin/modprobe mpls_tunnel # tunnel interface
    /sbin/modprobe ip6t_mpls  # iptables IPv6
    /sbin/modprobe ipt_mpls  # iptables IPv4
    echo
    echo "———— MPLS MODULES ———— LOADED"
}

help_exit()
{
    echo "Usage: $0 {start | crash}"
    echo "> lab start "
}
```

```

    echo "Starts all virtual machines for the lab "
    echo "> lab crash "
    echo "Crashes all virtual machines and delete .disk files "
    exit 1
}
#####
##      START / CRASH virtual machines      ##
#####
waitfinish ()
{
    SEMAPHORE=$1.booting
    touch $SEMAPHORE
    while [ -f $SEMAPHORE ]; do
        sleep 1
    done
    #sleep 3
}

notifyfinish ()
{
    rm $1.booting
}

startvm ()
{
    #In case you want to add another filesystem to your vm.
    #—append=ubdl=/home/umpls/Netkit/netkit2/fs/my-netkit—fs

X="—new —exec=/hosthome/${PWD#$HOME}/${SCRIPTNAME}"

if      [ -e r_IR1-IR1.disk ] ||
        [ -e r_IR2-IR2.disk ] ||
        [ -e r_CR1-CR1.disk ] ||
        [ -e r_CR2-CR2.disk ] ||
        [ -e r_CR3-CR3.disk ] ||
        [ -e r_CR4-CR4.disk ] ||
        [ -e r_ER-ER.disk ] ; then
    echo "$0: some .disk file exists in this directory!"
    echo " launch "$0 crash" first"
    exit 1
fi

vstart r_SRC-SRC —eth0=SRC1 —eth1=SRC2 $X
waitfinish r_SRC-SRC
vstart r_IR1-IR1 —eth0=OIR1 —eth1=SRC1 $X
waitfinish r_IR1-IR1
vstart r_IR2-IR2 —eth0=OIR2 —eth1=SRC2 $X
waitfinish r_IR2-IR2
vstart r_CR1-CR1 —eth0=OIR1 —eth1=OIR2 —eth2=ICN1 $X
waitfinish r_CR1-CR1
vstart r_CR2-CR2 —eth1=OER1 —eth2=ICN1 $X
waitfinish r_CR2-CR2
vstart r_CR3-CR3 —eth1=ICN2 —eth2=ICN1 $X
waitfinish r_CR3-CR3
vstart r_CR4-CR4 —eth1=ICN2 —eth2=OER1 $X
waitfinish r_CR4-CR4
vstart r_ER-ER —eth0=OER1 —eth1=DST1 $X
waitfinish r_ER-ER
vstart r_DST-DST —eth0=DST1 $X
echo ''
echo ' *** all machines started *** '
}

```

```

crashvm()
{
    vcrash r_SRC-SRC
    vcrash r_IR1-IR1
    vcrash r_IR2-IR2
    vcrash r_CR1-CR1
    vcrash r_CR2-CR2
    vcrash r_CR3-CR3
    vcrash r_CR4-CR4
    vcrash r_ER-ER
    vcrash r_DST-DST
}

#####
#####          MPLS FUNCTIONS          #####
#####

# SETUP two LSPs, each one is a 2-way LSP.
mplsconf()
{
    case "$HOSTNAME" in
        r_SRC-SRC)

        # routing info
        ip route add 192.168.70.2 via 192.168.30.1 # DST via IR1

        ;;

#-----
    r_IR1-IR1)

        # A      DOWNSTREAM      SRC->CR1      eth1->eth0
        # Create a NHLFE entry to add label 1000 and
        # forward the packets to 192.168.10.2 using outgoing interface eth0.
        key_1='mpls nhlfe add key 0 instructions push gen 1000 nexthop eth0 \
        ipv4 192.168.10.2 | grep key | cut -c 17-26' #(returns key 0x2)

        #FEC to NHLFE
        /usr/sbin/ip route add 192.168.70.2/32 via 192.168.30.1 mpls $key_1

        # A      UPSTREAM      CR1->SRC      eth0->eth1
        mpls labelspace set dev eth0 labelspace 0
        mpls ilm add label gen 1007 labelspace 0
        key_2='mpls nhlfe add key 0 instructions nexthop eth1 \
        ipv4 192.168.30.2 | grep key | cut -c 17-26' #(returns key 0x3)
        mpls xc add ilm_label gen 1007 ilm_labelspace 0 nhlfe_key $key_2

        # B      DOWNSTREAM      SRC->CR1      eth1->eth0
        key_3='mpls nhlfe add key 0 instructions push gen 2000 nexthop eth0 \
        ipv4 192.168.10.2 | grep key | cut -c 17-26' #(returns key 0x4)

        #/usr/sbin/ip route add 192.168.70.2/32 via 192.168.30.1 mpls 0x4

        # B      UPSTREAM      CR1->SRC      eth0->eth1
        mpls labelspace set dev eth0 labelspace 0
        mpls ilm add label gen 2005 labelspace 0
        key_4='mpls nhlfe add key 0 instructions nexthop eth1 ipv4 192.168.30.2 | grep key | cut -c 17-26'
        #(returns key 0x5)
        mpls xc add ilm_label gen 2005 ilm_labelspace 0 nhlfe_key $key_4

```

```

;;
#-----
r_IR2-IR2)
#Not really useful right now!!
;;
#-----
r_CR1-CR1)

# A      DOWNSTREAM      IR1->CR3      eth0->eth2
mpls labelspace set dev eth0 labelspace 0
mpls ilm add label gen 1000 labelspace 0
key_1='mpls nhlfe add key 0 instructions push gen 1001 nexthop eth2 \
ipv4 192.168.80.3 | grep key | cut -c 17-26' # (returns key 0x2)
mpls xc add ilm_label gen 1000 ilm_labelspace 0 nhlfe_key $key_1

# A      UPSTREAM      CR3->IR1      eth2->eth0
mpls labelspace set dev eth2 labelspace 0
mpls ilm add label gen 1006 labelspace 0
key_2='mpls nhlfe add key 0 instructions push gen 1007 nexthop eth0 \
ipv4 192.168.10.1 | grep key | cut -c 17-26' # (returns key 0x3)
mpls xc add ilm_label gen 1006 ilm_labelspace 0 nhlfe_key $key_2

# B      DOWNSTREAM      IR1->CR2      eth0->eth2
mpls labelspace set dev eth0 labelspace 0
mpls ilm add label gen 2000 labelspace 0
key_3='mpls nhlfe add key 0 instructions push gen 2001 nexthop eth2 \
ipv4 192.168.80.2 | grep key | cut -c 17-26' # (returns key 0x4)
mpls xc add ilm_label gen 2000 ilm_labelspace 0 nhlfe_key $key_3

# B      UPSTREAM      CR2->IR1      eth2->eth0
mpls labelspace set dev eth2 labelspace 0
mpls ilm add label gen 2004 labelspace 0
key_4='mpls nhlfe add key 0 instructions push gen 2005 nexthop eth0 \
ipv4 192.168.10.1 | grep key | cut -c 17-26' # (returns key 0x5)
mpls xc add ilm_label gen 2004 ilm_labelspace 0 nhlfe_key $key_4

;;
#-----
r_CR2-CR2)

# B      DOWNSTREAM      CR1->ER      eth2->eth1
mpls labelspace set dev eth2 labelspace 0
mpls ilm add label gen 2001 labelspace 0
key_1='mpls nhlfe add key 0 instructions push gen 2002 nexthop eth1 \
ipv4 192.168.50.2 | grep key | cut -c 17-26' # (returns key 0x2)
mpls xc add ilm_label gen 2001 ilm_labelspace 0 nhlfe_key $key_1

# B      UPSTREAM      ER->CR1      eth1->eth2
mpls labelspace set dev eth1 labelspace 0
mpls ilm add label gen 2003 labelspace 0
key_2='mpls nhlfe add key 0 instructions push gen 2004 nexthop eth2 \
ipv4 192.168.80.1 | grep key | cut -c 17-26' # (returns key 0x3)
mpls xc add ilm_label gen 2003 ilm_labelspace 0 nhlfe_key $key_2

;;
#-----
r_CR3-CR3)

# A      DOWNSTREAM      CR1->CR4      eth2->eth1
mpls labelspace set dev eth2 labelspace 0
mpls ilm add label gen 1001 labelspace 0

```

```

key_1='mpls nhlfe add key 0 instructions push gen 1002 nexthop eth1 \
ipv4 192.168.90.2 | grep key | cut -c 17-26' $(returns key 0x2)
mpls xc add ilm_label gen 1001 ilm_labelspace 0 nhlfe_key $key_1

# A      UPSTREAM      CR4—>CR1      eth1->eth2
mpls labelspace set dev eth1 labelspace 0
mpls ilm add label gen 1005 labelspace 0
key_2='mpls nhlfe add key 0 instructions push gen 1006 nexthop eth2 \
ipv4 192.168.80.1 | grep key | cut -c 17-26' $(returns key 0x3)
mpls xc add ilm_label gen 1005 ilm_labelspace 0 nhlfe_key $key_2

;;

#-----
r_CR4-CR4)

# A      DOWNSTREAM    CR3—>ER      eth1->eth2
mpls labelspace set dev eth1 labelspace 0
mpls ilm add label gen 1002 labelspace 0
key_1='mpls nhlfe add key 0 instructions push gen 1003 nexthop eth2 \
ipv4 192.168.50.2 | grep key | cut -c 17-26' $(returns key 0x2)
mpls xc add ilm_label gen 1002 ilm_labelspace 0 nhlfe_key $key_1

# A      UPSTREAM      ER—>CR3      eth2->eth1
mpls labelspace set dev eth2 labelspace 0
mpls ilm add label gen 1004 labelspace 0
key_2='mpls nhlfe add key 0 instructions push gen 1005 nexthop eth1 \
ipv4 192.168.90.1 | grep key | cut -c 17-26' $(returns key 0x3)
mpls xc add ilm_label gen 1004 ilm_labelspace 0 nhlfe_key $key_2

;;

#-----
r_ER-ER)

# A      DOWNSTREAM    CR4—>DST      eth0->eth1
mpls labelspace set dev eth0 labelspace 0
mpls ilm add label gen 1003 labelspace 0
key_1='mpls nhlfe add key 0 instructions nexthop eth1 \
ipv4 192.168.70.2 | grep key | cut -c 17-26' $(returns key 0x2)
mpls xc add ilm_label gen 1003 ilm_labelspace 0 nhlfe_key $key_1

# A      UPSTREAM      DST—>CR4      eth1->eth0
key_2='mpls nhlfe add key 0 instructions push gen 1004 nexthop eth0 \
ipv4 192.168.50.3 | grep key | cut -c 17-26' $(returns key 0x3)
/usr/sbin/ip route add 192.168.30.2/32 via 192.168.70.1 mpls $key_2

# B      DOWNSTREAM    CR4—>DST      eth0->eth1
mpls labelspace set dev eth0 labelspace 0
mpls ilm add label gen 2002 labelspace 0
key_3='mpls nhlfe add key 0 instructions nexthop eth1 \
ipv4 192.168.70.2 | grep key | cut -c 17-26' $(returns key 0x4)
mpls xc add ilm_label gen 2002 ilm_labelspace 0 nhlfe_key $key_3

# B      UPSTREAM      DST—>CR4      eth1->eth0
key_4='mpls nhlfe add key 0 instructions push gen 2003 nexthop eth0 \
ipv4 192.168.50.1 | grep key | cut -c 17-26' $(returns key 0x5)

#usr/sbin/ip route add 192.168.30.2/32 via 192.168.70.1 mpls $key_4

;;

#-----
r_DST-DST)
ip route add 192.168.30.2 via 192.168.70.1 # SRC via ER

```

```

;;
*)
exit 1

esac

}

#####
#####          MAIN SCRIPT          #####
#####

if [ "$1" = "start" ]; then
    startvm

elif [ "$1" = "crash" ]; then
    crashvm
    rm -f *.disk *.booting

elif [ "$1" = "clean" ]; then
    rm -f *.disk *.booting *.log
    echo "All \"*.disk\" \"*.booting\" \"*.log\" files are deleted"

elif [ ! -z $1 ]; then
    help_exit

elif [ -z $1 ]; then
    if [ 'id -u' != "0" ]; then
        help_exit
    fi

    SCRIPTFILE='cat /proc/cmdline | awk -v FS== -v RS=' ' '{if($1=="exec") print $2}' '
    SCRIPTDIR=${SCRIPTFILE%/${SCRIPTNAME}}
    echo changing directory to $SCRIPTDIR
    cd $SCRIPTDIR

    case "$HOSTNAME" in

        r_SRC-SRC)

### SRC ###

        /sbin/ifconfig eth0 192.168.30.2 netmask 255.255.255.0 up
        /sbin/ifconfig eth1 192.168.40.2 netmask 255.255.255.0 up
        notifyfinish r_SRC-SRC
        alleexecute
        mplsconf

        ;;

        r_IR1-IR1)

### IR1 ###

        /sbin/ifconfig eth0 192.168.10.1 netmask 255.255.255.0 up
        /sbin/ifconfig eth1 192.168.30.1 netmask 255.255.255.0 up
        notifyfinish r_IR1-IR1
        alleexecute
        mplsconf

        ;;

```



```

r_IR2-IR2)
### IR2 ###

/sbin/ifconfig eth0 192.168.20.1 netmask 255.255.255.0 up
/sbin/ifconfig eth1 192.168.40.1 netmask 255.255.255.0 up
notifyfinish r_IR2-IR2
allexecute
mplsconf

;;

r_CR1-CR1)
### CR1 ###

/sbin/ifconfig eth0 192.168.10.2 netmask 255.255.255.0 up
/sbin/ifconfig eth1 192.168.20.2 netmask 255.255.255.0 up
/sbin/ifconfig eth2 192.168.80.1 netmask 255.255.255.0 up
notifyfinish r_CR1-CR1
allexecute
mplsconf

;;

r_CR2-CR2)
### CR2 ###

/sbin/ifconfig eth1 192.168.50.1 netmask 255.255.255.0 up
/sbin/ifconfig eth2 192.168.80.2 netmask 255.255.255.0 up
notifyfinish r_CR2-CR2
allexecute
mplsconf

;;

r_CR3-CR3)
### CR3 ###

/sbin/ifconfig eth1 192.168.90.1 netmask 255.255.255.0 up
/sbin/ifconfig eth2 192.168.80.3 netmask 255.255.255.0 up
notifyfinish r_CR3-CR3
allexecute
mplsconf

;;

r_CR4-CR4)
### CR4 ###

/sbin/ifconfig eth1 192.168.90.2 netmask 255.255.255.0 up
/sbin/ifconfig eth2 192.168.50.3 netmask 255.255.255.0 up
notifyfinish r_CR4-CR4
allexecute
mplsconf

;;

r_ER-ER)
### ER ###

/sbin/ifconfig eth0 192.168.50.2 netmask 255.255.255.0 up

```

```
        /sbin/ifconfig eth1 192.168.70.1 netmask 255.255.255.0 up
        notifyfinish r_ER-ER
        allexecute
        mplsconf

        ;;

        r_DST-DST)

### DST ###

        /sbin/ifconfig eth0 192.168.70.2 netmask 255.255.255.0 up
        notifyfinish r_DST-DST
        allexecute
        mplsconf

        ;;

        *)
        echo "error:don\'t know how to configure $HOSTNAME "
        exit 1
    esac
fi
```