Link Failure Recovery

for MPLS Networks with Multicasting

A Thesis

Presented to

the faculty of the School of Engineering and Applied Science

University of Virginia

In Partial Fulfillment

of the requirements for the Degree

Master of Science

Computer Science

by

Yvan Pointurier

August 2002

Approvals

This thesis is submitted in partial fulfillment of the requirements for the degree of Master of Science

Yvan Pointurier

Approved:

Jörg Liebeherr (Advisor)

Stephen D. Patek (Chair)

Maïté Brandt-Pearce

Accepted by the School of Engineering and Applied Science:

Richard W. Miksad (Dean)

August 2002

Abstract

Link failures are a common cause of service disruption in computer networks. When a link of a network fails, all communications which were using the failed link are temporarily interrupted. Techniques have been engineered to alleviate the consequences of hardware failure in a network by rerouting traffic from the failed link to other links. When performed by low communication layers, rerouting is fast but expensive as additional hardware is needed. On the other hand, it is possible to reroute traffic at higher layers using software mechanisms, but such solutions prove to be slow. Moreover, most rerouting schemes are not optimized for multicasting applications such as teleconferencing where one sender can send data to several receivers.

The Internet is a datagram packet switching network where data is carried in IP packets. Recently, Multiprotocol Label Switching (MPLS) has been devised to carry IP packets over virtual circuits, thus combining the advantages of datagram packet switching and virtual circuit switching. In this thesis, we devise a solution to protect multicast communications in MPLS networks from a link failure. We present a graph algorithm which selects a backup path in a multicast routing tree that carries multicast traffic. The backup path aims at minimizing the number of receivers of the multicast routing tree that are dropped from the communication if a single link of the tree fails. We present MPLS multicast Fast Reroute, a new mechanism for MPLS networks which reroutes traffic over a backup path when a link of the multicast routing tree fails. We provide an implementation of MPLS multicast Fast Reroute on PC routers running the Linux operating system. We experimentally show that MPLS multicast Fast Reroute can repair multicast routing trees in less than 50 ms, making link failures unnoticeable to all end users.

Acknowledgements

First, I would like to thank here my advisor Dr Jörg Liebeherr, who gave me the opportunity to work on an this research problem, did not fail in trusting in my commitment, and patiently accepted to review hundreds of pages of the different versions of this thesis. Of course, this work would not have been possible without the support of his full research group, the Multimedia Networks Group (MNG). Working as a member of the MNGroup has truly been enjoyable.

I also do not forget that my presence here has been made possible by my former institution, the French Engineer School "Ecole Centrale de Lille", which does not let many students leave to the USA to pursue graduate studies.

Last but not least, thanks to my parents who have always been dedicated to my academic success.

Contents

List of Figures

List of Tables

1

Introduction

Many characteristics of switched communication networks are directly dependent on how data is relayed over the wires. Early networks carried continuous bitstreams over physical links in a technique called circuit switching, well suited to transmit voice or real time data from a single sender to a single receiver (unicast communication). However, a physical link failure in circuit switching networks has dramatic consequences leading to the interruption of all communications using the failed link. Datagram packet switching networks like the Internet fix these drawbacks by cutting data into small chunks called *packets*. In datagram packet switching networks, two consecutive packets from the same communication are independently handled by the network. Therefore, when a link fails, packets previously sent on the failed link can be *rerouted* to avoid the failed link and communications are not interrupted. Datagram packet switching networks are said to be *resilient* to link failures because link failures are hidden to end-users. On the other hand, it is more difficult to manage end-to-end flows of data in datagram packet switching networks than in circuit switching networks due to the lack of a separate circuit for each flow.

Virtual circuit packet switching, deployed with X.25, Asynchronous Transfer Mode (ATM), and more recently, with Multiprotocol Label Switching (MPLS), keeps the advantages of both circuit and datagram packet switching by sending packets instead of bitstreams over so-called virtual circuits (VC), but also suffers from the same lack of resilience as circuit switching when a link fails. Moreover, techniques have been designed to improve the resilience of unicast communications over datagram packet switching networks, but they are not adapted to multicast communications where one or several senders send traffic to several receivers. This thesis presents a new technique to improve the resilience of multicast communications in virtual circuit packet switching networks.

1.1 Switching technology

In the next three subsections, we present the three switching techniques used in networks: circuit switching, datagram packet switching and virtual circuit packet switching.

1.1.1 Circuit switching

Circuit switching is the transmission technology that has been used since the first communication networks in the nineteenth century. In circuit switching, a caller

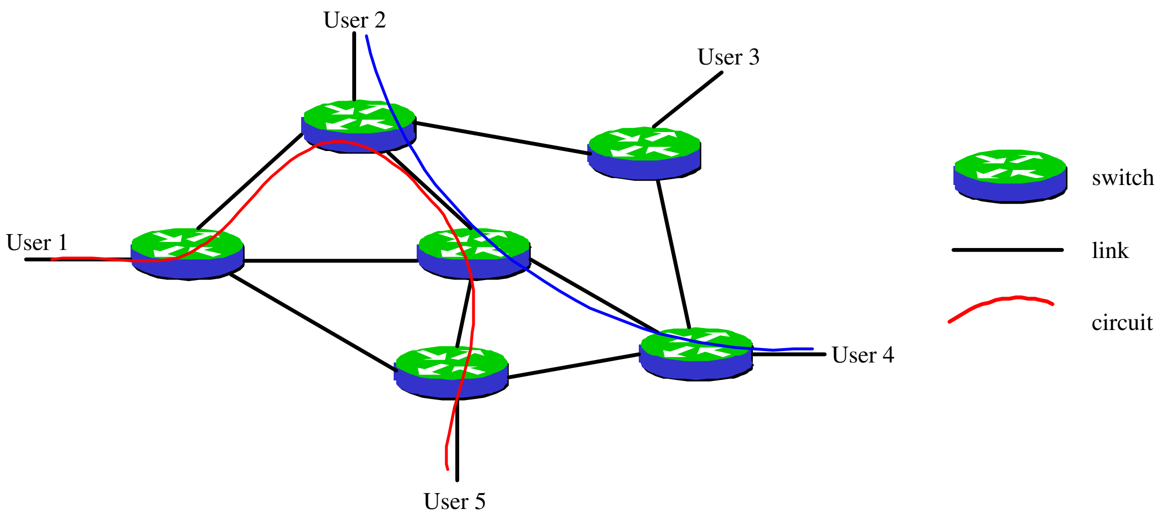


Figure 1.1: **Circuit switching.** The two different bitstreams flow on two separate circuits.

must first establish a connection to a callee before any communication is possible. During the connection establishment, resources are allocated between the caller and the callee. Generally, resources are frequency intervals in a Frequency Division Multiplexing (FDM) scheme or more recently time slots in a Time Division Multiplexing (TDM) scheme. The set of resources allocated for a connection is called a circuit, as depicted in Figure 1.1. A path is a sequence of links located between nodes called *switches*. The path taken by data between its source and destination is determined by the circuit on which it is flowing, and does not change during the lifetime of the connection. The circuit is *terminated* when the connection is closed.

In circuit switching, resources remain allocated during the full length of a communication, after a circuit is established and until the circuit is terminated and the allocated resources are freed. Resources remain allocated even if no data is flowing on a circuit, hereby wasting link capacity when a circuit does not carry as much traffic as the allocation permits. This is a major issue since frequencies (in FDM) or time slots (in TDM) are available in finite quantity on each link, and establishing a circuit consumes one of these frequencies or slots on each link of the circuit. As a result, establishing circuits for communications that carry less traffic than allocation permits can lead to resource exhaustion and network saturation, preventing further connections from being established. If no circuit can be established between a sender and a receiver because of a lack of resources, the connection is *blocked*.

A second characteristic of circuit switching is the time cost involved when establishing a connection. In a communication network, circuit-switched or not, nodes need to lookup in a *forwarding table* to determine on which link to send incoming data, and to actually send data from the input link to the output link. Performing a lookup in a forwarding table and sending the data on an incoming link is called *forwarding*. Building the forwarding tables is called *routing*. In circuit switching, routing must be performed for each communication, at circuit establishment time. During circuit establishment, the set of switches and links on the path between the sender and the receiver is determined and messages are exchanged on all the links between the two end hosts of the communication in order to make the resource allocation and build the routing tables. In circuit switching, forwarding tables are hardwired or implemented using fast hardware, making data forwarding at each switch almost instantaneous. Therefore, circuit switching is well suited for long-lasting connections where the initial circuit establishment time cost is balanced by the low forwarding time cost.

The circuit identifier (a range of frequencies in FDM or a time slot position in a TDM frame) is changed by each switch at forwarding time so that switches do not need to have a complete knowledge of all circuits established in the network but rather only local knowledge of available identifiers at a link. Using local identifiers instead of global identifiers for circuits also enables networks to handle a larger number of circuits.

Traffic engineering (TE) consists in optimizing resource utilization in a network by choosing appropriate paths followed by flows of data, according to static or dynamic constraints [39]. A main goal of traffic engineering is to balance the load in the network, i.e., to avoid congestion on links on a network while other links are under-utilized. To achieve such goals, traffic engineering methods can vary from offline capacity planning algorithms to automatic, dynamic changes. Since circuit switching allocates a fixed path for each flow, circuits can be established according to traffic engineering algorithms.

On the other hand, circuit switching networks are not reactive when a network topology change occurs. For instance, on a link failure, all circuits on a failed link are cut and communication is interrupted. Special mechanisms that handle such topological changes have been be devised. Traffic engineering can alleviate the consequences of a link failure by pre-planning failure recovery. A backup circuit can be established at the same time or after the primary circuit used for a communication is set up, and traffic can be rerouted from the failed circuit to the backup circuit if a link of the primary circuit fails. Circuit switching networks are intrinsically sensitive to link failures and rerouting must be performed by additional traffic engineering mechanisms.

1.1.2 Datagram packet switching

Conceived in the 1960's, *packet switching* is a more recent technology than circuit switching which addresses a disadvantage of circuit switching: the need to allocate resources for a circuit, thus incurring link capacity wastes when no data flows on a circuit. Packet switching introduces the idea of cutting data on a flow into packets which are transmitted over a network without any resource being allocated. If no data is available at the sender at some point during a communication, then no packet is transmitted over the network and no resources are wasted. Packet switching is the generic name for a set of two different techniques: datagram packet switching and virtual circuit packet switching. Here, we give an overview of datagram packet switching.

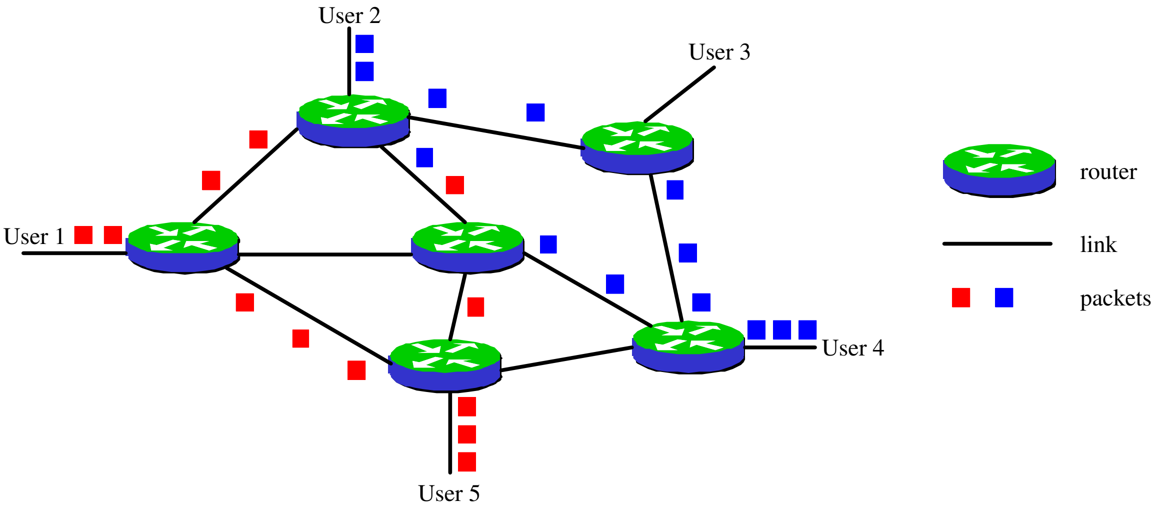


Figure 1.2: **Datagram Packet Switching.** Packets from a given flow are independent and a router can forward two packets from the same flow on two different links.

Different from circuit switching, datagram packet switching does not require to establish circuits prior to transmission of data and terminate circuits after the transmission of data. The switches, called routers, have to make a lookup in the forwarding table, called *routing table*, for each incoming packet. A routing table contains a mapping between the possible final destinations of packets and the outgoing link on their path to the destination. Routing tables can be very large because they are indexed by possible destinations, making lookups and routing decisions computationally expensive, and the full forwarding process relatively slow compared to circuit switching. In datagram packet switching networks, each packet must carry the address of the destination host and use the destination address to make a forwarding decision. Consequently, routers do not need to modify the destination addresses of packets when forwarding packets.

Since each packet is processed individually by a router, all packets sent by a host to another host are not guaranteed to use the same physical links. If the routing algorithm decides to change the routing tables of the network between the instants two packets are sent, then these packets will take different paths and can even arrive out of order. In Figure 1.2 for instance, packets use two different paths to go from User 1 to User 5. Second, on a network topology change such as a link failure, the routing protocol will automatically recompute routing tables so as to take the new topology into account and avoid the failed link. As opposed to circuit switching, no additional traffic engineering algorithm is required to reroute traffic.

Since routers make routing decisions locally for each packet, independently of the flow to which a packet belongs. Therefore, traffic engineering techniques, which heavily rely on controlling the route of traffic, are more difficult to implement with datagram packet switching than with circuit switching.

1.1.3 Virtual circuit packet switching

Virtual circuit packet switching (VC-switching) is a packet switching technique which merges datagram packet switching and circuit switching to extract both of their

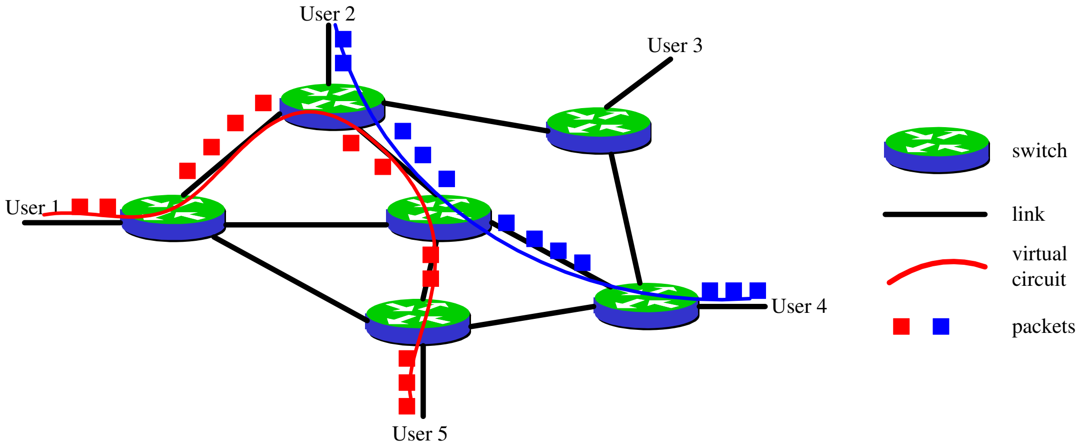


Figure 1.3: **Virtual circuit packet switching.** All packets from the same flow use the same virtual circuit.

advantages. VC-switching is a variation of datagram packet switching where packets flow on so-called logical circuits for which no physical resources like frequencies or time slots are allocated (see Figure 1.3). Each packet carries a circuit identifier which is local to a link and updated by each switch on the path of the packet from its source to its destination. A virtual circuit is defined by the sequence of the mappings between a link taken by packets and the circuit identifier packets carry on this link. This sequence is set up at connection establishment time and identifiers are reclaimed during the circuit termination.

We have seen the trade-off between connection establishment and forwarding time costs that exists in circuit switching and datagram packet switching. In VC-switching, routing is performed at circuit establishment time to keep packet forwarding fast. Other advantages of VC-switching include the traffic engineering capability of circuit switching, and the resources usage efficiency of datagram packet switching. Nevertheless, a main issue of VC-Switched networks is the behavior on a topology change. As opposed to Datagram Packet Switched networks which automatically recompute routing tables on a topology change like a link failure, in VC-switching all virtual circuits that pass through a failed link are interrupted. Hence, rerouting in VC-switching relies on traffic engineering techniques.

In practice, major implementations of VC-switching are X.25 [70], Asynchronous Transfer Mode (ATM [6]) and Multiprotocol Label Switching (MPLS [50]). The Internet, today's most used computer network, is entirely built around the Internet Protocol (IP), which is responsible for routing packets from one host to another. Because of the central role of IP in the Internet, we now discuss how ATM and MPLS interact with IP.

1.2 Virtual circuit packet switching with IP

Protocols which route data from a node or hop to another hop between two end hosts in a network are called network-layer or L3 protocols. In the Internet, the only currently available network-layer protocol is IP, which comes in two flavors: IPv4 [59] makes use of 32-bit long addresses and IPv6 [22] uses 128-bit long addresses.

We will mostly focus on IPv4 as IPv4 is the currently deployed version of IP in the Internet. Because of the advantages of virtual circuit packet switching and the growing popularity of IP, many Internet service providers send IP packets over virtual circuits. Virtual circuit packet switching technologies that have been used in the Internet backbones are ATM and, more recently, MPLS.

1.2.1 IP-over-ATM

ATM is a VC-switching technology which was standardized starting in the late 1980s. ATM uses fixed-length payloads with a length of 48 bytes and a 5-byte header, yielding 53-byte long ATM *cells*. Among the 40 header bits of a cell, 28 are reserved to identify the virtual circuit to which the cell belongs. The corresponding fields are called VCI/VPI (Virtual Circuit Identifier/Virtual Path Identifier). The VCI/VPI fields are updated at each switch.

A first issue that arises when trying to send IP packets over ATM virtual circuits is the need for a definition of an encapsulation of IP packets in ATM cells, i.e., how to put IP data inside ATM cells. Encapsulation is performed by an *adaptation layer* (AAL) as defined in [6]. Moreover, most IP packets are too large to fit in a 53-byte ATM cell. An IP header is at least 20 bytes long, and hosts cannot be coerced to send IP packets of at most 53 bytes. Therefore, IP packets must be cut into smaller pieces in a process called segmentation before they can be encapsulated and put in

ATM cells. The last router on the path of IP packets must reassemble the fragments to reconstitute the original IP packets. Segmentation and Reassembly (SAR) is a complex and time consuming process.

Sending IP traffic over an ATM infrastructure proves to be complex [19]. Far from solving the circuit interruption problem that arises on link failure with VC-switching, IP over ATM introduces the aforementioned issues and requires additional hardware as combined IP routers/ATM switches do not exist.

1.2.2 MPLS

Multiprotocol Label Switching (MPLS) is a comprehensive IP over virtual circuits technology which has been specifically engineered to interface better with IP than ATM. MPLS runs over many existing network hardware like Ethernet [38] or even ATM and supports the forwarding of IP packets over virtual circuits. MPLS can be implemented in IP routers.

In MPLS, each packet carries a virtual circuit identifier, called *label*, as a field of a *shim header* inserted between the IP header and the MAC/link layer header of a packet. A single packet can carry more than one shim header. The set of all headers carried by a packet is called an MPLS stack. Figure 1.4 depicts a stack of MPLS headers, and the position of the MPLS stack in the headers of a packet.

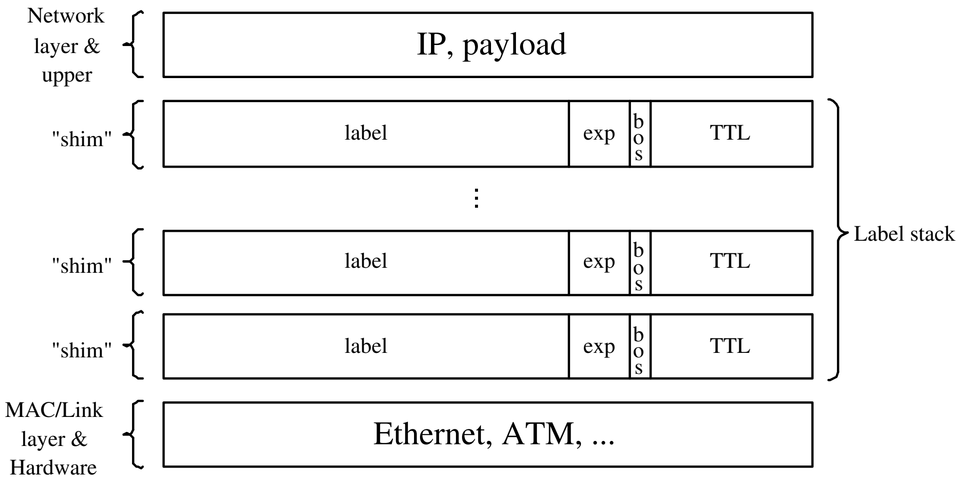


Figure 1.4: **Position of the MPLS stack in the network protocol stack.** Shim headers are inserted between the IP layer and the MAC or link layer. Each shim header consists of four fields.

MPLS handles labels just like all other virtual circuit identifiers are handled in other virtual circuit switching technologies. Consider an IP packet sent by host A to host B in Figure 1.5. The packet is forwarded through an MPLS network or *domain* between A and B. When a packet arrives at the first MPLS router, also called *ingress Label Edge Router* (ingress LER) of the MPLS domain, the source and destination IP addresses of the packet are analyzed and the packet is classified in a Forwarding Equivalence Class (FEC). All packets within the same FEC use the same virtual circuit, called *Label Switched Path* or *LSP*. Suppose a virtual circuit has already been established for the FEC of the packet sent by A to B: then, the ingress LER inserts or *pushes* an MPLS header on the packet (L1 in the Figure). Subsequent routers

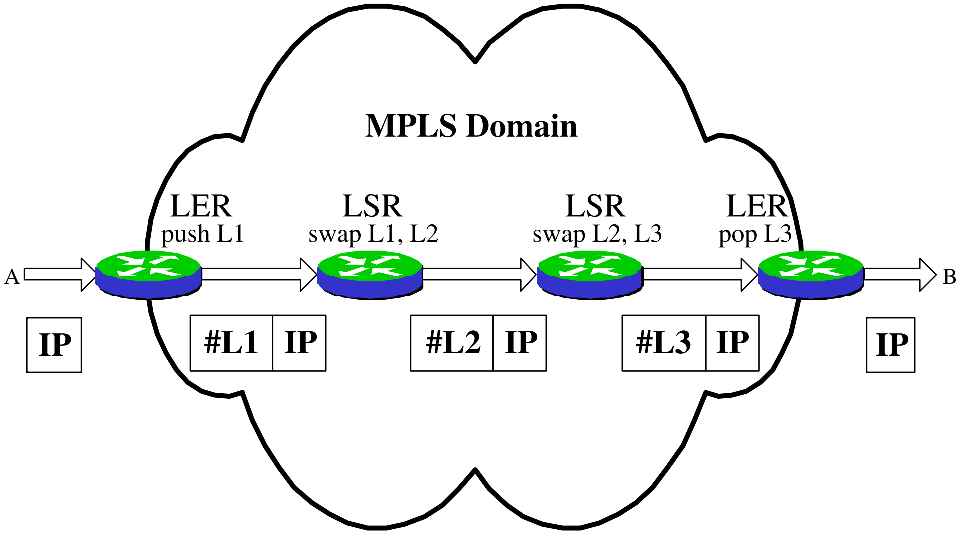


Figure 1.5: **MPLS forwarding.** The ingress LER determines the FEC of packets sent by A to B and pushes a label on the packets. Subsequent LERs swap labels. The egress LER pops the label and outputs an IP packet with no MPLS header.

of the MPLS domain update the MPLS header by *swapping* the label (L1 against L2, L2 against L3). Finally, the last router of the LSP, called *egress LER*, removes or *pops* the MPLS header (L3 in the Figure), so that the packet can be handled by subsequent MPLS-unaware IP routers or hosts.

MPLS routers push, swap and pop MPLS headers according to rules contained in a forwarding table called *Forwarding Information Base* (FIB) that is distinct for each MPLS router. The FIB can contain three different types of entries. A *Next Hop Label Forwarding Entry* (NHLFE) contains the information necessary to forward a packet for which a label has already been assigned. A NHLFE contains two pieces of information: the packet's next hop address, and whether the MPLS header of the packet must be swapped or popped. If the MPLS header of the packet must be swapped, then the NHLFE also contains the new label of the packet. The *Incoming Label Map* (ILM) contains the mappings between labels carried by incoming packets and NHLFE entries. Last, the *FEC-to-NHLFE* (FTN) contains the mappings between incoming packet FECs and NHLFE entries. MPLS routers use their FIB as follows. Suppose a packet with no label arrives at an MPLS router. The MPLS router first determines the FEC for the packet, then looks up in the FIB for the FTN that matches the FEC of the packet. This FTN contains a label and a NHLFE which in turn contains the next hop for the packet. The MPLS router pushes an MPLS header that contains the label read in the FTN and forwards the packet according to the information contained in the NHLFE. Now suppose that a labeled packet arrives at an MPLS router. The MPLS router searches in the FIB for an ILM that matches the label of the packet and reads the associated NHLFE. The NHLFE can either indicate that the MPLS header must be swapped against a new label, or popped. In the former case, the MPLS router swaps the MPLS header and forwards the packet to the next hop specified in the NHLFE. In the latter case, the MPLS router pops the label and forwards the packet to the next hop specified in the NHLFE.

MPLS headers are 32 bits long and labels take only twenty bits out of these thirty-two bits, therefore leaving room for further information inside the header. The twelve remaining bits are used as follows. First, MPLS is designed to be able to take into account Quality of Service (QoS), and three *experimental* (*exp*) bits have been allocated to handle up to eight QoS classes. Second, MPLS allows hierarchical domain nesting: when a packet enters an MPLS domain which is contained in another MPLS domain, a new label is appended to the packet, which was already carrying one label. This is referred to as *label stacking*. For instance, Figure 1.6 shows two nested MPLS domains. A *bottom of stack* (*bos*) bit indicates whether a label is the last in the stack or not. Finally, each MPLS router decrements an 8-bit Time To Live (TTL) field and discards packets when the value of this field initially set by the ingress LER reaches zero. The reason for this mechanism is to avoid packets from indefinitely looping in case a circular virtual circuit is mistakenly created. The division of an MPLS header into the four fields we have just described is summed up in Figure 1.4.

A main point of interest with FECs in a traffic engineering context is that they support aggregation. All packets from different sources but entering the MPLS domain through the same LER, and bound to the same egress LER, can be assigned to the same FEC and therefore the same virtual circuit. In other words, there is no need to establish a new virtual circuit for each (source, destination) pair read in the headers of incoming packets. Once an ingress LER has determined the FEC of a packet, the ingress LER assigns a virtual circuit to the packet via a label number. Also, FEC definitions can take into consideration IP packet sources in addition to destinations.

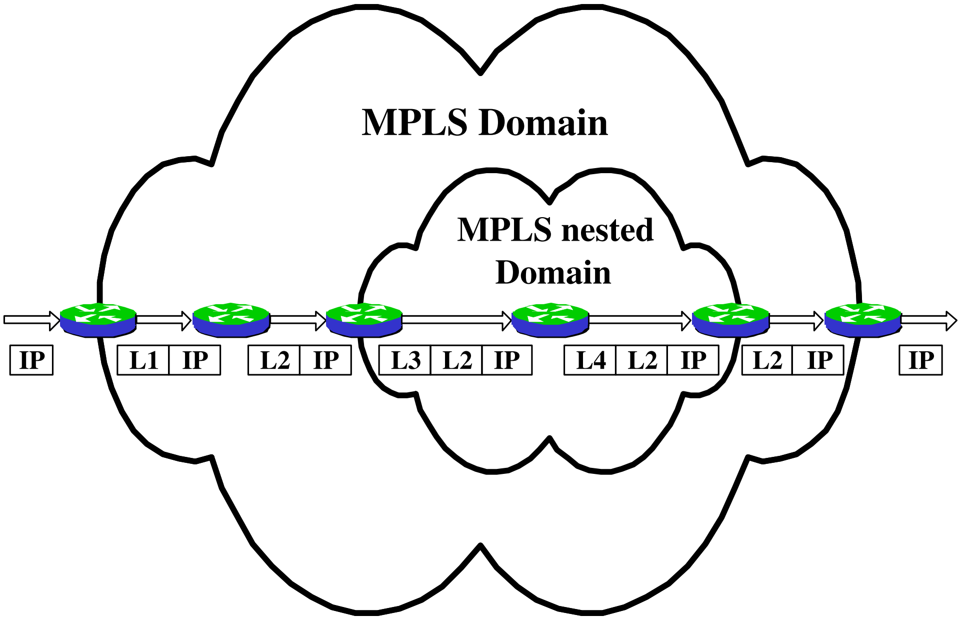


Figure 1.6: **Nested MPLS domains.** In the inner MPLS domain, two labels are stacked.

Two packets that enter the MPLS domain through the same LER and going to the same destination can use different sets of links so as to achieve *load balancing*, that is, put the same amount of traffic on all links hereby distributing the load of traffic on each link. A FEC can also depend on additional parameters such as the Type of Service bits of the IP header to provide differentiated services to IP traffic [23].

Establishing virtual circuits and mappings or *bindings* between FECs and labels and building the FIB at each MPLS router is the responsibility of a *signaling protocol.* The MPLS architecture [62] does not impose to use any specific signaling protocol. The only requirement is that, on a given link and for a given LSP, labels are assigned

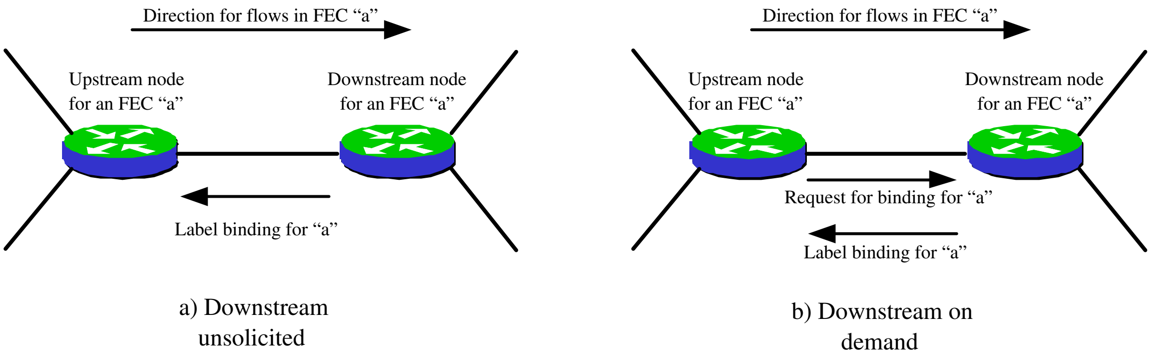

Direction for flows in FEC “a”
Upstream node for an FEC “a”
Downstream node for an FEC “a”
Label binding for “a”
a) Downstream unsolicited
Direction for flows in FEC “a”
Upstream node for an FEC “a”
Downstream node for an FEC “a”
Request for binding for “a”
Label binding for “a”
b) Downstream on demand

Figure 1.7: **Label distribution modes.** Downstream unsolicited and downstream on demand label distribution modes.

by the downstream node and advertised to the upstream node. Two refinements have been devised from this requirement, as shown in Figure 1.7. In *downstream unsolicited label distribution* mode, a node recognizes that it is a downstream node for a FEC and sends a label binding message to the upstream node for that FEC. The downstream node decides by itself to send the binding message without any trigger from the upstream node. Conversely, in *downstream on demand* mode, the upstream node identifies that it needs a label binding message for a particular FEC and requests that the downstream node sends this label binding message.

Although no label signaling protocol is imposed by the MPLS standards, only two signaling protocols for MPLS have been developed. RSVP-TE [8] is based on a resource reservation protocol for the Internet, RSVP [13], to which it adds the capability to advertise LSPs. Another protocol, LDP (Label Distribution Protocol [3]), has been defined from scratch as a part of the MPLS design effort. An extension to LDP, namely CR-LDP (Constraint Routing LDP [41]), adds important features to LDP with respect to traffic engineering. One of the main improvements of CR-LDP is the support of Explicit Routing, where a single node or an offline server which precomputes paths can fully define and advertise LSPs. The differences between these two signaling protocols are examined in [14].

MPLS is still a new technology. The need for a new VC-switching technology with IP packets to replace IP over ATM has been identified in 1996 and standardization of MPLS commenced in 1997 [62]. The standardization process is still continuing at the time of this writing, and one of the big pieces of work that remains to be added to this effort is multicasting.

1.3 Multicast

So far, we have concentrated on *point-to-point* communication or *unicast*, where a single source is sending data to a single receiver. Many applications however require that data be sent simultaneously to several receivers. The best example of such a requirement is teleconferencing between a *group* of three or more people. When one of the members of the group is speaking, his/her voice must be delivered to all other group members: this is called *point-to-multipoint* communication, or *multicast*.

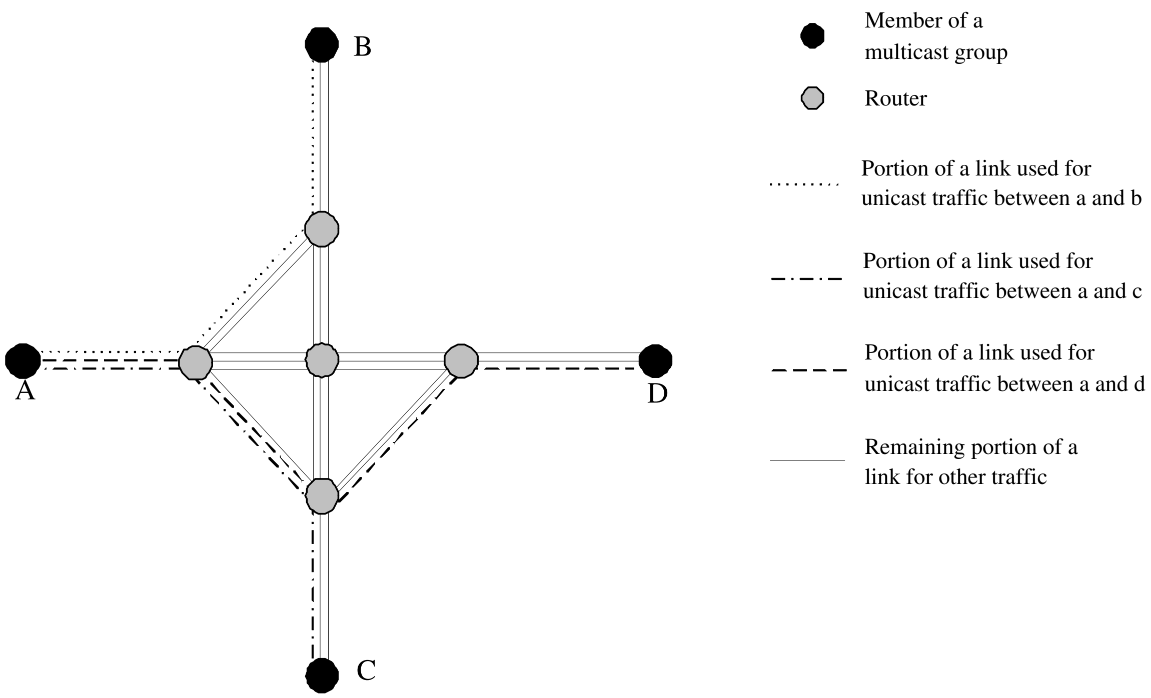

B
A
D
C
Member of a multicast group
Router
Portion of a link used for unicast traffic between a and b
Portion of a link used for unicast traffic between a and c
Portion of a link used for unicast traffic between a and d
Remaining portion of a link for other traffic

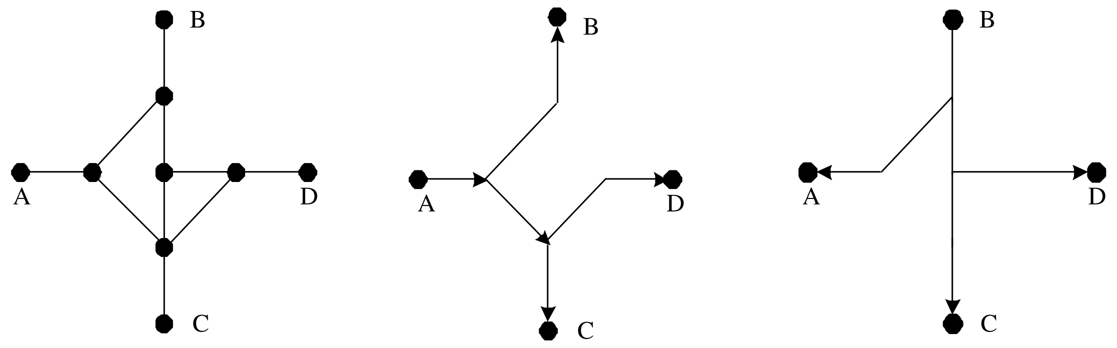
Figure 1.8: **Multicast achieved through unicast.** Member A sends the same data three times to members B, C and D.

A simple solution to implement multicasting is to send the same data in turn to all other members of the conference. In other words, point-to-multipoint communication can be achieved through multiple point-to-point communications. However, such a solution is expensive in terms of bandwidth utilization: the same information must be sent $n-1$ times if the teleconference gathers n people, thus wasting bandwidth. In Figure 1.8, A sends the same data three times, wasting two thirds of the bandwidth on the leftmost link. A more efficient support of multicast in a network sets up a *multicast routing tree*, where the switches are the nodes of the tree and the links are the edges of the tree. Each switch that is a bifurcation of the tree duplicates packets to each outgoing link. By forwarding data on this tree structure and duplicating data at intermediate nodes of the tree, the same information flows on each link of the tree only once, hereby saving bandwidth. Multicast routing trees can be either Shortest Path Trees or Core Based Trees. We review each of these structures in the next subsection.

1.3.1 Multicast routing tree structure

In a *shortest path tree* structure, each possible source of a multicast group is the root of a separate tree. The edges of the tree are network links, nodes are routers, and leaves are members of the multicast group. If the group contains n possible sources, then n trees must be built. A shortest path tree is obtained by computing the shortest path (in terms of some link cost — usually the cost is one for each link) between the source and each other group member. Figure 1.9 shows examples of shortest path trees rooted at two different nodes in a multicast group of four nodes (Figure 1.9(a)). The two shortest path trees do not use the same set of links, but all paths from node A to other nodes (Figure 1.9(b)) and from node B to other nodes (Figure 1.9(c)) use a minimum number of links.

In *center-based* or *core based tree* multicast, all participants are leaves of the same, unique shared tree. All the traffic generated by the multicast group flows through a

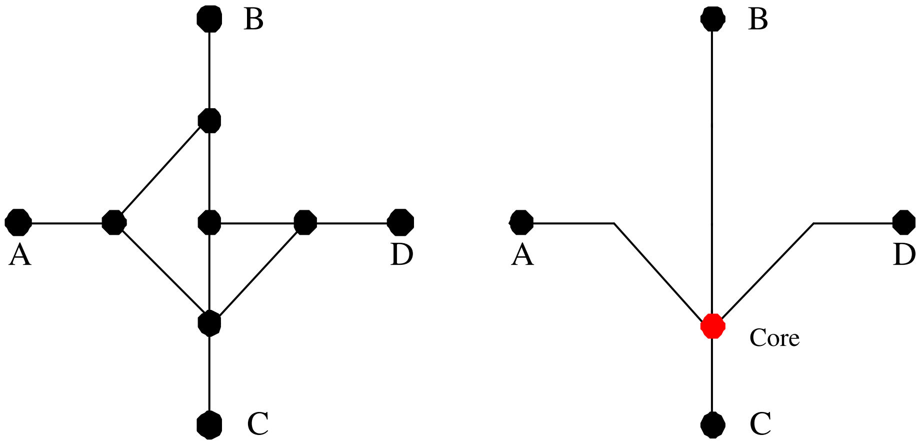


(a) A communication network and a multicast group of four participants: A, B, C, D (b) Shortest Path Tree rooted at A (c) Shortest Path Tree rooted at I

Figure 1.9: **Shortest path trees.** This figure only shows the shortest path trees rooted at A and B.

special node on this tree, called *center* or *core*, as shown in Figure 1.10. Contrary to shortest path trees, the path between two group members is not guaranteed to be the shortest. For instance, in Figure 1.10, data sent by node A uses five links before reaching C when the shortest path between A and C is only three links long. It can be shown that a particular core based tree, called Steiner Tree [30] is optimal in terms of bandwidth utilization. However, it can also be proven that computing Steiner trees is NP-hard [42]. A more thorough comparison between shortest path trees and core based trees can be found in [68].

In Section 1.1, we discussed link failures with unicast traffic and explained that traffic has to be rerouted to a different path to avoid a failed link. With datagram



(a) A communication network and a multicast group of four participants: A, B, C, D

(b) A center-based tree

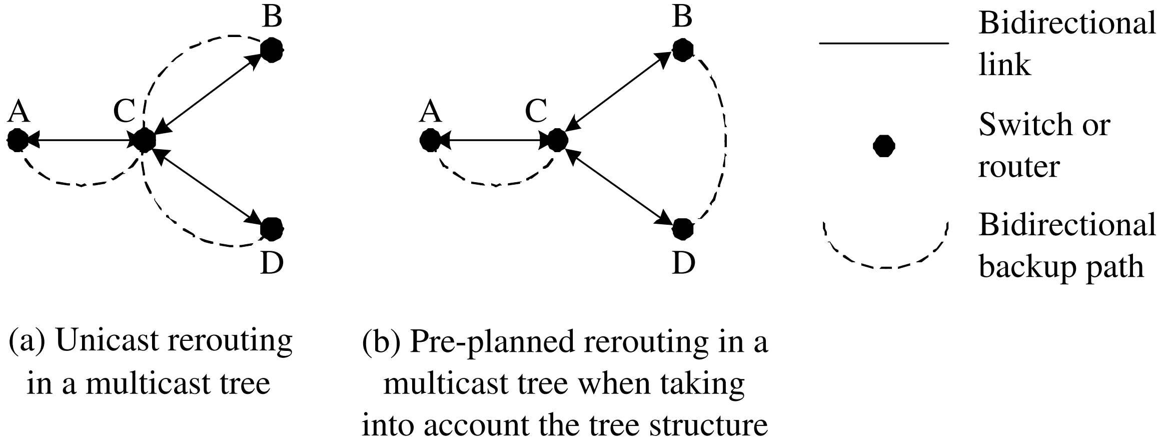
Figure 1.10: **Center-based multicast routing tree.** The multicast group shares a single tree.

packet switching, rerouting is performed automatically online after a link fails while traffic engineering techniques allow to pre-plan backup paths in circuit switching and VC-switching. A multicast group member is *dropped* from a multicast group when this member cannot reach or be reached by the core or source of the tree anymore. In order to protect a multicast routing tree from any single link failure, it is possible to pre-plan a different backup path between each group member and the source or the core of a multicast routing tree. For instance, in Figure 1.11(a), a core-based tree centered in C with three group members A, B and D is fully protected by three pre-planned backup paths. If any single link in the tree fails, the connectivity between

C and any dropped group member is restored by rerouting traffic over the backup path between C and the dropped group member. However, pre-planning involves additional computations, circuit advertising, and possibly resource reservation on the backup path to make sure that the rerouted traffic will not overload the backup path. Backup path pre-planning is therefore costly in terms of computations and bandwidth reservation. Thus, it is desirable to limit the number of pre-planned backup paths in a network. In Figure 1.11(b), we show how only two backup paths are required to protect the multicast routing tree from any single link failure. If link (CA) fails, traffic between A and C is rerouted over the leftmost backup path. If link (CB) fails, then multicast traffic can still reach node C via B and the rightmost backup path. The same backup path is also used when link (CD) fails, hereby keeping node D reachable from any other member of the tree. Therefore, a single backup path can be used to protect the multicast routing tree from different link failures.

1.3.2 Multicast with IP

Multicast with IP has been studied since 1988 [21] and a certain range of IP addresses (all addresses beginning with the four bits 1110, or, in dotted-decimal notation, 224.0.0.0 to 239.255.255.255) has been allocated for multicast communications. An IP multicast address is an identifier for a multicast group. IP multicast is defined for UDP only, an unreliable datagram oriented transport protocol.



(a) Unicast rerouting in a multicast tree

(b) Pre-planned rerouting in a multicast tree when taking into account the tree structure

Figure 1.11: **Pre-planning backup paths in a multicast routing tree.** It is possible to protect the multicast routing tree with three group members from any single link failure with only two pre-planned backup paths.

Since a multicast routing tree can span over the whole Internet, establishing a multicast routing tree and then routing multicast packets is the main issue encountered by IP multicast. Another issue is that few routers are multicast-enabled in the Internet. IP multicast is achieved with two protocols.

First, a signaling protocol like IGMP [20] [29] runs between end hosts and routers and is responsible for managing host group membership. Multicast groups are open and dynamic. To join a multicast group, a host needs to know the IP address of the group and send an IGMP message to its next-hop router; on the other hand, a host can leave a group at any time by sending the appropriate IGMP message.

Second, multicast routing protocols run between routers to create and manage multicast routing trees. Many multicast routing protocols have been designed in the last decade. The Distance Vector Multicast Routing Protocol (DVMRP [67]) builds source-rooted trees using a Distance Vector protocol, where each router is able to determine the outgoing link of a packet based on local information only. MOSPF [48] extends a unicast routing protocol, OSPF. MOSPF also builds source-rooted trees, but because OSPF is a link-state routing protocol, each OSPF router knows the full topology of the network. Thus, MOSPF can build shortest path trees using the well-known Dijkstra shortest path algorithm [24]. One protocol uses core based trees: Core Based Tree or CBT [9]. Finally, PIM [27] can build both types of trees. In multicast groups that contain many receivers (*PIM Dense Mode*), PIM builds shortest path trees. Conversely, with smaller groups (*PIM Sparse Mode*), PIM builds core based trees.

Last but not least, IP routers are able to duplicate packets. Routers with three or more links involved in a multicast routing tree must duplicate packets before they can forward them simultaneously on several links.

Although IP multicasting is well defined and standardized, it is not widely deployed in the Internet [26] mainly because of current multicast routing protocols scalability issues. We now present how VC-switching technologies, ATM and MPLS, support multicasting.

1.3.3 Multicast with ATM

Three different approaches have been proposed to add multicast support to ATM. The first one, the usage of point-to-multipoint virtual circuits to carry multicast traffic [2], requires cell duplication by switches and a specific signaling protocol to advertise the point-to-multipoint virtual circuits. In [32], the authors make a survey of the different techniques available to duplicate cells with specific hardware. However, the current standard ATM signaling protocol does not support point-to-multipoint virtual circuits implemented in hardware [7]. Moreover, we have seen that ATM switches perform Segmentation and Reassembly to fit IP packets into ATM cells. With AAL5 [36], the main adaptation layer currently deployed in ATM networks, cells arriving at a destination switch are identified solely by a virtual circuit number. ATM switches that receive multicast cells have no means of knowing the origin of a cell. Therefore, ATM switches cannot reassemble IP packets contained in cells from different sources that have been interleaved by the ATM network [18].

The other two ATM multicast approaches emulate point-to-multipoint virtual circuits with point-to-point virtual circuits [25]. In the multicast virtual circuit mesh model, every member of a multicast group establishes a point-to-point virtual circuit to every other member of the group. This approach is not scalable with the number of multicast group members. Indeed, when a node wants to join or leave a group, every group member must create or terminate a virtual circuit to the new member. Third, in the *Multicast Server model* (*MCS*), a centralized server is responsible for handling multicast traffic. For instance, in *LANE* [65], all multicast traffic on an ATM network is sent to a *Broadcast and Unknown Server* (*BUS*) which establishes a point-to-point virtual circuit with all other members of the group. While MCS does not have the scalability issue of mesh virtual circuits, the BUS is a single point of failure and can become a bottleneck in the network.

The *Multicast Address Resolution Server* (*MARS*) architecture [5] implements both mesh virtual circuits and MCS. With MARS, members of a multicast group must contact a MARS server to join or leave a group. Then, the MARS server can either act as an ARP server and let a joining host establish a point-to-point virtual circuit to all other group members, or act as the center of a core based tree and create a unicast path to every member of the group in order to replace one point-to-multipoint virtual circuit by several point-to-point virtual circuits. In summary, current implementations of multicast with ATM only emulate multicasting with unicast virtual circuits and therefore do not have efficient multicast support.

1.3.4 Multicast with MPLS

Although MPLS natively supports multicasting in its design, MPLS multicast has not been given a lot of attention and is still at the proposal stage [56] [71]. As a side note, Alcatel stated to have an implementation running as early as 1999 [17]. This prototype of multicast over MPLS uses a proprietary signaling protocol or proprietary extensions to an existing signaling protocol.

Multicast and unicast traffic require different types of processing from routers. For instance, IP identifies multicast packets by looking at the multicast address range. In MPLS, unicast and multicast packets have already been assigned a different type code in the link-layer header [61]. Therefore, MPLS routers know whether a packets is from a unicast or a multicast flow.

MPLS multicast is fundamentally different from ATM multicast. First, MPLS routers do not need to perform Segmentation and Reassembly, thus the aforementioned frame interleaving issue does not exist in MPLS multicast networks. Second, MPLS routers are mainly IP routers enhanced to support MPLS. The packet duplication mechanism that is implemented in IP routers to support IP multicast can be used to duplicate MPLS packets. MPLS routers at the bifurcation of a multicast routing tree duplicate packets and send copies of the same packet on different outgoing links. Each copy of an incoming MPLS multicast packet is assigned a different label before it is forwarded on an outgoing link. Furthermore, when a packet is duplicated, one copy can be forwarded using MPLS (the label of the incoming packet is swapped and the packet is sent to another MPLS router) and another copy can be sent using IP (the label is incoming is popped and the packet is forwarded to an IP router). Therefore, a multicast MPLS router can be at the same time a LSR and a LER for the same multicast virtual circuit.

Each member of a multicast group can build a shortest path tree multicast LSP to reach all other members. Alternatively, all members of a group can be leaves of a common core based tree whose center can be any node of the network. A signaling protocol performs multicast LSP establishment and termination, online or offline. MPLS can rely on an IP multicast routing protocol to build the tree online, and then create a multicast LSP that matches the IP multicast routing tree. The other alternative is to have MPLS multicast LSPs built offline by a dedicated server. This solution may be preferable in situations where substantial offline computing is necessary. The dedicated server computes a multicast routing tree and uses signaling protocol messages to advertise the multicast LSP. This particular form of routing where a LSP is fully defined and advertised by a particular node is called *Explicit Routing*. Explicit Routing is a traffic engineering technique which is a novelty of MPLS over ATM. However, the main issue with MPLS multicast is that no signaling protocol currently supports MPLS multicast virtual circuits (*multicast LSPs*).

1.4 Contributions of this thesis

Different from datagram switching, virtual circuit packet switching technologies like MPLS require traffic engineering mechanisms to compute backup paths and to perform rerouting after a link has failed. In this thesis, we address the problem of fast recovery of a multicast routing tree after a link failure.

Consider an MPLS network over which a multicast routing tree has been established, as shown in Figure 1.12. An MPLS network receives traffic directly from multicast hosts attached to MPLS routers, or from networks which simply relay multicast traffic from other multicast hosts. When a link of the multicast routing tree fails, a certain number of multicast hosts accessing the tree directly or through other networks are dropped from the communication. In this thesis, we present an algorithm which aims at selecting one backup path in a given multicast routing tree to improve the resilience of the tree for a single link failure. The backup path selected by the algorithm minimizes the number of group members dropped from a multicast communication on a single link failure. We provide a specification, complexity analysis and implementation of the algorithm.

Our second contribution is the addition of multicast support to the Linux implementation of MPLS. A unicast implementation of MPLS is available for the Linux operating system, but MPLS multicast is barely standardized and no implementation is available at the time of this writing. We also provide the definition and implementation of the signaling protocol needed to establish multicast LSPs in an MPLS network. Our signaling protocol implements Explicit Routing to establish multicast LSPs. Any MPLS router or alternatively a dedicated server can establish mLSPs.

Our third contribution is the design and implementation of an MPLS multicast rerouting mechanism, MPLS multicast Fast Reroute. MPLS multicast Fast Reroute

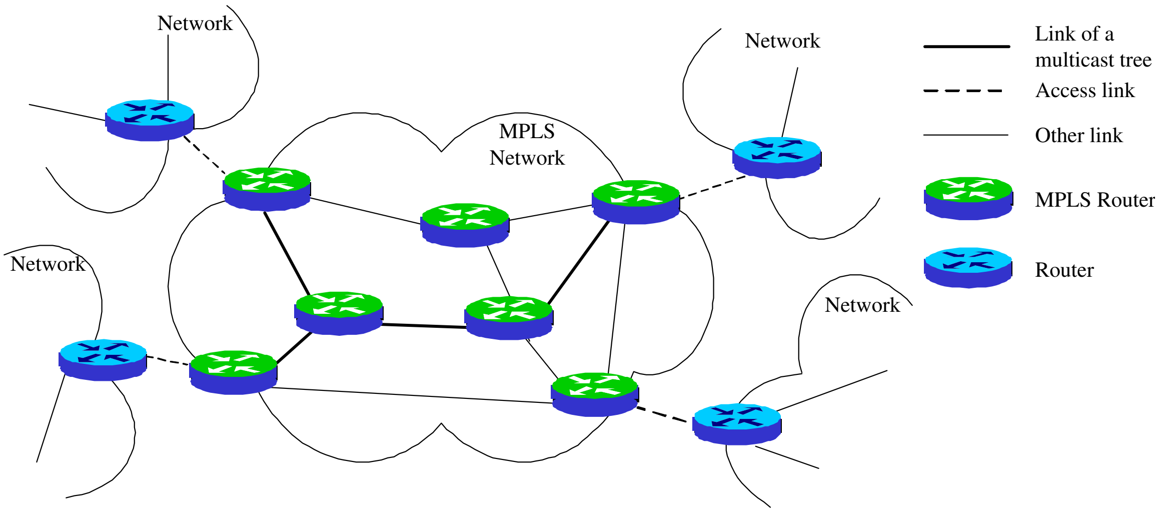


Figure 1.12: **MPLS multicast routing tree.** In this example, multicast group members are not represented and send multicast traffic over the tree established in the MPLS domain via other networks.

is an extension to MPLS Fast Reroute [34], a unicast MPLS unicast rerouting mechanism. We provide a description and the implementation of both the MPLS multicast Fast Reroute mechanism itself and the signaling protocol extensions necessary to support the rerouting mechanisms. Although our implementation of MPLS multicast Fast Reroute runs over off-the-shelf Linux PC-routers, it can reroute multicast traffic in less than 50 ms (without propagation delay) depending on the size of the tree, thus making link failures unnoticeable to users of multicast applications like teleconferencing.

The remainder of this thesis is structured as follows. In Chapter 2, we present techniques to protect networks from link failures. These techniques range from mechanisms implemented in the physical layer to a traffic engineering mechanism specific to MPLS unicast. In Chapter 3, we propose a graph algorithm that builds a particular backup path that minimizes the number of group members dropped from a multicast communication on a single link failure. In Chapter 4, we present MPLS Multicast Fast Reroute, an extension to MPLS that implements rerouting in multicast routing trees and makes use of the backup path computed in Chapter 3. In Chapter 5, we show how to add multicast support to a unicast implementation of MPLS using PC hardware and the Linux operating system. Then, we define the signaling protocol required to implement MPLS Fast Reroute, and describe our implementation of MPLS Multicast Fast Reroute. In Chapter 6, we present experiments which show how MPLS Multicast Fast Reroute compares with traditional rerouting techniques.

2

Resilience and protection in networks

Resilience refers to the ability of a network to keep services running despite a failure. Failure can have many causes, and we will focus in this thesis on “fiber cuts” or “link cuts”, which result in the loss of all traffic that is forwarded on a failed link. Link cut is a common failure. For instance, in telephone networks, link cuts are the largest cause of service interruption time. Indeed, in a study conducted from April 1992 to March 1994, Kuhn [44] showed that cable cutting was the cause of 25% of telephone networks downtime.

Resilient networks recover from a failure by repairing themselves automatically. More specifically, failure recovery is achieved by *rerouting* traffic from the failed part of the network to another portion of the network. Rerouting is subject to several constraints. End-users want rerouting to be fast enough so that the interruption of service time due to a link failure is either unnoticeable or minimal. The new path taken by rerouted traffic can be either computed at the time failures occur or before failures. In the second case, rerouting is said to be *pre-planned.* Compared with recovery mechanisms that do not pre-plan rerouting, pre-planned rerouting mechanisms permit to decrease interruption of service times but may require additional hardware to provide redundancy in the network and consume valuable resources like computational cycles to compute backup paths. The different techniques we present in this chapter illustrate the trade-off between recovery speed and costs incurred by pre-planning.

In this chapter, we review existing techniques that improve network resilience. We first present an overview of rerouting and formalize the notion of total repair time for a network that reroutes traffic after a link failure in Section 2.1. In Section 2.2, we describe lower (physical and MAC) layer rerouting techniques. Low layer rerouting techniques rely solely on hardware and are therefore the fastest rerouting techniques available. However, they also require expensive hardware redundancy. We compare rerouting at lower layers with rerouting performed by the network layer without pre-planning in Section 2.3. Network layer rerouting is purely implemented in software and is therefore slower than lower layer rerouting. However, network layer rerouting does not use pre-planning, hence saving costs in hardware and CPU cycles compared with lower layer rerouting. Performing rerouting between lower layers and the network layer with ATM or MPLS presents a trade-off between recovery speed and pre-planning costs. In Section 2.4, we give an overview of MPLS Fast Reroute, a unicast rerouting technique that takes advantage of this trade-off. As shown in Section 1.3.1, taking the tree topology of multicast groups into account for routing

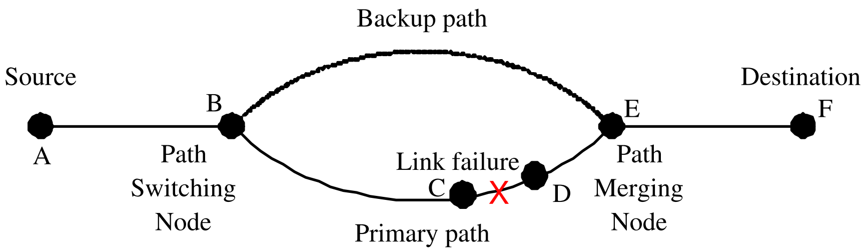


Figure 2.1: **Rerouting overview and terminology.** Traffic between A and B on the primary path is rerouted over the backup path when link (CD) fails.

purposes leads to cost savings in terms of computations and bandwidth. As a result, techniques specifically designed for multicast traffic have been developed. We describe these multicast rerouting techniques in Section 2.5.

2.1 Overview of rerouting

We present here general concepts and terminology concerning rerouting. Rerouting is a technique that can be used in both Circuit Switching and Packet Switching networks. When a link in a network fails, traffic that was using the failed link must change its path in order to reach its destination: it is rerouted from a *primary path* to a *backup path*. The primary and the backup path can be totally disjoint or partially merged. Figure 2.1 presents an example where a source node A sends traffic to a destination node F, and where a link on the primary path fails. In the remainder of this section, we will refer to this example to illustrate rerouting. A complete rerouting technique consists in seven steps. The first four concern rerouting after a link has failed to switch traffic from the primary to the backup path, while the last three concern rerouting after the failed link has been repaired to bring back traffic to the primary path.

First, the network must be able to *detect link failures*. Link failure detection can be performed by dedicated hardware or software by the end nodes C and D of the failed link. Second, nodes that detect the link failure must *notify* certain nodes in the network of the failure. Which nodes are actually notified of the failure depends on the rerouting technique. Third, a *backup path must be computed*. In pre-planned rerouting schemes however, this step is performed before link failure detection. Fourth, instead of sending traffic on the primary, failed path, a node called Path Switching Node must send traffic on the backup path. This step in the rerouting process is called *switchover*. Switchover completes the repairing of the network after a link failure.

When the failed link is physically repaired, traffic can be rerouted to the primary path, or keep being sent on the backup path. In the latter case, no further mechanism is necessary to reroute traffic to the primary path while three additional steps are needed to complete rerouting in the former case. First, a mechanism must *detect the link repair*. Second, nodes of the network must be *notified* of the recovery, and third the Path Switching Node must send traffic back on the primary path in the so-called *switchback* step.

Consider a unicast communication. When a link of the path between the sender and the receiver fails, users experience service interruption until the path is repaired. The length of the interruption is the time between the instant the last bit that went through the failed link before the failure is received, and the instant when the first bit of the data that uses the backup path after the failure arrives at the receiver. Let T_{detect} denote the time to detect the failure, T_{notif} the notification time, $T_{switchover}$ the switchover time, and d_{ij} the sum of the queuing, transmission and propagation delay needed to send a bit of data between two nodes i and j. Then, for the example given in Figure 2.1, the total service interruption time for the communication $T_{service}$ is given by:

$$T_{service} = T_{detect} + T_{notif} + T_{switchover} + (d_{BE} + d_{EF}) - (d_{DE} + d_{EF}). \tag{2.1}$$

The quantity $(d_{BE} - d_{EF}) - (d_{DE} - d_{EF})$ does not depend on the rerouting technique but rather on the location of the failure. Therefore, we define the total repair time T_{repair} which only depends on the rerouting mechanism by

$$T_{repair} = T_{detect} + T_{notif} + T_{switchover}. \tag{2.2}$$

The total repair time is the part of the service interruption time that is actually spent by a rerouting mechanism to restore a communication after a link has failed.

2.2 Protection at the MAC and physical layers: self-healing rings

A ring network is a network topology where all nodes are attached to the same set of physical links. Each link forms a loop. In counter rotating ring topologies, all links are unidirectional and traffic flows in one direction on one half of the links, and in the reverse direction on the other half. Self-healing rings are particular counter rotating ring networks which perform rerouting as follows. In normal operation, traffic is sent from a source to a destination in one direction only. If a link fails, then the other direction is used to reach the destination such that the failed link is avoided. Self-healing rings require expensive specific hardware and waste up to half of the available bandwidth to provide full redundancy. On the other hand, lower layer protection mechanisms are the fastest rerouting mechanisms available as self-healing rings can reroute traffic in less than 50 ms. In this section, we present four MAC and physical rerouting mechanisms which all rely on a counter rotating ring topology: SONET UPSR and BLSR Automatic Protection Switching, FDDI protection switching, and RPR Intelligent Protection Switching.

SONET is a physical layer technology for optical transmission [11] [12]. In SONET, protection with self-healing rings is called “Automatic Protection Switching” (APS [10]) and comes in two flavors. The first one, Unidirectional Path-Switched Ring architecture (UPSR, see Figure 2.2), benefits from 1+1 protection. In 1+1 protection, two rings are used. A source injects exactly the same traffic in reverse directions on both rings. The destination receives the same data on each ring, but takes into account traffic from one ring only. On link failure, the receiver detects the increase of the bit error rate or the absence of traffic on one of the rings, and then decides to take into account the traffic from the other ring. The SONET standards specify that the service interruption time should not exceed 50 ms, which is low enough for the outage to be unnoticeable by customers who participate in a live conversation where voice is carried over a SONET network. While service recovery with this technique meets the 50 ms goal, SONET UPSR requires a substantial amount of dedicated backup resources as half of the links are used for path restoration purpose only.

The second protecting scheme, Bidirectional Link-Switched Ring architecture (BLSR, see Figure 2.3), benefits from 1:1 protection. In 1:1 protection, every link can carry both regular traffic and backup traffic at the same time and thus does not require dedicated backup links. On a link failure, the node upstream of the failed link wraps traffic from one ring to another ring in the reverse direction so that traffic still can reach its destination. BLSR is as fast as Unidirectional Path-Switched protection and does not waste as many resources, as there is no notion of dedicated primary and backup link [11].

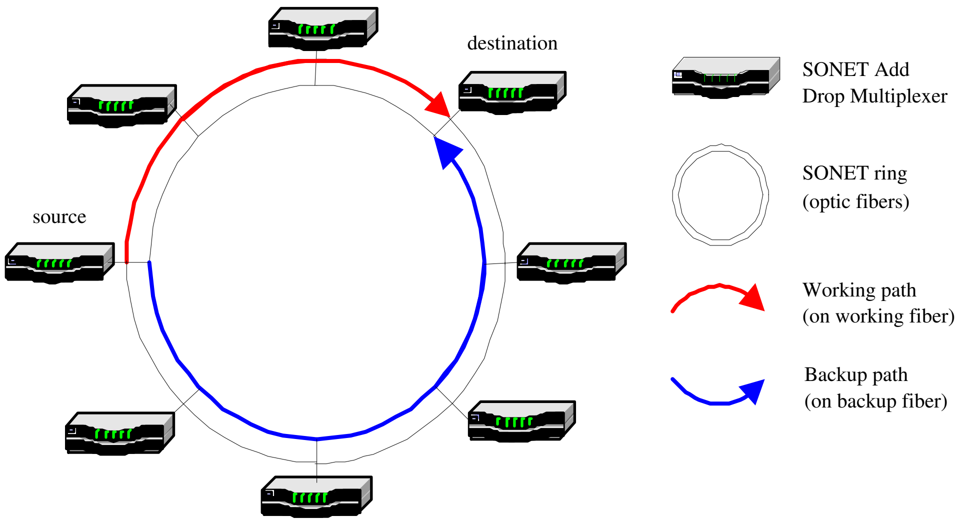


Figure 2.2: **SONET self-healing ring: Unidirectional Path-Switched Ring architecture.** UPSR achieves 1+1 protection.

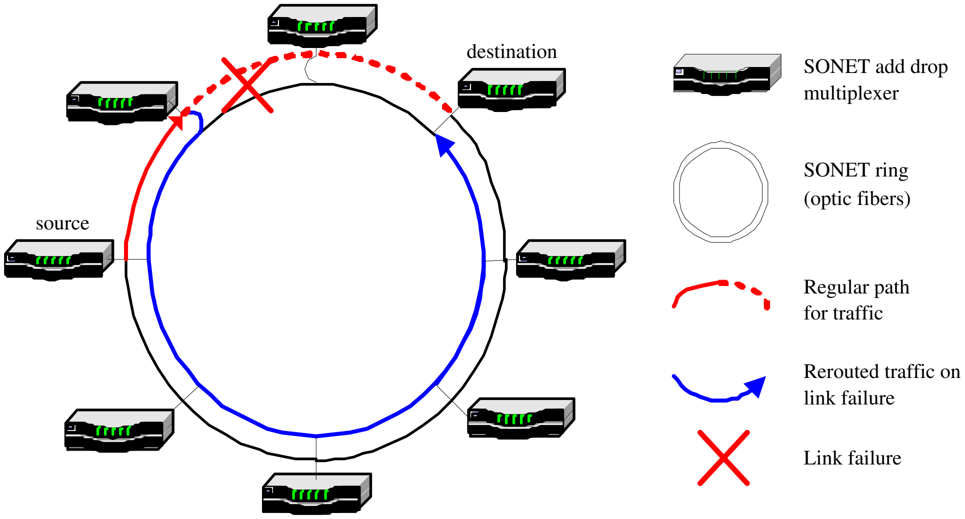
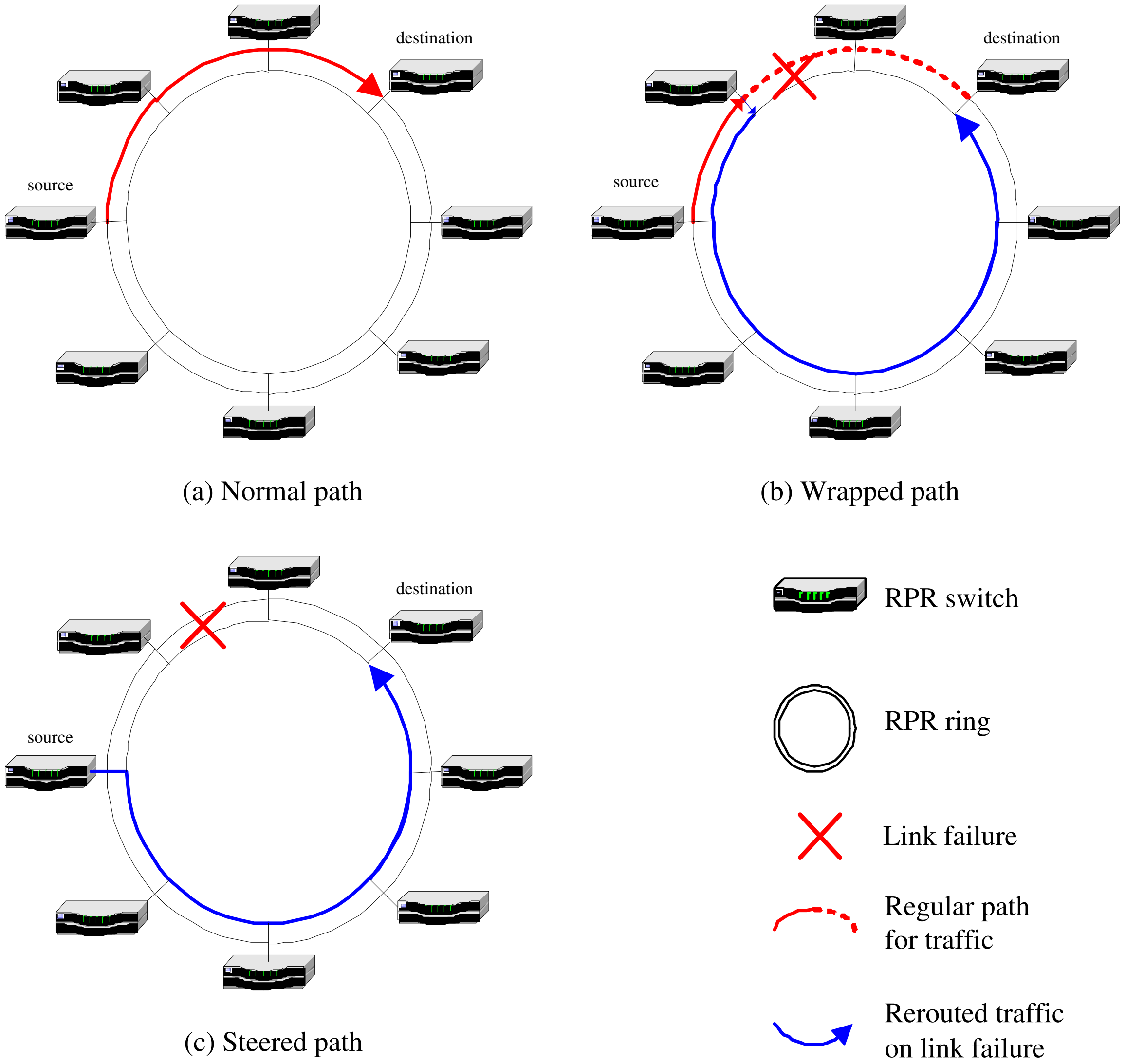


Figure 2.3: **SONET self-healing ring: Bidirectional Link-Switched Ring architecture.** BLSR achieves 1:1 protection.

The MAC layer provides the means for IP to send packets over a local area network. Fiber Distributed Data Interface (FDDI [4]) implements at the MAC layer a protection mechanism that is similar to SONET BLSR. FDDI runs over dual counter rotating rings. In normal operation, traffic is sent on one ring only. Like BLSR, FDDI wraps paths when a link failure is detected and uses the second ring only as a backup ring. Therefore, FDDI implements 1+1 protection and requires full link redundancy.

Resilient Packet Ring is a more recent MAC protocol designed to run on multiple counter-rotating rings (see Figure 2.4(a)) [37]. In RPR, path protection is called Intelligent Protection Switching (IPS). IPS can be viewed as an enhanced SONET BLSR mechanism. Indeed, when a link failure occurs, traffic is first wrapped exactly like SONET BLSR does (Figure 2.4(b)). The emitting node is notified of the failure and changes the ring on which it sends traffic (Figure 2.4(c)). The new path taken by packets is therefore shorter than the wrapped path, resulting in both shorter delays for packets and a better utilization of the available resources.

The lower layer rerouting mechanisms are fast because the nodes that detect the failure perform themselves instantaneously the switchover step, bypassing the notification step. The total repair time is therefore reduced to the detection time ($T_{repair} = T_{detect}$).



(a) Normal path

(b) Wrapped path

(c) Steered path

Figure 2.4: **Intelligent Protection Switching with a Resilient Packet Ring.**

IPS is an enhancement of BLSR.

2.3 Network layer protection

Packet switching networks like the Internet are inherently resilient to link failures. Routing protocols [35] [46] [49] [57] [60] [63] take account for topology changes such as a link failure and recompute routing tables accordingly using a shortest path algorithm. When all routing tables of the network are recomputed and have *converged*, all paths that were using a failed link are rerouted through other links. However, convergence is fairly slow and takes usually several tens of seconds. Part of the reason for this is that routing protocols use timers to detect link failure with coarse granularity (1 second) making the T_{detect} term in Equation 2.2 large compared with lower layer rerouting mechanisms. Second, all routers in the network have to be notified of the failure. Propagating notification messages is done in an order of magnitude of tens of millisecond which makes T_{notif} negligible compared with T_{detect}. Indeed routers only need to forward the messages with no additional processing. Finally routing tables have to be recomputed before paths are switched. Recomputing routing tables implies using CPU intensive shortest path algorithms which can take a time $T_{switchover}$ of several hundred milliseconds in large networks.

In [1], the authors argue that it is possible to perform IP rerouting in less than one second by shrinking the T_{detect} and $T_{switchover}$ terms of Equation 2.2. First, they propose to use subsecond timers to detect failures and decrease the value of the T_{detect} term. Second, they suggest that routing convergence is slow due to the obsolescence of the shortest path algorithms employed in current routing protocols which would be able to recompute routing tables at the millisecond scale if faster, more modern algorithms were used. In summary, expected rerouting times in networks using modified routing protocols are below one second, but the authors also argue that millisecond network layer rerouting is achievable. However, implementation of those guidelines requires major modifications in current routing algorithms and routers.

2.4 MPLS Unicast Fast Reroute

Rerouting at the MAC and physical layer is fast but requires dedicated hardware. On the other hand, IP rerouting is slow but does not rely on any specific topology and is implemented in every router over the Internet. MPLS, which is implemented between the IP and MAC layers, supports rerouting mechanisms that provide a trade-off between repair speed and deployment cost.

Several methods have been proposed to reroute unicast traffic in MPLS [58]. We present here the fastest MPLS rerouting mechanism available, *MPLS Fast Reroute* [34]. A slower, less complex mechanism can be found in [64]. A comparison of different MPLS rerouting mechanisms can be found in [28].

Fast Reroute relies on pre-planning and requires that a backup path is computed and advertised before a link failure can be repaired. Figure 2.5 illustrates unicast Fast

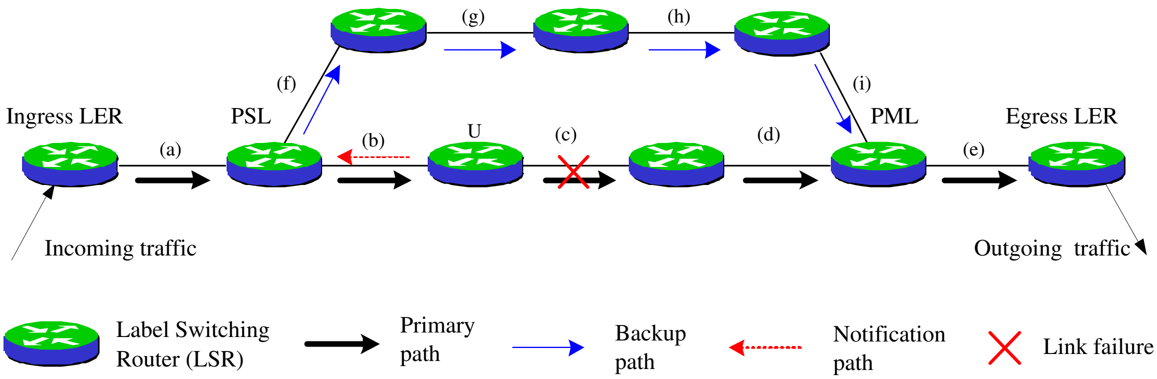


Figure 2.5: **Unicast Fast Reroute mechanism.** When link (c) fails, the traffic that flows from the ingress LER to the egress LER is rerouted from the primary path to the backup path.

Reroute. Suppose traffic goes from the Ingress LER to the Egress LER of an MPLS domain through the primary LSP (a, b, c, d, e), and that the backup LSP (f, g, h, i) has already been set up. The first router of the backup path is called PSL (Path Switching LSR), and the last router of the backup path is called PML (Path Merging LSR). If link c fails, the router U upstream of c detects the failure and sends the packets whose destination was the Egress router back to the Ingress router. When the first of those packets reaches the PSL, the PSL knows that a failure has occurred. Alternatively, U could send a notification message to the PSL to let it know of the failure. The PSL then forwards on the backup path the packets coming back from U. This ensures that no packet is lost after the fault is detected by U, during the notification step of the rerouting mechanism. The switchover step is instantaneous as the PSL only needs to start forwarding the packets coming from the Ingress LSR going to the Egress LSR on the backup path instead of the primary path. A disadvantage of Fast Reroute is that the packets sent during the notification step arrive out of order. Also, some packets will cover up to three times the distance between the ingress and the egress router if the PSL is the ingress node, the PML is the egress node, and the failed link is the last link on the primary path before the PML. The major advantage of MPLS unicast Fast Reroute, however, is that rerouting is fast and no packet is lost after the fault is detected. When the failed link is physically repaired, node U sends a notification message to the PSL which can send traffic back from the backup path to the primary path in the switchback step. Switchback, like switchover, is instantaneous.

MPLS Fast Reroute is faster than IP rerouting but slower than MAC or physical layer rerouting. Indeed, Fast Reroute saves the switchover step that is expensive in IP rerouting, but does not get rid of the notification step as lower layer mechanisms do. Detection can be performed as in SONET using dedicated hardware or like in routing by sending probes over the link regularly. Detection times are expected to be in the order of 1 to 100 milliseconds. Notification takes the same amount of time as with network layer rerouting, that is, a few milliseconds. All in all, the repair time is $T_{service} = T_{detect} + T_{notif}$ and is expected to be as fast as SONET [34].

2.5 Multicast fault recovery

Unicast rerouting mechanisms can protect multicast routing trees from link failures by setting up a backup path from each group member to the core of a core based tree or the source of a shortest path tree. Protecting multicast routing trees from link failures requires to compute, advertise and reserve bandwidth for many unicast backup paths, some of them possibly having links in common and therefore leading to link capacity wastes if bandwidth is reserved on the backup paths. A few rerouting mechanisms applicable to ATM or MPLS that take multicasting into account have been proposed in the literature.

In [43], an algorithm that builds a primary and a backup tree at the same time is presented. The algorithm minimizes the bandwidth that is used by the primary and the backup paths. The algorithm selects in turn every member of the group, starting with the source (in the case of shortest path trees) or center (in the case of center-based trees). For each member, two disjoint paths from the source or center to this member that respect certain resource availability properties are computed. One path is inserted in the primary tree and the other in the backup tree. Bandwidth used by the trees is minimized. However, since a backup path protects the tree for all possible link failures, the total bandwidth that should be reserved for the backup tree is the same as the bandwidth reserved for the primary tree. Similar algorithms that do not take bandwidth utilization into consideration but also build a primary and a backup tree simultaneously are discussed in [40] and [47].

In [69], the authors introduce an online mechanism able to repair ATM multicast routing trees. When a failure occurs in an ATM multicast routing tree, the multicast routing tree is split into two smaller trees, T_1 and T_2. One of these smaller trees T_1 contains the source or center of the tree, and the other T_2 is the tree rooted at the switch S downstream of the failed link with regards to the source or center of the tree. When S detects the link failure, S sends a failure notification message that contains its unique switch identifier to all of its neighbors. Each neighbor forwards in turn the notification message to its own neighbors and so on, thus flooding the network with the notification message. The first switch S' of T_1 that receives the notification message replies to S in order to set up a backup path between itself and S. A candidate backup path is the path taken by the notification message from S to S', but any other path between S and S' can be chosen as a backup path. This backup path is inserted in the multicast routing tree such that S becomes a downstream node or child of S' in the reconfigured multicast routing tree. As expected, this mechanism compares well compared with IP rerouting but poorly compared with SONET, since the backup path is computed online instead of being pre-planned. According to the simulations in [69], this mechanism can repair multicast routing trees in 400 ms. The advantage of this ATM multicast routing tree rerouting mechanism is that it repairs a tree with a single backup path.

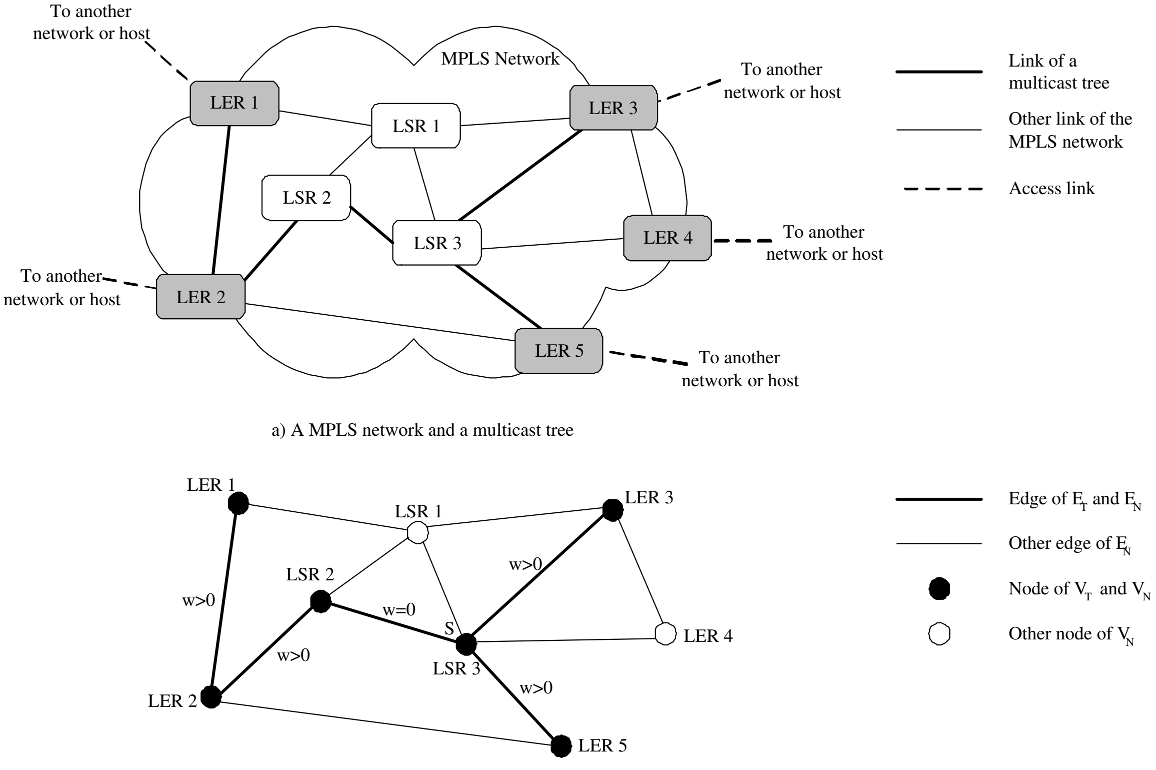
Current multicast rerouting mechanisms protect multicast groups from any link failure. As a result many backup paths must be computed, advertised and bandwidth must be allocated for the backup traffic which will use the backup paths only on a link failure. In this thesis, we consider the case where CPU and bandwidth resources are sparse, and assume that only one backup path is deployed in a multicast routing tree. In the next section, we develop an algorithm that computes this backup path.

3

A multicast routing tree repair algorithm

In this chapter, we propose an algorithm for rerouting multicast communications. The presented algorithm presents a trade-off between resource requirements and the level of protection when pre-planning rerouting. Given a network and a multicast group, we compute a single, pre-planned backup path for the multicast routing tree corresponding to the multicast group. The backup path is selected so that the number of group members dropped from the multicast communication if a single link of the network fails is minimized.

In Section 3.1, we model the network as a graph and the multicast group as a tree. In Section 3.2, we propose a graph algorithm that aims at computing the aforementioned backup path. Then, we propose extensions to our algorithm in order to update a backup path computed prior to changes in the multicast group, e.g. joining and leaving members. Finally, in Section 3.3, we determine the time complexity of the algorithms in the average case and the worst case.



a) A MPLS network and a multicast tree

b) Model

Figure 3.1: **Network and multicast group model.**

3.1 Problem modeling

The network considered in this chapter consists of a set of *routers* which can be either *Label Switching Routers* (LSRs) or *Label Edge Routers* (LERs). The modeling process is shown in Figure 3.1 for an example. Routers are connected by point-to-point *links.* End-hosts are attached to LERs of the MPLS network either directly or via other networks. Links between an LER of the considered network and an end-host or a router of another network are called *access links*. A *multicast group* is a set of hosts which communicate together. Data that is sent by a multicast group member to a multicast group is received by all other group members.

In this section, we model the network as a *graph* $G = (V, E)$. The set V of the *vertices* of the graph contains the routers (LSRs and LERs) of the network. The set E of the *edges* of the graph contains the links of the network. Set E does not contain the access links. All links are assumed to be bidirectional.

Further, we assume that a multicast communication has been established over the network. Let $T = (V_T, E_T)$ be the undirected link weighted tree which maps the multicast routing tree of the multicast communication. The multicast routing trees can be organized as a shortest path tree or a core based tree. In the case where the multicast routing tree is a core based tree, we assume that the core is not a LER. We denote by S the source of T if T is a shortest path tree or the core of T if T is a core based tree. The set V_T of the *nodes* of the tree contains the routers of the multicast routing tree. The set E_T of the *links* of the tree contains the links of the multicast routing tree. Tree T is embedded in graph G and therefore $V_T \subset V$ and $E_T \subset E$.

In a tree, the children of a node i are the nodes immediately downstream of i with regards to S and the parent of i is the node immediately upstream of i with regards to S. Let $child_i \subset V_T$ be the set of the children of node $i \in V_T$ and $parent_i \in V_T$ be the parent of node i. Similarly, given a link l, let $down_l \subset E_T$ be the set of the links immediately downstream of link l with regards to S. Let $sub_T(i) = (V_{sub_T(i)}, E_{sub_T(i)})$ be the subtree of T with regards to S and which root is node i. Let $lsub_T(l) = (V_{lsub_T(l)}, E_{lsub_T(l)})$ be the subgraph of T which contains node i upstream of l with regards to S, link l, and the subtree rooted at the node j downstream of l with regards to S. By definition, $lsub_T(l) = (\{i\} \cup V_{sub_T(j)}, \{l\} \cup E_{sub_T(j)})$ is a tree.

We define the weight $w_{T,l}$ of a link $l \in E_T$ as follows. Consider a node $i \in V_T$ and link $up_i \in E_T$ immediately upstream of i with regards to S. If i is a LER, w_{T,up_i} is the number of multicast group members that send or receive multicast traffic via an access link attached to node i. If i is a LSR, then let $w_{T,up_i} = 0$. LERs of the MPLS network can learn the number of multicast hosts accessible via each access link by a signaling protocol like IGMP v3 [16]. If the information on the number of multicast hosts accessible via each access link is not available in the MPLS network, we set to "1" the weight of the link upstream of each LER which sends or receives multicast traffic for the considered multicast group via at least one of its access links. In summary, link weights are assumed to be known at multicast routing tree establishment time. We call a *receiver* a node immediately downstream of a link l with regards to S such that $w_l \neq 0$. Receivers are LERs which transmit multicast traffic to or from multicast group members over an access link. A *leaf* of a tree T is a node $i \in V_T$ such that $child_i = \emptyset$. Let $L_T \subset V_T$ be the set of the leaves of T. Note that all leaves are receivers, but a receiver is not necessarily a leaf. Indeed, a LER that forwards traffic

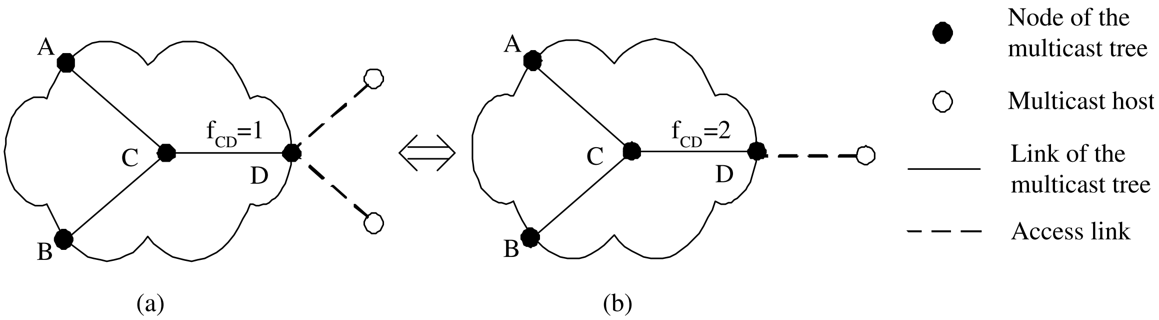


Figure 3.2: **Link failure rate weight.** We consider that the failure of link CD when f_{CD}=1 and when two multicast hosts are attached to D is equivalent to the failure of link CD when f_{CD}=2 and when one multicast host is attached to D.

from the tree over access links can also forward the same traffic to LSRs or other LERs of the tree before this traffic reaches an access link. In practice, such a LER pops labels before sending packets over the access link and swaps labels before sending packets to other routers of the MPLS domain. We introduced this capability as mixed L2/L3 forwarding in Section 1.3.4. In Figure 3.1 for instance, LER 2 performs mixed L2/L3 forwarding. If only one host is attached to LER 2 via the access link of LER 2, then the weight of the link between LER 2 and LSR 2 is "1" while LER 2 is not a leaf for tree T.

We assume that the only possible failure for a link is a link cut. When a link l fails, it is removed from E. The failure rate F_l for a link $l \in E$ is the average number of failures per unit of time for link l. The *Mean Time Between Failures* (*MTBF*) of a link is the time that passes before the link fails. If the Mean Time Between Failures $MTBF_l$ of a link l is known, then $F_l = \frac{1}{MTBF_l}$. We assume that the failure rate is known for each link of the network and that link failure rates are low enough to consider that at most one link fails at any given time. We associate a positive *link failure rate weight* f_l to each link $l \in E$ which is proportional to the link failure rate, relatively to the failure rates of the other links of the network. For instance, if the network is constituted by two different kinds of links, and if links of the first kind fail twice as often as links of the second kind, then $f = 2$ for the links of the first kind and $f = 1$ for the links of the second kind. In the case where failure rate information is not available, we assume that the failure rate is the same for every link in the network, i.e.:

$$\forall l \in E, f_l = 1.$$

If $l \in E_T$ fails, then tree T is split in two trees T_1 and T_2. Let T_1 be the tree which contains S and let T_2 be the other tree. All multicast hosts attached to a LER of T_2 are *dropped* from the multicast communication, and all LERs of T_2 are dropped from tree T. If LER i is dropped from T then w_{T,up_i} multicast hosts are dropped from the multicast communication. In this thesis, we make the following design choice. We consider that dropping twice a given number of hosts if a link with a given failure rate fails (Figure 3.2(a)) is equivalent to dropping the given number of hosts if a link with twice the failure rate fails (Figure 3.2(b)).

A path P_{N_1,N_p} between two vertices N_1 and N_p of graph G is a sequence of vertices $(N_1, N_2, \ldots, N_p)$ such that $\forall i \in [1..p-1], (N_i, N_{i+1}) \in E$ and $\forall i \neq j, V_i \neq V_j$. We denote by $V_{P_{N_1,N_p}} = \{N_i | i \in [1..p]\}$ the set of the nodes of P_{N_1,N_p} and by $E_{P_{N_1,N_p}} = \{(N_i, N_{i+1}) | i \in [1..p-1]\}$ the set of the links of P_{N_1,N_p}. In a tree $T = (V_T, E_T)$, there exists a unique path between any two nodes $A, B \in V_T$. Therefore, there exists a unique path $P_{S,A}$ between root S and node A on the one hand, and $P_{S,B}$ between S and B on the other hand. Let node $S'_{A,B}$ be the node $N_q \in V_T$ such that $P_{S,A} = (S, N_1, N_2, \ldots, N_q, N_r, \ldots, A)$, $P_{S,B} = (S, N_1, N_2, \ldots, N_q, N_s, \ldots, B)$ and $N_r \neq N_s$. Node $S'_{A,B}$ is called the *Least Common Ancestor* (LCA) of A and B.

Definition 1 *The* Protected Path $PP_{i,j}$ *is the unique path* $(N_1, \ldots, N_p)$ *between nodes* $i, j \in V_T$ *such that:*

$$N_1 = i, N_p = j \textit{ and } (\forall i \in [1..p-1], N_i \in V_T \wedge (N_i, N_{i+1}) \in E_T).$$

Definition 2 *A* Backup Path $BP_{i,j}$ *is a path* $(N_1, \ldots, N_p)$ *between two nodes* $i \in V_T$ *and* $j \in V_T$ *such that:*

$$N_1 = i, N_p = j \textit{ and } (\forall i \in [2..p-1], N_i \in (V \backslash V_T) \cup \{i, j\}) \textit{ and } (\forall i \in [1..p-1] (N_i, N_{i+1}) \in E \backslash E_T).$$

While a protected path is unique, a backup path may not be unique for a given pair of nodes. Backup paths protect trees from link failures by providing path redundancy. A backup path and tree T are link disjoint so that the failure of a link in the tree does not prevent the backup path to reroute traffic. Similarly, a backup path and tree T are vertex disjoint so as to avoid any interference between non-rerouted traffic and rerouted traffic at any node of a backup path.

We now show that, given two nodes i and j of tree T, if a link of the protected path $PP_{i,j}$ fails then it is possible to reroute traffic through a backup path $BP_{i,j}$ without dropping any member of the multicast group. More formally, it is possible to transform a tree where a link has failed into a new tree T' *with the same total link weight* by appending the backup path to the original tree.

Claim 1 *Given a tree $T = (V_T, E_T)$, two nodes $i, j \in V_T$, the protected path $PP_{i,j}$, a backup path $BP_{i,j}$, and a link $b \in E_{PP_{i,j}}$, there exists a tree $T' = (V_{T'}, E_{T'})$ such that*

1. $V_{T'} = V_T \cup V_{BP_{i,j}}$
2. $E_{T'} = \{E_T \cup E_{BP_{i,j}}\} \setminus \{b\}$
3. $\sum\limits_{l \in E_{T'}} w_{T',l} = \sum\limits_{l \in E_T} w_{T,l}$

Proof:

Let $T_1 = (V_{T_1}, E_{T_1})$ and $T_2 = (V_{T_2}, E_{T_2})$ be the two smaller trees that result from the removal of link b in tree T. Trees T and T_1 are rooted at node S. Assume without loss of generality that $j \in V_{T_1}$ and $i \in V_{T_2}$. Let i be the root of tree T_2. Trees T_1 and T_2 are link and node disjoint therefore the graph $T' = (V_T \cup V_{BP_{i,j}}, \{E_T \cup E_{BP_{i,j}}\} \setminus \{b\})$

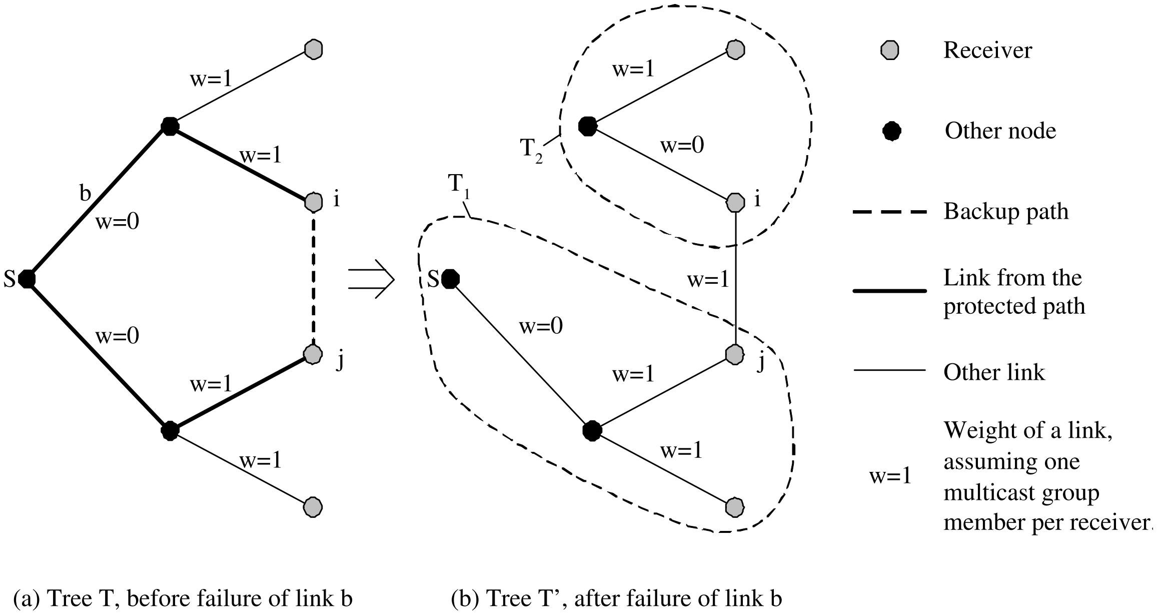

w=1
w=1
b
w=0
i
S
w=0
j
w=1
w=1
T2
w=1
w=0
i
T1
S
w=1
w=0
j
w=1
w=1
Receiver
Other node
Backup path
Link from the protected path
Other link
w=1
Weight of a link, assuming one multicast group member per receiver.
(a) Tree T, before failure of link b
(b) Tree T', after failure of link b

Figure 3.3: **Protection of a tree from a link failure on the protected path with a backup path.** When link b fails, multicast traffic that was using tree T is rerouted to use tree T' so that no multicast group member is dropped.

which results from merging T_1 and T_2 via backup path $BP_{i,j}$ is a tree, hence (1) and (2). Let S be the root of T'. The sum $\sum_{l \in E_T} w_{T,l}$ is the total number of multicast hosts attached to all LERs of T. Similarly, the sum $\sum_{l \in E_{T'}} w_{T',l}$ is the total number of multicast hosts attached to all LERs of T'. Since $V_{T'} = V_T \cup V_{BP_{i,j}} = V_T \cup (V_{BP_{i,j}} \setminus \{i,j\})$ and no multicast host is attached to a vertex of $V_{BP_{i,j}} \setminus \{i,j\}$, the numbers of multicast hosts attached to any LER of T and T' are the same, hence (3). □

On a link failure in tree T rooted at S, T is split into two trees T_1 and T_2 as

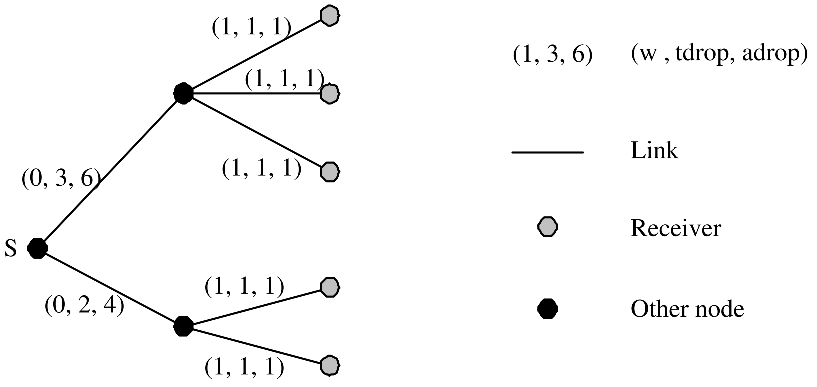


Figure 3.4: **Weight *w* and metrics *tdrop* and *adrop* for all links of a sample tree.** In this example, we assume that all seven links of the tree have the same failure rate ($\forall l \in E, f_l = 1$) and that each receiver is attached to one multicast group member.

described above and all receivers in T_2 are disconnected from S. If a backup path $BP_{i,j}$ is inserted in T (see Figure 3.3(a)), then when any link b from the protected path $PP_{i,j}$ fails a new tree T' replaces T to carry multicast traffic and no receiver is dropped (see Figure 3.3(b)), thus improving the resilience of the tree as defined in the previous section.

Let $tdrop_l$ be the number of hosts attached to receivers that are dropped from tree T when link $l \in E_T$ fails and no backup path is set up. Since $E_{lsub_T(l)}$ is the set that contains at the same time link l and all links in the subtree of T rooted at the node immediately downstream of l with regards to S, then, by definition of $tdrop_l$:

$$tdrop_l = \sum_{k \in E_{lsub_T(l)}} w_k.$$

Similarly, we denote by $adrop_l$ the number of hosts attached to receivers that are dropped from the tree T on the single failure of link l or any link downstream of l when no backup path is set up weighted by the link failure rates. By definition:

$$adrop_l = \sum_{k \in E_{lsub_T(l)}} f_k \; tdrop_k.$$

In Figure 3.4, we give the values of w, *tdrop* and *adrop* for all links on an example. In the following, we use metric *tdrop* to formally define the *resilience* of a tree, and metric *adrop* to speed up resilience computations.

We now introduce a formal definition for the resilience of a tree. We have seen how a link failure partitions a tree T into two trees T_1 and T_2. If a backup path $BP_{i,j}$ is set up in T, then according to Claim 1 it is possible to build a new tree T' such that no receiver is dropped from the communication if any link from the protected path $PP_{i,j}$ fails. Therefore the weighted number of receivers dropped from a multicast communication using a tree protected by a backup path $BP_{i,j}$ on a single link failure is:

$$R_d(i,j) = R_d(j,i) = \sum_{\substack{k \in E_T \\ k \notin E_{PP_{i,j}}}} f_k \; tdrop_k.$$

Moreover, assuming that no backup path is set up in T, the number of hosts dropped from a multicast communication on an single link failure is:

$$R_t = \sum_{k \in E_T} f_k \; tdrop_k.$$

Quantity R_t is a constant that depends on the tree topology only whereas R_d also depends on the end nodes i and j of the backup path $BP_{i,j}$. Quantity R_t measures the impact of a link failure on a tree that is not protected by any backup path, while R_d measures the impact of a link failure on a tree where a backup path is set. The difference between these two quantities is the number of hosts *not* dropped from the communication after traffic is rerouted over the backup path.

Definition 3 *The resilience of a tree $T = (V_T, E_T)$ protected by a backup path between nodes $i, j \in V_T$ is defined as:*

$$R(i,j) = R_t - R_d(i,j).$$

In Chapter 2, we have defined the resilience of a network as the *"ability of a network to keep services running despite a failure"*. Therefore, according to the definition from Chapter 2, a tree protected by a backup path is resilient when a link failure in the tree with the backup path set up leads to a low number of group members being dropped from the communication. The metric $R(i,j)$ measures how well a backup path protects a multicast routing tree and thus matches the definition of the resilience given in Chapter 2. The higher this metric, the more resilient the tree protected by the backup path. A resilience is a number between 0 and R_t. A resilience of R_t

means that on any link failure in the tree, no multicast group member is dropped from the multicast communication. On the other hand, a resilience of 0 implies that the backup path does not prevent any group member from being dropped on any single link failure.

Definition 4 *A backup path $BP_{A,B}$ achieves optimal path protection in a tree T when:*

$$R_d(A, B) = \min_{i,j \in V_T} R_d(i, j).$$

According to Definition 3, $R(i, j) = R_t - R_d(i, j)$ where R_t is a constant. Therefore maximizing the resilience $R(i, j)$ boils down to minimizing the quantity $R_d(i, j)$. A backup path which maximizes the resilience of a tree also achieves optimal path protection for the tree. In the next section, we propose an algorithm which aims at finding a backup path that maximizes the resilience of a tree by minimizing quantity R_d.

3.2 Maximization of the resilience of a tree with a single backup path

In this section, we present an algorithm that aims at determining the backup path which maximizes the resilience of a tree. Finding the backup path itself is performed by a shortest path algorithm. We introduce an incremental version of this algorithm that takes into account modifications of a tree which models a multicast group where hosts can leave and join dynamically.

3.2.1 Main algorithm

We suppose here that no backup path has been previously computed for tree T. Our algorithm (see Algorithm 1) proceeds as follows. We first compute the *tdrop* and *adrop* metrics for each link of the tree, and preprocess the tree so as to speed up further Least Common Ancestor (LCA) computations required to compute metrics R_d. Then, we compute in turn R_d for all pairs of nodes constituted by a leaf of the tree and another node of the tree. We denote by (A, B) the pair of nodes of the tree which minimizes the set of the metrics R_d previously computed. A shortest path algorithm computes the backup path between A and B. If the shortest path algorithm fails to find a backup path, then no backup path achieves optimal path protection in tree T.

Claim 2 *If Algorithm 1 returns a backup path then this backup path maximizes the resilience of T.*

Proof:

Suppose that metrics *adrop* and *tdrop* are known for each link of the tree (lines 1 to 3). We first prove by contradiction that at least one end node of a backup path which minimizes R_d is a leaf of T. Suppose the end nodes i and j of a backup path that minimizes R_d are known and that, by contradiction, i is not a leaf of T.

Let $k \in child_i$ (see Figure 3.5). Then:

$$R_d(i,j) = K + adrop_{ik}$$

and:

$$R_d(k,j) = K + \sum_{h \in child_k} adrop_{kh}$$

where K is the part of metric R_d that is common to $R_d(i,j)$ and $R_d(k,j)$. More specifically, K takes into account all link failures except for link (i, k) and for all links downstream of node k with regards to root S. Since:

$$adrop_{ik} = f_{ik}\ tdrop_{ik} + \sum_{h \in child_k} adrop_{kh}$$

then:

$$R_d(k,j) < R_d(i,j)$$

therefore the pair of nodes (i,j) does not minimize R_d, which contradicts the hypothesis. Therefore, at least one of the end nodes A and B of the backup path $BP_{A,B}$ which maximizes the resilience of T is a leaf. We call A the end node that is a leaf. The other end node B has no such restriction and may or may not be a leaf. By construction of nodes A and B in lines 5 to 8, a backup path between A and B minimizes R_d and thus maximizes the resilience of the tree. The backup path is computed in line 9 using a shortest path algorithm between A and B over the graph

Algorithm 1 Computation of the optimal backup path that maximizes the resilience of a tree T in a graph G.

```
$FIND\_OPTIMAL\_BACKUP\_PATH(tree\ T, graph\ G)$

1. for each $l \in E_T$ do
2.         compute $tdrop_l$ and $adrop_l$;
3. endfor
4. Preprocess $T$ (build auxiliary trees) to speed up the computations of Least
   Common Ancestors;
5. for each pair $(Y, Z) \in L_T \times V_T | Y \neq Z$ do
6.         compute $R_d(Y, Z)$;
7. endfor
8. Find the pair of nodes $(A, B)$ such that $R_d(A, B) = \min_{\substack{Y \in L_T \\ Z \in V_T \\ Y \neq Z}} R_d(Y, Z)$;
9. Compute shortest path between $A$ and $B$ in graph $G' = ((V \setminus V_T) \cup \{A, B\}, E \setminus E_T)$;
```

$G' = ((V \setminus V_T) \cup \{A, B\}, E \setminus E_T)$ so that the backup path and the tree are node and edge disjoint. □

Definition 5 *A near-optimal backup path $BP_{A,B}$ is a backup path such that:*

$$R_d(A, B) = \min_{\substack{i,j \in V_T \\ BP_{i,j}\ exists}} R_d(i, j).$$

If no optimal backup path is found, then it is possible to extend Algorithm 1 to find a *near-optimal* backup path. The extended algorithm (see Algorithm 2) proceeds as follows. As in Algorithm 1, metrics *tdrop* and *adrop* are computed for all links of the tree and T is preprocessed to speed up LCA computations in lines 1 to 4.

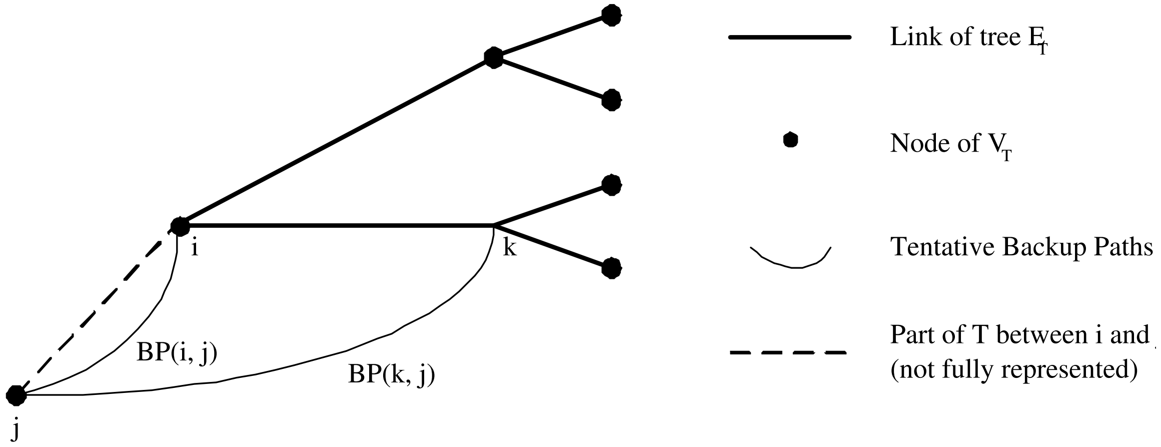


Figure 3.5: **Proof of the algorithm.** It can be shown that $R_d(j, k) < R_d(j, i)$. Using a recursion, we show that at least one of the end nodes of the backup path must be a leaf.

For any pair of nodes $n_k \in V_T \times V_T$, the value of the metric R_d does not depend on the order of the nodes of the pair, and is zero if both nodes of the pair are the same. Therefore, in a graph which contains $|V_T|$ nodes, there is a total number of $N_{pairs} = \frac{|V_T|(|V_T|-1)}{2}$ possible pairs of end nodes for a backup path.

In lines 5 to 7 of Algorithm 2, metrics R_d are computed for all N_{pairs} pairs of nodes in the tree as defined above and the set of these metrics is ordered from the lowest to the highest in line 8. In lines 9 to 13, considering in turn each pair of nodes from the ordered set previously defined and starting from the pair of nodes which minimizes the set of metrics R_d, a shortest path algorithm computes the backup path between the two nodes of the pair until a backup path is found. As with optimal

Algorithm 2 Computation of the near-optimal backup path.

```
$FIND\_NEAR\_OPTIMAL\_BACKUP\_PATH(tree\ T, graph\ G)$

for each $l \in E_T$ do
    compute $tdrop_l$ and $adrop_l$;
endfor
Preprocess $T$ (build auxiliary trees) to speed up the computations of Least Common Ancestors;
for each pair $(Y, Z) \in V_T \times V_T | Y \neq Z$ do
    compute $R_d(Y, Z)$;
endfor
Define the sequence of pairs of nodes $(n_0, n_1, \ldots, n_{N_{pairs}})$ such that
$\forall k \in [0..N_{pairs}], n_k \in V_T \times V_T$ and $\forall i \leq j, R_d(n_i) \leq R_d(n_j)$;
i:=0;
repeat
    Compute shortest path between the nodes of pair $n_i$ in graph
    $G' = ((V \setminus V_T) \cup \{A, B\}, E \setminus E_T)$;
    i:=i+1;
until a backup path is found
```

backup paths, a near-optimal backup path is not guaranteed to exist. More precisely, since Algorithm 2 considers in turn all possible pairs of nodes of the tree and tries to find a backup path between these two nodes until a backup path is found, if no near-optimal backup path exists for tree then no backup path at all exists for the tree. For instance, in the case where $V = V_T$ and $E = E_T$, no optimal nor near-optimal backup path can be found for T.

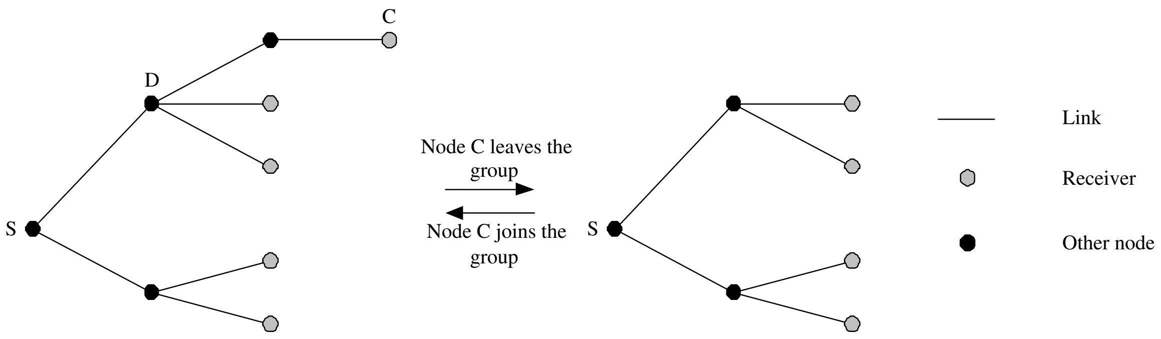


Figure 3.6: **Tree topology modification after a node leaves or joins a multicast group.** Links and nodes are removed or added downstream of node D with regards to S when node C leaves or joins the group.

3.2.2 Incremental version

Multicast group members can join or leave a multicast group at all time, making multicast groups and trees dynamic. When the topology of a tree is modified, it is possible to avoid rerunning all of Algorithm 1 to compute a new backup path. We assume here that a backup path has been established between two end nodes A and B for a tree T with Algorithm 1. We propose Algorithm 3 to handle topological modifications in a multicast routing tree T.

When a node C leaves (or joins) a multicast group, a certain number of nodes and links are removed from (or added to) the multicast routing tree. Consider the path $P_{C,S}$ in tree T. Let D be the first node on $P_{C,S}$ that is either a LER or has more than one child in T. Therefore, when C leaves (or joins) the multicast group, all

Algorithm 3 Computation of the optimal backup path in a graph G that maximizes the resilience of a tree T' resulting from a modification downstream of node D in tree T with regards to the root S of T.

```
$UPDATE\_BACKUP\_PATH(tree\ T, graph\ G, node\ D)$

1. for each $l \in E_{P_{D,S}}$ do
2.         compute $tdrop'_l$;
3. endfor
4. for each pair $(Y, Z) \in L_T \times V_T | Y \neq Z$ do
5.         update $R_d(Y, Z)$;
6. endfor
7. $(A, B) := (Y, Z) \in L_T \times V_T | R_d(Y, Z) = \min_{\substack{Y \in L_T \\ Z \in V_T \\ Y \neq Z}} R_d(Y, Z)$;
8. Compute shortest path between $A$ and $B$ in graph $G' = ((V \setminus V_T) \cup \{A, B\}, E \setminus E_T)$;
```

links and nodes except D of the path $P_{C,D}$ are removed from (or added to) the tree as shown in Figure 3.6. We denote by T' the modified tree and $tdrop'$ the modified metrics $tdrop$ of T'. If $tdrop_l$ is the value of metric $tdrop$ for link l in T, then $tdrop'_l$ is the value of metric $tdrop$ for link l in T'. According to the definition of $tdrop$, if a topology modification occurs downstream of D with regards to S, then the values of the metrics $tdrop$ for links upstream of D with regards to S only are changed.

Algorithm 3 proceeds as follows. First, metrics $tdrop$ are recomputed for links on the path between nodes D and S in lines 1 to 3. Then, in lines 4 to 6, the metrics R_d are updated for all pairs of nodes of tree T'. Let $R'_d(i, j)$ be the new value of the metric R_d taken between nodes i and j in tree T'. It is possible to compute $R'_d(i,j)$ in T' using the value of $R_d(i,j)$ in T and therefore avoid doing all computations again. For instance, if Definition 3 is used to compute the metrics R_d and R'_d, then, when C leaves the group:

$$\begin{aligned} R_d(i,j) &= \sum_{k \in E_T \setminus E_{PP_{i,j}}} f_k \; tdrop_k \\ &= \sum_{k \in E_{P_{C,D}} \setminus E_{PP_{i,j}}} f_k \; tdrop_k \\ &\quad + \sum_{k \in E_{P_{D,S}} \setminus E_{PP_{i,j}}} f_k \; tdrop_k \\ &\quad + \sum_{k \in E_T \setminus (E_{PP_{i,j}} \cup E_{P_{C,D}} \cup E_{P_{D,S}})} f_k \; tdrop_k. \end{aligned}$$

Since all nodes of $P_{C,D}$ are removed from the tree when C leaves the group, neither i nor j can be between C and D. Therefore:

$$\sum_{k \in E_{P_{C,D}} \setminus E_{PP_{i,j}}} f_k \; tdrop_k = \sum_{k \in E_{P_{C,D}}} f_k \; tdrop_k.$$

If we call l the first link on the path from D to C, then the expression above can be further simplified:

$$\sum_{k \in E_{P_{C,D}}} f_k \; tdrop_k = adrop_l$$

and thus:

$$\begin{aligned} R_d(i,j) &= adrop_l \\ &\quad + \sum_{k \in E_{P_{D,S}} \setminus E_{PP_{i,j}}} f_k \; tdrop_k \\ &\quad + \sum_{k \in E_T \setminus (E_{PP_{i,j}} \cup E_{P_{C,D}} \cup E_{P_{D,S}})} f_k \; tdrop_k. \end{aligned}$$

Furthermore:

$$\begin{aligned}
R'_d(i,j) &= \sum_{k \in E_{T'} \setminus E_{PP_{i,j}}} f_k \ tdrop'_k \\
&= \sum_{k \in E_{P_{C,D}} \setminus E_{PP_{i,j}}} f_k \ tdrop'_k \\
&\qquad + \sum_{k \in E_{P_{D,S}} \setminus E_{PP_{i,j}}} f_k \ tdrop'_k \\
&\qquad + \sum_{k \in E_{T'} \setminus (E_{PP_{i,j}} \cup E_{P_{C,D}} \cup E_{P_{D,S}})} f_k \ tdrop'_k \\
&= 0 \\
&\qquad + \sum_{k \in E_{P_{D,S}} \setminus E_{PP_{i,j}}} f_k \ tdrop'_k \\
&\qquad + \sum_{k \in E_{T'} \setminus (E_{PP_{i,j}} \cup E_{P_{D,S}})} f_k \ tdrop_k.
\end{aligned}$$

Consequently:

$$R'_d(i,j) = R_d(i,j) - adrop_l + \sum_{k \in (E_{P_{D,S}} \setminus E_{PP_{i,j}})} f_k (tdrop'_k - tdrop_k). \tag{3.1}$$

Similar expressions are available in the case where C joins a group. Therefore, instead of completely recomputing all metrics R'_d, it is possible to compute the new metrics R'_d by substracting from the old metrics R_d a value of the metric *adrop* and adding products $f_k(tdrop'_k - tdrop_k)$ for links k of a path of a tree. When all metrics R_d are updated, a backup path is computed in lines 7 to 8 using a shortest path algorithm between the two nodes of the pair which maximizes the set of all metrics R_d as in Algorithm 1.

Algorithm 3 is able to handle the modification of a multicast routing tree incurred by any leaving or joining node. In the case where a node is leaving the multicast group, and where this node is not an end node of the backup path computed for the tree topology prior to the modification, a trivial alternative to Algorithm 3 is available to handle the modification. This alternative merely consists in leaving the formerly computed backup path unchanged. Obviously, the unchanged backup path is not guaranteed to be optimal anymore, but no further computation is required to update the backup path. Therefore, to reduce the amount of computations required to update a backup path on a tree topology change, it is possible to set a threshold on the number of modifications of the multicast routing tree and not update the backup path until this threshold is reached. Then, Algorithm 3 can be used to recompute the backup path.

3.3 Complexity analysis

In this section, we first present a fast computation method for the metric R_d based on an expression for R_d derived from its definition, and methods to compute metrics *tdrop* and *adrop*. Second, we give a general expression for the complexity of Algorithms 1, 2 and 3. Then, we give a complexity analysis in both the average case and worst case of a tree embedded in a network for all three algorithms.

3.3.1 Computation of the metrics

Algorithm 1 is articulated around the computation of several metrics. In line 6 of Algorithm 1, metric R_d is computed between two nodes of T. It is possible to compute R_d between two nodes with the definition given in Section 3.1. In Claim 3 we derive a new expression for R_d which makes use of the *adrop* metrics. Computing the *adrop* metrics for all links of the tree and then computing a value for R_d between two nodes in the tree is faster than computing R_d using the definition. We postpone the discussion on the gain in time complexity incurred by this method until Section 3.3.3.

Claim 3 *Let $A, B \in V_T$ and $S' = S'_{A,B}$. Then*

$$R_d(A,B) = \underbrace{\sum_{i \in V_{PP_{A,B}}} \sum_{\substack{j \in children_i \\ j \notin V_{PP_{A,B}}}} adrop_{(i,j)}}_{(1)} + \underbrace{\sum_{k \in E_{P_{S,S'}}} f_k \; tdrop_k}_{(2)} + \underbrace{\sum_{\substack{i \in V_{P_{S,S'}} \\ i \neq S'}} \sum_{\substack{j \in children_i \\ j \notin V_{P_{S,S'}}}} adrop_{(i,j)}}_{(3)} . \tag{3.2}$$

Proof:

We prove Claim 3 using a partitioning of E_T. Let $A, B \in V_T$. We assume that a backup path has been established between nodes A and B. A partitioning of E_T is

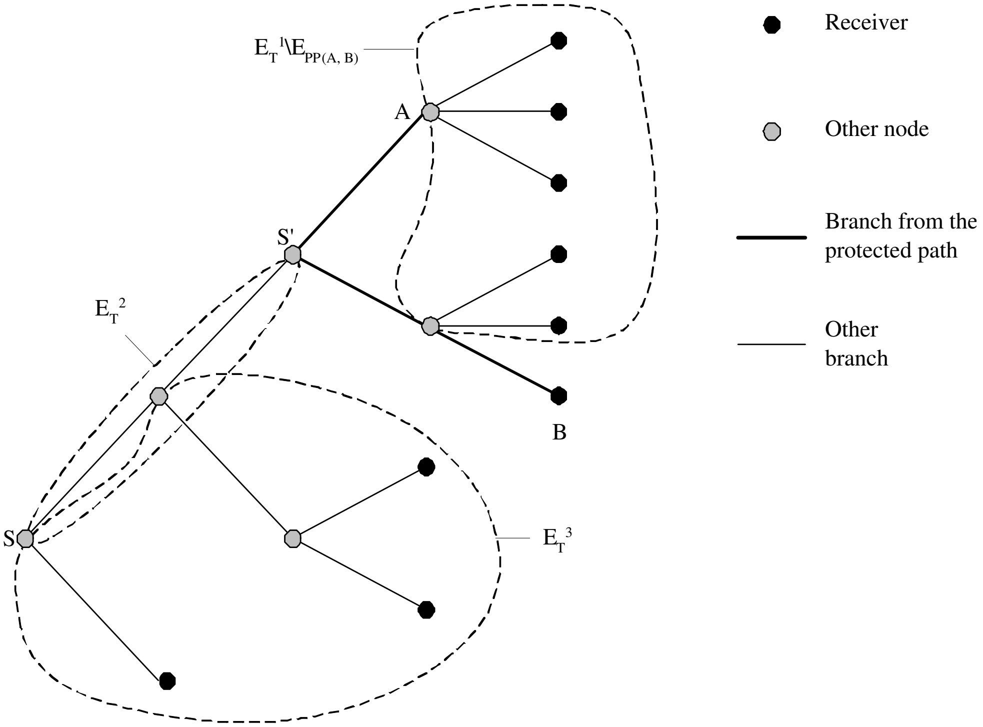

$E_T^1 \backslash E_{PP(A, B)}$
A
S'
E_T^2
S
B
E_T^3
Receiver
Other node
Branch from the protected path
Other branch

Figure 3.7: **Computation of $R_d(A, B)$.**

(E_T^1, E_T^2, E_T^3) with:

$$\begin{cases} E_T^1 &= E_{sub_T(S')}, \\ E_T^2 &= E_{P_{S,S'}}, \\ E_T^3 &= E_{lsub_T}(l) | l \in E_T \wedge parent_l \in V_{P_{S,S'}} \wedge parent_l \neq S' \wedge l \notin E_{P_{S,S'}}. \end{cases}$$

Let $E = E_T \setminus E_{PP_{A,B}}$. By definition of S', $E_{PP_{A,B}} \subset E_T^1$ therefore a partitioning of E is $(E_T^1 \setminus E_{PP_{A,B}}, E_T^2, E_T^3)$ as shown in Figure 3.7. According to the definition of R_d:

$$R_d(A, B) = \sum_{k \in E} f_k \ tdrop_k.$$

Thus, using the partitioning of E:

$$\begin{aligned}
R_d(A,B) &= \sum_{k \in E_T^1 \setminus E_{PP_{A,B}}} f_k \; tdrop_k \\
&\quad + \sum_{k \in E_T^2} f_k \; tdrop_k \\
&\quad + \sum_{k \in E_T^3} f_k \; tdrop_k \\
&= \sum_{i \in V_{PP_{A,B}}} \sum_{\substack{j \in children_i \\ j \notin V_{PP_{A,B}}}} \sum_{l \in lsub_T(i,j)} f_l \; tdrop_l \\
&\quad + \sum_{k \in E_{P_{S,S'}}} f_k \; tdrop_k \\
&\quad + \sum_{\substack{i \in V_{P_{S,S'}} \\ i \neq S'}} \sum_{\substack{j \in children_i \\ j \notin V_{P_{S,S'}}}} \sum_{l \in lsub_T(i,j)} f_l \; tdrop_l
\end{aligned}$$

When injecting the definition of *adrop* in the last line, the equation becomes:

$$R_d(A,B) = \sum_{i \in V_{PP_{A,B}}} \sum_{\substack{j \in children_i \\ j \notin V_{PP_{A,B}}}} adrop_{(i,j)} + \sum_{k \in E_{P_{S,S'}}} f_k \; tdrop_k + \sum_{\substack{i \in V_{P_{S,S'}} \\ i \neq S'}} \sum_{\substack{j \in children_i \\ j \notin V_{P_{S,S'}}}} adrop_{(i,j)}.$$

□

The computation method for R_d we have presented requires the knowledge of the metrics *adrop* and *tdrop*, and computations of Least Common Ancestors. Each of the metrics *adrop* and *tdrop* can be computed for all links of T prior to any evaluation of R_d (lines 1 to 3 in Algorithm 1) with the following recursive relations:

$$tdrop_l = \begin{cases} w_l & \text{if the node downstream of } l \text{ with regards to } S \text{ is a leaf,} \\ \left(\sum_{i \in down_l} tdrop_i \right) + w_l & \text{otherwise.} \end{cases}$$

for *tdrop* and:

$$adrop_l = \begin{cases} f_l \; tdrop_l & \text{if the node downstream of } l \text{ with regards to } S \text{ is a leaf,} \\ \left(\sum\limits_{i \in down_l} adrop_i \right) + f_l \; tdrop_l & \text{otherwise.} \end{cases}$$

for *adrop*.

In [33] the authors show that it is possible to determine a Least Common Ancestor for any pair of nodes in a tree in constant time after a linear preprocessing of T. This tree preprocessing is performed in line 4 of Algorithm 1.

3.3.2 General case

We now analyze the time complexity of Algorithms 1, 2 and 3 in the general case. Let $N = |V|$ be the number of vertices of the graph G, $n = |V_T|$ be the number of nodes of the tree T. The number of links in T is $|E_T| = n - 1$. Let $\ell = |L_T|$ be the number of leaves of the tree.

Consider Algorithm 1. In lines 1 to 3, the metrics *tdrop* and *adrop* are computed for every link of T. Computing each metric for all links of the tree can be performed in linear time with the recursive formula given in Section 3.3.1. Therefore, the time complexity of lines 1 to 3 is $O(n)$.

In line 4, tree T is preprocessed so as to speed up further LCA computations. After the tree preprocessing, it is possible to determine the LCA of any pair of nodes in constant time. A linear tree preprocessing algorithm is proposed in [33], hence a time complexity of $O(n)$ for line 4.

Nodes A and B are determined in lines 5 to 8 by computing the metric R_d between a leaf and another node. Thus, $O(\ell n)$ metrics R_d have to be computed and compared. Computing R_d with Definition 3 involves the addition of products between link failure rate weights and *tdrop* metrics for all links of the tree except on the path between two nodes, hence a complexity of $O(n)$. The complexity of lines 5 to 8 is thus $O(\ell n^2)$ if R_d is computed with its definition. If R_d is computed with the expression from Equation 3.2, then three terms must be added. Term (1) involves to determine the LCA S' of a pair of nodes and the addition of metrics *adrop* for links downstream of vertices of the protected path between the two nodes for which R_d is computed. Determining a LCA is performed in constant time as already stated and at most all n metrics *adrop* are added in Term (1), hence a time complexity of $O(n)$ to compute Term (1) of Equation 3.2. Term (2) involves the addition of products between f and *tdrop* metrics for all links on the path between S' and root S. Since a path can be up to n links long, the time complexity for computing Term (2) of Equation 3.2 is $O(n)$. Same as Term (1), Term (3) requires the addition of up to n terms hence a time complexity of $O(n)$. In summary, computing a single metric R_d between two leaves can be done in time $O(n)$ with either the definition or the method previously introduced, and the time complexity of lines 5 to 8 is also $O(\ell n^2)$ in the general case.

Last, line 9 computes a backup path between two vertices in a graph of $N - n$ edges. Classic Dijkstra's shortest path algorithm [24] solves this problem in time $O((N - n)\log(N - n))$.

Overall, the time complexity of Algorithm 1 is:

$$O(n) + O(n) + O(\ell n^2) + O((N - n)\log(N - n)) = O(\ell n^2 + (N - n)\log(N - n)).$$

Since $\ell = O(N)$ and $n = O(N)$, an upper bound for the complexity of Algorithm 1 is $O(N^3)$.

Now consider Algorithm 2. Lines 1 to 7 of Algorithms 1 and 2 are identical. In line 8, metric R_d is determined for $N_{pairs} = O(n^2)$ nodes, then the set of all these metrics is sorted. Since computing a metric R_d takes time $O(n)$ and sorting a set of size n^2 takes time $O(n^2 \log(n^2)) = O(n^2 \log(n))$, the time complexity of line 8 of Algorithm 2 is $O(n^2 n + n^2 \log(n)) = O(n^3)$. In lines 9 to 13, a shortest path is computed for up to N_{pairs} pairs of nodes, thus the time complexity of lines 9 to 13 is $O(n^2(N - n)\log(N - n))$. Overall the time complexity of Algorithm 2 is:

$$O(n) + O(n) + O(n^3) + O(n^2(N - n)\log(N - n)) = O(n^3 + n^2(N - n)\log(N - n)).$$

Since $n = O(N)$, an upper bound for the complexity of Algorithm 2 is $O(N^3 \log(N))$.

Finally, consider Algorithm 3. The maximum length for a path in a tree of n nodes is n thus the time complexity of lines 1 to 3 is $O(n)$. In lines 4 to 6, ℓn metrics R_d are updated. Updating a single metric R_d has a time complexity of $O(n)$ hence a time complexity of $O(\ell n^2)$ for lines 4 to 6. In line 7, the minimum of a set of N_{pairs} is determined in time $O(n^2)$ and in line 8 a shortest path is computed in time $O((N-n)\log(N-n))$. Therefore, the complexity of Algorithm 3 is:

$$O(n) + O(\ell n^2) + O(n^2) + O((N-n)\log(N-n)) = O(\ell n^2 + (N-n)\log(N-n)).$$

Since $n = O(N)$, an upper bound for the complexity of Algorithm 3 is $O(N^3)$. Although Algorithms 1 and 3 have the same asymptotic time complexity, Algorithm 3 is faster than Algorithm 1 as Algorithm 3 most of the metric computations do need not to be recomputed and updating R_d for a pair of node is requires fewer operations.

In the next sections, we show that the time complexity of Algorithm 1 in the average case is $O(N^2\log(N))$ and give a worst case example where the upper bound $O(N^3)$ is reached. We present similar results for Algorithms 2 and 3.

3.3.3 Average case

The *height* of a tree is the maximum number of edges on the path between the root S of the tree and any leaf. We consider as the average case a tree T with height $\log n$, where the number of children of any node of the tree is bounded by a constant k and where the number of leaves is bounded by n ($\ell = \Theta(n)$).

First, consider Algorithm 1. The computations performed in lines 1 to 4 are the same as in the general case, hence a complexity of $O(n)$. In lines 5 to 8, metric

R_d must be computed between each of the ℓ leaves and all other nodes. Therefore $O(\ell n) = O(n^2)$ metrics R_d must be computed. Suppose we use the definition to compute R_d. Computing R_d involves the addition of products between f and $tdrop$ metrics for all n links of the tree except those on the protected path between A and another leaf. Therefore, $n - 2\log(n)$ products have to be added hence a time complexity of $O(n - 2\log(n)) = O(n)$ to compute each of the $n-1$ metrics R_d, and a complexity of $O(n^3)$ for lines 5 to 8 of Algorithm 1. This is the same result as in the general case.

Now suppose that we use the method presented in Section 3.3.1 to compute R_d between two nodes Y and Z of T. First, part (1) of Equation 3.2 requires the addition of at most k terms for each vertex on the protected path $PP_{Y,Z}$ for which R_d is computed. A protected path consists of at most $2\log(n)$ vertices, thus complexity of Term (1) is $O(2k\log(n)) = O(\log(n))$. Second, the number of links on the path from $S'_{A,B}$ to S is at most $\log(n)$. Determining an LCA S' is done in constant time as previously stated therefore part (2) of Equation 3.2 has a complexity of $O(\log(n))$. Third, Term (3) involves the addition of at most k metrics for each link of the path between S and S', hence a time complexity of $O(k\log(n)) = O(\log(n))$ for Term (3) of Equation 3.2. Overall, the time complexity of the computation of all metrics R_d performed by lines 5 to 8 of Algorithm 1 is $O(n^2)(O(\log(n))+O(\log(n))+O(\log(n))) = O(n^2\log(n))$, instead of $O(n^3)$ if the definition of R_d was used.

The last step of Algorithm 1 computes a shortest path between two nodes in a graph with $(N-n)$ edges. The classic Dijkstra's shortest path Algorithm [24] solves the problem in time $O((N-n)\log(N-n))$.

All in all, the complexity of Algorithm 1 is:

$$O(n)+O(n)+O(n^2\log(n))+O((N-n)\log(N-n)) = O(n^2\log(n)+(N-n)\log(N-n)).$$

Note that if we used the definition of R_d to compute all metrics R_d in lines 5 to 8 then the complexity of Algorithm 1 would be $O(n^3+(N-n)\log(N-n))$, which justifies a posteriori why we introduced an alternate method to compute R_d in Section 3.3.1.

Now consider Algorithm 2. We have shown above that the time complexity of lines 1 to 7 is $O(n^2\log(n))$. Computing and sorting the set of all R_d metrics in line 8 takes time $O(n^2\log(n))$. Computing all possible backup paths takes time $O(n^2(N-n)\log(N-n))$, hence the time complexity of Algorithm 2 in the general case:

$$O(n^2\log(n)+n^2\log(n)+n^2(N-n)\log(N-n)) = O(n^2\log(n)+n^2(N-n)\log(N-n)).$$

Last, consider Algorithm 3. The length of the path between D and S is $\log(n)$ thus computing metrics $tdrop'$ in lines 1 to 3 is done in time $O(\log(n))$. The size of the set $E_{P_{D,S}} \backslash E_{PP_{i,j}}$ is $\log(n)$ therefore updating a metric R_d takes time $O(\log(n))$. Updating all the metrics R_d takes time $O(n^2\log(n))$ (lines 4 to 6). In line 7, determining a minimum of N_{pairs} values takes time $O(n^2)$. In line 8, a shortest path is computed

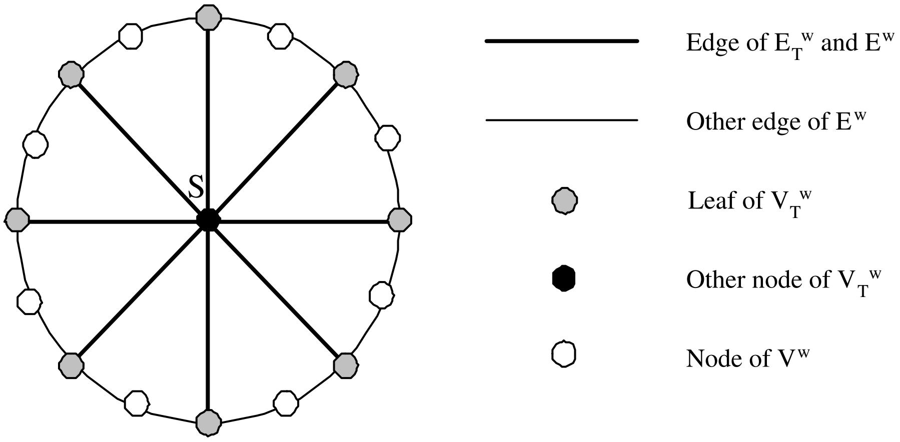


Figure 3.8: **Worst case for Algorithm 1.** With this topology, the complexity of Algorithm 1 reaches the higher bound $O(N^3)$.

in time $O((N-n)\log(N-n))$. The time complexity of Algorithm 3 in the average case is:

$$O(\log(n))+O(n^2\log(n))+O(n^2)+O((N-n)\log(N-n)) = O(n^2\log(n)+(N-n)\log(N-n)).$$

All three Algorithms 1, 2 and 3 are faster in the average case than in the general case. In the next section, we determine the worst case for each algorithm.

3.3.4 Worst case

Consider Algorithm 1. The general expression for its time complexity is $O(\ell n^2 + (N-n)\log(N-n))$. The upper bound $O(N^3)$ can be reached only when $\ell = \Theta(N)$ and $n = \Theta(N)$. Consider the graph $G^w = (E^w, V^w)$ and the tree $T^w = (E_T^w, V_T^w)$

depicted in Figure 3.8. Suppose that all links from the tree have the same relative failure probability and the same weight $w = 1$. Tree T^w has $n = \frac{N+1}{2} = \Theta(N)$ links and $\ell = \frac{N-1}{2} = \Theta(N)$ leaves.

Lines 1 to 4 take time $O(n) = O(N)$ to complete. Then, $\ell n = \frac{N^2-1}{4} = O(N^2)$ metrics R_d must be computed to determine A and B. Computing each metric R_d takes time $O(N)$. Indeed, in this case the protected path between a leaf $Y \in L_T$ and any node $Z \in V_T$ is reduced to $PP_{A,Z} = (Y, S, Z)$. Term (1) of Equation 3.2 is the sum of the metrics *adrop* for all links of the tree except the two links of the protected path, hence a complexity $O(N - 2) = O(N)$ for Term (1). Since $S = S'$, Terms (2) and (3) of Equation 3.2 are equal to zero. In this case, using the alternate method to compute each metric R_d is as complex as using the definition of R_d. Complexity of lines 5 to 8 is $O(N^2N) = O(N^3)$. Determining the shortest path in this graph is performed in time $O(N \log(N))$.

In summary, in this case the complexity of Algorithm 1 is $O(N^3)$. We have shown that N^3 was an upper bound for the complexity of Algorithm 1, and presented a case where this bound was attained.

We now show that the upper bound $O(N^3 \log(N))$ can be attained for Algorithm 2. The general expression of the time complexity is $O(n^3 + n^2(N - n)\log(N - n))$. Consider the network and the tree depicted in Figure 3.9. The tree is disconnected from the other nodes of the network and therefore Algorithm 2 computes a backup

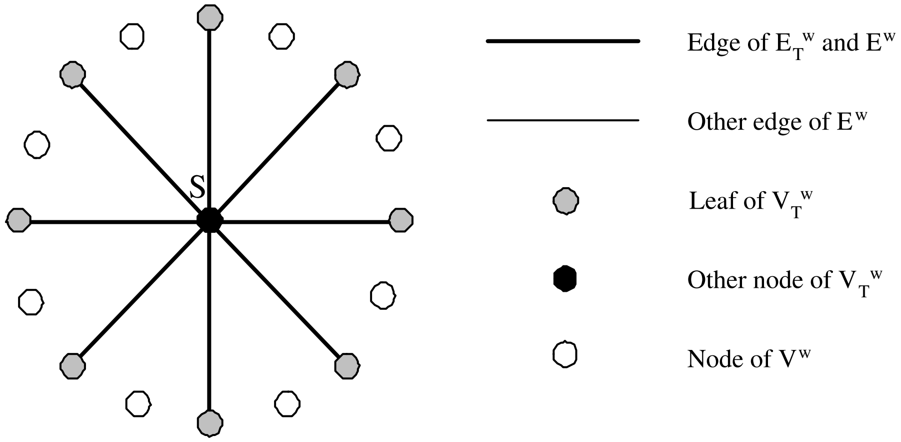


Figure 3.9: **Worst case for Algorithm 2.** With this topology, the complexity of Algorithm 2 reaches the higher bound $O(N^3 \log(N))$.

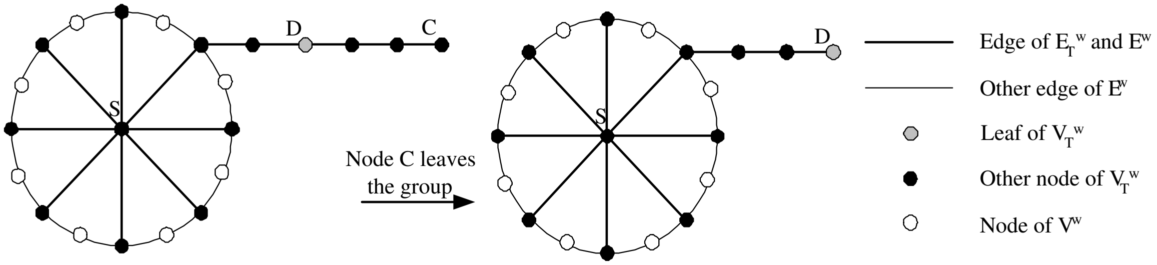


Figure 3.10: **Worst case for Algorithm 3.** With this topology, the complexity of Algorithm 3 reaches the higher bound $O(N^3)$.

path between all possible pairs of nodes of the tree before failing to find a backup path. Since $n = \Theta(N)$, the time complexity of Algorithm 2 in this particular case is $O(N^3 \log(N))$.

Last, we show that the upper bound $O(N^3)$ can be attained for Algorithm 3. Consider Figure 3.10 where node C leaves the group. The path between S and C is such that $|E_{P_{S,C}}| = \Theta(n)$ and D is located in the middle of this path. After C leaves the group we have $\ell = \Theta(N)$ and $n = \Theta(N)$. Then, $\ell n = \Theta(N^2)$ metrics R_d must be recomputed. The path between D and S has $\Theta(N)$ links thus for any two nodes i and j such that $i, j \notin E_{P_{D,S}}$ the size of the set $E_{P_{D,S}} \setminus E_{PP_{i,j}}$ is $\Theta(N)$. There are $\Theta(N^2)$ such pairs of nodes. According to Equation 3.1, updating one metric R_d takes time $O(n) = O(N)$. Therefore, updating all metrics R_d takes time $O(N^3)$. In this case, the time complexity of Algorithm 3 is $O(N^3)$.

4

MPLS Multicast Fast Reroute

In this chapter, we describe MPLS multicast Fast Reroute, a rerouting mechanism adapted to protect multicast routing trees from single link failures. MPLS multicast Fast Reroute extends the unicast MPLS Fast Reroute mechanism presented in Chapter 2. MPLS multicast Fast Reroute makes it possible to repair a multicast routing tree if a single link of the so-called protected path fails by rerouting traffic on a pre-planned backup path. This rerouting mechanism uses the same components as MPLS unicast Fast Reroute. A backup path must first be established in a multicast routing tree. Then, when a node detects a link failure or recovery, it sends a notification message to the *Path Switching LERs* (PSLs, see Section 2.4) that performs either switchover of switchback. MPLS multicast Fast Reroute assumes that multicast routing trees are core based trees.

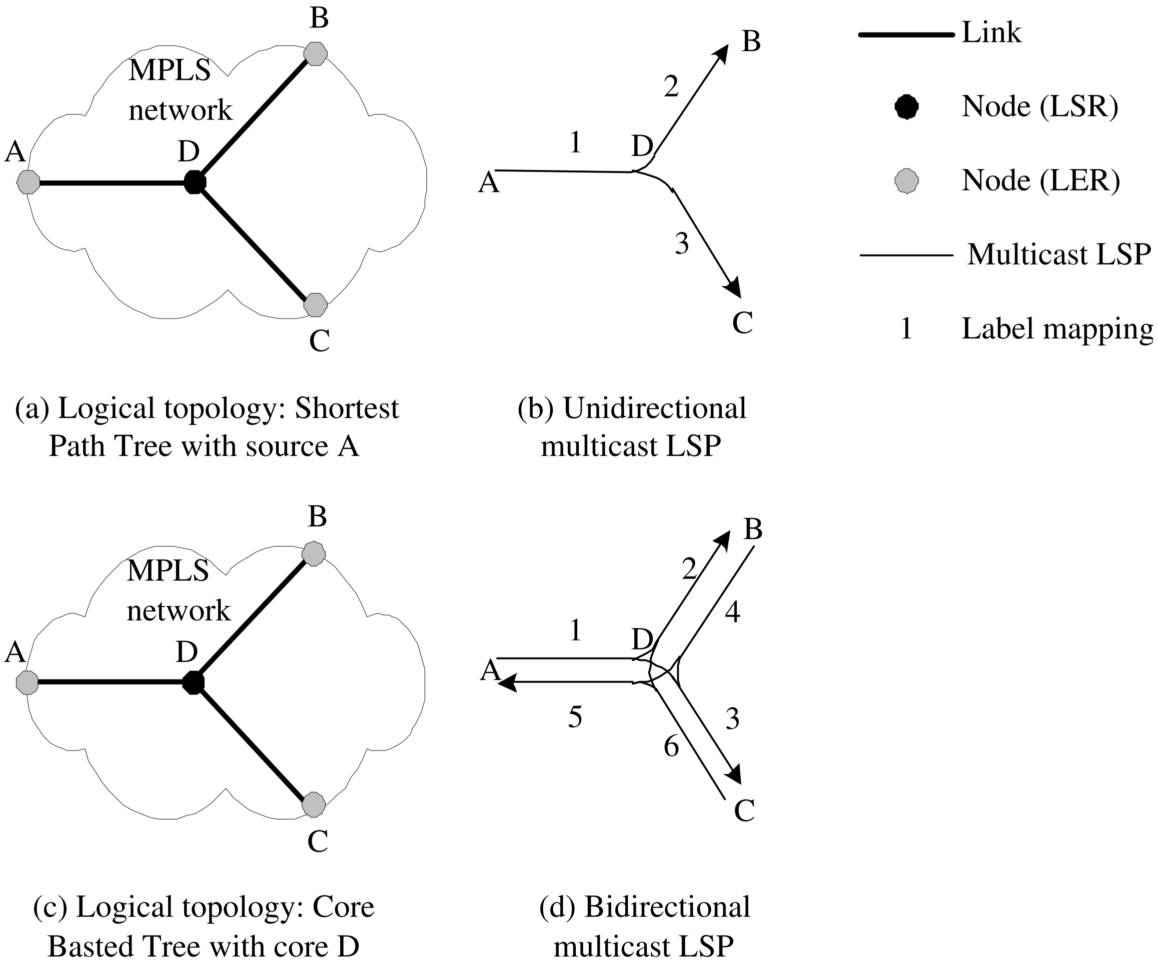
4.1 Overview

Before we give an overview of the multicast extensions to MPLS Fast Reroute, we introduce *multicast LSPs* (mLSPs), which are the multicast counterpart to unicast

Label Switching Paths. Different from other virtual circuit switching techniques like ATM, MPLS is able to create virtual circuits that map multicast routing trees without the need of establishing a distinct virtual circuit between a particular node and all multicast hosts of a multicast group.

A *multicast LSP* (mLSP) is a point-to-multipoint MPLS virtual circuit. Packets are forwarded on mLSPs the same way as they are forwarded on unicast LSPs, except that they can be duplicated by MPLS routers and forwarded on several links. Like unicast LSPs, mLSPs are virtual circuits. Therefore, a mLSP must be established before a multicast communication can actually take place and terminated when the communication is over. Such tasks are performed by a signaling protocol. In MPLS networks, LSPs are associated with FECs. Packets that enter an MPLS network are assigned to a FEC and all packets from the same FEC entering in the network via the same ingress LER are forwarded on the same LSP. The FEC associated to a multicast LSP is the IP address of the multicast group whose traffic is carried by the mLSP.

In Section 1.3, we introduced the notion of shortest path tree and core based tree. Consider Figure 4.1(a) which depicts a shortest path tree spanning over an MPLS network. The source of the tree is A. Node A receives traffic from other networks (not represented) and sends this traffic over the multicast routing tree. Leaves of the tree, B and C, receive the multicast traffic sent by A over the tree. A shortest path tree is mapped by a *unidirectional LSP* where label mappings are established



(a) Logical topology: Shortest Path Tree with source A

(b) Unidirectional multicast LSP

(c) Logical topology: Core Basted Tree with core D

(d) Bidirectional multicast LSP

Figure 4.1: **Multicast Label Switching Path examples.**

on each link of the tree in a single direction. For instance, the mLSP represented in Figure 4.1(b) maps the shortest path tree. Suppose that node A receives packets that must be forwarded over the tree. Node A pushes label “1” on the packets and forwards these packets to D. Node A is an ingress LER of the mLSP. Then, node D

duplicates the packets and swaps the label of one of the two copies first with label "2", then with label "3". Node D forwards the packets labeled "2" to B and the packets labeled "3" to C. Node D is therefore a LSR of the mLSP. Node B pops labels from packets labeled "2" coming from D and node C pops labels from packets labeled "3" coming from D. Nodes B and C are egress LERs of the mLSP.

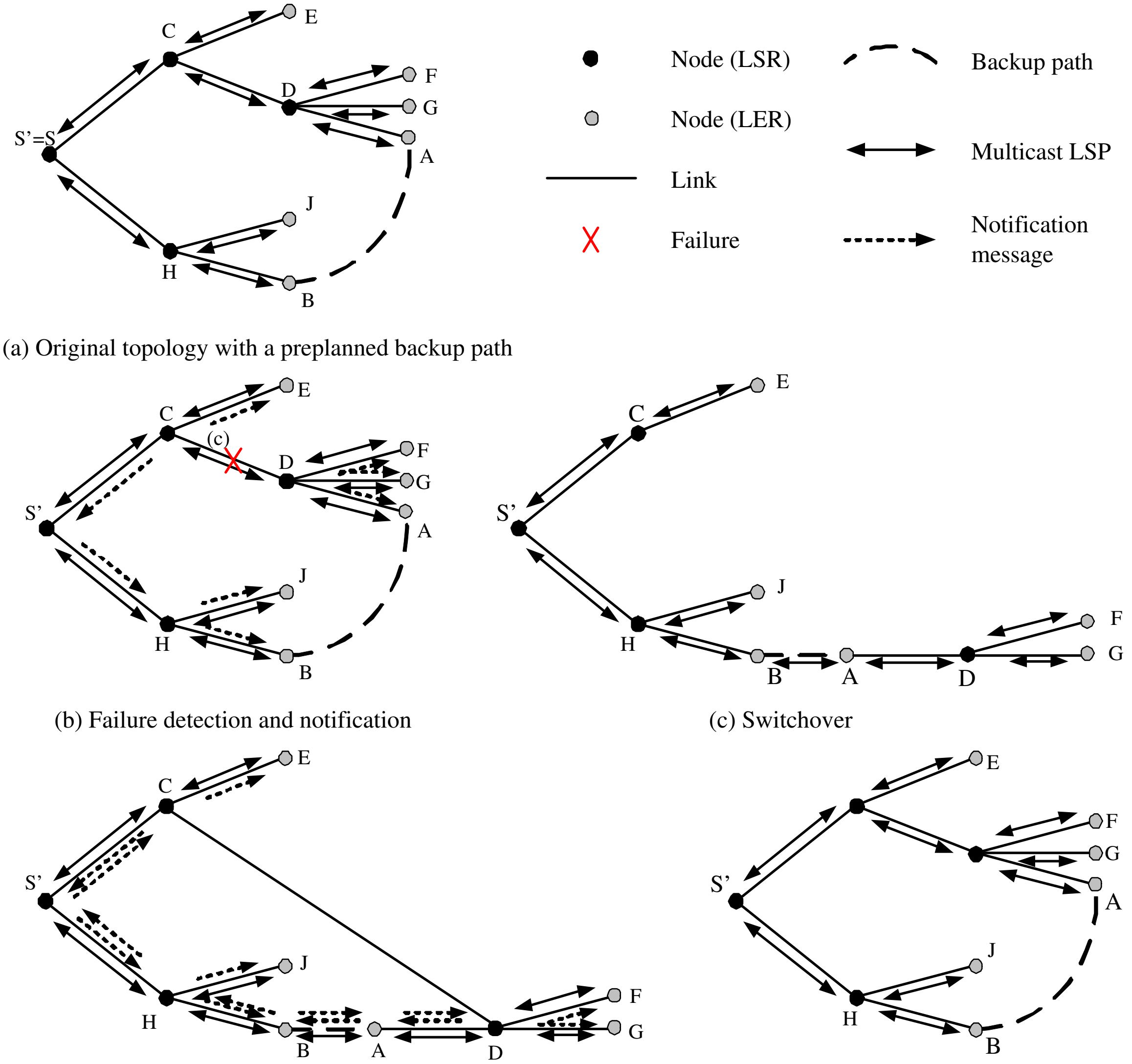
On the other hand, core based trees allow any multiple sources to send data to the members of a multicast group. It is possible to map a core based tree of n sources with n unidirectional mLSPs. Each of the unidirectional mLSPs would map a tree rooted at one of the n different sources. However, the main advantage of core based trees is that a single tree is shared by all sources. We introduce the notion of *bidirectional mLSP* where label mappings are established on each link of the tree in both directions, assuming that all links of the tree are bidirectional. For instance, consider the core based tree represented in Figure 4.1(c), where node D is the core and nodes A, B, C are the sources . This core based tree is mapped by the single bidirectional mLSP represented in Figure 4.1(d). Node A pushes label "1" on packets bound to the multicast group. Node D swaps label "1" against labels "2" and "3" and forwards packets labeled "2" to B and packets labeled "3" to C. Node B pops label "2" and node C pops label "3". If node B is a source of the multicast group, then when node B needs to send packets to the multicast group it pushes label "4" on the transmitted packets. Node D swaps label "4" against labels "5" and "3", and node

A pops label “5” while C pops “3”. If node C is a source of the multicast group, node C pushes label “6” on packets bound to the group, node D swaps label “6” against labels “5” and “2”, node A pops label “6” and node B pops label “2”. Since A, B and C both push and pop labels for packets of the bidirectional mLSP, they are ingress and egress LERs at the same time. Node D is a LSR of the bidirectional mLSP. In the remainder of this thesis, we only consider core based trees mapped by bidirectional mLSPs.

We now give an overview of MPLS Multicast Fast Reroute, the multicast extension to the MPLS Fast Reroute mechanism described in Section 2.4. We discuss an example that illustrates each step of MPLS Multicast Fast Reroute. The example is illustrated in Figure 4.2.

MPLS multicast Fast Reroute is a pre-planned rerouting mechanism. This means that a backup path is computed and advertised before rerouting is performed. A backup path is computed with Algorithm 1 presented in Chapter 3. However, the choice of the algorithm and the rerouting mechanism are orthogonal. In the following, we do not make any assumption on how the backup path is computed. We continue to use here the notations introduced in Chapter 3.

Consider the tree rooted at S mapped by the bidirectional mLSP depicted in Figure 4.2(a). A backup path $BP_{A,B}$ has been computed between nodes A and B and the protected path $PP_{A,B}$ is constituted by links AD, DC, CS', $S'H$ and HB.



(a) Original topology with a preplanned backup path

(b) Failure detection and notification

(c) Switchover

(d) Recovery detection and notification

(e) Switchback

Figure 4.2: **Overview of MPLS multicast Fast Reroute.**

Node S' is defined as the Least Common Ancestor of A and B with regards to S. In this example, since S belongs to the protected path between A and B, then $S' = S$.

MPLS multicast Fast Reroute can repair the multicast routing tree if any of these five links fails. Failure is detected by the end nodes of the failed link. All nodes of a multicast routing tree regularly send probe messages on each of their outgoing links. Each node also listens to such probe messages. If a node stops receiving probe messages on a link, then this node considers that the link has failed.

If a link fails, such as link CD in Figure 4.2(b), then both nodes C and D detect the failure of link CD. Then, both nodes C and D send failure notification messages which are propagated through the tree (Figure 4.2(b)). All nodes of the tree are notified of the link failure. When the end nodes A and B of the backup path receive failure notification messages, they perform switchover by merging the backup path in the mLSP (Figure 4.2(c)). A new mLSP results from the merging of the backup path and the mLSP that mapped the multicast routing tree before the failure. This new mLSP maps the multicast routing tree described in Claim 1 of Chapter 3. Nodes now forward packets over the new mLSP and no LER is dropped from the tree. We introduced the notion of *Path Switching LSR* (PSL) in Section 2.4 in the context of MPLS unicast Fast Reroute. In MPLS multicast Fast Reroute, since mLSPs are bidirectional, both nodes at the end of the backup path are PSLs. Indeed, node A forwards traffic coming from the LERs A, F and G over the backup path, therefore A is a PSL, and node B forwards traffic coming from the LERs B, E and J over the backup path, therefore B is also a PSL.

When nodes C and D resume receiving probes over the previously failed link CD, they detect that the failed link has been repaired. Then C and D send link recovery notification messages which are propagated through the multicast routing tree (Figure 4.2(d)). When nodes A and B receive those messages, they perform switchback by stopping forwarding packets over the backup path (Figure 4.2(e)). After switchback is completed, the multicast routing tree is exactly the same as the original multicast routing tree that was in use before the failure occurred (Figure 4.2(a)).

In the next sections, we describe the steps of MPLS multicast Fast Reroute: link failure and recovery detection, link failure and recovery notification, and switchover and switchback.

4.2 Link failure and recovery detection

To detect link failure or recovery, nodes regularly send small messages called *probes* on all the links on which they send traffic, and listen for probes on the links from which they receive traffic. In the context of bidirectional mLSPs, nodes actually send and listen for probes on each link they are attached to. Probes are small messages that take a low percentage of the bandwidth of the link on which they are sent. In our implementation (see Chapter 5), we send small UDP messages with a payload of 4 bytes.

A link failure must be detected as early as possible in order to keep the total repair time low. To do so, probes should be sent at a high frequency. However, since detecting a link failure triggers traffic switchover, link failures should not be detected when a link has not actually failed. We call T_p the period used by nodes to send probes. A probe can be detected as missing while a link has not failed in two cases. First, due to delay jitter, a delayed probe may be considered as missing. Second, since probes are sent as UDP messages, a lost probe is not retransmitted. Therefore, a single lost probe should not be interpreted as a consequence of a link failure.

We consider that a link has failed only when several probes are missing in sequence. Let the *beat checking number* $n \geq 2$ be the number of probes that must be missing before a node reports a link failure. The failure detection time $T_{fdetect}$ is the time between the instant at which a link fails and the instant at which a node that receives probes via this link considers that the link has failed. We now determine the distribution of the failure detection time. Suppose node C is sending probes to node D as shown in Figure 4.3. At time $t = 0$, C sends the last probe before link CD fails. Link CD fails at time T_1. Time T_1 is uniformly distributed between 0 and T_p. Every period nT_p, node D checks whether it received at least one probe from C. Since the sender of probes at node C and the receiver of probes at node D are not synchronized, the time T_2 at which D checks and records the presence of the last probe sent by C is uniformly distributed between 0 and nT_p. Node D detects the

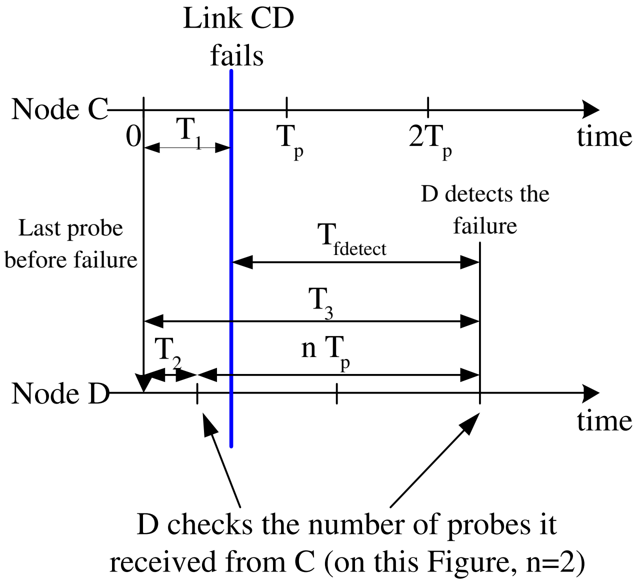


Figure 4.3: **Link failure detection mechanism.** Node C starts sending probes with period T_p at time 0, node D starts checking the number of probes it received from C every period nT_p at time T_2. Link CD fails at time T_1 and D detects the failure at time T_3. The failure detection time is $T_{fdetect} = T_3 - T_1$.

failure at time $T_2 + nT_p$ since no probe is received between $t = T_2$ and $t = T_2 + nT_p$. Therefore the time T_3 at which the failure is detected is uniformly distributed between nT_p and $2nT_p$. The failure detection time $T_{fdetect}$ is given by $T_3 - T_1$. The distribution of $T_{fdetect}$ is represented in Figure 4.4.

There is a trade-off between the speed of the failure detection and the accuracy of the detection, i.e., the ability of the nodes to detect failures only when a link has failed. Low values of n may lead to a high number of false failure detection. Since

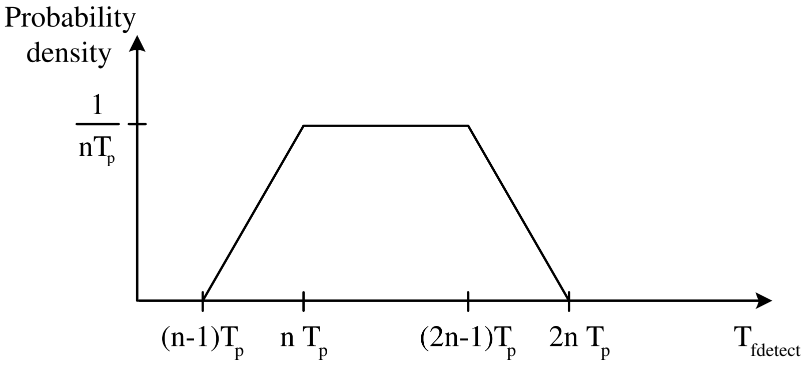


Figure 4.4: **Link failure detection time probability density function.** The probability density function of the difference $T_{fdetect} = T_3 - T_1$ is obtained with a convolution of T_1 and T_3.

the link failure detection time is comprised between $(n-1)T_p$ and $2nT_p$, high values of n yield long times before link failures are reported.

The same probing mechanism can be used to detect the repair of a link. When a link is reported as failed, its two end nodes keep trying sending probes with period T_p. When one of these two nodes receives such a probe, then the link is detected as repaired. Different from link failure detection where a single missing probe is not enough for an end node to infer that a failure has occurred, the arrival of the first probe on a link previously reported as failed indicates the recovery. For instance, suppose that node C sends a probe on link CD time T_1 after link CD has been repaired as shown in Figure 4.5. Since node C tries to send probes every period T_p, T_1 is uniformly distributed between 0 and T_p. Node D detects the repair as soon as it

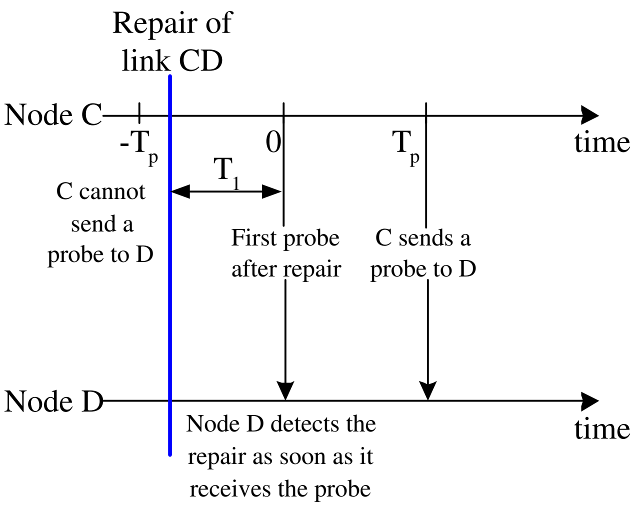


Figure 4.5: **Link repair detection mechanism.** Link CD is repaired at time $t = T_1$ and node C sends a probe at time $t = 0$. Node D detects the link repair as soon as it receives the probe, thus the repair detection time is $T_{rdetect} = T_1$.

receives a probe, thus the recovery detection time $T_{rdetect}$ to detect the repair is equal to T_1 and is uniformly distributed between 0 and T_p (Figure 4.6).

The average value of $T_{fdetect}$ is $\overline{T}_{fdetect} = \frac{3n-1}{2}T_p$ and the average value of $T_{rdetect}$ is $\overline{T}_{rdetect} = \frac{T_p}{2}$. Using millisecond timers on the nodes that perform failure or repair detection, it is possible to detect the failure or the repair of a link in a few milliseconds or tens of milliseconds depending on n.

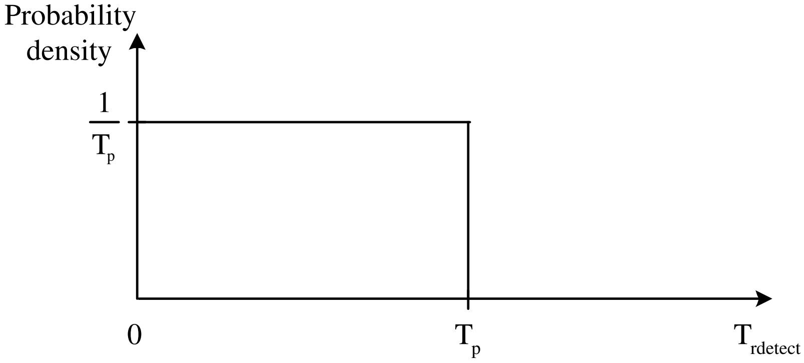


Figure 4.6: **Link repair detection time probability density function.** $T_{rdetect}$ is uniformly distributed between 0 (a probe is sent immediately after the link is repaired) and T_p (a probe is sent time T_p after the link is repaired).

4.3 Failure and recovery notification

When a link failure or recovery is detected, the PSLs must be notified of the failure or recovery so that they can perform switchover or switchback. The nodes that detect the failure or recovery are responsible for notifying the PSLs. We successively consider link failure and recovery in a multicast routing tree.

As explained in Chapter 3, when a link fails the multicast routing tree is split in two trees. For instance, if nodes C and D detect the failure of link CD as illustrated in Figure 4.2(b), the original multicast routing tree is split into one tree rooted at C and one tree rooted at node D. These two trees are represented in Figure 4.7. Each of the two nodes that detect the failure sends out a signaling protocol *link failure notification message* to each of their children. This notification message contains in its payload the IP address of the interface that ends the failed link. Nodes that receive a link failure notification send in turn link failure notification to their children until all leaves of the two trees previously defined are reached. When a node sends a link failure notification message, it does not change the IP address contained in the payload of the message such that all nodes which receive the message know which link has failed. For instance, in Figure 4.7, D sends a link failure notification to F, G and A. The notification message sent by D contains the IP address of the interface of node D that ends link CD. Node C sends a link failure notification to nodes E and S'. The notification message sent by C contains the IP address of the interface of node C that ends link CD. Upon reception of the notification, S' sends a link failure notification to H which in turn sends link failure notifications to J and B.

All nodes of the original multicast routing tree are notified of the failure. In particular, if the failed link is on the protected path protected by a backup path, then the PSLs for the backup path are both notified of the link failure. Each PSL contains a list of the IP addresses of the interfaces that end links of the protected path. When a PSL receives a link failure notification message, it checks that the IP address contained in the payload of the notification message matches an IP address of its internal list. If such is the case, then the PSL must perform switchover. Otherwise, the failed link is not on the protected path protected by the PSL and the PSL does not perform switchover.

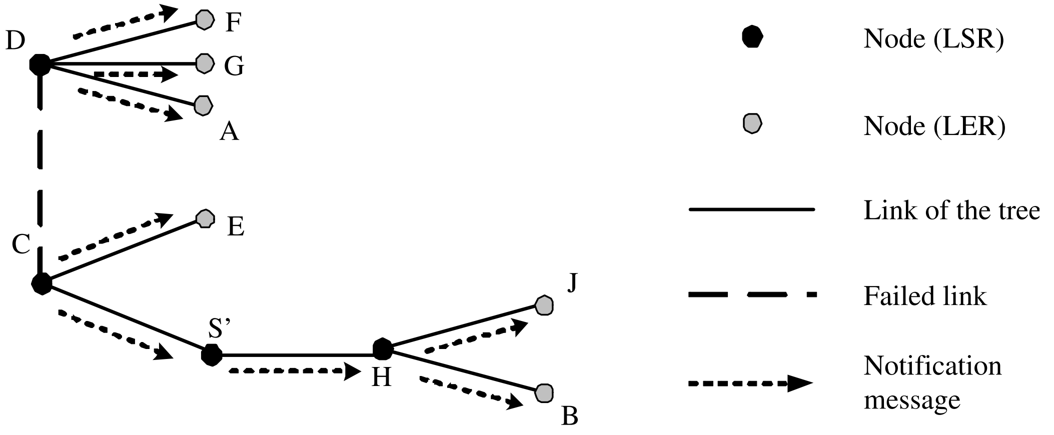


Figure 4.7: **Failure notification mechanism.** After failure of link CD, the original multicast routing tree is split into two trees, one rooted at C and one rooted at D. Nodes C (resp. D) sends a link failure notification message on the tree for which it is the root. The failure notification messages are propagated on both trees until all leaves are reached.

When a link repair is detected, we use the exact same mechanism to propagate the repair information. Only the type of message used changes, i.e. signaling protocol *link recovery notification messages* are used. Messages contain the IP address of the interface of the repaired link. In our example, when link CD is repaired, then node D sends a link recovery notification message to F, G and A which contain the IP address of the interface of D that ends the recovered link. Node C sends a link recovery notification message to S' which contain the IP address of the interface of C that ends the recovered link. Node S' sends a link recovery notification message to

H, which sends link recovery notification messages to J and B. If a backup path has been established and if the link that has been repaired is part of the protected path, then the end nodes of the backup path perform switchback.

The failure notification time is the time between the instant at which a failure is detected and the instant at which both PSLs are notified of the failure. Likewise, the recovery notification time is the time between the instant at which a link repair is detected and the instant at which both PSLs are notified of the recovery. Since the notification mechanism is the same for link failure and link recovery, failure notification time and recovery notification time are the same. We call notification time and we note T_{notif} the common value for failure notification time and recovery notification time. If we denote by T_{nnotif} the time taken by a node to send and process a notification message (*node notification delay*), and if the protected path of the multicast routing tree consists of l links, then the recovery time is bounded by:

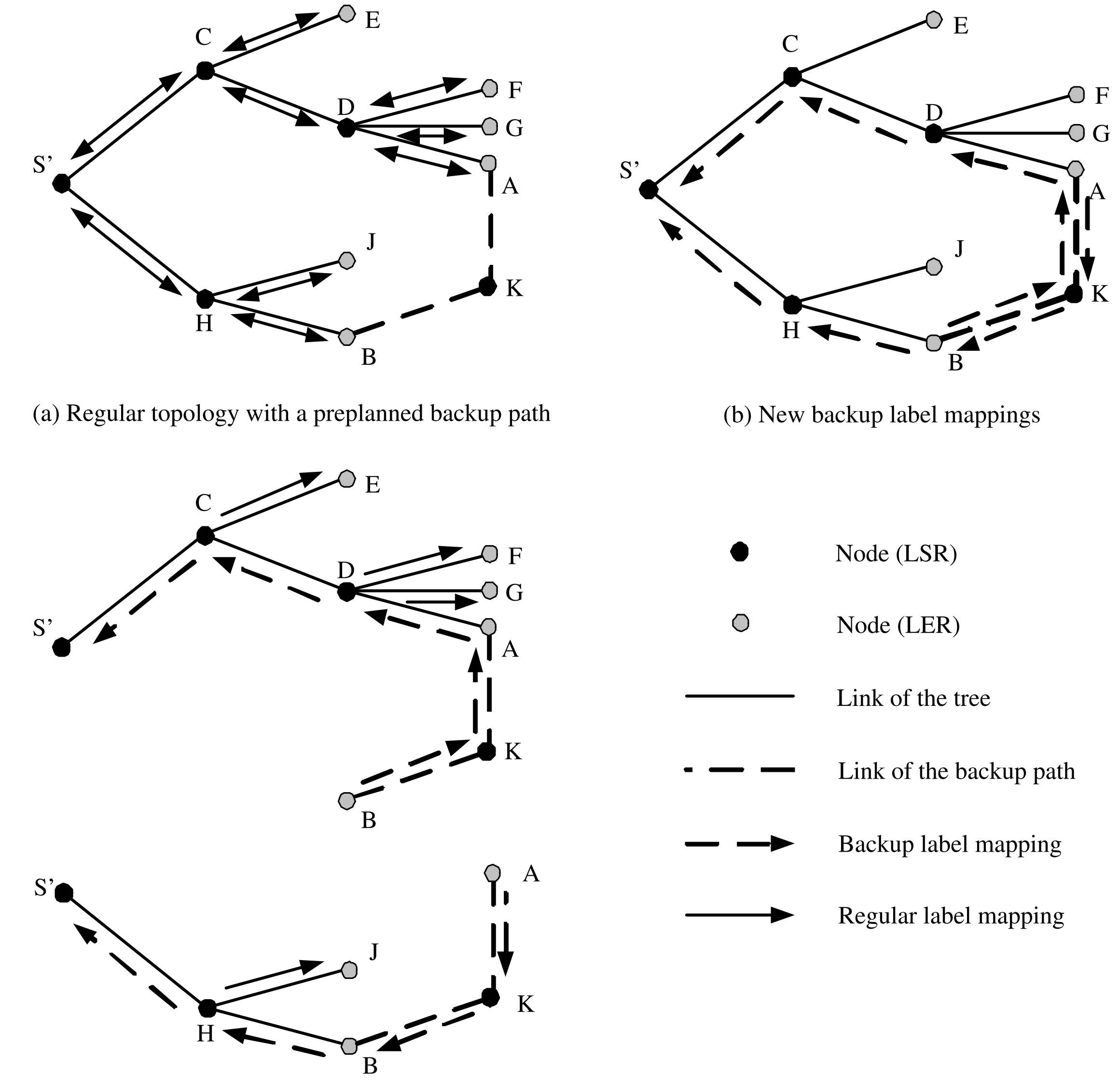
$$T_{notif} \leq lT_{nnotif}.$$

Using 100 Mbits/s Ethernet hardware, the node notification delay is in the order of 1 millisecond. Therefore, it takes a few milliseconds to notify the PSLs after a link failure has been detected.

4.4 Switchover and switchback

Switchover consists in merging the backup path with the mLSP that maps the original multicast routing tree (before link failure). After switchover is performed, traffic flows on a new mLSP. On the other hand, switchback restores the original mLSP by ceasing to send traffic over the backup path.

Let us consider the backup label mappings that need to be advertised before a multicast routing tree can be repaired. Consider a multicast routing tree and the mLSP that has been established for the multicast routing tree. In the following, we will refer to the label mappings that define this mLSP as "original label mappings". We establish two series of new mappings for the backup path as follows. First, label mappings are established on the path between A and S' via B. Second, label mappings are established on the path between B and S' via A. Therefore, backup mappings are established on all links of the backup path in both directions, and on all links of the protected path in a single direction. Consider the example network in Figure 4.8(a). We show these additional backup mappings for this network in Figure 4.8(b). With both mappings from the original mLSP and these additional mappings, two new unidirectional mLSPs are defined (Figure 4.8(c)). For instance, suppose B sends a packet on the backup path using the new backup label mapping. Node K forwards the packet to A which forwards the packet to D, still using backup label mappings. Node D duplicates the packet and sends a copy to C using a backup label mapping,



(a) Regular topology with a preplanned backup path

(b) New backup label mappings

(c) Trees over which the PSLs send traffic after switchover

Figure 4.8: **Backup label mappings.**

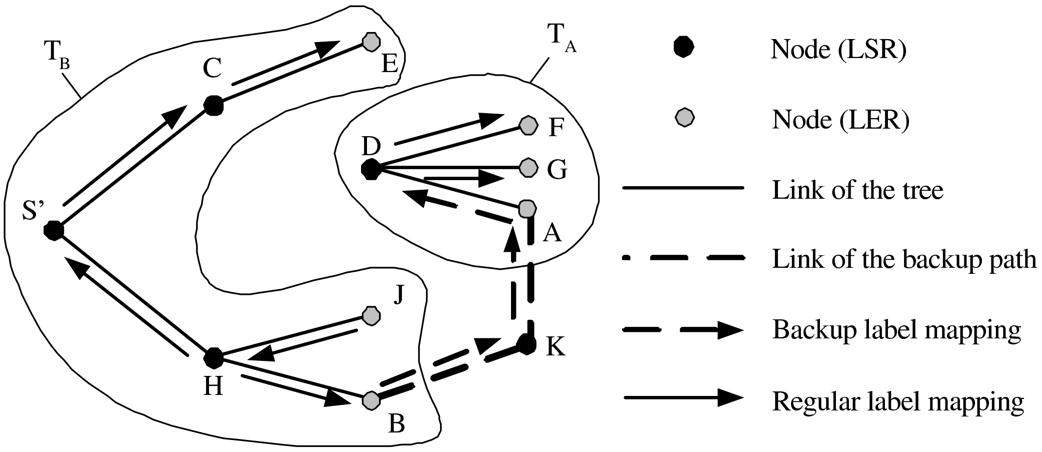


Figure 4.9: **Path followed by packets sent by J after switchover.**

one copy to F and one copy to G using original label mappings. When C receives the packet, it forwards the packet to both S' using a backup label mapping and to E using an original label mapping. Node S' does not forward packets that use backup label mappings and come from a link of the protected path to another link of the protected path. We will explain why when we focus on switchback.

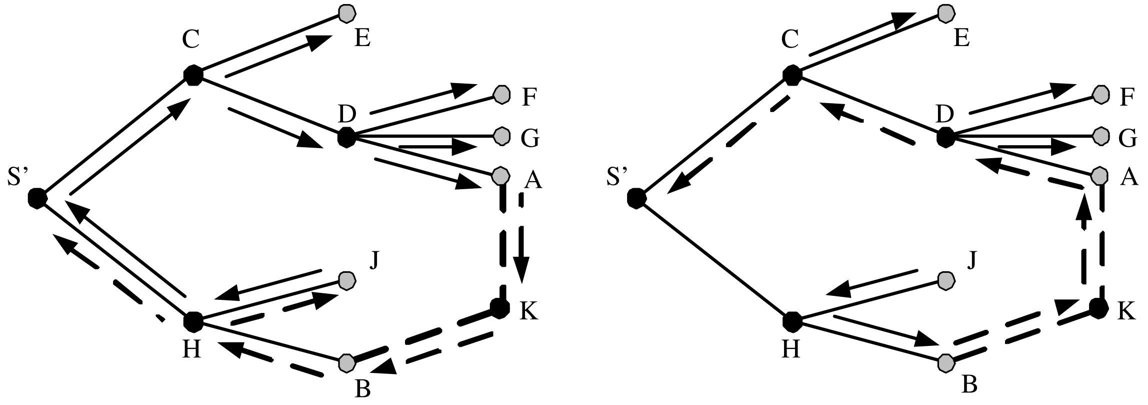
Switchover is performed as follows: when a PSL is notified of a link failure, it forwards all the packets it receives on the backup path using the backup mapping. Suppose for instance that link CD fails. The PSLs A and B perform switchover and the backup path is merged with the original multicast routing tree, yielding a new, *repaired* multicast routing tree. Suppose J sends a packet on the new multicast routing tree. The packet reaches nodes H, S', C, E and B using original label mappings. When B receives the packet, it forwards the packet on the backup path as described above, except that D does not forward the packet to C. We show the path followed by packets sent by J after switchover in Figure 4.9.

If a second link of the protected path fails, then the PSLs are notified of the second link failure and ignore it. The repaired multicast routing tree is split into two trees and the multicast group is partitioned. We do not consider the case where only one direction of a link fails, since links are assumed to be bidirectional and a failure generally affects both directions.

Both PSLs do not perform switchover simultaneously. When a link fails, a multicast routing tree is split into two smaller subtrees T_A and T_B. Suppose T_A is the subtree that contains PSL A and T_B is the subtree which contains PSL T_B (see Figure 4.9). After the link failure and before A and B are notified of the failure, traffic sent by nodes from T_A cannot reach nodes of T_B and traffic sent by nodes from T_B cannot reach nodes of T_A. Suppose A is notified of the failure and performs switchover before B. After A has performed switchover and before B has performed switchover, traffic sent by nodes of T_A can reach nodes of T_B but conversely traffic sent by nodes of T_B cannot reach nodes of T_A. After both PSLs have performed switchover, no node is dropped from the multicast routing tree. Switchover consists in a change in the MPLS forwarding table of the LSRs, thus switchover is almost instantaneous. The total time to repair the tree is therefore $T_{repair} = T_{fdetect} + T_{notif} \approx T_{fdetect}$. The order of magnitude of T_{repair} is a few tens of milliseconds.

We now discuss the switchback mechanism. When a node detects a link failure, it stops forwarding traffic over the failed link. When this node detects the link repair, it sends out notification messages as explained in Section 4.3 and resumes forwarding traffic over the repaired link. When a PSL is notified that a failed link is repaired, it stops forwarding traffic over the backup path. Like switchover, switchback is not performed simultaneously by both PSLs. After the link repair detection and before A and B are notified of the link recovery, when a node sends a packet then all other nodes receive two copies of this packet. Consider for example Figure 4.10 and suppose that nodes C and D have detected the repair of link CD and that neither A nor B have performed switchback. When J sends a packet to the multicast group, node H duplicates the packet and sends one copy to S' (Figure 4.10(a)) and the other copy to B (Figure 4.10(b)). We now follow the path of the first copy. Node S' forwards it to C which forwards the packet to both node E and, since the repair of link CD has been detected, node D. Node D forwards the packet to F, G and A. Node A has not performed switchback thus it forwards the packet to K. The packet then reaches B. Node B forwards the packet to H, which forwards the packet to J and S'. Node S' does not forward the packet coming from a link of the protected path (link $S'H$) to another link of the protected path (link $S'C$), thus the packet is not forwarded to C. If the packet was forwarded to C, then it would loop on the path formed by the protected and the backup paths. Now consider the second copy of the packet made by H after H receives the packet from J. Node H forwards the packet to B which forwards the packet over the backup path to A via K. Node A sends the packet to D which forwards the packet to F and G. Since the repair of link CD has been detected, D also sends the packet to C, C forwards the packet to E and S'. Node S' does not forward the packet to H and breaks the loop. Therefore, during this transient period, all nodes receive duplicate copies of all packets. When only one PSL, for instance A, has performed switchback then traffic from nodes of T_A is delivered twice to the nodes of T_A and only once to the nodes of T_B, and traffic from nodes of T_B is delivered twice to the nodes of T_A and only once to the nodes of T_B. When the second PSL performs switchback then no node is forwarded on the backup path anymore and traffic is forwarded on the original mLSP as before the link failure.

We define the time $T_{repairback}$ to switch traffic back on the original tree as the time between the instant at which the failed link is repaired and the instant at which both PSLs have performed switchback. Therefore $T_{repairback} = T_{rdetect} + T_{notif}$. During the time $T_{repairback}$, certain links must carry twice the same data, possibly leading to congestion. If the traffic that uses the mLSP before switchback represents less than 50% of the capacity of the links that is allocated to the mLSP then no congestion will occur and nodes will simply receive the same packets twice. It is up to the application layer to drop the redundant packets. On the other hand if traffic that flows on the mLSP before switchback represents more than 50% of the capacity of the links that is allocated to the mLSP then the mLSP will be congested and packets will be dropped.



(a) Path followed by one of the copies of a packet sent by J, forwarded by H to S'

(b) Path followed by the other copy of a packet sent by J, forwarded by H to B

Figure 4.10: **Duplicate packets during switchback.** When a failed link is repaired and switchback has not been performed, all nodes receive two copies of the packets sent by any node. In this example, all nodes receive two copies of a packet sent by J.

However, since $T_{rdetect} < T_{fdetect}$ the time during which congestion may occur is smaller than the interruption of service due to link failure. Moreover congestion that may be caused by switchback is very limited in time (a few milliseconds) and therefore does not prevent the network from functioning properly.

In this chapter, we have exposed the principles of MPLS multicast Fast Reroute. Implementing this mechanism requires the ability to advertise a mLSP and the backup path mappings. Second, the probing and notification mechanisms must be implemented. Finally, nodes must perform switchover or switchback when they are notified to. In the next chapter, we present an implementation of all these mechanisms.

5

Implementation

In this chapter, we present the implementation of multicast MPLS for Linux and multicast MPLS Fast Reroute. First, in Section 5.1, we discuss an implementation of MPLS multicast for the Linux operating system. We have added multicast support to a previously existing implementation of MPLS for Linux, which formerly supported unicast only. To our knowledge, our implementation is the only MPLS multicast implementation available in the public domain. In Section 5.2, we present a new signaling protocol, *MULticast TREe rEpair Label Distribution Protocol* (*MulTreeLDP*). MulTreeLDP runs on top of our implementation of MPLS multicast to support multicast Label Switched Path (mLSP) establishment and implements the MPLS multicast Fast Reroute mechanism from Chapter 4.

5.1 Multicast MPLS-Linux

MPLS-Linux is a recent implementation of MPLS for PCs running the Linux operating system [51]. MPLS-Linux is freely modifiable under the GNU license [31] and conforms to the MPLS specifications [61] [62]. Other MPLS implementations for PCs have been proposed in the past [54] [55], but are not maintained by their authors. Thus, we chose to implement our multicast rerouting mechanism on PCs running MPLS-Linux. MPLS-Linux does not support multicast forwarding, therefore we augmented MPLS-Linux with multicast capabilities. Before we explain how we extended MPLS-Linux, we provide a background on the existing MPLS-Linux implementation for unicast.

5.1.1 Unicast MPLS-Linux implementation

MPLS-Linux is implemented as a layer between Ethernet and IP. Ethernet is a MAC layer protocol which encapsulates IP packets in *frames*. In Section 1.2.2, we gave an overview of the three operations that MPLS routers can perform on packets (*push*, *swap* and *pop*) and we described the *Forwarding Information Base* (FIB) which contains the rules according to which MPLS routers forward packets. We now describe how the MPLS operations and the FIB are implemented in MPLS-Linux.

MPLS-Linux defines five instructions to implement shim header pushing, swapping and popping. Each of these instructions can be applied to IP packets or Ethernet frames in the MPLS layer as they are being processed by the Linux kernel. We give an overview of these five instructions in Table 5.1 and we describe how they implement the three MPLS operations in Table 5.2. The `PUSH` instruction adds an MPLS shim header to a packet which comes from the IP layer. The `SET` instruction passes an IP

Instruction	Input layer	Output layer	Description
PUSH	IP	MPLS	Adds a shim header to an IP packet.
SET	MPLS	Ethernet	Passes an MPLS unicast packet to an Ethernet interface.
POP	Ethernet	MPLS	Removes a shim header from an Ethernet frame.
FWD	MPLS	MPLS	Calls PUSH for a packet coming from POP.
DLV	MPLS	IP	Passes an MPLS packet to the IP layer.

Table 5.1: **MPLS-Linux unicast instructions overview.** MPLS-Linux unicast implements the three MPLS operations (push, swap, pop) with five different instructions.

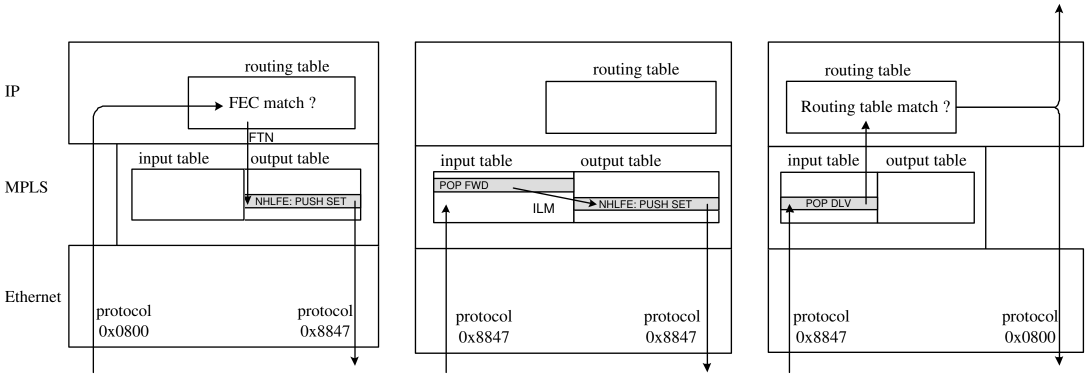
MPLS Operation	Corresponding sequence of instructions in MPLS-Linux
push	PUSH, SET
swap	POP, FWD, PUSH, SET
pop	POP, DLV

Table 5.2: **Implementation of the three MPLS operations with the five MPLS-Linux instructions.**

packet with a shim header from the MPLS layer to the Ethernet layer and tells the Ethernet layer on which Ethernet interface the MPLS packet should be forwarded. Together, the PUSH and SET instructions implement the MPLS “push” operation. The POP instruction removes the shim header of a packet that comes from the Ethernet layer. Packets processed by POP must be subsequently processed by either FWD or SET. The FWD instruction takes as an input a packet processed by POP and calls the PUSH instruction. Together, the POP, FWD, PUSH and SET instructions implement the MPLS “swap” operation. We will see in the remainder of this section why this FWD instruction is made necessary to swap labels. Last, the DLV instruction takes as an input a packet processed by POP and passes it to the IP layer. The POP and DLV instructions implement the MPLS “pop” operation.

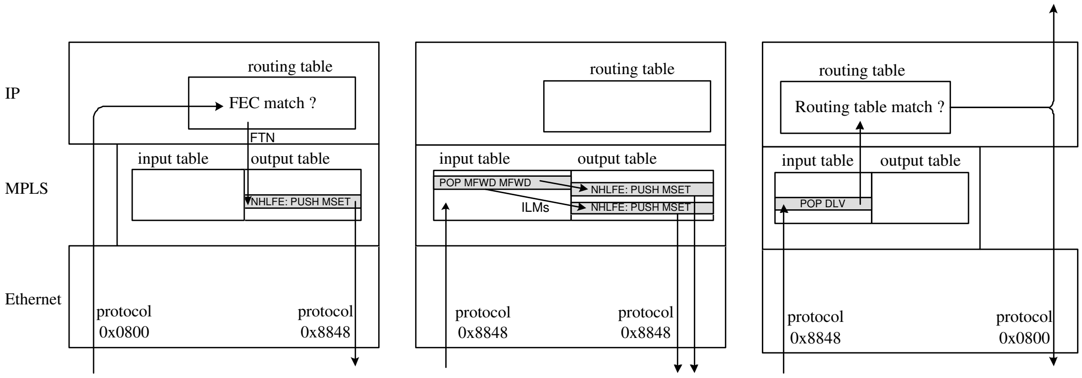
We now describe the implementation of the FIB in MPLS-Linux. In MPLS-Linux, the FIB is split into three tables: the *MPLS input and output tables*, and the IP routing table. MPLS-Linux defines a Forwarding Equivalence Class (FEC) with a *prefix* and a *prefix length*. A prefix is a 32-bit IP address and a prefix length is a number comprised between 1 and 32. A packet with destination IP IP_d matches the FEC P/P_{len} constituted by the prefix P and the prefix length P_{len} if and only if the first P_{len} bits of IP_d and P are the same. A requirement of MPLS-Linux is the presence in the IP routing table of a specific entry for each FEC that is defined at an MPLS ingress LER. It is not possible to define a FEC if no matching entry exists in the routing table. Indeed, MPLS-Linux relies on the IP routing table to determine the FEC of an IP packet. In MPLS-Linux, IP routing table entries are extended and contain FEC to Next Hop Label Forwarding Entry (FTN) mappings in addition to the IP routing information. Both the MPLS input and output table contain Next Hop Label Forwarding Entries (NHLFEs), while the MPLS input table implements the Incoming Label Map (ILM).

Consider Figure 5.1(a) which shows how a shim header is pushed on an incoming Ethernet frame by an ingress LER. The Ethernet layer of the LER receives a frame



a) At an ingress LER b) At a LSR c) At an egress LER

Figure 5.1: **Processing of a packet in the MPLS layer with MPLS-Linux unicast.**



a) At an ingress LER b) At a LSR c) At an egress LER

Figure 5.2: **Processing of a packet in the MPLS layer with MPLS-Linux multicast.**

with a protocol field in the Ethernet header set to 0x0800, which is the protocol code for IPv4. The Ethernet layer passes the incoming frame to the IP layer. The MPLS router searches for an entry in the IP routing table to make the routing decision, but since this entry matches a FEC it has been modified so that the packet is passed to the MPLS layer instead of being routed by the IP layer. The additional information contained in the IP routing table is a FTN, that is, a pointer to an MPLS output table entry. This output table entry is a NHLFE that contains two instructions. A `PUSH` instruction defines the label number of the packet, and a `SET` instruction defines on which interface the packet should be sent on. The MPLS layer adds at the beginning of the packet an MPLS header which contains the label found in the NHLFE, and passes the packet to the Ethernet layer. The Ethernet layer generates a frame with the protocol field set to the code assigned to MPLS unicast packets (0x8847) and sends the frame over the wire.

Consider now Figure 5.1(b) which shows how a label is swapped by a LSR. The Ethernet layer of the LSR receives a frame with a protocol field in the Ethernet header set to 0x8847. Since 0x8847 is the code assigned to MPLS unicast packets encapsulated in Ethernet frames, the Ethernet layer passes the frame to the MPLS layer of the LSR. The MPLS layer searches in the MPLS input table for the entry that matches the label embedded in the shim header of the packet. The input table implements the ILM and tells the MPLS layer what to do with the packet. The input table entry contains two instructions. The `POP` instruction tells the LSR to remove the MPLS header, and the `FWD` instruction points to an entry of the MPLS output table. This entry in turn contains two instructions: the `PUSH` instruction contains the new label for the packet and tells the LSR to add a shim header on the packet with this new label, while the SET instruction tells the LSR on which Ethernet interface the packet should be sent. The Ethernet layer then builds a frame with a protocol field of 0x8847 and sends it over the wire. By definition, the NHLFE tells an MPLS router whether a header must be popped or swapped. In MPLS-Linux the SWAP operation is implemented by successively popping and pushing a shim header, and the instructions required to pop an push a label are located in each of the MPLS tables. In this case, the NHLFE is contained at the same time in the input table and the output table.

Last, consider Figure 5.1(c) which shows how a label is popped by an egress LSR. The Ethernet layer of the LSR receives a frame with a protocol field in the Ethernet header set to 0x8847 and therefore passes the frame to the MPLS layer. The MPLS input table entry that matches the label of the packet contains two instructions. The POP instruction tells the LER to remove the shim header from the packet, and the DLV instruction tells the LER to pass the packet to the IP layer where it will be processed like any other IP packet. In this case, the NHLFE is fully contained in the input table entry and tells the packet to pop the shim header.

Labelspaces define the scope of forwarding rules. If two interfaces of the same MPLS router belong to the same labelspace, then they apply the same set of forwarding rules to MPLS packets. For example, if interfaces “2” and “4” are part of the same labelspace, then two packets with the same label arriving one on interface “2” and the other on interface “4” will follow the same forwarding rule. On the other

Instruction	Input layer	Output layer	Description
MSET	MPLS	Ethernet	Passes an MPLS multicast packet to an Ethernet interface.
MFWD	MPLS	MPLS	Calls PUSH for a multicast packet coming from POP.

Table 5.3: **MPLS-Linux multicast instructions overview.** MPLS-Linux multicast extensions require two additional instructions to forward multicast packets.

hand, if multiple interfaces do not belong to the same labelspace then the incoming MPLS packets follow different forwarding rules. In our implementation, we do not use labelspaces and for each Ethernet interface we set the labelspace to be equal to the interface index assigned by the kernel.

5.1.2 Multicast MPLS-Linux implementation

Unicast MPLS-Linux provides five instructions to implement MPLS headers header operations. In order to support multicasting, we added two new instructions MSET and MFWD inside the kernel implementation of MPLS-Linux. Table 5.3 gives an overview of these two new instructions.

A first difference between MPLS unicast and MPLS multicast lies in the protocol number in the Ethernet frames. The value of the protocol number is 0x8847 for MPLS unicast and 0x8848 for MPLS multicast. When a frame that contains an MPLS multicast packet is *transmitted* by the Ethernet layer, the protocol number should be set to the correct value in the Ethernet header. This is done with the new

MSET instruction which replaces the unicast SET instruction when an MPLS router forwards multicast packets. On the other hand, frames *received* by the Ethernet layer with a multicast MPLS protocol number should be processed by the MPLS layer rather than the IP layer. The Linux kernel API defines the dev_add_pack() instruction to associate Ethernet protocol numbers with upper layer handlers. For instance, the protocol number 0x0800 is associated with the IP layer handler so that the Ethernet layer passes to the IP layer the frames that contain IP packets. We wrote the handler that redirects MPLS multicast packets to the MPLS layer.

Second, different from MPLS unicast, in MPLS multicast the MPLS layer must be able to duplicate packets. MPLS multicast forwards the same incoming packet to several interfaces. We define the new MPLS operation *mswap* (multicast swap) on MPLS headers. When an MPLS packet shim header is *mswapped*, the packet is duplicated and the shim header of each copy of the packet is swapped against a new one. Then, each copy of the packet is sent on a different interface. We implement the *mswap* operation with the POP and PUSH instructions, and the use of the new MFWD and MSET instructions are described in Table 5.4.

We have has designed the new MFWD (Multicast FWD) instruction as a replacement for FWD. While an input table entry can contain only one FWD instruction in unicast MPLS-Linux, several MFWD instructions can be placed in a single MPLS input table entry in our MPLS-Linux extensions. The MFWD instruction supports packet du-

MPLS Operation	Corresponding sequence of instructions in MPLS-Linux
push	PUSH,MSET
mswap	POP , MFWD, PUSH, MSET MFWD, PUSH, MSET ⋮ MFWD, PUSH, MSET
pop	POP, DLV

Table 5.4: **Implementation of the multicast MPLS operations.** The new instructions MFWD and MSET replace FWD and SET.

plication. If an MPLS input table entry contains n MFWD instructions, then incoming packets are duplicated $n-1$ times using a software mechanism provided by the kernel API. Each MFWD instruction points to a different MPLS output table entry. Each of the n copies of the packet is processed according to the contents of the MPLS output table entries pointed by one of the MFWD instructions. Therefore, the MPLS layer can push a different shim header on each copy of the packet and forward it on a different interface.

Figures 5.2(a), 5.2(b) and 5.2(c) respectively show how shim headers are pushed, mswapped and popped for multicast MPLS packets. In Figure 5.2(a), the only difference between pushing a shim header on an MPLS unicast packet and pushing a shim header on an MPLS multicast packet lies in the protocol number in the Ethernet frame. The MPLS layer uses the MSET instruction instead of the SET instruction in the MPLS output table when pushing shim headers on MPLS multicast packets. In Figure 5.2(b), we illustrate how multicast packets are forwarded on several interfaces at the same time. The MPLS input table contains a POP instruction and two MFWD instructions for incoming packets. The MPLS layer first removes the MPLS shim header of each incoming packet. Then, it duplicates the packet in order to get two copies of the packet. The first MFWD instruction points to an entry in the MPLS output table which contains a PUSH and a MSET instruction. A shim header is pushed on the first copy of the packet and the Ethernet layer sends the corresponding frame over the wire via the interface specified by the MSET instruction. The second MFWD instruction points to a different entry in the MPLS output table which contains another PUSH and another MSET instruction. A shim header containing a different label is pushed on the second copy, and the corresponding Ethernet frame is sent using another interface. In Figure 5.2(c), we show how MPLS routers pop shim headers from MPLS multicast packets. There is no difference with popping the shim header of an MPLS unicast packet, except for the protocol number in the Ethernet header of incoming frames.

Last, our implementation supports mixed L2/L3 forwarding. The concept of mixed L2/L3 forwarding has been introduced in Section 1.3.4 and refers to the ability of a router to forward a multicast packet both with an IP and an MPLS mechanism. We perform mixed L2/L3 forwarding by using in the same MPLS input table entry one or several MFWD instructions to forward the packet with an MPLS mechanism, and a DLV instruction to forward the packet with an IP forwarding mechanism. Mixed L2/L3 forwarding support is illustrated in Figure 5.3. Incoming MPLS packets are duplicated by the MPLS layer. One copy remains in the MPLS layer and the other copy is passed to the IP layer.

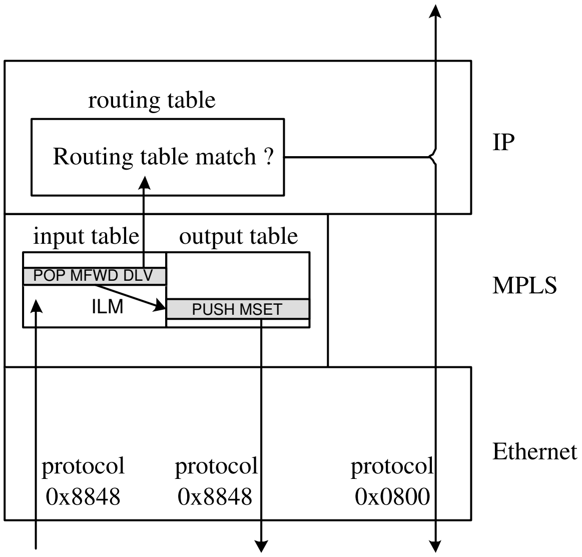


Figure 5.3: **Mixed L2/L3 forwarding implementation.** The same incoming packet is passed to both the IP layer and the Ethernet layer by the MPLS layer. The shim header of the copy of the packet that remains in the MPLS layer is mswapped and the packet is passed to the Ethernet layer, while the copy of the packet that is sent to the IP layer is routed by the IP layer and can either be sent to the Ethernet layer, or be delivered to the transport layer of the MPLS router.

5.1.3 FIB management API

Our implementation provides an API to let user processes modify the FIB of MPLS routers. In MPLS-Linux, the FIB is located inside the Linux kernel. Therefore the

File	Contents
/proc/net/mpls_labelspace	Mapping between physical interfaces and labelspaces.
/proc/net/mpls_fec	FEC mappings.
/proc/net/mpls_in	Input table.
/proc/net/mpls_out	Output table.

Table 5.5: **The /proc files related to the MPLS FIB.** All files are in text format.

implementation of MPLS-Linux requires that a user program communicates with the Linux kernel.

The three communication channels between user programs and the kernel provided by Linux are *ioctl system calls*, *netlink sockets*, and the */proc file system*. MPLS-Linux uses netlink sockets and the /proc file system to access the FIB. Netlink is a datagram oriented socket based interface between the kernel and user programs. The assigned domain for netlink sockets is `PF_NETLINK`. The netlink API provides functions to encapsulate and decapsulate information in *netlink datagrams* which have a specific format. Users can send and receive information encapsulated in netlink datagrams to the kernel via the classic socket calls `send` and `recv`. The /proc file system is a virtual filesystem where files contain information used by the kernel. MPLS-Linux uses certain files of the /proc filesystem (see Table 5.5) to represent the FIB in human-readable text format. Our implementation uses solely the netlink communication interface to communicate with the kernel [53].

MPLS-Linux provides four functions to manipulate the FIB. These functions can either create or remove entries in these tables. The function `send_nhlfe()` creates an entry in the MPLS output table if called with parameter RTM_NEWNHLFE, and deletes an entry in the MPLS output table when called with RTM_DELNHLFE. The function send_xc() binds or unbinds an entry in the input table to an entry in the output table. Therefore, since NHLFE are implemented in both the input and output tables, NHLFE are created or deleted using combinations of send_nhlfe() and send_xc(). A *mcast* field in the message sent by send_nhlfe() specifies whether the NHLFE refers to unicast or multicast packets. The function send_ilm() manages the ILM by creating or deleting entries in the MPLS input table. The function send_ftn() creates or deletes FTN mappings in the IP routing table. We added the function send_mc() to create and delete mappings between an entry in the MPLS input table and several entries in the MPLS output table. MPLS forwarding rules are created with the netlink functions described above. However, a single forwarding rule such as "push unicast label 10" requires to send two netlink messages: a NHLFE and a FTN must be created via send_nhlfe and send_ftn. To assist users in creating forwarding rules, we provide a C API, which is described in Tables 5.6 and 5.7. This simple API simplifies the creation of MPLS forwarding rules by hiding to the user the crafting of complex netlink messages and the call of the netlink functions.

MPLS Operation	Instructions	Netlink functions called	C API
push (unicast)	`PUSH`, `SET`	`send_ftn` `send_nhlfe` with *mcast=0*	`push_label()`, `remove_push_label()`
push (multicast)	`PUSH`, `MSET`	`send_ftn` `send_nhlfe` with *mcast=1*	`push_label()`, `remove_push_label()`

swap	`POP,`
	`PUSH,`
	`SET,`
	`FWD`
		`send_ilm`
		`send_nhlfe`
		`send_xc`
			`swap_label()`,
			`remove_swap_label()`

mswap	`POP,`
	`PUSH,`
	`MSET,`
	`MFWD`
		`send_ilm`
		`send_nhlfe`
		`send_mc`
			`mswap_label()`,
			`remove_mswap_label()`

pop	`POP,`
	`DLV`
		`send_ilm`	`pop_label()`,
			`remove_pop_label()`

Table 5.6: **Netlink functions and the corresponding C API used to set the MPLS forwarding rules.** The C API simplifies the creation and removal of MPLS forwarding rules.

MPLS Operation	C API function	Description
push	`push_label( int ifindex, struct in_addr next_hop, u_int label_id, struct in_addr fec_prefix, u_char fec_len)`	Push a header with label `label_id` on packets of the FEC `fec_prefix`/`fec_len` that are to be transmitted on interface `ifindex` towards `next_hop`.
	`remove_push_label( int ifindex, u_int label_id)`	Remove a rule created with `push_label()`.
swap, mswap	`swap_label( u_int in_label_id, int in_labelspace, u_int out_label_id, int out_if_index, struct in_addr out_next_hop)`	Swap a header with label `in_label_id` of packets that are to be transmitted on interface `in_labelspace` against a header containing label `out_label_id` and forward the packets on the interface indexed by `out_if_index` towards `out_next_hop`.
	`remove_swap_label( u_int in_label_id, int in_labelspace, u_int out_label_id, int out_if_index)`	Remove a rule created with `swap_label()`.
pop	`pop_label( u_int label_id,` int labelspace)	Pop headers containing label `label_id` of packets arriving via an interface that belongs to labelspace `labelspace`.
	`remove_pop_label( u_int label_id, int labelspace)`	Remove a rule created with `pop_label()`.

Table 5.7: **Details on the FIB manipulation API.** The C API hides to the user the crafting of complex netlink messages and the calls of the netlink functions.

5.2 The MulTreeLDP protocol

In this section, we describe *MulTreeLDP*, our signaling protocol for label distribution of multicast LSPs (*mLSPs*). Two signaling protocols have been developed for MPLS unicast, *TE-RSVP* [8] and *LDP/CR-LDP* [3] [41]. Neither supports multicasting, nor have they been implemented for Linux. Unlike RSVP-TE, CR-LDP has been specifically designed for MPLS, thus it is easier to engineer multicast extensions for CR-LDP than for RSVP-TE.

Our MulTreeLDP signaling protocol is a new label distribution protocol which conforms to the requirements of the MPLS architecture [62]. The MulTreeLDP protocol is different from CR-LDP, however we kept the CD-LDP message formats, making it possible to envision a merging of MulTreeLDP with CR-LDP. MulTreeLDP defines three different categories of messages. Advertisement messages create label mappings and LSPs. Link failure and recovery detection messages are used to detect the failure or the recovery of links attached to interfaces of an MPLS router. For proper MulTreeLDP operation, advertisement and notification messages must be delivered reliably and in-order. To satisfy this requirement, we chose TCP (port 2646) as the transport protocol for these messages. On the other hand, a link failure or repair needs to be detected as early as possible and TCP retransmission timers would increase the detection time. Therefore, link failure and recovery detection messages run over UDP (port 2646).

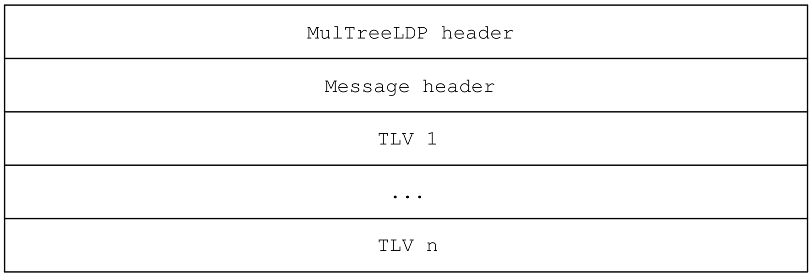


Figure 5.4: **General format of a MulTreeLDP message.** MulTreeLDP messages carry information in the TLV objects.

All MulTreeLDP routers listen for advertisement and notification messages on TCP port 2646. When a router needs to send a MulTreeLDP message to another router, it opens a new connection on port 2646 of the destination router. MulTreeLDP advertisement and notification messages consist of a MulTreeLDP header, a message header, and a series of Type-Length-Value (*TLV*) objects as illustrated in Figure 5.4. A TLV is a 3-tuple which contains the type of the message, the length of the value field, and a value field.

Our implementation of MulTreeLDP consists of three threads. The main thread handles all advertisement and notification messages. On reception of an advertisement message, the main thread can modify the FIB of the router and send other MulTreeLDP messages in accordance with the MulTreeLDP protocol described in the remainder of this section. The main thread can also perform switchback and switchover when it receives notification messages. The second and the third threads detect failures between a router and its neighbors in the multicast routing tree. The second thread sends link failure and recovery detection messages to every neighbor of a router while the third thread listens for these messages. In the remainder of this chapter, we describe the MulTreeLDP signaling protocol.

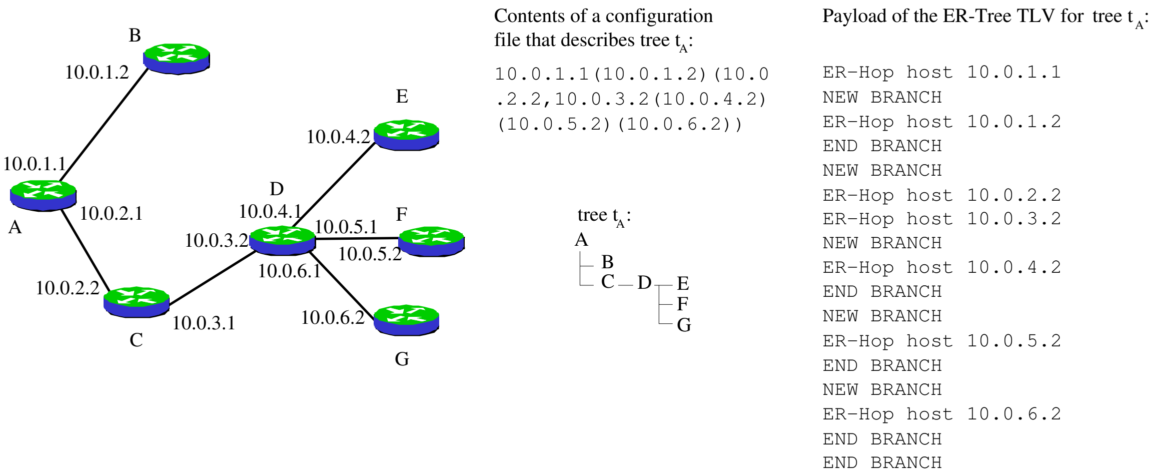
5.2.1 Multicast Explicit Routing

We now describe how MulTreeLDP supports multicast Explicit Routing. Explicit Routing is the technique which enables any node in a network to set up a LSP. This path is usually computed offline. MPLS uses Explicit Routing for traffic engineering purposes but so far no support for multicast routing trees has been defined. MulTreeLDP defines messages and procedures that enable Explicit Routing for core based trees. Our implementation can easily be modified to support shortest path trees. We denote by S the core of the tree.

First we explain how a user can provide the description of a multicast routing tree that is understandable by MulTreeLDP. Users must write in a *configuration file* the multicast routing tree description before MulTreeLDP establishes the mLSP that maps the multicast routing tree. Multicast routing tree description files contain IP addresses separated by a “,” symbol, a “(” symbol, a “)” symbol or a combination of these three symbols. We next describe the rules for describing multicast routing trees.

Trees must be described depth-first, starting from S. The IP address chosen for a router is the address of the interface via which packets would arrive if they were sent by S over the multicast routing tree. The IP address of any interface of S can be used for S. A "," separates the IP addresses of two consecutive routers if the only router downstream of the first router of the expression with regards to S is the second router of the expression. If a router has several children in T, then all subtrees of this router are successively described in the configuration file. The descriptions of the subtrees are put between a "(" and a ")" symbol and the description of the first subtree follows immediately the IP address of the parent router. Subtrees are described using the above rules recursively. Let us take an example to illustrate the rules. Consider Figure 5.5 and tree T_A rooted at A. The configuration file first contains any IP address of A. This IP address can be either `10.0.1.1` or `10.0.2.1`. We arbitrarily choose `10.0.1.1` as the address that describes S and write this address in the configuration file. Node S has two children B and D, therefore two subtrees T_B and T_C of T_A originate from a child node of S. We put the description of these subtrees in brackets. The first subtree T_B is constituted by node B only. If packets are sent from S to B then they arrive at B via the interface associated with IP address `10.0.1.2`. Therefore T_B is described by `(10.0.1.2)`. The second subtree T_C is the subtree of T_A rooted at C. Node C has only one child D so the description of T_C will start with `(10.0.2.2,10.0.3.2`. Then D has three children E, F and G that are leaves of the tree, hence T_C is described by `(10.0.2.2,10.0.3.2(10.0.4.2)(10.0.5.2)(10.0.6.2))`. Consequently the configuration file for T_A contains:

```
10.0.1.1(10.0.1.2)(10.0.2.2,10.0.3.2(10.0.4.2)(10.0.5.2)(10.0.6.2)).
```



Contents of a configuration file that describes tree t_A:

```
10.0.1.1(10.0.1.2)(10.0
.2.2,10.0.3.2(10.0.4.2)
(10.0.5.2)(10.0.6.2))
```

Payload of the ER-Tree TLV for tree t_A:

```
ER-Hop host 10.0.1.1
NEW BRANCH
ER-Hop host 10.0.1.2
END BRANCH
NEW BRANCH
ER-Hop host 10.0.2.2
ER-Hop host 10.0.3.2
NEW BRANCH
ER-Hop host 10.0.4.2
END BRANCH
NEW BRANCH
ER-Hop host 10.0.5.2
END BRANCH
NEW BRANCH
ER-Hop host 10.0.6.2
END BRANCH
END BRANCH
```

Figure 5.5: **Description of a tree in a file and in a Explicit Route Tree TLV.** Node A is the core of the tree.

We now explain how MulTreeLDP builds the bidirectional mLSP that maps a multicast routing tree described in a tree configuration file. Any computer can establish an explicit mLSP. For example, a multicast routing tree can be computed by a computer offline and this computer can establish the mLSP. The computer responsible for establishing the tree first parses the configuration file. The computer creates an *Explicit Route Hop TLV* for each IP address found in the file, a *New Branch TLV* for each "(" symbol and an *End Branch TLV* for each "(" symbol. Then the computer responsible for establishing the mLSP creates an *Explicit Route Tree TLV* that contains all Explicit Route, New Branch and End Branch TLVs it has created in the order their corresponding entries were parsed in the file. In Figure 5.5, we show the contents of the Explicit Route Tree TLV (ER-Tree TLV) for the example developed in the previous paragraph. Also the computer creates a FEC TLV which contains the IP multicast address of the group for which we want to create the mLSP. Finally the computer sends an *Explicit Route Request message* that contains the Explicit Route Tree TLV and the previously created FEC TLV to the router designated by the IP address of the first Explicit Route Hop TLV. This IP address is the address of one of the interfaces of the core of the tree. The size of a multicast routing tree established with MulTreeLDP is limited by the maximum size of a TCP datagram.

Each child of the core is the root of a subtree of the multicast routing tree. When the core receives the Explicit Route Request message, it extracts the tree topology from the contents of the message. The core of the multicast routing tree builds a new Explicit Route Tree TLV that contains the description of one of these subtrees for each of its children. Then the core sends a *Label Request Message* that contains one of the Explicit Route Tree TLVs to the corresponding child. When a node B receives a Label Request Message from a node A, it performs two actions. First, B sends a label mapping to A. Node B chooses a label, creates the corresponding *Label TLV* and sends to A a *Label Mapping message* that contains the *Label TLV*. Because the mLSP that maps the multicast routing tree is bidirectional, node B that sent the *Label Mapping message* also sends a *Label Request message* to A. The Explicit Route Tree TLV sent in this message contains only the IP address of the interface of A on the link between A and B. Node A replies with a Label Mapping message. At that point, a bidirectional LSP is established between A and B, or in this case the core and one of its children. Second, for each child C of B, B builds an Explicit Route Tree TLV that contains the description of the subtree rooted at C and sends a Label Request Message that contains this Explicit Route Tree TLV to C. The mLSP formerly created between the core and its children is thus expanded until all leaves of the tree are reached. When all leaves of the tree are reached, the mLSP maps the entire multicast routing tree.

As an example, consider the tree depicted in Figure 5.6(a). Figures 5.6(b), (d), (f), (g), (h), (j), (l) depict the MulTreeLDP messages exchanges in the MPLS network while Figures 5.6(c), (e), (g), (i), (k), (m) depict the expansion of the mLSP that maps the multicast routing tree after a message exchange. Suppose node A receives a Label Request message that contains the Explicit Route Tree TLV presented in Figure 5.5. Node A sends a Label Request message to nodes B and C (Figure 5.6(b)). Both messages contain an Explicit Route Tree TLV. The Explicit Route Tree TLV that is sent to B contains the IP address `10.0.1.2` of B. The Explicit Route Tree TLV that is sent to C describes the subtree of the multicast routing tree that is rooted at node C: `10.0.2.2,10.0.3.2(10.0.4.2)(10.0.5.2)(10.0.6.2)`. At that point no label mapping is established yet as shown in Figure 5.6(c). Then, nodes B and C send a Label Mapping message to A (Figure 5.6(d)). The portions of

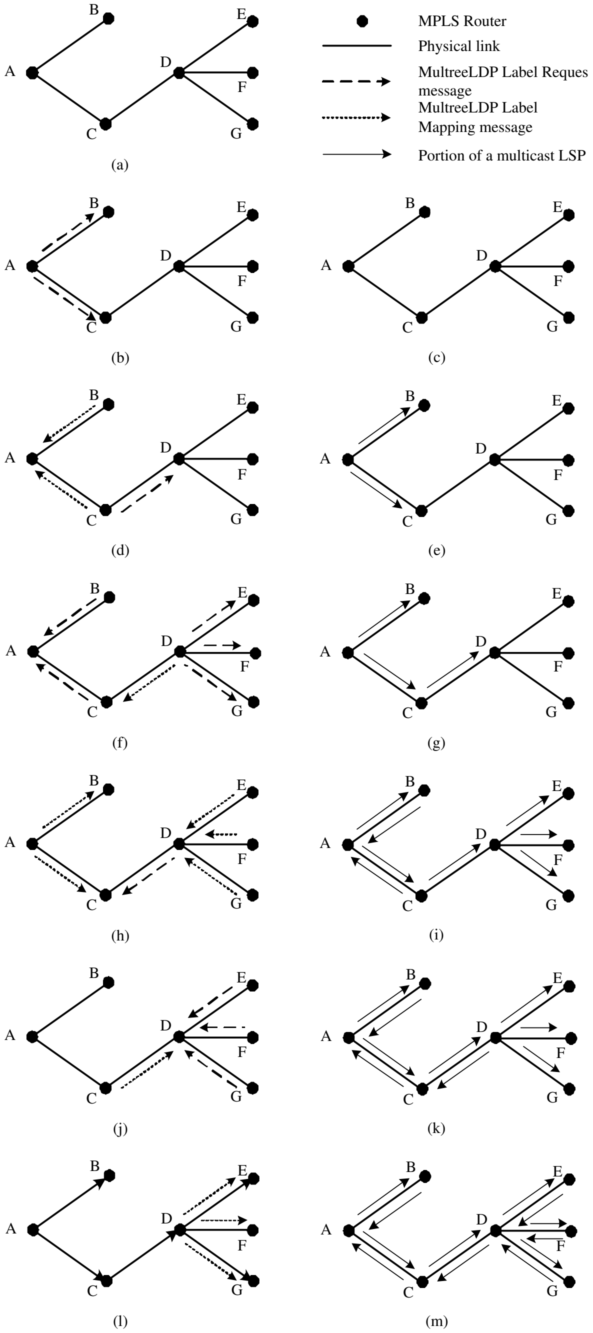

B
E
A
D
F
C
G
(a)
MPLS Router
Physical link
MultreeLDP Label Reques message
MultreeLDP Label Mapping message
Portion of a multicast LSP
(b)
(c)
(d)
(e)
(f)
(g)
(h)
(i)
(j)
(k)
(l)
(m)

Figure 5.6: **Advertisement of a multicast routing tree.**

the mLSP that correspond to the label mapping established after A receives the Label Mapping messages from B and C are shown in Figure 5.6(e). Also, node C sends to D a Label Request message that contains the Explicit Route Tree TLV which contains the description the subtree of the multicast routing tree rooted at D: `10.0.3.2(10.0.4.2)(10.0.5.2)(10.0.5.2)`. In Figure 5.6(f), nodes B and C send a Label Request message to A in order to establish the second direction of the portions of the mLSP between B and A on the one hand, and between C and A on the other hand. The Label Request message sent by B to A contains the Explicit Route Tree TLV `10.0.1.1` and the Label Request message sent by C to A contains the Explicit Route Tree TLV `10.0.2.1`. Node D sends a Label Request message to E, F and G. The Label Request message sent to E contains the Explicit Route Tree TLV `10.0.4.2`, the Label Request message sent to F contains the Explicit Route Tree TLV `10.0.5.2`, and the Label Request message sent to G contains the Explicit Route Tree TLV `10.0.6.2`. Node D also sends a Label Mapping message to C and establishes the portion of the mLSP from C to D as shown in Figure 5.6(g). In Figure 5.6(h), node A sends a Label Mapping message to B and C and a bidirectional mLSP is established between A and B on the one hand, and A and C on the other hand. Node D sends a Label Request message to C to establish the other direction of the part of the mLSP between C and D. The Label Request message sent by D to C contains the Explicit Route Tree TLV `10.0.3.1`. Nodes E, F and G send a Label Mapping message to D and the three corresponding mLSP portions are established.

The new mLSP portions are depicted in Figure 5.6(i). In Figure 5.6(j), node C sends a Label Mapping message to D. The part of the mLSP between C and D becomes bidirectional (Figure 5.6(k)). Nodes E, F and G send a Label Request message to C. In Figure 5.6(l), D replies to E, F and G with Label Mapping messages to establish a bidirectional mLSP between D and E, D and F, and D and G. The mLSP that maps the multicast routing tree is complete (Figure 5.6(m)).

Backup path establishment is very similar to mLSP establishment, except that unidirectional mLSPs are created instead of bidirectional mLSPs. When a computer wants to advertise a backup path, it sends a *Backup Route Request message* to each of the end nodes of the backup path. This message contains the FEC TLV that corresponds to the FEC of the protected multicast routing tree, the Explicit Route Tree TLV that corresponds to the backup path as described in Section 4.4, and a *Routers TLV*. Since a path can be viewed as a tree where every node has only one child, we kept the format of the Explicit Route Tree TLV to describe a backup path. Therefore the Explicit Route Tree TLV contains Explicit Route Hop TLVs only. The routers TLV contains all the IP addresses of MPLS routers interfaces on the protected path. Then, the node that receives the Backup Route Request message removes the first hop from the Explicit Route Tree TLV and sends a *Backup Label Request message* to the next hop of the backup path. The Backup Label Request message contains the new Explicit Route Tree TLV and the FEC TLV of the protected tree. When a node receives a Backup Label Request message, it sends another Backup Label Request message to the next hop of the backup path after it has removed a Explicit Route Hop TLV from the Explicit Route Tree TLV which describes the backup path. Also, this node sends a *Backup Label Mapping message* to the sender of the Backup Label Request message. The Backup Label Mapping message contains a Label TLV. At that point a label mapping is established in one direction between the first node and the second node of the backup path. Other mappings are established similarly with exchanges of Backup Label Request and Backup Label Mapping messages.

The process we have described establishes a LSP that maps only one direction of a backup path. Using the notations of Section 4.4, this path is for instance the path between PSL A and node S' via the backup path. In order to establish the second direction, the node that sent the Backup Route Request message needs to send a Backup Route Request message to the other end of the backup path. This second Backup Route Request message describes the reverse direction of the backup path, that is, the path between PSL B and node S' via the backup path.

5.2.2 Link failure and recovery detection

Link failure and recovery detection is implemented with two threads. Consider an MPLS router A. Every period T_p, the first thread of A sends a probe to each of its neighbors on UDP port 2646. A node of the MPLS network can be an neighbor for

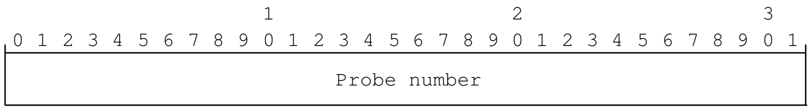


Figure 5.7: **Link failure and recovery detection probe format.** Probes have been kept as small as possible in order to decrease the failure or recovery detection time.

A with regards to several multicast routing trees. In that case, A sends only one probe every period T_p to this multiple neighbor. Router A maintains a counter per neighbor and increments this counter by “1” every period nT_p and puts the value of this counter in the probes. This counter value is the only payload of a probe (see Figure 5.7).

The second thread listens for probe messages on UDP port 2646. Every period nT_p ($n \geq 2$), the second thread checks the number of received probes for each neighbor. If this number is zero, then the link from A to its neighbor has failed. Router A has detected a link failure and sends link failure notification messages using the mechanism described in Section 5.2.3. Router A internally records the link failure. When router A receives a probe on a link recorded as failed, it immediately sends link recovery notification messages using the mechanism described in Section 5.2.3.

The payload of the probes is not used in our implementation of MulTreeLDP but can be used in future work for probe numbering to dismiss reordered or duplicate probes.

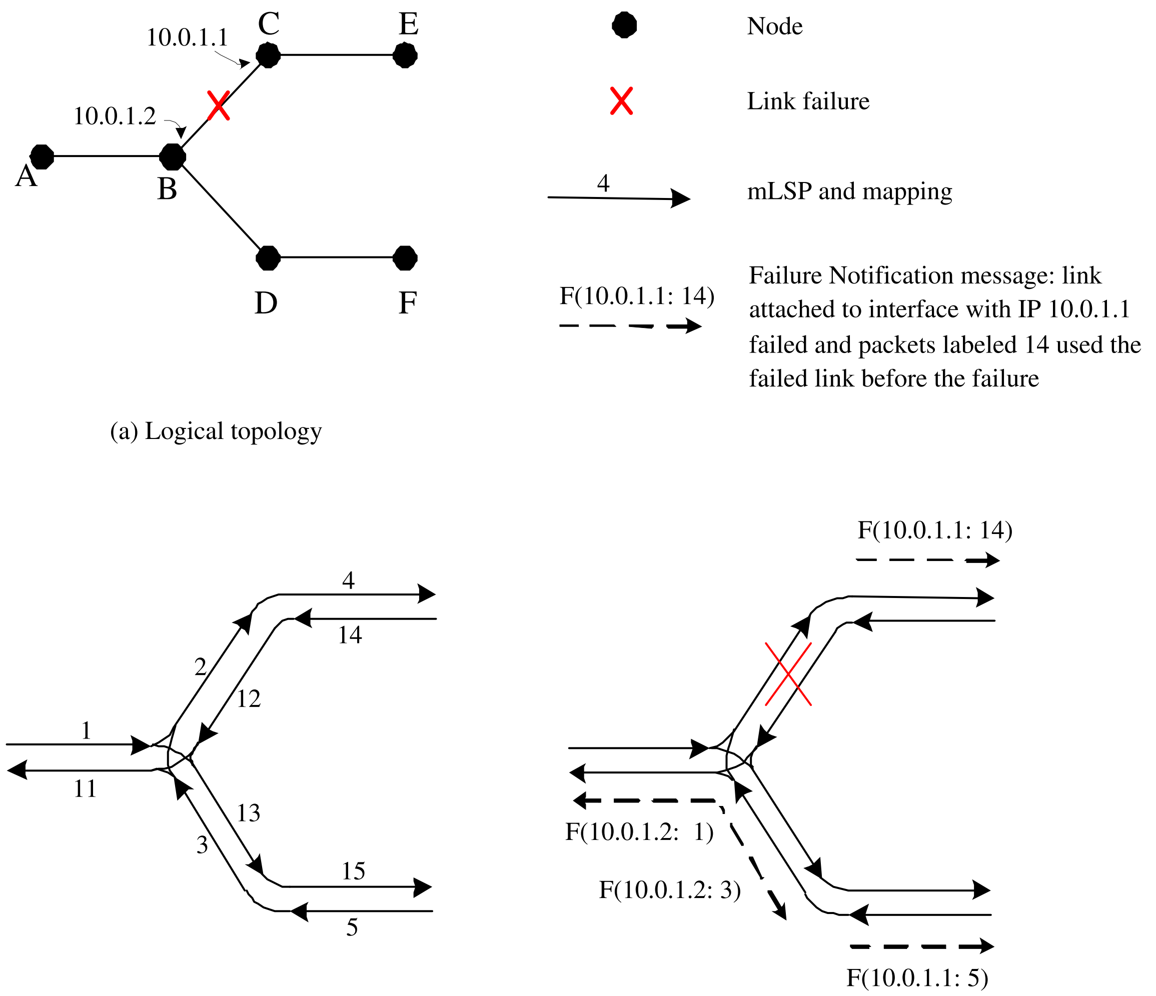
5.2.3 Link failure and recovery notification

When a node detects a link failure or a link recovery, it must notify the two PSLs of the mLSP of the event so that they can perform either a switchover in case of a link failure on the protected path or a switchback in case of a link recovery on the protected path. A node that detects the failure or recovery may be a LSR or LER for several mLSPs and does not know which routers are the PSLs for each of these mLSP, therefore it must notify all routers of each mLSP of the failure or recovery.

Suppose a node detects a link failure using the link failure detection mechanism presented in Section 5.2.2. After the link has failed, the node creates an *IPv4 node TLV* which contains the IP address of the interface on which the failed link is connected. The node then considers in turn each link from which it receives traffic, except the failed link. For each of these links, it builds a list of the labels of packets for flows forwarded from the considered link to the failed link. Then the node sends a *Link Failure Notification message* that contains the IPv4 node TLV and a Label TLV per label in the aforementioned list on all links except the failed link. When a node receives a Link Failure Notification message, it knows that all outgoing packets that are labeled with a label of the list embedded in the Link Failure Notification message will eventually be dropped due to the failed link. A node which receives a Link Failure Notification message acts as if the link through which it received the message had failed and builds a list of incoming labels for packets that are forwarded on this link. This node sends on every link, except the link through which it has been notified of the failure, a Link Failure Notification message which contains a Label TLV for each label of the list and the same IPv4 node TLV as the one it received in the Link Failure Notification message. The failure notification information is thus propagated by each node that detects the failure to all routers of all mLSPs that use the failed link.

Consider, for instance, the multicast routing tree topology from Figure 5.8(a) and the mLSP that maps the tree in Figure 5.8(b). Suppose that the link BC between nodes B and C fails. Figure 5.8(c) shows the Failure Notification process. Both B and C detect the failure. Before the failure, node B forwards packets labeled with “2” on link BC. Those packets are the packets labeled with “1” coming from A, and the packets labeled with “3” coming from D. The IP address of the interface of B at the failed link is 10.0.1.2. Therefore, B sends a Link Failure Notification message with an IPv4 node TLV containing the IP address 10.0.1.2 and the Label TLV for label “1” to A, and a Link Failure Notification message with an IPv4 node TLV containing the IP address 10.0.1.2 and the Label TLV for label “3” to D. When D receives the Link Failure Notification message from B, it knows that the packets forwarded with label “3” will eventually reach the interface with IP address 10.0.1.2 and then will be dropped. Node D forwards packets with label “5” coming from node F on link FD with label “3”. Therefore, D sends a Link Failure Notification message with an IPv4 node TLV containing the IP address 10.0.1.2 and the Label TLV for label “5” to *F*. On the other end of the failed link, node *C* also detects the failure. Before the failure, node *D* forwards packets with label “14” coming from node *E* on link *BC* with label “12”. The IP address of the interface of *C* at the failed link is 10.0.1.1. Therefore, *C* sends a Link Failure Notification message with an IPv4 node TLV containing IP address 10.0.1.1 and the Label TLV for label “14” to *E*. At this point, all nodes of the mLSP are notified of the link failure. If the mLSP is protected by a backup path, then the end nodes of the backup path must perform a switchover.

There is no difference between the link recovery notification and link failure notification processes except in the type of the messages used. Indeed, the link recovery notification process uses *Link Recovery Notification messages* to notify the other nodes of mLSP whose traffic was forwarded on the recovered link before the failure. On the example illustrated by Figure 5.8(c), suppose that the formerly failed link *BC* is repaired. Nodes *B* and *C* detect the recovery with the mechanism described in Section 5.2.2. Node *B* sends a Link Recovery Message to node *A* with the IPv4 node TLV containing the IP address 10.0.1.2 and a Label TLV for label “1”, and a Link Recovery Message to node *A* with the same IPv4 node TLV and a Label TLV for label “3” to *D*. When it receives the Link Recovery Notification message from *B*, *D* sends a Link Recovery notification message to *F* which contains the same IPv4 node TLV again and a Label TLV for label “5”. Node *C* also detects the link recovery and sends


10.0.1.1
C
E
10.0.1.2
A
B
D
F
Node
Link failure
4
mLSP and mapping
F(10.0.1.1: 14)
Failure Notification message: link attached to interface with IP 10.0.1.1 failed and packets labeled 14 used the failed link before the failure
4
14
2
12
1
11
13
3
15
5
F(10.0.1.1: 14)
F(10.0.1.2: 1)
F(10.0.1.2: 3)
F(10.0.1.1: 5)

(a) Logical topology

(b) Multicast label mappings

(c) Notification messages contents

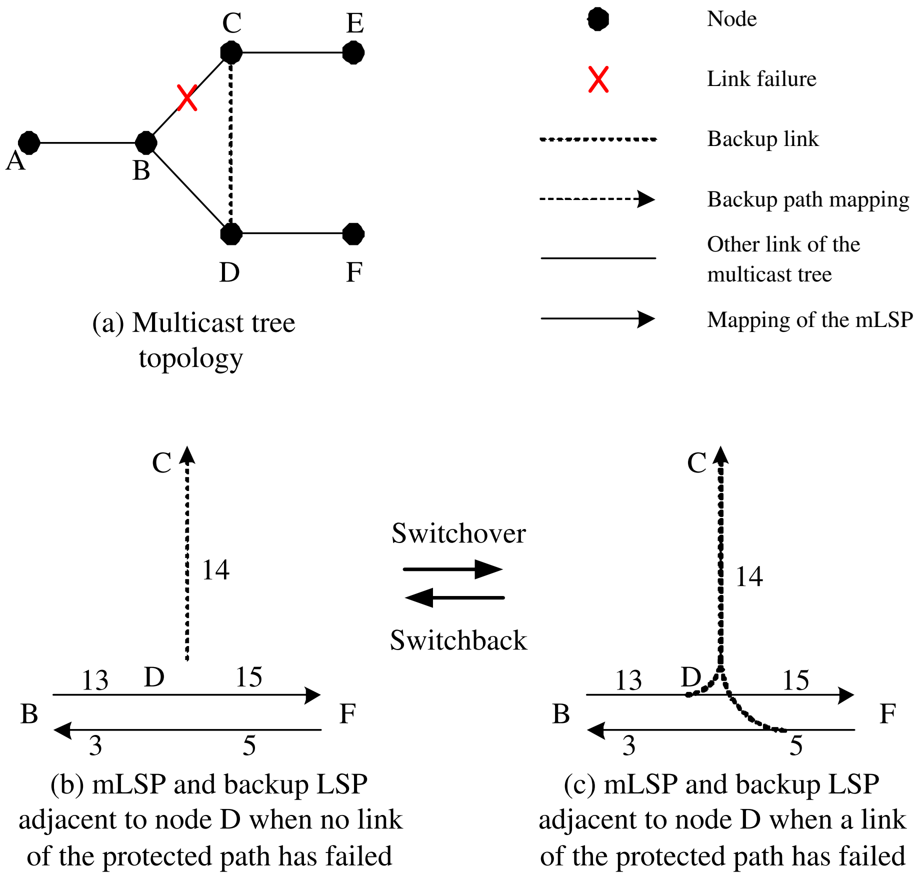
Figure 5.8: **Failure and recovery notification process.**

Link Recovery notification message to E which contains the IPv4 node TLV for the IP address 10.0.1.1 and a Label TLV for label “14”. At this point all nodes of the mLSP are notified of the link recovery. If the mLSP is protected by a backup path, then the end nodes of the backup path must perform a switchback.

5.2.4 Switchover and switchback

Consider a mLSP protected by a backup path and the associated protected path of the backup path. A PSL of a mLSP must perform switchover when itself detects the failure of a link of the protected path or when it is notified of the failure of a link of the protected path. Likewise, a PSL of a mLSP must perform switchback when it detects the recovery of a link of the protected path or when it is notified of the recovery of a link of the protected path.

When a node receives a Link Failure Notification message or detects the failure of a link attached to one of its interfaces, it first checks if it is a PSL for a mLSP that uses the failed link. PSLs are nodes which receive a Backup Route Request message. Backup Route Request messages contain the list of the interfaces that end a link of the protected path. We call *failed interface* the IP address contained in the IPv4 Node TLV of the Link Failure Notification message if the PSL received such a message, or the IP address of the interface at the failed link if the PSL itself detected the failure. If the failed interface matches one of the addresses previously received in the Router TLV of a Backup Route Request message, the PSL knows that one link on the protected path has failed and must perform a switchover. The switchover consists of modifying the MPLS forwarding tables of the LSRs and is performed instantaneously on each of the LSRs. Tables are changed such that all incoming traffic from the protected mLSP is forwarded also on the backup path.



(a) Multicast tree topology

(b) mLSP and backup LSP adjacent to node D when no link of the protected path has failed

(c) mLSP and backup LSP adjacent to node D when a link of the protected path has failed

Figure 5.9: **Switchback and switchover.** Modification of the forwarding tables at a PSL.

Consider, for example, Figure 5.9(a). Nodes C and D are the PSLs for the tree represented as solid lines protected by the backup path represented in dotted lines. The backup path consists of link CD and the protected path consists of links BC and BD. We study the behavior of PSL D on failure and recovery of link BC . When link BC has not failed, D forwards traffic from F to B and from B to F as shown in Figure 5.9(b) where only one direction of the backup path between C and

D is represented. Packets coming from B with label “13” are forwarded to F with label “15” and packets coming from F with label “5” are forwarded to B with label “3”. Node C does not forward packets over the backup path (Figure 5.9(b)). When D learns that it must perform switchover because a link of the protected path has failed, D forwards packets with label “13” from B both to F using label “15” and over the backup path to C using label “14”. Node D also forwards packets with label “5” from F both to B using label “3” and over the backup path to C using label “14” (Figure 5.9(c)). When node D learns it must perform switchback because the failed link has been repaired, it stops forwarding packets over the backup path as in Figure 5.9(b).

5.2.5 MulTreeLDP messages and TLV formats

The messages and TLVs in MulTreeLDP are similar to those in LDP and CR-LDP. We indicate for each message or TLV whether it is a modified CR-LDP message which has been amended or a new message. The general form of MulTreeLDP messages is a MulTreeLDP header, followed by a message header and any number of TLV objects (Figure 5.4). A TLV object itself consists of a TLV header which contains the type and the length of the TLV, and a payload which contains the value of the TLV.

MulTreeLDP header

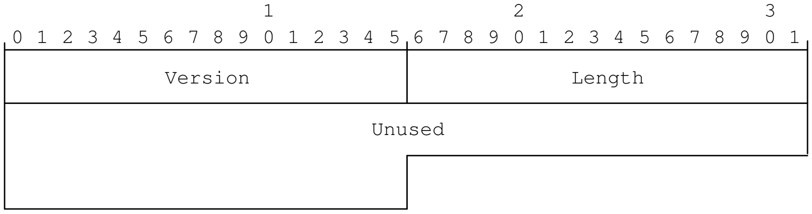


Figure 5.10: **MulTreeLDP header.**

The `Version` field is set to 0x1. The `Length` is the length in bytes of the MulTreeLDP message, version and length fields of the MulTreeLDP message excluded.

We now give the format of all TLVs and messages presented in this chapter. MulTreeLDP TLVs and messages are compatible with CR-LDP. The “u” and “unused” bits that appear in the MulTreeLDP TLV and messages formats correspond to CR-LDP fields that are not used in MulTreeLDP.

MulTreeLDP TLV formats

FEC TLV

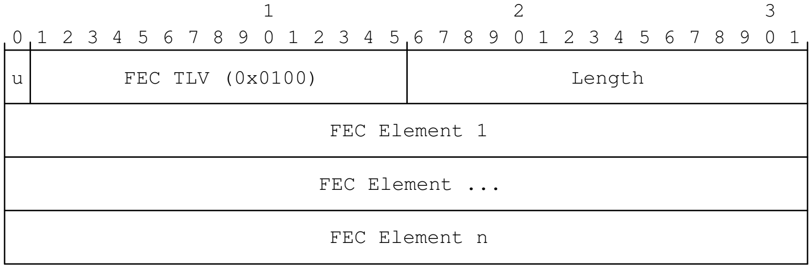


Figure 5.11: **FEC TLV format.**

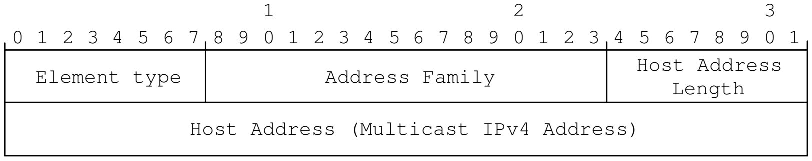


Figure 5.12: **FEC element format.**

The *FEC TLV*, of type `0x0100`, is adapted from CR-LDP. A FEC is defined in a *FEC element* by a prefix and a prefix length. For an IPv4 address like an IP multicast group address, the LDP specification [3] requires that the *element type* field is set to “2”, the *Address Family* to “1” and the *Host Address Length* to “4” which is the number of bytes of the IP address. The *host address* is the IPv4 address of the multicast group described by the FEC element.

Label TLV

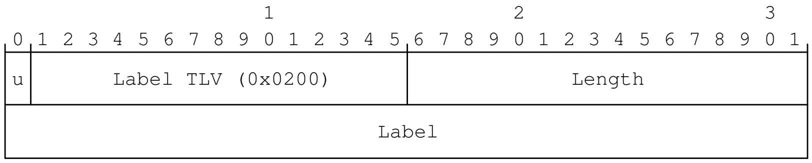


Figure 5.13: **Label TLV format.**

The *Label TLV*, of type `0x0200`, is adapted from CR-LDP and contains a label number.

IPv4 node TLV

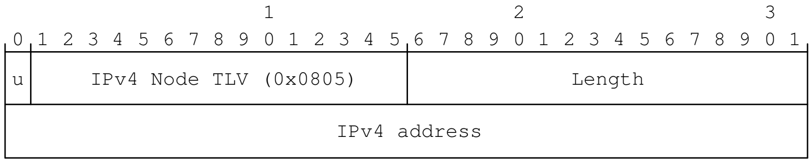
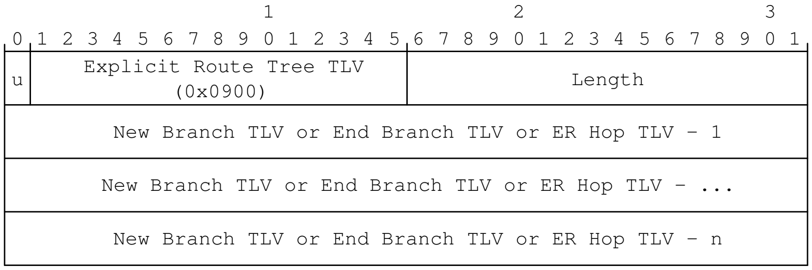


Figure 5.14: **IPv4 node TLV format.**

The *IPv4 node TLV*, of type `0x0805`, is a new TLV and does not exist in CR-LDP. It contains an IPv4 address and is used in the notification messages.

Explicit Route Tree TLV

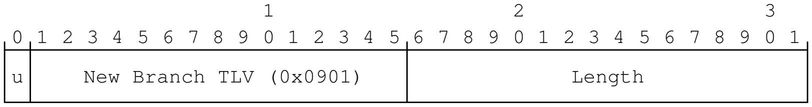


```
                    1                   2                   3
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
u | Explicit Route Tree TLV (0x0900) | Length
New Branch TLV or End Branch TLV or ER Hop TLV - 1
New Branch TLV or End Branch TLV or ER Hop TLV - ...
New Branch TLV or End Branch TLV or ER Hop TLV - n
```

Figure 5.15: **Explicit Route Tree TLV format.**

The *Explicit Route Tree TLV*, of type `0x0900`, is a new TLV and does not exist in CR-LDP. It defines a multicast explicit route tree. A multicast routing tree definition consists of New Branch TLVs (Figure 5.16), End Branch TLVs (Figure 5.17), and Explicit Route Hop TLVs (Figure 5.18).

New Branch TLV



```
                    1                   2                   3
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
u | New Branch TLV (0x0901) | Length
```

Figure 5.16: **New Branch TLV format.**

The *New Branch TLV*, of type `0x0901`, is a new TLV and does not exist in CR-LDP. It has no payload and therefore the length field is set to “0”.

End Branch TLV

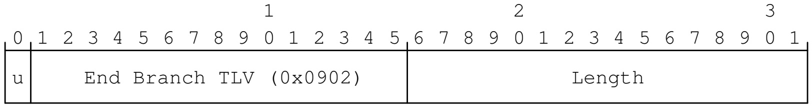


Figure 5.17: **End Branch TLV format.**

The *End Branch TLV*, of type `0x0902`, is a new TLV and does not exist in CR-LDP. It has no payload and the length field is set to “0”.

Explicit Route Hop TLV

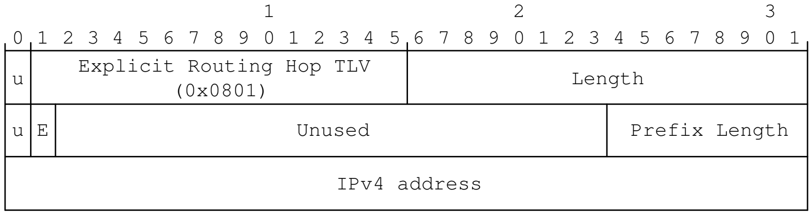


Figure 5.18: **Explicit Route Hop TLV format.**

The *Explicit Route Hop TLV*, of type `0x0801`, is adapted from CR-LDP. An Explicit Route Hop TLV contains the IP address in the *IPv4 address* field of an interface of a router part of an multicast route tree. The *Prefix Length* field is set to “32”, which is the number of bits of an IPv4 address. If the “E” bit is set then packets received on this interface must be passed to the IP layer and the host can perform mixed L2/L3 forwarding.

Routers TLV

	1	2 3
0	1 2 3 4 5 6 7 8 9 0 1 2 3 4 5	6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
u	Routers TLV (0x0910)	Length
Explicit Routing Hop TLV - 1		
Explicit Routing Hop TLV - ...		
Explicit Routing Hop TLV - n		

Figure 5.19: **Routers TLV format.**

The *Routers TLV* of type `0x0910` is a new TLV and does not exist in CR-LDP. The Routers TLV contains a list of Explicit Route Hop TLV.

MulTreeLDP message formats

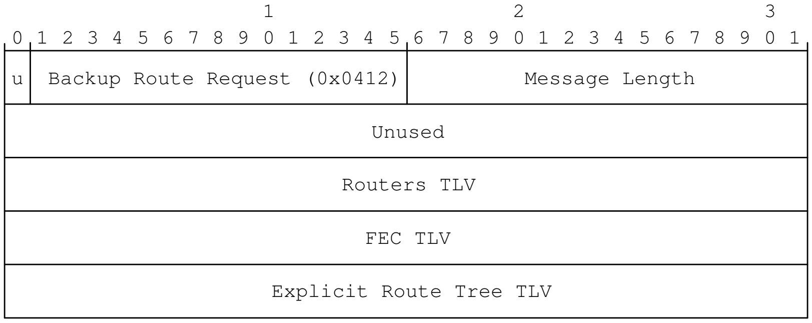
Explicit Route Request message

	1	2 3
0	1 2 3 4 5 6 7 8 9 0 1 2 3 4 5	6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
u	Explicit Route Request (0x0411)	Message Length
Unused		
FEC TLV		
Explicit Route Tree TLV		

Figure 5.20: **Explicit Route Request message format.**

The *Explicit Route Request message*, of type `0x0411`, is a new message and does not exist in CR-LDP. Any router can send a route request message to initiate the creation of an Explicit Route Tree. The Explicit Route Request message must be sent to the core of the tree. The FEC TLV (Figure 5.11) contains the definition of the FEC for the advertised tree. The Explicit Route Tree TLV (Figure 5.15) contains the definition of the tree.

Explicit Backup Route Request message



```
                    1                   2                   3
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
u  Backup Route Request (0x0412)       Message Length
                           Unused
                         Routers TLV
                           FEC TLV
                   Explicit Route Tree TLV
```

Figure 5.21: **Explicit Backup Route Request message format.**

The *Explicit Backup Route Request message*, of type `0x0412` is a new message and does not exist in CR-LDP. Any router can send an Explicit Backup Route Request message to initiate the creation of a backup path. This message has to be sent to the first hop of the backup path. Two Explicit Backup Route Request messages must be sent to establish a bidirectional backup path. The FEC TLV (Figure 5.11) contains the definition of the FEC for the tree protected by the backup path. The Routers TLV (Figure 5.19) contains the IP addresses of interfaces at the end of links of the protected path. The Explicit Route Tree TLV (Figure 5.15) contains the definition of the backup path.

Label Request message

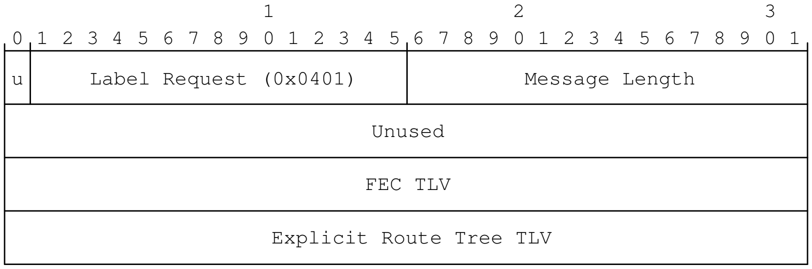


Figure 5.22: **Label Request message format.**

The *Label Request message*, of type `0x0401`, is adapted from CR-LDP. A router sends a Label request message to request a label mapping for a LSP defined in the Explicit Route Tree TLV (Figure 5.15) associated to the FEC defined in the FEC TLV (Figure 5.11).

Backup Label Request message

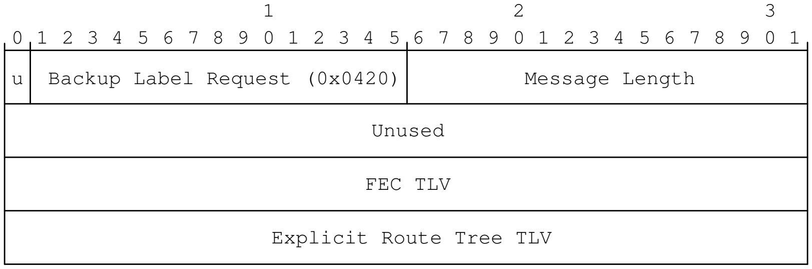
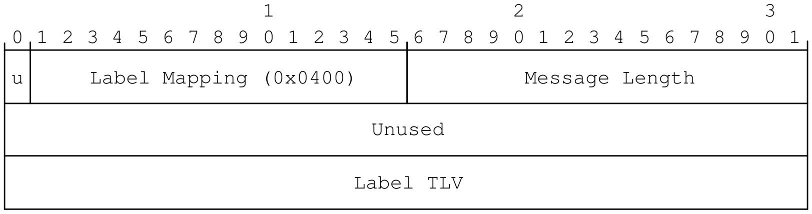


Figure 5.23: **Backup Label Request message format.**

The *Explicit Backup Route Request message*, of type `0x0412`, is a new message and does not exist in CR-LDP. This message is the same as Label Request message, except that the Explicit Route Tree TLV describes a backup LSP associated with the FEC defined by the FEC TLV.

Label Mapping message

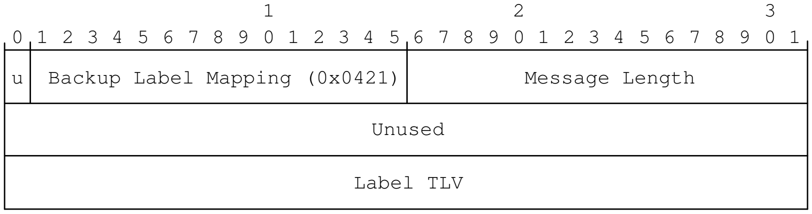


u	Label Mapping (0x0400)	Message Length
Unused		
Label TLV		

Figure 5.24: **Label Mapping message format.**

The *Label Mapping message*, of type `0x0400`, is adapted from CR-LDP. Label Mapping messages are sent from downstream nodes to upstream nodes for a given LSP. A Label Mapping message contains a label mapping in the Label TLV (Figure 5.13) for the LSP associated with the FEC defined in the FEC TLV (Figure 5.11).

Backup Label Mapping message

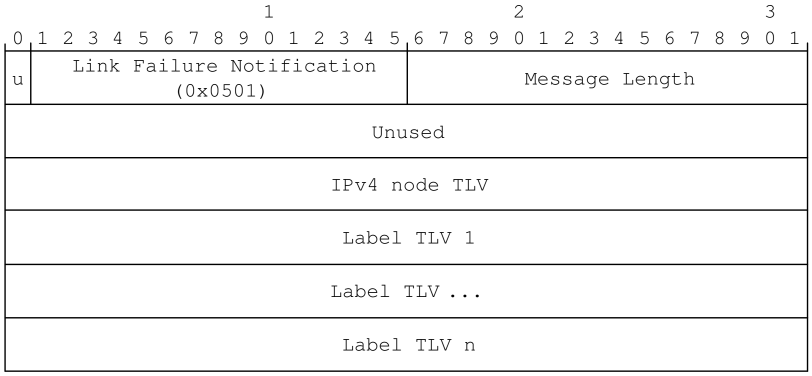


u	Backup Label Mapping (0x0421)	Message Length
Unused		
Label TLV		

Figure 5.25: **Backup Label Mapping message format.**

The *Backup Label Mapping*, of type `0x0421`, is a new message and does not exist in CR-LDP. This message is the same as the Label Mapping message, except that the label in the label TLV refers to the backup path associated with the FEC defined in the FEC TLV.

Link Failure Notification message

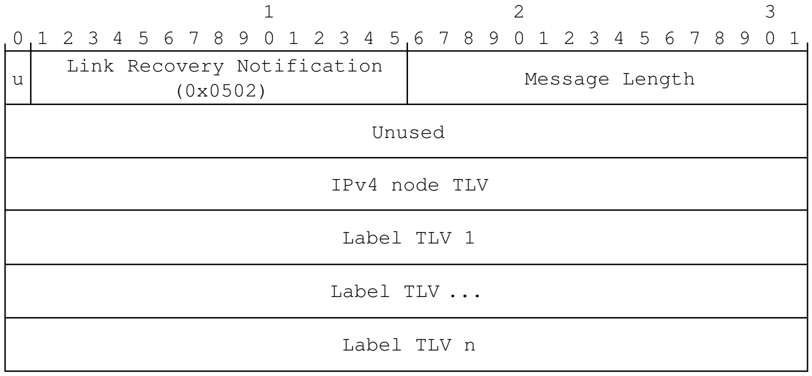


```
                    1                   2                   3
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
u | Link Failure Notification (0x0501) | Message Length
Unused
IPv4 node TLV
Label TLV 1
Label TLV ...
Label TLV n
```

Figure 5.26: **Link Failure Notification message format.**

The *Link Failure Notification message*, of type `0x0501`, is a new message and does not exist in CR-LDP. This message contains the IP address in the IPv4 node TLV (see Figure 5.14) of an interface on which a failed link is attached. It also contains a list of (incoming) labels in the Label TLVs (Figure 5.13) as explained in Section 5.2.3.

Link Recovery Notification message



```
                    1                   2                   3
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-----------------------------+-------------------------------+
|u| Link Recovery Notification  |        Message Length         |
| |         (0x0502)            |                               |
+-+-----------------------------+-------------------------------+
|                            Unused                             |
+---------------------------------------------------------------+
|                         IPv4 node TLV                         |
+---------------------------------------------------------------+
|                          Label TLV 1                          |
+---------------------------------------------------------------+
|                         Label TLV ...                         |
+---------------------------------------------------------------+
|                          Label TLV n                          |
+---------------------------------------------------------------+
```

Figure 5.27: **Link Recovery Notification message format.**

The *Link Recovery Notification message*, of type `0x0502`, is a new message and does not exist in CR-LDP. This message contains the IP address in the IPv4 node TLV (see Figure 5.14) of an interface on which a recovered link is attached. It also contains a list of (incoming) labels in the Label TLVs (Figure 5.13) as explained in Section 5.2.3.

6

Experiments

In this chapter, we present the experiments that assess the performance of MPLS multicast Fast Reroute. All experiments are performed on Linux PCs equipped with Fast Ethernet network adapters. We successively evaluate the performance of all components of MPLS multicast Fast Reroute. These components are MPLS-Linux multicast, the link failure and recovery detection mechanisms, the link failure and recovery notification mechanisms, and the switchover and switchback mechanisms. In Section 6.1, we discuss the hardware used to conduct the experiments. In Section 6.2, we assess the performance of MPLS-Linux augmented with our multicast extensions by measuring the maximum throughput of data that can flow on a multicast LSP. We compare this maximum throughput with the throughput achieved with MPLS unicast and IP unicast. We show that MPLS-Linux with multicast support can fully exploit the capacity of Fast Ethernet links. In Section 6.3, we evaluate the link failure and recovery detection mechanisms. We simulate link failure and recovery by manually enabling up and disabling down an interface attached to a link of a mLSP. Disabling an interface is equivalent to physically cutting a link and enabling a disabled interface is equivalent to repairing a link. We measure the time required by the end nodes of the link to detect the simulated failures and repairs. We show that the time to detect a failure ($T_{fdetect}$) and the recovery ($T_{rdetect}$) of a link conforms to the distributions determined in Section 4.2, with an average value of a few tens of milliseconds. In Section 6.4, we measure the node notification delay T_{nnotif} for notification messages. We show that the node notification delay is close to 1 millisecond and therefore the order of magnitude of the notification time T_{notif} is a few milliseconds to tens of milliseconds depending on the multicast routing tree topology. In Section 6.5, we measure the total time to repair a multicast routing tree when a link fails. We send traffic over a mLSP and monitor the packet arrivals at the receivers. We measure the interruption time T_{repair} seen by the receivers when we simulate a link failure. We compare T_{repair} with the sum of $T_{fdetect}$ and T_{notif} measured in the previous sections and show that the network is repaired in less than the SONET requirement of 50 ms.

6.1 Hardware used for the experiments

We perform our experiments on a testbed of six identical Intel PCs PII-450 with 128MB of memory running Linux and MPLS-Linux with our multicast extensions (see Figure 6.1).

The network topology is depicted in Figure 6.2. Each PC is equipped with five Fast Ethernet adapters (100BaseTX). Only one interface of each PC, *eth0*, has Internet

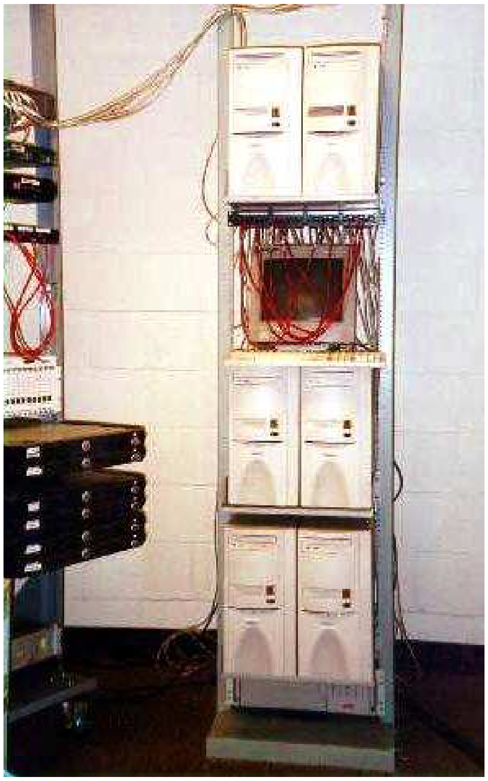


Figure 6.1: **The testbed used for the experiments.** Our testbed consists of 6 PC-routers and Fast Ethernet hardware.

Name	eth1	eth2	eth3	eth4
PC1	N/A	N/A	N/A	10.0.4.2
PC2	10.0.23.2	N/A	10.0.3.2	10.0.4.1
PC3	10.0.21.2	10.0.2.2	10.0.3.1	10.0.20.1
PC4	10.0.23.1	10.0.22.2	N/A	10.0.20.2
PC5	N/A	10.0.2.1	N/A	N/A
PC6	10.0.21.1	10.0.22.1	N/A	N/A

Table 6.1: **IP addresses used in the Indra testbed.** The interfaces of *Monitor* do not need to be assigned IP addresses to monitor the traffic on a link.

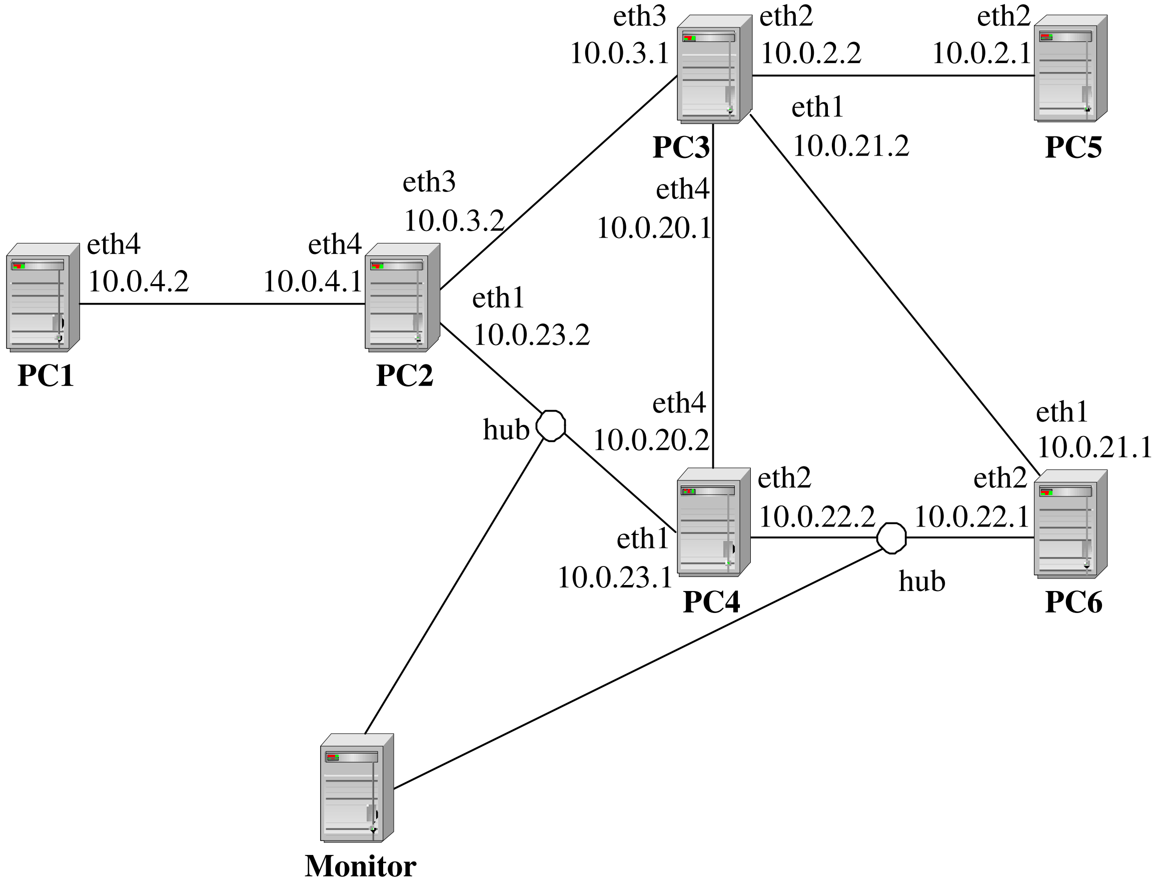


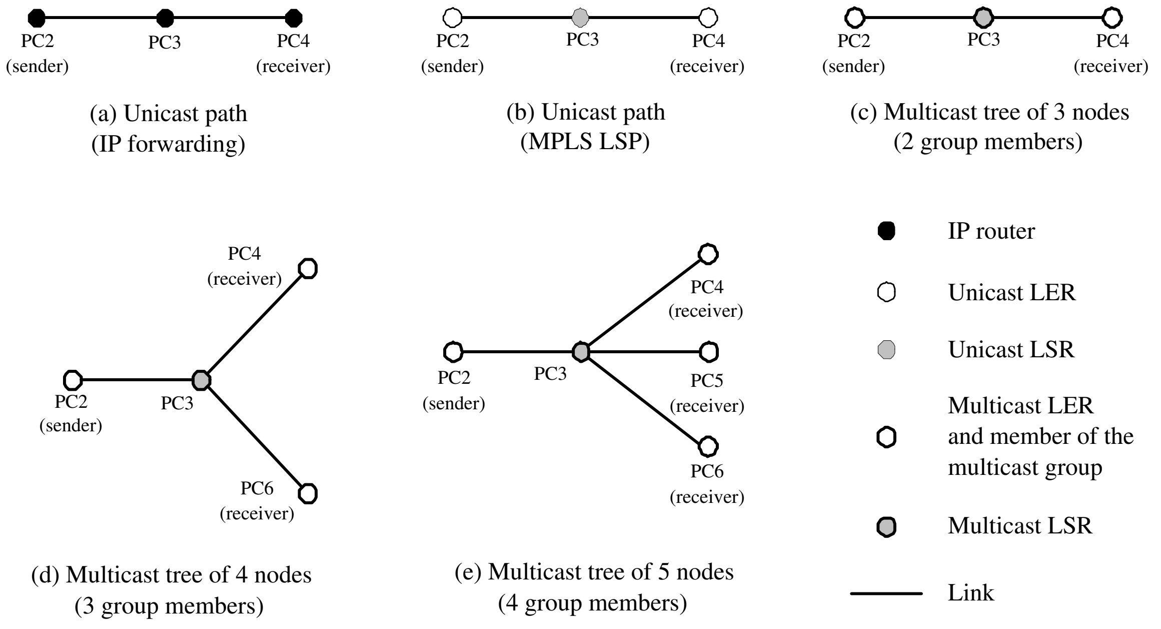
Figure 6.2: **Setup of the network used during the experiments.** *Monitor* does not participate actively in the experiments and only monitors traffic on the link between *PC2* and *PC4* and on the link between *PC4* and *PC6*.

access. This interface is not used during the experiments and is not represented in Figure 6.2. All other interfaces are not connected to the Internet. On each of the six PCs (*PC1*, *PC2*, *PC3*, *PC4*, *PC5*, *PC6*), we use up to four of the other interfaces (*eth1*, *eth2*, *eth3* and *eth4*) to conduct our experiments. All links are full-duplex point-to-point 100BaseTX Ethernet cables. Table 6.1 shows the IP addresses of the interfaces.

In addition to the six PCs of the testbed, we use a seventh machine, "*Monitor*", for measurement purposes. *Monitor* has three interfaces. The first interface is connected to the Internet and is not used in the experiments. The second interface is used to capture the traffic on the link between *PC2* and *PC4* while the third interface is used to capture the traffic between *PC4* and *PC6*. *Monitor* is connected to the link between *PC2* and *PC4* and to the link between *PC4* and *PC6* via Fast Ethernet hubs. Therefore, when *PC2* sends a frame to *PC4*, this frame is received by both *Monitor* and *PC4*. Since *Monitor* only captures traffic on the wires of the network, we do not assign IP addresses to the second and third interfaces. *Monitor* only has a passive role and the presence of *Monitor* in the network is transparent to the experiments.

6.2 Experiment 1: Measuring MPLS multicast throughput

The goal of the first set of experiments is to evaluate the throughput achievable by our multicast MPLS forwarding engine and compare it with the data throughput on a unicast path. In each experiment, a source is sending data to one or several receivers



(a) Unicast path
(IP forwarding)

(b) Unicast path
(MPLS LSP)

(c) Multicast tree of 3 nodes
(2 group members)

(d) Multicast tree of 4 nodes
(3 group members)

(e) Multicast tree of 5 nodes
(4 group members)

Figure 6.3: **Topology of the network used to measure the performance of the MPLS-Linux multicast implementation.** We compare the throughput achieved with IP routing, MPLS unicast, MPLS multicast in groups of two to four members.

using a modified version of the ttcp tool [52] [66] that supports multicast traffic. The ttcp tool is a traffic generation tool in which it is possible to choose the size, the total amount of data sent and the packet sending rate of the generated traffic. At the sender, we configure the ttcp tool to send 100,000 UDP packets of 8192 bytes at the maximum speed offered by the hardware. Since we use UDP packets to measure throughputs, packets may be dropped between the source and the receivers and the receivers do not necessarily receive every packet. The throughput seen by a receiver is the amount of data received by the receiver divided by the time taken to receive the data. A program runs on each receiver and computes the throughput seen by the receiver. We run each experiment five times, thus for each experiment we collect five values for the throughput seen by each receiver.

In Experiment 1.1.1 (see Figure 6.3(a)), the source *PC2* sends IP packets to the receiver *PC4*. All three PCs involved in the experiments, *PC2*, *PC3* and *PC4*, exclusively use IP forwarding mechanisms and MPLS forwarding is disabled. In Experiment 1.1.2 (see Figure 6.3(b)), we set up a unicast LSP between the source *PC2* and the receiver *PC4*. *PC2* is setup as an ingress LER, *PC3* is a LSR and *PC4* is an egress LER. In Experiments 1.1.3 to 1.1.8, we set up multicast LSPs for multicast routing trees with two (Figure 6.3(c)), three (Figure 6.3(d)) and four (Figure 6.3(e)) group members. The core of the multicast routing tree is *PC3*, and *PC2* is the sender. In Experiments 1.1.3 and 1.1.6, *PC4* is the only receiver. In Experiments 1.1.4 and 1.1.7, both *PC4* and *PC5* are receivers and in Experiments 1.1.5 and 1.1.8, *PC4*, *PC5* and *PC6* are receivers. We perform Experiments 1.1.1 and 1.1.2 with the probing mechanism that detects link failures and repairs deactivated. We perform the experiments that involve MPLS multicast with the probing mechanism deactivated (Experiments 1.1.3, 1.1.4 and 1.1.5), and then with the probing mechanism activated (Experiments 1.1.6, 1.1.7 and 1.1.8).

	Exp. number	Description	Throughput th_a^1 (Standard variation) Mbits/s	Difference to MPLS Unicast ($rdiff_a^1$)
	Probing mechanism deactivated			
	1.1.1	IP unicast	93.552 (0.0039)	+0.279 %
	1.1.2	MPLS Unicast	th_0^1 =93.292 (0.0001)	(reference)
MPLS multicast	1.1.3	Group of 2 members	93.286 (0.0106)	-0.006 %
	1.1.4	Group of 3 members:		
		PC4	93.291 (0.0017)	-0.001 %
		PC5	93.292 (0.0013)	-0.001 %
	1.1.5	Group of 4 members:		
		PC4	93.270 (0.0258)	-0.024 %
		PC5	93.273 (0.0222)	-0.021 %
		PC6	93.282 (0.0174)	-0.012 %
	Probing mechanism activated, T_p=10 ms			
MPLS multicast	1.1.6	Group of 2 members	93.215 (0.0090)	-0.083 %
	1.1.7	Group of 3 members:		
		PC4	93.215 (0.0011)	-0.081 %
		PC5	93.217 (0.0008)	-0.081 %
	1.1.8	Group of 4 members:		
		PC4	92.034 (0.2169)	-1.35 %
		PC5	92.108 (0.1635)	-1.35 %
		PC6	92.108 (0.1729)	-1.27 %

Table 6.2: **Multicast MPLS forwarding engine performance with UDP packets of 8192 bytes.** Multicast MPLS achieves throughputs comparable with MPLS unicast and IP unicast. The probing mechanism has little influence on the maximum throughputs achieved. The size of the group has a limited impact on the performance of multicast MPLS.

Consider Table 6.2. The first and second columns contain the experiment number and a short description for each experiment. Each experiment consists of five runs. In the third column, we give the average throughput seen by each receiver for the five runs of each experiment, and the standard deviation for the set of the five values of the throughput. In the fourth column, we give the relative difference $rdiff_a^1$ between the throughput th_a^1 seen by a receiver a and the throughput achieved with MPLS unicast when the probing mechanism is not activated th_0^1; therefore, $rdiff_a^1 = \frac{th_a^1 - th_0^1}{th_0^1}$.

First, we notice that IP unicast is faster than MPLS (Experiments 1.1.1 and 1.1.2). For each incoming packet, an ingress LER performs a lookup in the IP routing table to find the FTN of the packet and then a lookup in the MPLS output table to find the NHLFE pointed by the FTN. Then the LER pushes a label according to the information contained in the NHLFE. With IP routing only one lookup is performed thus IP routing is faster than MPLS forwarding. Nevertheless, routers perform these lookups fast and the difference between the throughputs achieved by IP and MPLS unicast is small (0.279 %).

Second, in all experiments, the throughput for all multicast group members is the same. For example, in the experiments with the group of four members (Experiments 1.1.5 and 1.1.8), the throughputs at *PC4*, *PC5* and *PC6* are the same (93.3 Mbits/s when the probing mechanism is deactivated, 92.1 Mbits/s when the probing mechanism is activated). However, the throughput decreases with the number of group members. In the group of four members, when the probing mechanism is activated, the throughput is 1.3 % lower than when IP unicast is used (Experiment 1.1.8), while this throughput loss is only 0.8 % in groups of two or three members (Experiments 1.1.6 and 1.1.7). Packet duplication is a time consuming operation and has a negative impact on the throughput on the network. We could not test the performance of the duplication mechanism for a larger number of duplications.

Third, the probing mechanism has a limited impact on the maximum throughput in the network (Experiments 1.1.3 and 1.1.6, 1.1.4 and 1.1.7, 1.1.5 and 1.1.8). Throughputs in the network are lower when the probing mechanism is activated. The throughput decrease due to the probing mechanism reaches 1.3 % (Experiment 1.1.8). The probes consume bandwidth on the links and this bandwidth is not available for the data that the sender has to transmit.

In summary, adding the multicast capability to the routers has a limited impact on the maximum throughput of the network when using UDP packets of 8192 bytes.

Now, we repeat the experiments but set the UDP packet size to 1024 (instead of 8192 bytes) in Experiments 1.2.1 to 1.2.8. Consider Table 6.3. In the third column, we give the average throughput seen by each receiver for the five runs of each experiment, and the standard deviation for the set of the five values of the throughput. In the fourth column, we give the relative difference $rdiff_a^2$ between the throughput th_a^2 seen

	Exp.	Description	Throughput th_a^2	Difference
	number		(Standard variation)	to MPLS
			Mbits/s	Unicast ($rdiff_a^2$)
	Probing mechanism deactivated			
	1.2.1	IP Unicast	91.694 (0.0025)	+0.64 %
	1.2.2	MPLS Unicast	th_0^2 =91.114 (0.2217)	(reference)
MPLS multicast	1.2.3	Group of 2 members	88.809 (0.2215)	-2.53 %
	1.2.4	Group of 3 members:		
		PC4	91.215 (4.2001)	+0.11%
		PC5	88.089 (3.1458)	-3.32%
	1.2.5	Group of 4 members:		
		PC4	91.237 (0.3782)	+0.13 %
		PC5	90.478 (0.1635)	-4.47 %
		PC6	87.044 (0.2680)	-0.70 %
	Probing mechanism activated, T_p=10 ms			
MPLS multicast	1.2.6	Group of 2 members	80.609 (1.4546)	-11.53 %
	1.2.7	Group of 3 members:		
		PC4	84.640 (0.2974)	-7.11 %
		PC5	82.120 (0.4136)	-9.87 %
	1.2.8	Group of 4 members:		
		PC4	81.327 (0.2583)	-10.74 %
		PC5	80.450 (0.2797)	-11.70 %
		PC6	84.321 (0.6687)	-7.46 %

Table 6.3: **Multicast MPLS forwarding engine performance with UDP packets of 1024 bytes.** The limits of the hardware are reached and the performance of multicast MPLS is severely degraded when the traffic consists of small packets.

by a receiver a and the throughput achieved with MPLS unicast when the probing mechanism is activated th_0^2; therefore, $rdiff_a^2 = \frac{th_a^2 - th_0^2}{th_0^2}$.

The throughputs achieved in the network are lower than with 8192-byte packets. With IP unicast and MPLS unicast (Experiments 1.2.1 and 1.2.2) the throughput is now 91 Mbits/s. With MPLS multicast, the throughput is comprised between 80 Mbits/s (Experiment 1.2.8) and 91 Mbits/s (Experiments 1.2.3 and 1.2.4). With smaller packets, the number of packets that routers need to process each second is larger than with large packets, thus decreasing the performance of the routers. In this experiment, we reach the processing capacity limit of the PC routers. Our PC routers are not able to forward UDP packets of 1024 bytes at the maximum speed allowed by the network hardware. In the following experiments, we use only 8192-byte UDP packets in order to use the full capacity of the links and not overload the CPUs of the PC routers.

6.3 Experiment 2: Measuring link failure and recovery detection times

In the second experiment, we determine the link failure and recovery detection times and compare the experimental values with the values from the analytical model presented in Section 4.2. The setup for this experiment is depicted in Figure 6.4. In this

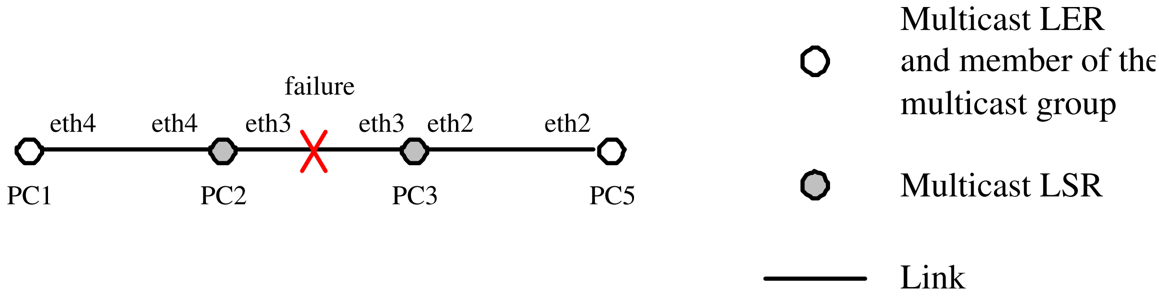


Figure 6.4: **Experimental setup for determining the link failure and recovery detection times.** No traffic flows on the tree. Once the mLSP is established, we successively simulate the failure and recovery of the link between *PC2* and *PC3* by bringing down and up the interface *eth3* of *PC2*.

experiment, we set up a multicast LSP but do not transmit data over the LSP. The core of the tree is *PC2*. The members of the multicast group are *PC1* and *PC5*. On each machine involved in the experiment, we set the beat checking number n to the minimum value $n = 2$ as defined in Section 4.2. In the Linux operating system, the most accurate timer has a resolution of 10 ms [15]. We use this resolution of 10 ms for the period T_p. The 10 ms timer is accurate when the machine is underloaded, however since Linux is not a real-time operating system, the accuracy becomes questionable when the system is overloaded [15] [45]. In this experiment, the PC routers are underloaded and we assume that the timer is accurate. When an interface is disabled, the kernel considers that no link is attached to the interface therefore disabling (or bringing down) an interface is equivalent to cutting the link attached to the interface. Reenabling (bringing up) an interface is equivalent to repairing the link attached to the interface.

We modify the code of MulTreeLDP on *PC2* to measure $T_{fdetect}$ and $T_{rdetect}$. We add a thread to MulTreeLDP that automatically brings down and brings up *eth3*. MulTreeLDP records in a timestamp the instant at which it brings *eth3* up. MulTreeLDP computes the difference between the time at which it detects the link failure and the instant at which *eth3* is brought down. This time difference is $T_{fdetect}$. Then, the thread we added to MulTreeLDP chooses a random time value using the internal random number generator of the PC and sleeps during that time. When the new thread wakes up, it brings up *eth3* and records in a timestamp the instant at which it brings *eth3* up. When the link repair is detected, *PC2* computes the time difference between the instant of the repair detection and the aforementioned timestamp. This difference is $T_{rdetect}$. The new thread successively brings down and brings up *eth3* 100 times and then quits. We record the 100 values for $T_{fdetect}$ and $T_{rdetect}$, stop the MulTreeLDP program and restart it on all four machines used in this experiment. We collect 25 series of 100 values for both $T_{fdetect}$ and $T_{rdetect}$. Therefore, we collect 2500 values for $T_{fdetect}$ and 2500 values for $T_{rdetect}$. According to our model presented in Section 4.2, the time to detect a link failure depends on two factors. The first factor is the length of the time interval between the instant at which *PC3* sends the last probe before the failure occurs and the instant at which the failure occurs. We called this time T_1 in Section 4.2. We randomize T_1 by bringing up and bringing down interface *eth3* at random times. The second factor is related to the synchronization between the timers on *PC2* and *PC3*. In Section 4.2, we called T_2 the difference of synchronization of the timers of *PC2* and *PC3*. We assume that manually stopping and restarting MulTreeLDP on all machines randomizes T_2.

In Figure 6.5, we show the distribution of the 2500 samples of $T_{fdetect}$ for 2 ms long time intervals, and compare this experimental distribution with the expected distribution derived from the analytical model in Section 4.2. The average for the 2500 samples of $T_{fdetect}$ is $\overline{T}_{fdetect}$=25.4 ms. With $n = 2$ and T_p=10 ms the theoretical average is $\frac{3n-1}{2}T_p = 25$ ms. Although the model we discuss in Section 4.2 is simple, our experimental results match the theoretical values determined with the model.

In Figure 6.6, we show the distribution of the 2500 samples of $T_{rdetect}$ for 2 ms long time intervals and compare this distribution with the expected distribution derived from our model. Here, the experimental results do not match the model well. We expect 10 % of the recovery detection times to be comprised between 0 and 10 ms and 0 % above 10 ms, but only 6.1 % of the samples are comprised between 0 and 10 ms and more than 3 % of the values are higher than 10 ms. Actually, the experimental recovery detection times are not comprised between 0 and 10 ms but between 0.4 and 10.4 ms, as shown in Figure 6.7. The average for the 2500 samples of $T_{rdetect}$ is $\overline{T}_{rdetect}$=5.48 ms, which is close to the theoretical average (5 ms).

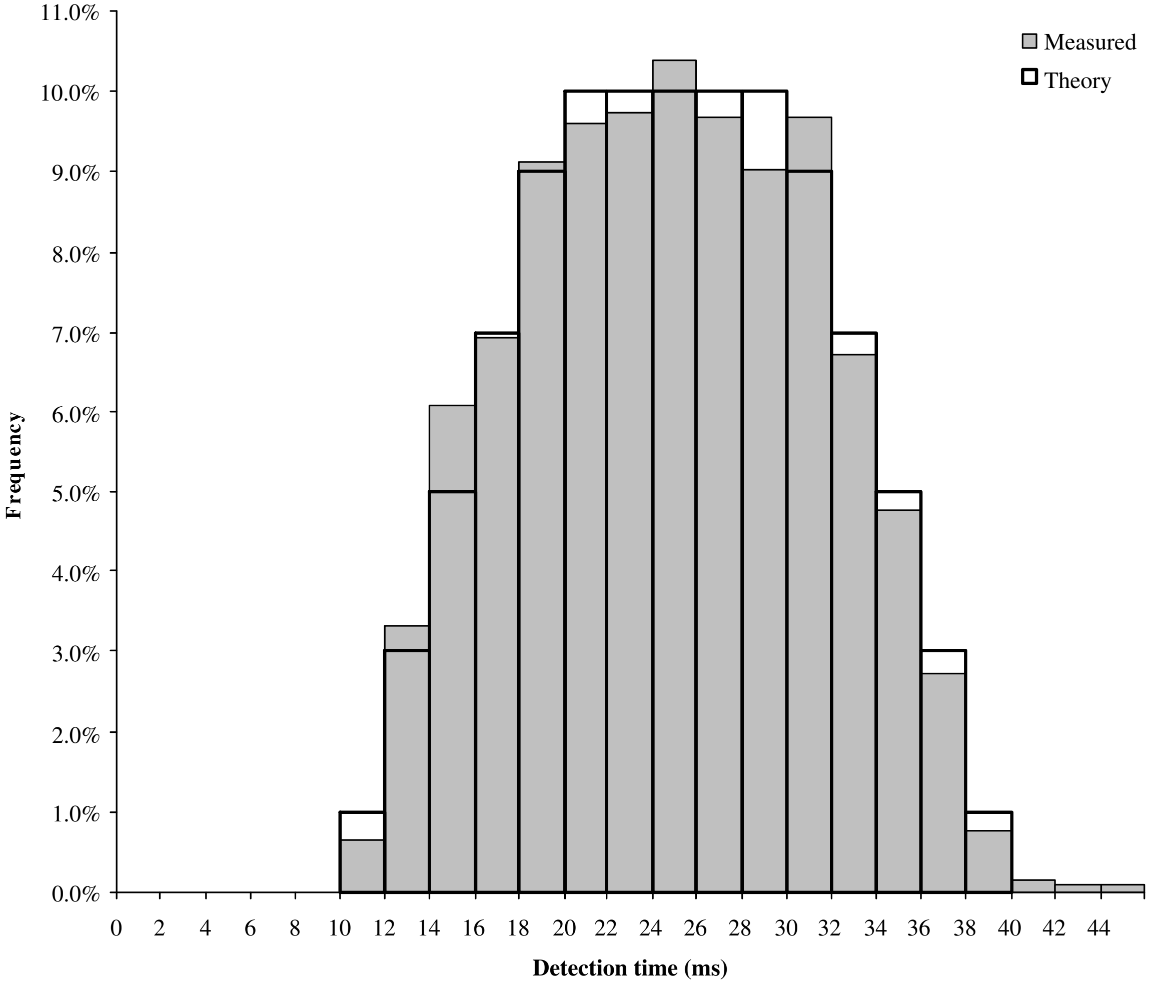


Figure 6.5: **Experimental distribution of the link failure detection time.** In bold lines, the theoretical distribution for time intervals of 2 ms.

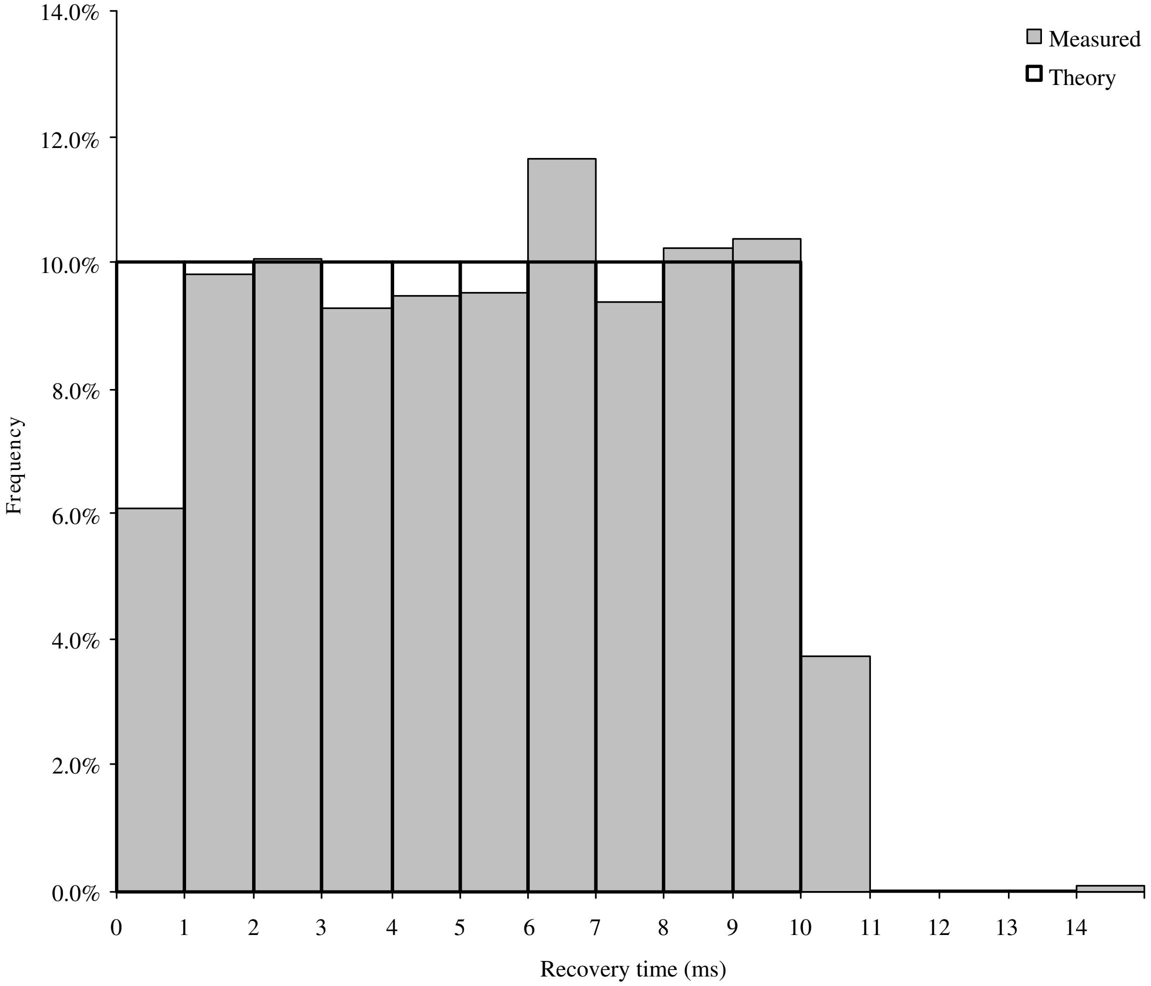


Figure 6.6: **Experimental distribution of the link recovery detection time.** In bold lines, the theoretical distribution for time intervals of 1 ms.

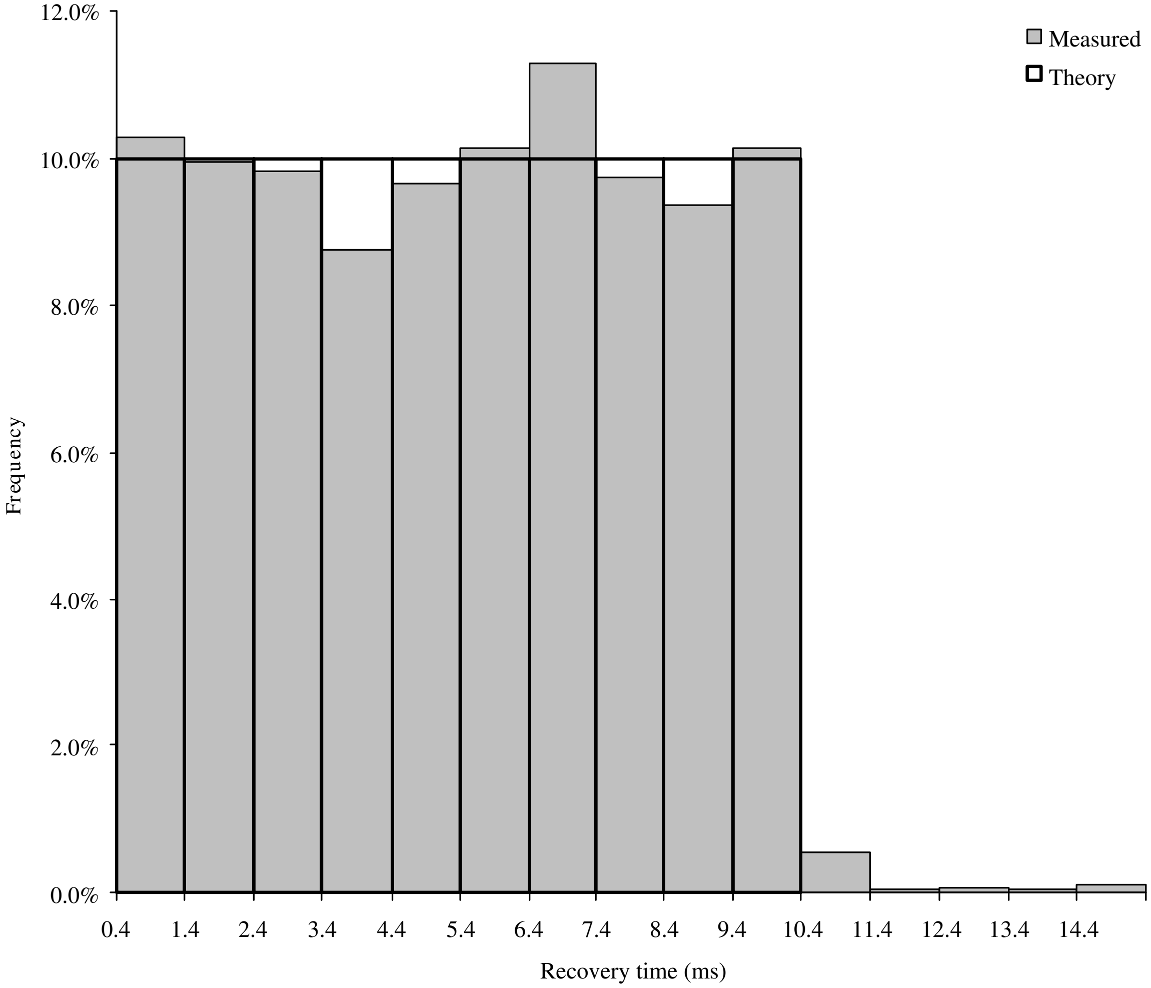


Figure 6.7: **Experimental distribution of the link recovery detection time compared with the theoretical distribution shifted by 0.4 ms.** The experimental distribution matches the theoretical distribution when we add 0.4 ms to the time intervals in the model. There is a difference of 0.4 ms between the experimental and theoretical distributions.

We conduct additional experiments to assess the behavior of the link failure detection mechanism for high link capacity utilization. We modify the setup of the experiment such that *PC1* sends traffic to *PC5* using the multicast LSP. The traffic consists of UDP packets of 8192 bytes. When we set the sending rates at 93 Mbits/s or more, we observe that *PC2* and *PC3* make false detections, i.e they detect that the link between *PC2* and *PC6* successively fails and is repaired several times per second. The PC routers are not fast enough to forward the packets and send or check the reception of the probes at the same time. As discussed earlier, Linux is not a real-time operating system therefore there is no guarantee that probes are sent exactly every T_p ms or that probe reception is checked exactly every n T_p ms under high load of the system. Solutions to this issue include increasing n or T_p (at the cost of higher link failure and detection times), using a real-time operating system, using faster routers, or using a fraction of the maximum throughput achievable with MPLS multicast to send traffic. In the remaining experiments, we send traffic at lower rates to avoid false detections.

6.4 Experiment 3: Measuring link failure and recovery notification times

In this experiment, we determine the time to notify a PSL of a link failure or link recovery by measuring the node notification delay T_{nnotif}. Indeed, the time to notify the PSLs after a link failure or recovery has been detected is proportional to the time T_{nnotif} to propagate the notification message between two nodes as explained in Section 4.3.

The setup of the experiment is illustrated in Figure 6.8. We set up a multicast LSP of six nodes. *PC1*, *PC5* and *PC6* are the LERs of the tree. *PC2*, *PC3* and *PC4* are LSRs. The link between *PC3* and *PC4* is the unique link of the backup path. The PSLs are *PC3* and *PC4*. The links between *PC3* and *PC2*, and *PC2* and *PC4* are the links of the protected path. *PC1* is a source and sends UDP packets of 8192 bytes at 40 Mbits/s on the tree. The receivers are *PC5* and *PC6*. *Monitor* captures traffic on the link between *PC2* and *PC4* and on the link between *PC4* and *PC6* using *tcpdump*.

To measure the node notification delay T_{nnotif}, we simulate the failure and recovery of the link between *PC3* and *PC2*. When we bring down interface *eth3* of *PC2*, *PC2* and *PC3* detect a link failure. *PC2* sends a link failure message to *PC4*. Upon reception of this link failure message, *PC4* sends a link failure message to *PC6*. We use *Monitor* to measure T_{nnotif}. Since *Monitor* captures traffic on the link between *PC2* and *PC4* and on the link between *PC4* and *PC6*, *Monitor* receives both notification

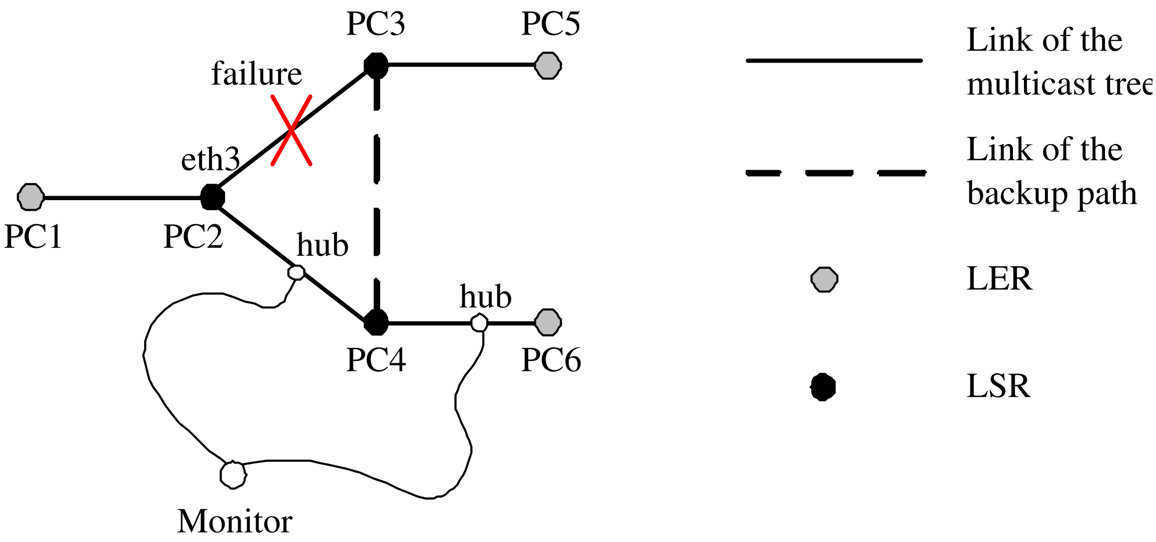


Figure 6.8: **Experimental setup for determining the link failure and recovery notification delays.** No traffic flows on the tree. Once the mLSP is established, we successively simulate the failure and recovery of the link between *PC2* and *PC3* by bringing down and bringing up the interface *eth3* of *PC2*. *Monitor* is passive and only monitors traffic on the two links to which is attached.

messages and is able to compute the time difference between the instants at which it receives each message. When we bring up interface *eth3* of *PC2*, *PC2* and *PC3* detect the link recovery. *PC2* sends a links recovery notification message to *PC4* and then *PC4* sends a link recovery message to *PC6*. *Monitor* records the time at which it captures the notification message on the link between *PC2* and *PC4*, the time at which it captures the notification message on the link between *PC4* and *PC6*, and determines T_{nnotif} for the link recovery by computing the difference between the two recorded times. To bring up and bring down interface *eth3* of *PC2*, we use the additional thread in MulTreeLDP we introduced in Section 6.3. This thread brings down and brings up *eth3* at instants randomly chosen by the random number generator of the machine. After the interface is brought down and brought up 100 times, we stop and restart MulTreeLDP manually on all six machines. We repeat the experiment 25 times. Therefore we collect 25 series of 200 node notification delays.

Figure 6.9 shows how the 5000 samples are distributed. The average for T_{nnotif} is $\overline{T}_{nnotif}$=1.18 ms with a minimum of 0.127 ms, a maximum of 4.172 ms and a standard deviation of 0.37 ms. The notification time is a multiple of T_{nnotif}. In our experiment, there is at most one hop between the routers which detects the link failure and the PSLs thus $l = 1$ and $T_{notif} = T_{nnotif}$. Therefore the average time to notify PSL *PC4* of the link failure is 1.2 ms. With larger networks and larger trees, the value of l is larger and the notification time can reach a few tens of milliseconds. It is possible to decrease T_{nnotif} by opening a TCP connection between every two neighbors in a multicast LSP at circuit establishment and closing the connection when the LSP is torn down instead of opening a new TCP connection each time a MulTreeLDP sends a message. This optimization is left for future work.

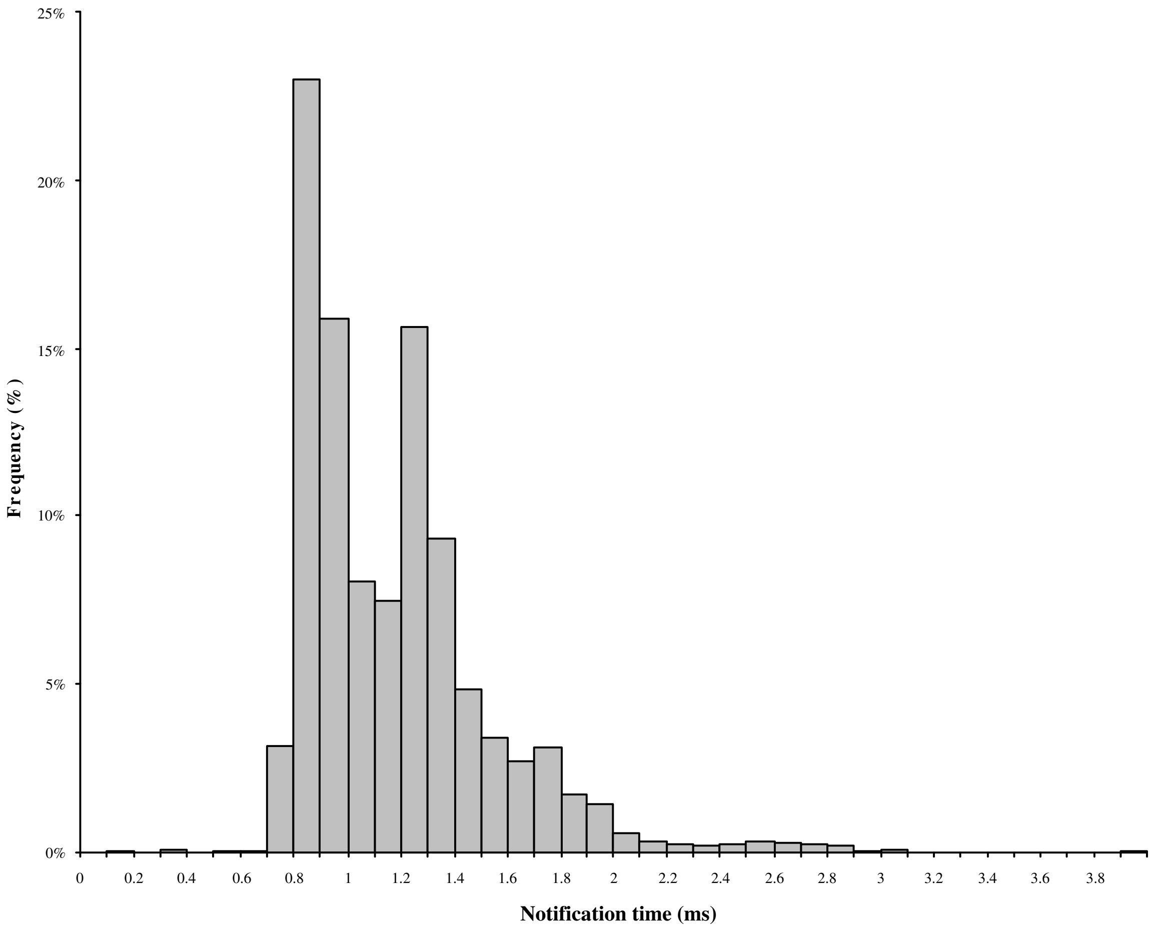


Figure 6.9: **Experimental distribution of the node notification delay.**

6.5 Experiment 4: Measuring the tree repair time

In this experiment, we measure the time T_{repair} to repair a tree on a link failure and study the behavior of the MPLS multicast Fast Reroute mechanism when a failed link is physically repaired. For each experiment, we present typical graphs to illustrate our discussion.

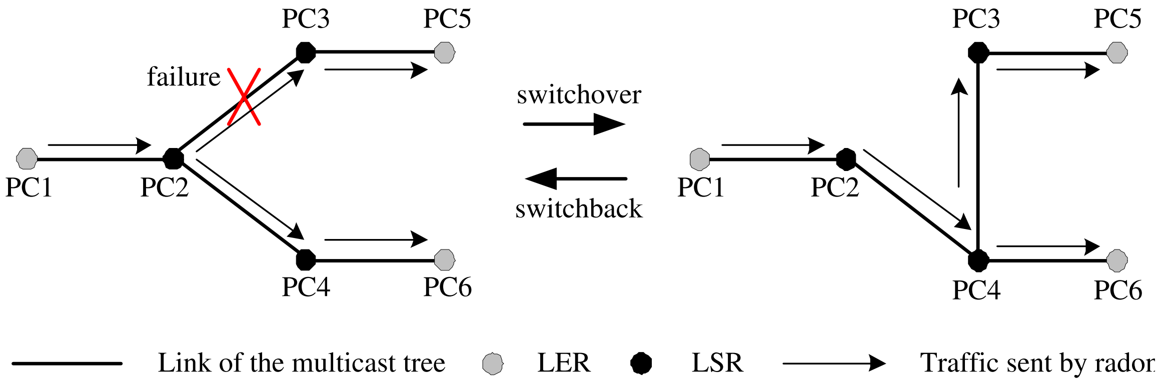


Figure 6.10: **Switchover and switchback in the multicast routing tree set up on our testbed.** *PC1* sends traffic over the tree. When the link between *PC2* and *PC3* fails, *PC4* reroutes traffic over the backup path between *PC4* and *PC3*. When the link is repaired, *PC4* stops forwarding traffic on the backup path.

6.5.1 Experiment 4.1: Measuring the service interruption time due to a link failure

In this experiment, we determine the distribution of the service interruption time due to a link failure. The interruption time is the time to repair the tree including propagation delays (see Section 2.1). We keep the setup from Section 6.4 and do not use *Monitor* (see Figure 6.10). We set up a multicast LSP of six nodes. *PC1*, *PC5* and *PC6* are the LERs of the tree. *PC2*, *PC3* and *PC4* are LSRs. The link between *PC3* and *PC4* is the unique link of the backup path. The PSLs are *PC3* and *PC4*.

The links between *PC3* and *PC2*, and *PC2* and *PC4* are the links of the protected path. *PC1* is a source and sends UDP packets of 8192 bytes at 40 Mbits/s on the tree. The receivers are *PC5* and *PC6*. We simulate the failure and repair of the link between *PC2* and *PC3* by bringing down and bringing up interface *eth3* of *PC2*. To bring up and bring down interface *eth3* of *PC2*, we use the additional thread in MulTreeLDP we introduced in Section 6.3. This thread brings down and brings up *eth3* at instants randomly chosen by the random number generator of the machine. After the interface is brought down and brought up 100 times, we stop and restart MulTreeLDP manually on all six machines. We repeat the experiment 25 times to collect 2500 values of the repair time.

When we simulate the link failure, *PC5* stops receiving traffic. *PC2* detects the link failure and notifies *PC4* of the failure. When *PC4* is notified of the failure, it switches traffic over the backup path and *PC5* resumes receiving the traffic sent by *PC1* (see Figure 6.10). The repair time is the time during which PC1 receives no packet. We measure the repair time as follows. On *PC5*, we record the arrival time of each packet sent by *PC1*. PC1 sends one packet roughly every 1.5 ms therefore the packet interarrival time at *PC5* is 1.5 ms before a link failure. We compute the interarrival time for any two packets successively received at *PC5*. According to Section 6.3, the minimum amount of time to detect a link failure is 10 ms, which is much larger than the packet interarrival time before link failure. Therefore, we consider that every interarrival time longer than 10 ms is a service interruption time due to a link failure. Since we are not able to distinguish whether an interarrival time is longer than 10 ms due to a link failure or an external phenomenon not related to our experiments, we collect more than 2500 values for the repair time. Actually we collect 2600 values; we present in Figure 6.11 the distribution for all collected samples.

The average for all samples is $\overline{T}_{repair}$=29.4 ms, with a minimum of 10 ms (by construction of the sample set) and a maximum of 49.6 ms. The standard deviation is 7.1 ms. The average $\overline{T}_{repair}$ is close to the sum $\overline{T}_{fdetect}+1\times\overline{T}_{nnotif}$=25.4+1.2=26.6 ms. Thus, there is a difference of less than 3 ms between the experimental and the analytical averages for the repair time. This difference takes into account the propagation delays and the time for the PCs to modify the MPLS tables. The experimental results show that a network can be repaired in average in less than 50 ms by MPLS multicast Fast Reroute. The main component of the repair time is the detection time. In larger trees, only the notification time increases. Our data shows that our implementation of MPLS multicast Fast Reroute can repair multicast routing trees with a protected path of up to 20 links in less than 50 ms. With a protected path of 20 links, the average repair time is $25.4 + 20 \times 1.2 = 49.4$ ms.

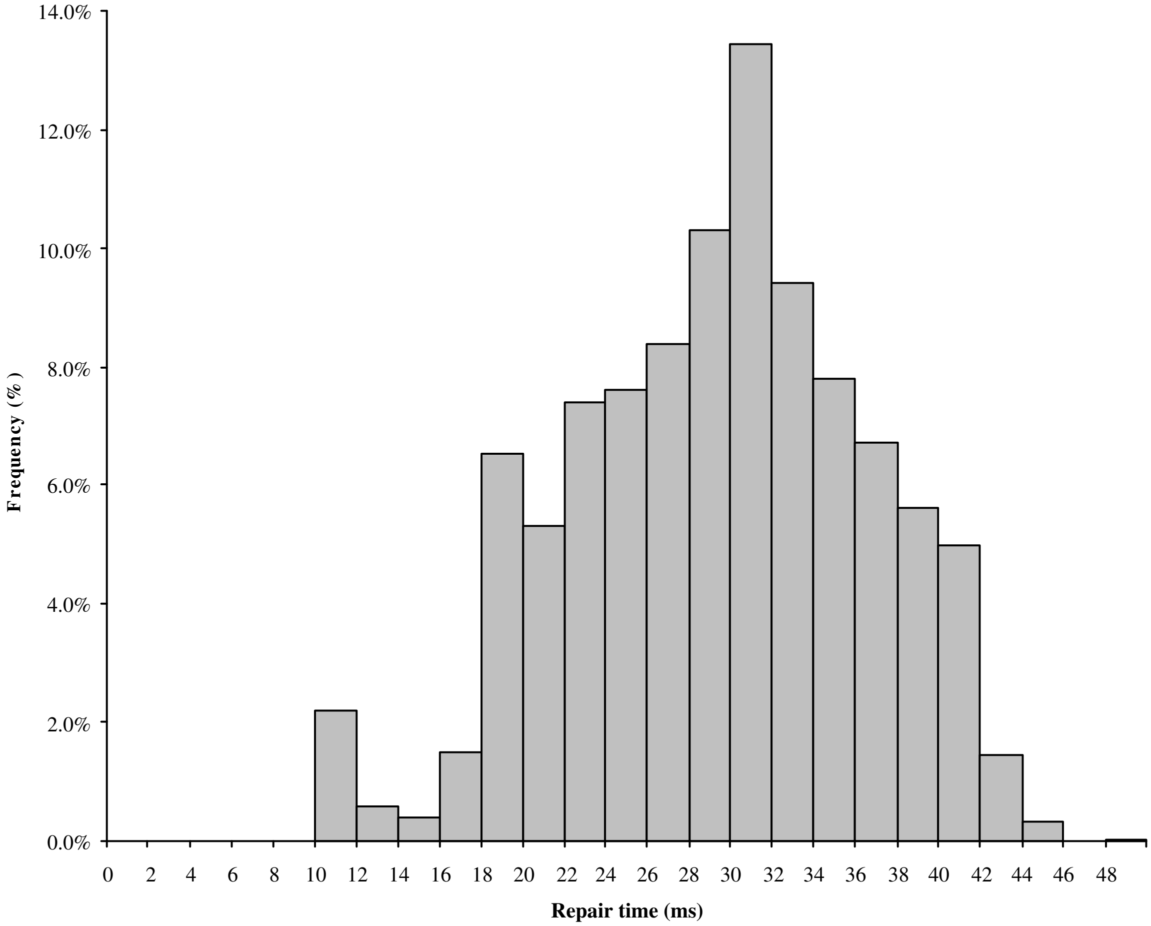


Figure 6.11: **Experimental distribution of the repair time.**

6.5.2 Experiment 4.2: Observing duplicate packets on the tree when a failed link is repaired

We perform additional experiments to show the duplicate packets that flow on the tree on switchback. We do not change the experiment setup. *PC1* starts sending

UDP packets of 8192 bytes at 85 Mbits/s on the tree at time $t = 0$. We simulate the failure of the link between *PC2* and *PC3* at time $t \approx 2.3$ s by bringing down interface *eth3* of *PC2*. We simulate the repair of the link between *PC2* and *PC3* at time $t \approx 4.6$ s by bringing up interface *eth3* of *PC2* and we record the reception time for each packet on *PC5* and *PC6*.

Figures 6.12 and 6.13 depict the evolution of the amount of data received by *PC5* and *PC6* when *PC1* sends data over the multicast routing tree. Figures 6.14, 6.15, 6.16 and 6.17 are enlarged views of Figures 6.12 and 6.13. We observe that *PC5* receives no traffic for 29 ms starting at time $t = 2.321$ s (Figure 6.14). *PC6* receives traffic during the full duration of the experiment.

The link failure occurs at time $t \approx 2.321$. Before *PC 3* and *PC4* have performed switchover, *PC 5* receives no traffic. At time $t = 2.350$ s, switchover is complete and *PC5* receives the traffic from *PC1* via the backup path. The repair time is 29 ms. Since the failure is not located between *PC1* and *PC 6*, *PC6* keeps receiving traffic during the link failure. We expect to observe a slope increase on Figures 6.16 and 6.17 due to packet duplication at switchback time (see Section 4.4), however the slope of the curves slightly decreases during a short time when the link is repaired. The reason why the traffic increase phenomenon is not visible in this experiment lies in the size of the UDP packets we used. Indeed, *PC1* sends packets of 8192 bytes. Ethernet can send frames with a payload of at most 1500 bytes over the links and therefore on *PC1* the IP layer has to segment the UDP packets before passing them to the Ethernet layer. On *PC5* and *PC6* the IP layer reassembles the fragments of the packets received in the Ethernet frames. During the reassembly process the IP layer detects the duplicate frames and discards them. Therefore, although there is a traffic increase on two links of the network, this increase is hidden by the IP layer of the receivers.

To make the duplicate packets on switchback apparent, we conduct an additional experiment. The experiment setup is kept unchanged except that PC1 sends UDP packets of 1024 bytes at 40 Mbits/s. We simulate the failure of the link at $t \approx 2.3$ s and the repair at time $t \approx 4.50$ s. We show the traffic received by *PC5* and *PC6* during the total length of the experiment in Figures 6.18 and 6.19. Figures 6.20 and 6.21 are enlarged views of Figures 6.18 and 6.19.

In Figures 6.20 and 6.21, the slope change due to switchback is visible at time $t \approx 4.50$ s. After the link is repaired and before switchback is complete, *PC5* and *PC6* receive duplicate packets, hence the slope change during approximately 5 ms on each receiver. The slope change lasts only a few milliseconds, making it impossible to distinguish from other curve irregularities via automated means and preventing us from measuring $T_{repairback}$ experimentally.

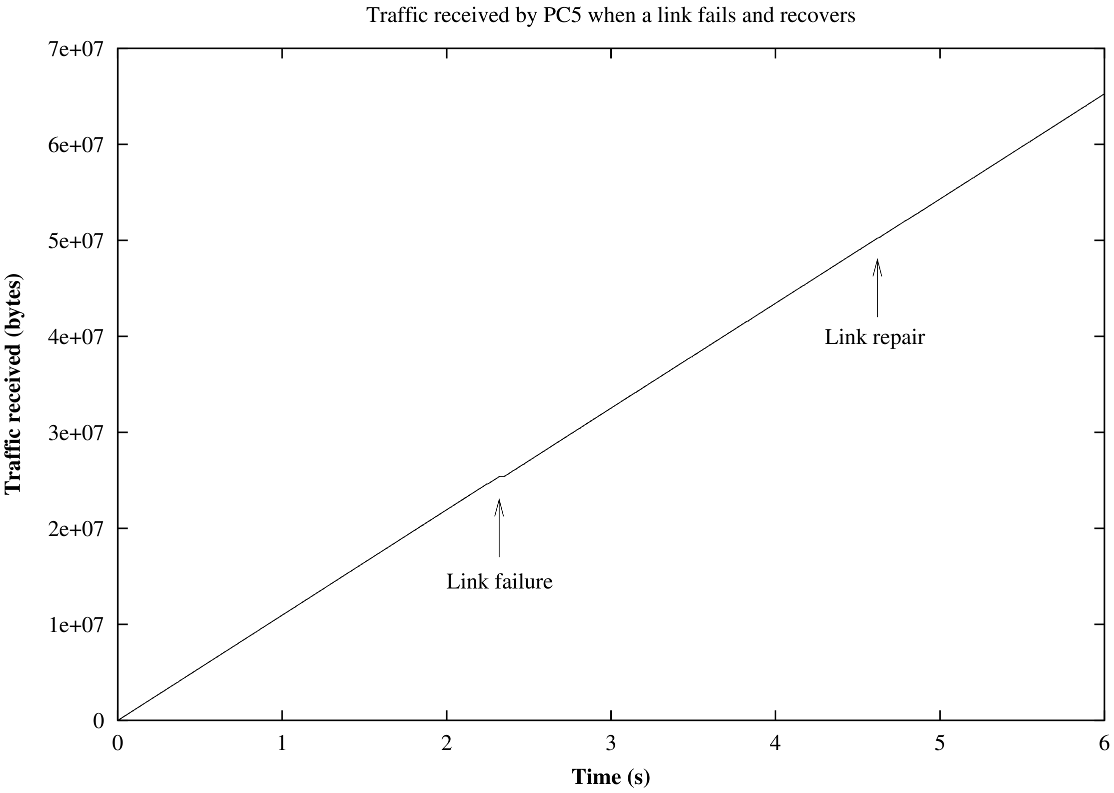


Figure 6.12: **Traffic received by *PC5* when the tree sustains a failure and a recovery (UDP packets of 8192 bytes).** The failure occurs at time $t \approx 2.3$ s and the link is repaired at $t \approx 4.6$ s.

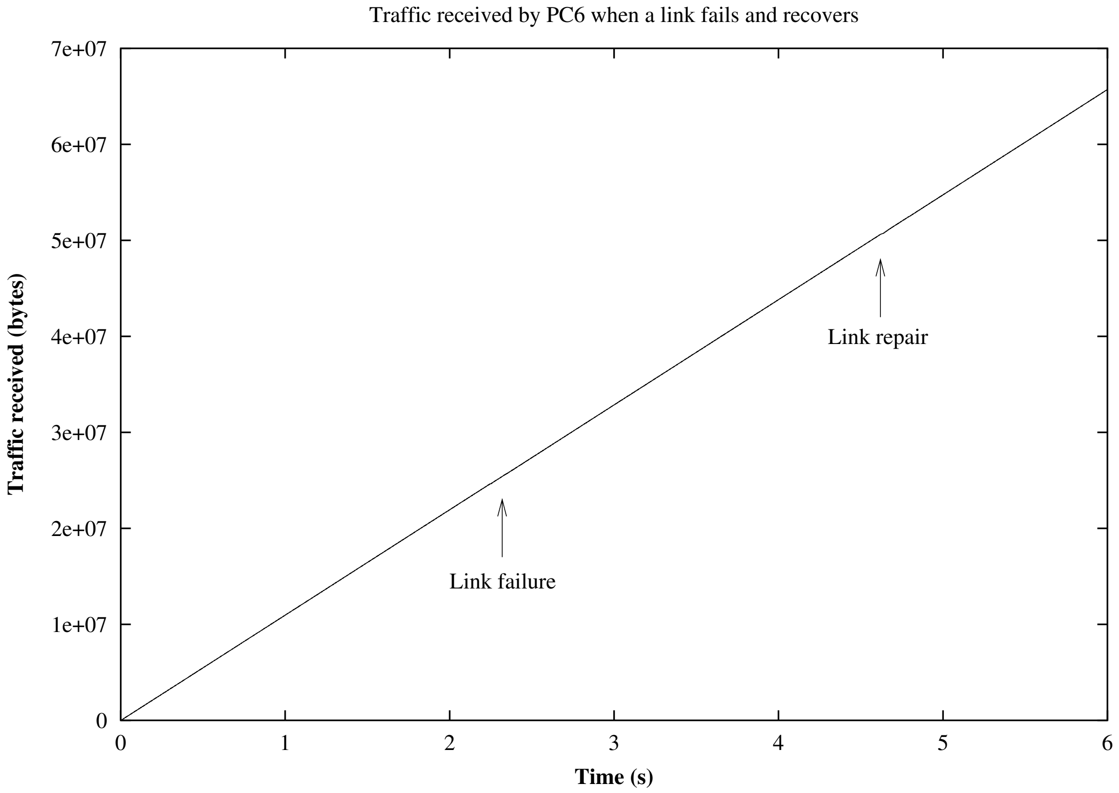


Figure 6.13: **Traffic received by *PC6* when the tree sustains a failure and a recovery (UDP packets of 8192 bytes).** The failure occurs at time $t \approx 2.3$ s and the link is repaired at $t \approx 4.6$ s.

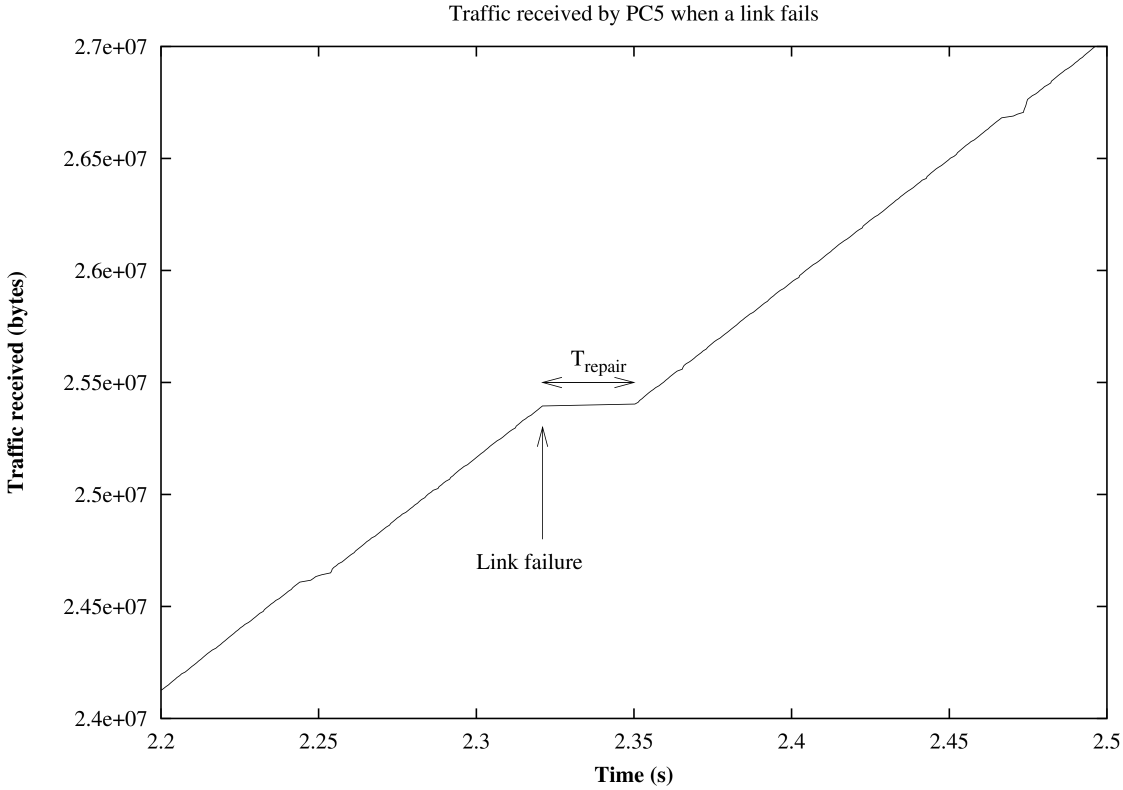


Figure 6.14: **Switchover on *PC5* (packets of 8192 bytes).** *PC5* receives no traffic between $t = 2.321$ s and $t = 2.350$ s. The interruption of service seen by *PC5* is 29 ms.

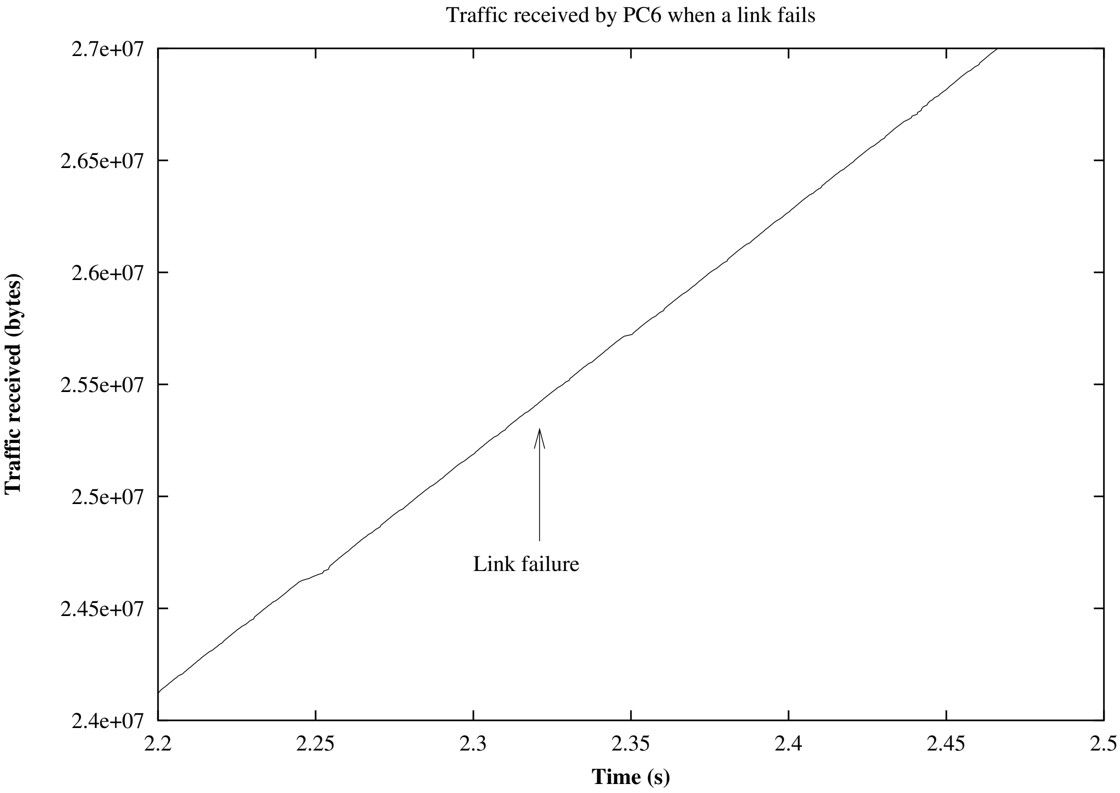


Figure 6.15: **Switchover on *PC6* (packets of 8192 bytes).** Instead of increasing, the slope of the curves slightly decreases on the link failure.

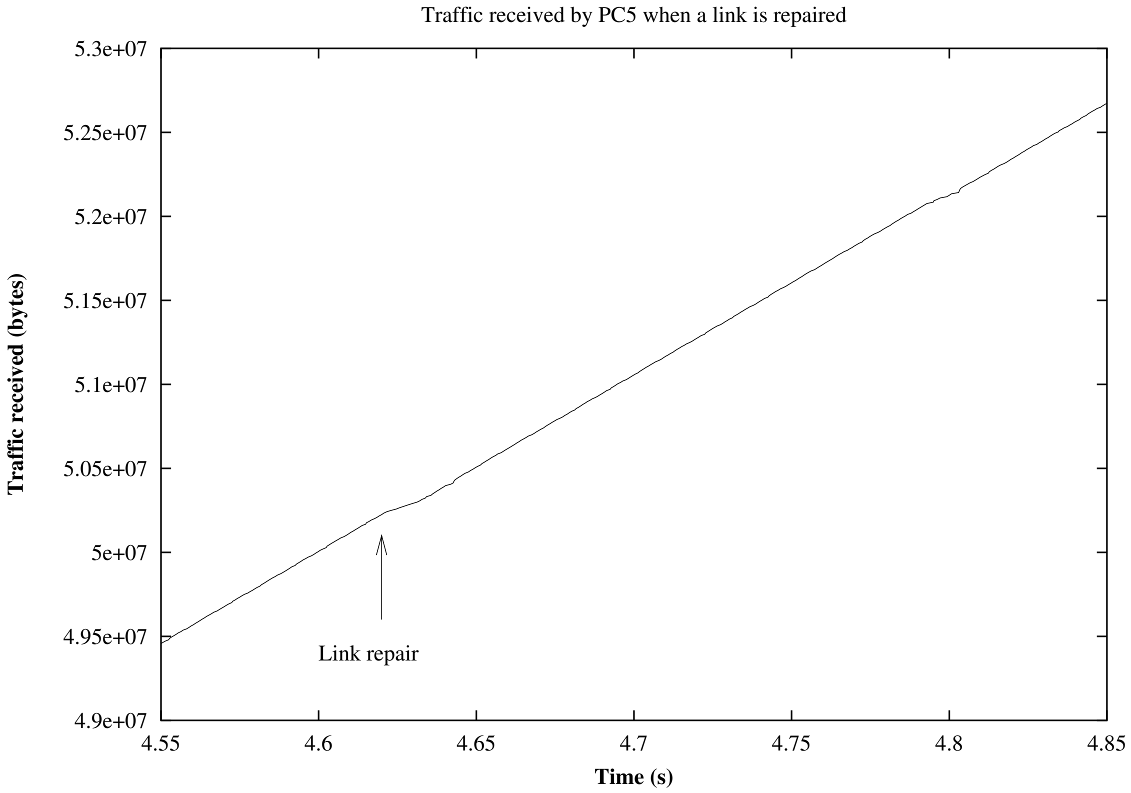


Figure 6.16: **Switchback on *PC5* (packets of 8192 bytes).** Instead of increasing, the slope of the curves slightly decreases on the link failure.

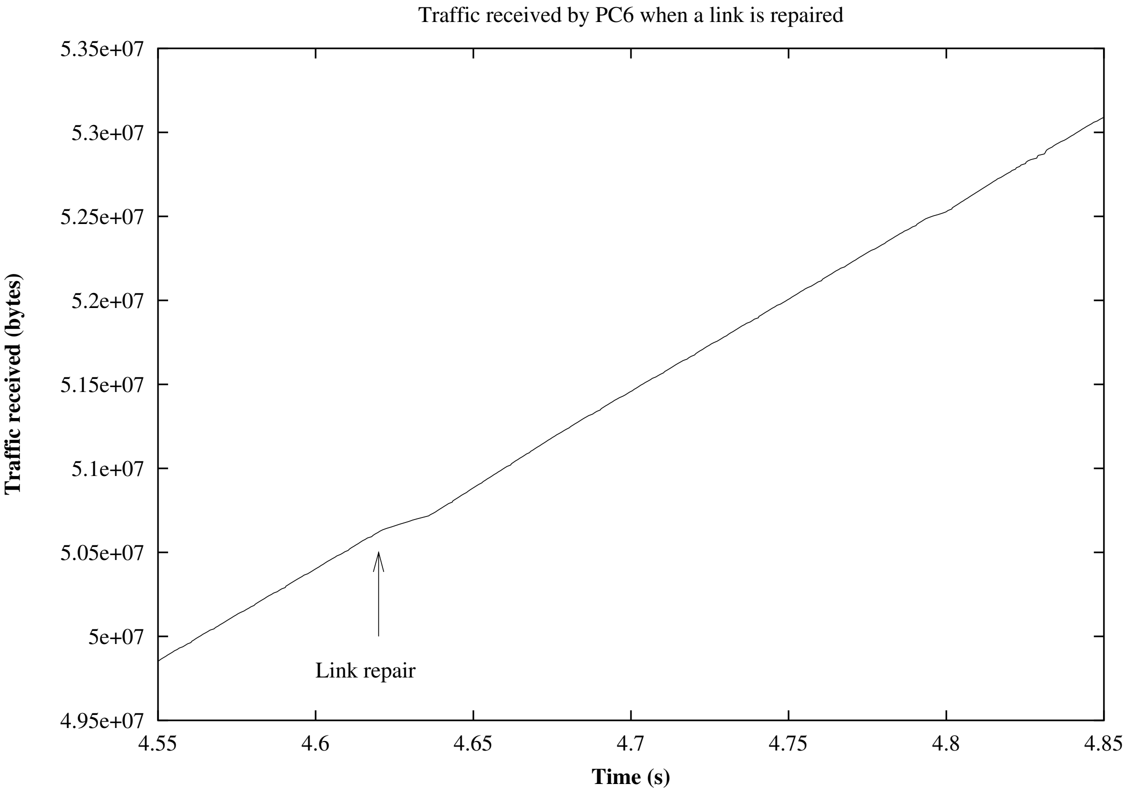


Figure 6.17: **Switchback on *PC6* (packets of 8192 bytes).** *PC6* is not affected by the link recovery.

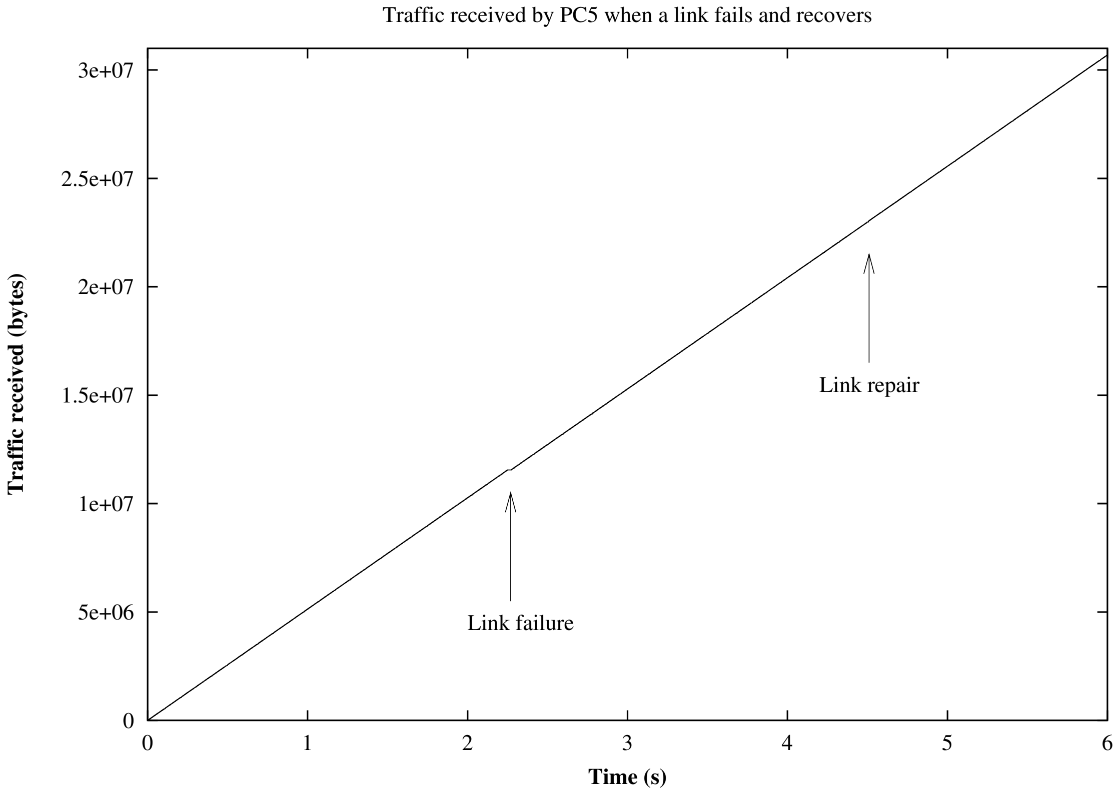


Figure 6.18: **Traffic received by *PC5* when the tree sustains a failure and a recovery (UDP packets of 1024 bytes).** The failure occurs at time $t \approx 2.3$ s and the link is repaired at $t \approx 4.50$ s.

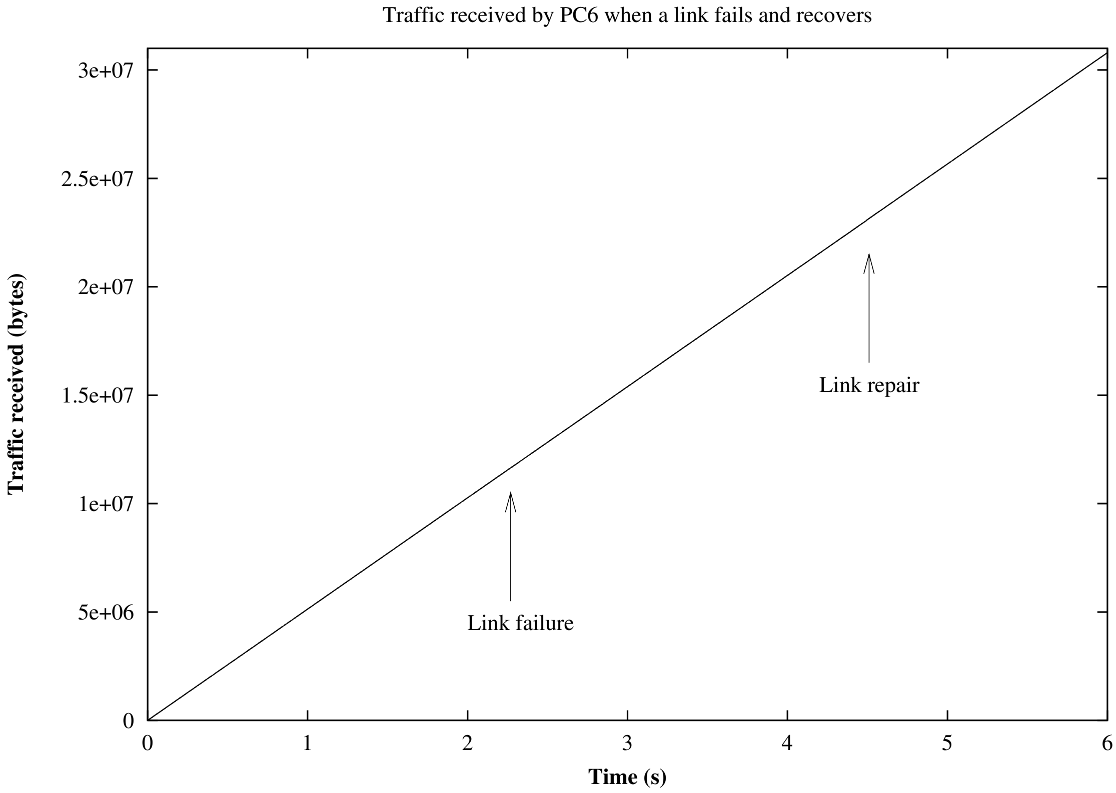


Figure 6.19: **Traffic received by *PC6* when the tree sustains a failure and a recovery (UDP packets of 1024 bytes).** The failure occurs at time $t \approx 2.3$ s and the link is repaired at $t \approx 4.50$ s.

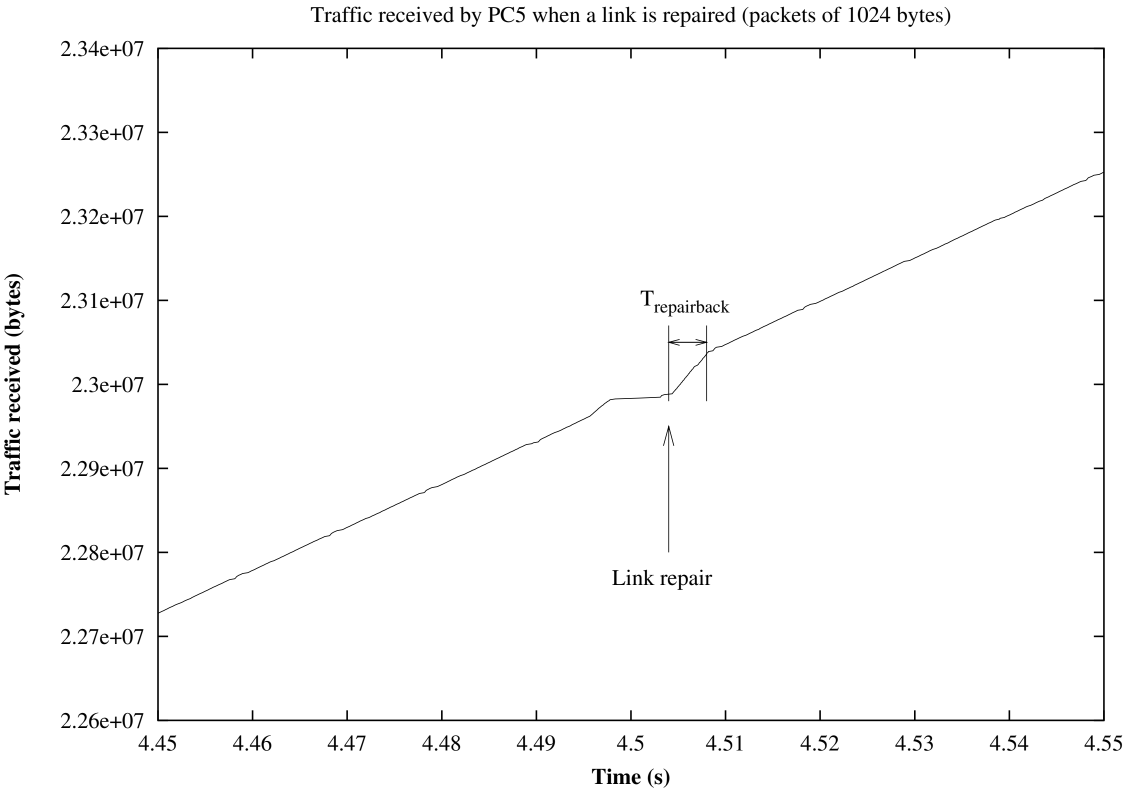


Figure 6.20: **Switchback on *PC5* (packets of 1024 bytes).** A traffic increase during a short period is visible at time $t \approx 4.50$ s.

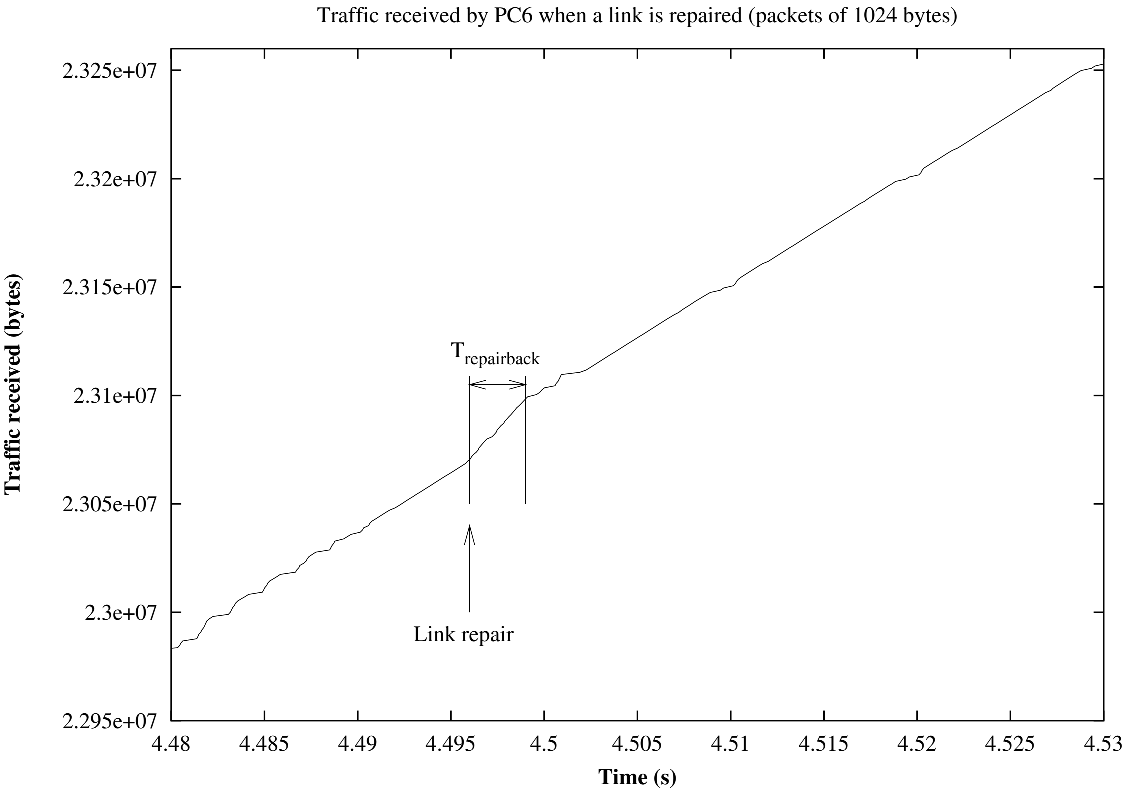


Figure 6.21: **Switchback on *PC6* (packets of 1024 bytes).** A traffic increase during a short period is visible at time $t \approx 4.50$ s.

7

Conclusion

Link failure is a common cause of service disruption in computer networks. Many techniques have been developed to alleviate the consequences of hardware failure in a network like the Internet by rerouting traffic from a failed link to a working or a set of working links. Rerouting is performed automatically in the Internet by recomputing routing tables. However routing convergence may be slow and faster techniques which require expensive hardware have been developed to protect networks from link failures. MPLS is a recent virtual circuit packet switching technology which has been designed to support the forwarding of IP packets over virtual circuits. MPLS Fast Reroute is a traffic engineering technique that is able to reroute IP traffic quickly without the need of additional hardware. Indeed, MPLS Fast Reroute relies on pre-planned backup path to reroute traffic on a link failure and can be implemented in existing routers.

An important delivery mode of the Internet is multicasting, where the information sent by a member of a multicast group is received by all other members of the group.

A popular example of a multicasting application is teleconferencing. In real-time applications like teleconferencing, if a link failure occurs, it is crucial to repair the multicast routing tree of the multicast communication in a short time. For example, an interruption of service of more than 50 ms is noticeable in a live transmission. Establishing a backup path to protect a multicast routing tree is a resource consuming process. Therefore, it is desirable to protect a large number of members of a multicast group with a low number of backup paths. In this thesis, we presented an algorithm which is able to choose such a backup path, and the design and implementation of an MPLS-based rerouting mechanism adapted to the protection of multicast routing trees. We now review our contributions and expose possible future work.

7.1 Contributions

In Chapter 3, we presented a graph algorithm which computes a single backup path to protect a multicast routing tree. The backup path is computed after the multicast routing tree establishment and before a link failure occurs, making it suitable for pre-planned rerouting mechanisms. The aim of the algorithm is to minimize the number of members of the multicast group dropped from the communication when a single link fails. We showed how a backup path determined by the algorithm could be used to reroute traffic so that no node is dropped from the tree when a single link of the protected path fails. We determined the complexity of the algorithm in the average and the worst case. We also gave extensions to the algorithm to support dynamic multicast groups where nodes can join and leave after the establishment of the communication.

In Chapter 4, we presented MPLS multicast Fast Reroute, a pre-planned rerouting mechanism that can use the backup path computed in Chapter 3 to protect a multicast routing tree when a single link fails. MPLS multicast Fast Reroute uses a probing mechanism to detect link failures. The nodes that detect the failure then propagate link failure notifications over the multicast routing tree. Two routers, one at each end of the backup path, switch traffic over the backup path when they are notified of the link failure. When the link is repaired, the nodes that detect the failure also detect the link repair and propagate link recovery information on the multicast routing tree. The mechanisms used to detect the repair and notify the routers of the tree of the link repair are the same as those used when a link fails. The two routers which switched traffic over the backup path perform switchback by stopping forwarding traffic on the backup path. During switchover, nodes downstream of the failed link with regards to the center of the tree are temporarily disconnected from the multicast routing tree. During switchback, certain links of the tree see an increase of traffic and certain nodes of the tree receive duplicate packets.

In Chapter 5, we presented our implementation of multicast extensions to MPLS-Linux. MPLS-Linux is a unicast MPLS implementation that runs on Linux PCs. To our knowledge, our implementation is the first non-proprietary MPLS multicast implementation. We also presented the MulTreeLDP protocol which implements the MPLS multicast Fast Reroute mechanism described in Chapter 4. We evaluated the performance of multicast MPLS-Linux and MPLS multicast Fast Reroute. Overall, multicast MPLS-Linux is able to forward fast Ethernet traffic on PC routers with no dedicated hardware. MPLS multicast Fast Reroute can repair a multicast routing tree in a few tens of milliseconds which mostly correspond to the time to detect the failure. Because of the timer resolution limitation on the Linux operating system, it is not possible to build faster software link failure detector. When the failed link is repaired, MPLS multicast Fast Reroute can switch the traffic back on the original multicast routing tree in a few milliseconds.

7.2 Directions for future work

Our work can be extended in several directions. First, our algorithm selects a single backup path to protect a multicast routing tree from a single link failure. Extensions to our algorithm may take into account node failures, multiple link or multiple node failure, or the computation of several backup paths to improve the resilience of the multicast routing tree.

Second, although the message format of MulTreeLDP is the same as the format of CR-LDP messages, MulTreeLDP and CR-LDP are different protocols. Since LDP/CR-LDP is the reference signaling protocol for MPLS, MulTreeLDP should be merged with CR-LDP and eventually standardized as a part of CR-LDP. Concerning MPLS multicast Fast Reroute, our experiments show that the transient network overload due to switchback does not disrupt network operations, temporary transmission of duplicate packets may occur. Clearly, it is desirable to totally avoid the forwarding and delivery of duplicate packets.

Finally, we implemented our solution on a small network of PC routers. We performed our experiments on a single multicast LSP and a single flow. An extension to our work includes an implementation in commercial routers and deployment in large scale networks.

Bibliography

[1] C. Alaettinoglu, V. Jacobson, and H. Yu. Toward millisecond IGP convergence, October 2000. NANOG 20, Washington, D.C., USA.

[2] Mostafa H. Ammar, Shun Yan Cheung, and Caterina M. Scoglio. Routing multipoint connections using virtual paths in an ATM network. In *Proceedings of IEEE INFOCOM*, pages 98–105, 1993.

[3] L. Andersson, P. Doolan, N. Feldman, A. Fredette, and B. Thomas. IETF RFC 3036: LDP Specification, January 2001.

[4] ANSI. Fiber Distributed Data Interface (FDDI) – Token Ring Media Access Control (MAC), ANSI X3.139-1987, 1987.

[5] G. Armitage. IETF RFC 2022: Support for multicast over UNI 3.0/3.1 based ATM networks, November 1996.

[6] The ATM forum, http://www.atmforum.com/.

[7] Traffic Management Specification Version 4.0, ftp://ftp.atmforum.com/pub/approved-specs/af-tm-0056.000.pdf.

[8] D. Awduche, L. Berger, D. Gan, T. Li, V. Srinivasan, and G. Swallow. IETF RFC 3209: RSVP-TE: Extensions to RSVP for LSP Tunnels, December 2001.

[9] A. Ballardie. IETF RFC 2201: Core Based Trees (CBT) multicast routing architecture, September 1997.

[10] Bellcore. SR-NWT-001756, Automatic Protection Switching for SONET, Issue 1, October 1990.

[11] Bellcore. GR-1230-Core, SONET Bidirectional Line-Switched Ring Equipment Generic Criteria, Issue 2, November 1995.

[12] Bellcore. GR-1400-Core, SONET Dual-Fed Unidirectional Path Switched Ring (UPSR) Equipment Generic Criteria, Issue 1, Revision 1, October 1995.

[13] R. Braden, L. Zhang, S. Berson, S. Herzog, and S. Jamin. IETF RFC 2205: Resource ReSerVation Protocol (RSVP), September 1997.

[14] P. Brittain and A. Farrel. MPLS Traffic Engineering: a choice of signaling protocol, January 2000.

[15] H. Bruyninckx. Real Time and embedded HOWTO, http://people.mech.kuleuven.ac.be/ bruyninc/rthowto/.

[16] B. Cain, S. Deering, B. Fenner, I. Kouvelas, and A. Thyagarajan. Internet Group Management Protocol, Version 3, http://www.ietf.org/internet-drafts/draft-ietf-idmr-igmp-v3-10.txt, April 2002. Work in progress.

[17] Alcatel Corporate Research Center. IP Multicast in MPLS Networks, technical leaflet, 1999.

[18] Inc Cisco Systems. Internetworking Technology Handbook, http://www.cisco.com/univercd/cc/td/doc/cisintwk/ito_doc/index.htm, 2002.

[19] R. Cole, D. Shur, and C. Villamizar. IETF RFC 1932: IP over ATM: A framework document, April 1996.

[20] S. Deering. IETF RFC 1112: Host extensions for IP multicasting, August 1989.

[21] S. Deering. *Multicast routing in a datagram internetwork*. PhD thesis, Stanford University, December 1991.

[22] S. Deering and R. Hinden. IETF RFC 2460: Internet Protocol, Version 6 (IPv6) specification, December 1998.

[23] Differentiated services charter (DiffServ), http://www.ietf.org/html.charters/diffserv-charter.html.

[24] E. W. Dijkstra. A note on two problems in connection with graphs. *Numerical Mathematics*, 1:269–271, 1959.

[25] C. Diot, W. Dabbous, and J. Crowcroft. Multipoint communication: A survey of protocols, functions and mechanisms. *IEEE Journal on Selected Areas in Communications*, 15(3), April 1997.

[26] C. Diot, B. Levine, J. Lyles, H. Kassem, and D. Balensiefen. Deployment issues for the IP multicast service and architecture. *IEEE Network*, 14(1):78–88, January / February 2000.

[27] D. Estrin, D. Farinacci, A. Helmy, D. Thaler, S. Deering, M. Handley, V. Jacobson, C. Liu, P. Sharma, and L. Wei. IETF RFC 2362: Protocol Independent Multicast-Sparse Mode (PIM-SM): Protocol specification, June 1998.

[28] A. Farrel and B. Miller. Surviving failures in MPLS networks. Technical report, Data Connection, February 2001.

[29] W. Fenner. IETF RFC 1112: Internet Group Management Protocol, Version 2, November 1997.

[30] E. N. Gilbert and H. O. Pollak. Steiner minimal trees. *SIAM journal of applied mathematics*, January 1968.

[31] GNU's Not Unix licenses, http://www.gnu.org/licenses/.

[32] M.-H. Guo and R.-S. Chang. Multicast ATM Switches: Survey and Performance Evaluation. *ACM SIGCOMM, Computer Communication Review*, 28(2), April 1998.

[33] D. Harel and R. Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM Journal on Computing*, 13(2), 1984.

[34] D. Haskin and R. Krishnan. A method for setting an alternative label switched paths to handle fast reroute, November 2000. Work in progress.

[35] C. L. Hedrick. IETF RFC 1058: Routing Information Protocol, June 1988.

[36] J. Heinanen. IETF RFC 1483: Multiprotocol encapsulation over ATM Adaptation Layer 5, July 1993.

[37] IEEE 802.17 Resilient Packet Ring Working Group, http://grouper.ieee.org/groups/802/17/.

[38] IEEE. 802.3-2002 Information Technology - Telecommunication & Information Exchange Between Systems - LAN/MAN - Specific Requirements - Part 3: Carrier Sense Multiple Access with Collision Detection (CSMA/CD) Access Method and Physical Layer Specifications 2002, 2002.

[39] IETF Internet Traffic Engineering Charter, http://www.ietf.org/html.charters/tewg-charter.html.

[40] Alon Itai and Michael Rodeh. The multi-tree approach to reliability in distributed networks. In *IEEE Symposium on Foundations of Computer Science*, pages 137–147, 1984.

[41] 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. IETF RFC 3212: Constraint-Based LSP Setup using LDP, January 2002.

[42] R. M. Karp. *Reducibility among combinatorial problems.* Plennum Press, New York, 1972.

[43] M. Kodialem and T. Lakshman. Dynamic routing of bandwidth guaranteed multicasts with failure backup. In *Proceedings of IEEE INFOCOM*, June 2002.

[44] R. Kuhn. Sources of failure in the public switched telephone network. *IEEE Computer*, 30(4), April 1997.

[45] Kansas University Real-Time Linux, http://www.ittc.ku.edu/kurt/.

[46] G. Malkin. IETF RFC 2453: RIP version 2, November 1998.

[47] M. Médard, S. Finn, R. Barry, and R. Gallager. Redundant trees for preplanned recovery in arbitrary vertex-redundant or edge-redundant graphs. *IEEE/ACM Transactions on Networking*, 7(5):641–652, 1999.

[48] J. Moy. IETF RFC 1584: Multicast extensions to OSPF, March 1994.

[49] J. Moy. IETF RFC 2328: OSPF version 2, April 1998.

[50] Multiprotocol Label Switching (MPLS) charter, http://www.ietf.org/html.charters/mpls-charter.html.

[51] MPLS for Linux, http://sourceforge.net/projects/mpls-linux/.

[52] M. Muuss. ttcp, http://ftp.arl.mil/ mike/ttcp.html.

[53] Linux manpages — Netlink.

[54] NetX: Networking research without a better home, University of Cambridge, UK, http://www.cl.cam.ac.uk/research/srg/netos/netx/.

[55] NIST Switch - NIST MPLS research platform, National Institute of Standards and Technology, information technology laboratory, http://snad.ncsl.nist.gov/nistswitch/.

[56] D. Ooms, B. Sales, W. Livens, A. Acharya, F. Griffoul, and F. Ansari. Framework for IP Multicast in MPLS, draft-ietf-mpls-multicast-07, January 2002. Work in progress.

[57] D. Oran. IETF RFC 1142: OSI IS-IS intra-domain routing protocol, February 1990.

[58] K. Owens, S. Makam, V. Sharma, B. MackCrane, and C. Huang. A path protection/restoration mechanism for MPLS networks, draft-chang-mpls-path-protection-03, July 2001. Work in progress.

[59] J. Postel. IETF RFC 791: Internet Protocol, September 1981.

[60] Y. Rekhter and T. Li. IETF RFC 1771: A Border Gateway Protocol 4 (BGP-4), March 1995.

[61] E. Rosen, D. Tappan, G. Fedorkow, Y. Rekhter, D. Farinacci, T. Li, and A. Conta. IETF RFC 3032: MPLS Label Stack Encoding, January 2001.

[62] E. Rosen, A. Viswanathan, and R. Callon. IETF RFC 3031: Multiprotocol Label Switching Architecture, January 2001.

[63] E. C. Rosen. IETF RFC 827: Exterior Gateway Protocol (EGP), October 1982.

[64] V. Sharma and F. Hellstrand. Framework for MPLS Based Recovery, draft-ietf-mpls-recovery-frmwrk-06, July 2002. Work in progress.

[65] H. Truong, W. Ellington, J. Le Boudec, A. Meier, and J. Pace. LAN Emulation on an ATM network. *IEEE Communications Magazine*, 33(5):70–85, May 1995.

[66] ttcp multicast, http://mng.cs.virginia.edu/software.html.

[67] D. Waitzman, C. Partridge, and S. E. Deering. IETF RFC 1075: Distance Vector Multicast Routing Protocol, November 1988.

[68] B. Wang and J. Hou. Multicast routing and its QoS extension: Problems, algorithms, and protocols. *IEEE Network Magazine*, 14(1), January / February 2000.

[69] Y.-F. Wang and R.-F. Chan. Self-healing on ATM multicast tree. *IEICE Transaction on Communication*, E81-B(8):590–598, August 1998.

[70] Interface between data terminal equipment(DTE) and data circuit-terminating equipment(DCE) for terminals operating in the packet mode and connected to public data networks by dedicated circuit.

[71] S. Yasukawa, M. Uga, H. Kojima, and K. Sugisono. Extended RSVP-TE for Multicast LSP tunnels, draft-yasukawa-mpls-rsvp-multicast-00, June 2002. Work in progress.