Contents

Abstract

The present Thesis is concerned about new generation's protocol MPLS (Multiprotocol Label Switching). Target of this work is been the realization of a first, simply, but also complete release of a new software layer to insert in the protocol stack of UNIX systems (more specifically for FreeBSD), that implements MPLS. The starting point were: the protocol's specifications, listed in [MPLS-ARC] and a public domain implementation of the protocol , on LINUX platform. As a matter of fact, the public domain software was realized before the final version of the specifications, due to this only a small part of it is been reused, because it was very different from the final specifications.

Three are the main topics concerning MPLS Architecture that have been implemented:

- Label's Table.
- Access Interface to Label's Table.
- Management of MPLS packet.

The Label's Table represents the database of the used labels and the information associated to them. The whole data structure is been implemented as a part of the kernel. Specifically it is a two levels hash indexed table with dynamic allocations of record's blocks, so to avoid wasting too much memory space. Along with the real data structure of the Label's Table there are the management routines, also implemented as a part of the kernel.

To use the Label's Table, likewise as the Routing Tables, it has been defined a new kind of socket, by means of the which is possible to access the Label's Table. The mechanism of MPLS sockets offers an interface to the Label's Table at User Level. Like Routing Sockets, MPLS Sockets have a set of possible operations, and a set of data structures that can be used for parameters passage. The realization of this kind of interface simplify, for example, the future development of demons for the labels distribution, argument not treated in the present work.

To manipulate MPLS packets it has been inserted a new software layer between the Data-Link layer and the Network layer of the protocol stack. In this first release that software layer is able to interact only with Ethernet as Data-link protocol and with IPv4 as Network layer protocol. Nevertheless, is our conviction that the code we implemented is quite open to allow the interaction with other protocols (both in the upper layer that in the bottom layer) with a small amount of work. This software layer manipulate the MPLS packet, using the information present in the Label's Table, to the which it can access in a direct manner, without using MPLS sockets. This because MPLS sockets are the interface towards the User Level, whereas management routines are inside the kernel itself, like the Label's Table.

Since this is a first release of a protocol still object of studies, there were no specific performance targets. Nevertheless it has been compared the traditional IPv4 forwarding and the new MPLS forwarding. The result is that MPLS is a bit faster in forwarding packets.

Chapter 1

Introduction to MPLS Architecture

1.1 Current Internet architecture

In the current Internet Architecture, but also in all packet switched networks, information travel fragmented in several packets, from one hop (router) to another until they reach the final destination (figure 1.1). Any Network Layer packet , travel from a router to another; on each router the forwarding operation is independent from the forwarding decision of the other routers.

Therefore every router analyses the Network Layer header of the packet, besides on every router is running a Network Layer routing algorithm. Every router decides independently in which direction to forward the packet, based on the packet's header and the result of the routing algorithm.

Usually in a router is implemented a protocol stack like that one shown in figure

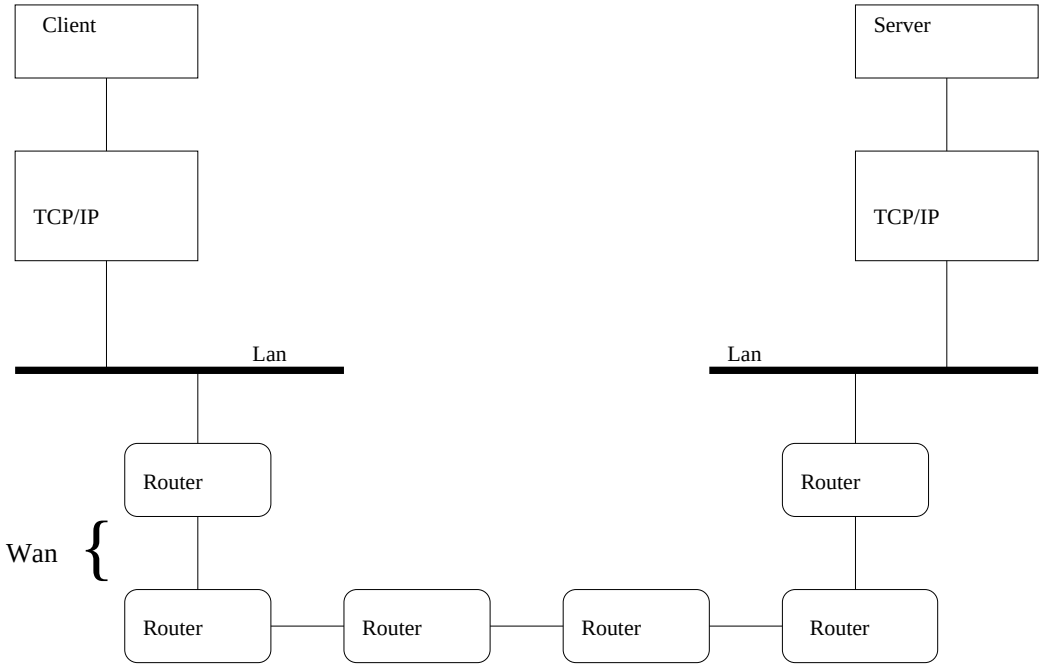

Figure 1.1: Typical Client/Server connection on Packet Switched Networks.

1.2[1]. The higher layers protocols do not perform any forwarding operation, but we find them to support one or more routing protocol, usually based on Transport Layer protocols. For example the BGP (Border Gateway Protocol) is based on TCP.

The Network Layer header of a packet, usually contains much more information that is needed just to choose the next hop. Choosing the next hop can be considered as the union of two functions. The first function partitions the set of all possible packets in a set of *"Forwarding Equivalence Classes"* (FEC). The second one map every FEC in a next hop.

Insofar as the forwarding decision is concerned, different packets which get mapped into the same FEC are indistinguishable. Indeed different packets, belonging to the same FEC, traversing a particular router will be forwarded in the same manner.

Therefore a FEC is the set of all the packets that on a particular router will be forwarded in the same way. Note that this definition is quite general, and may include

[1]In the figure 1.2 it has been given prominence to the co-existence of two version of the IP protocol (by now is a real common thing), also to underline how above the Data-Link Layer several different Network Layer protocols may be present.

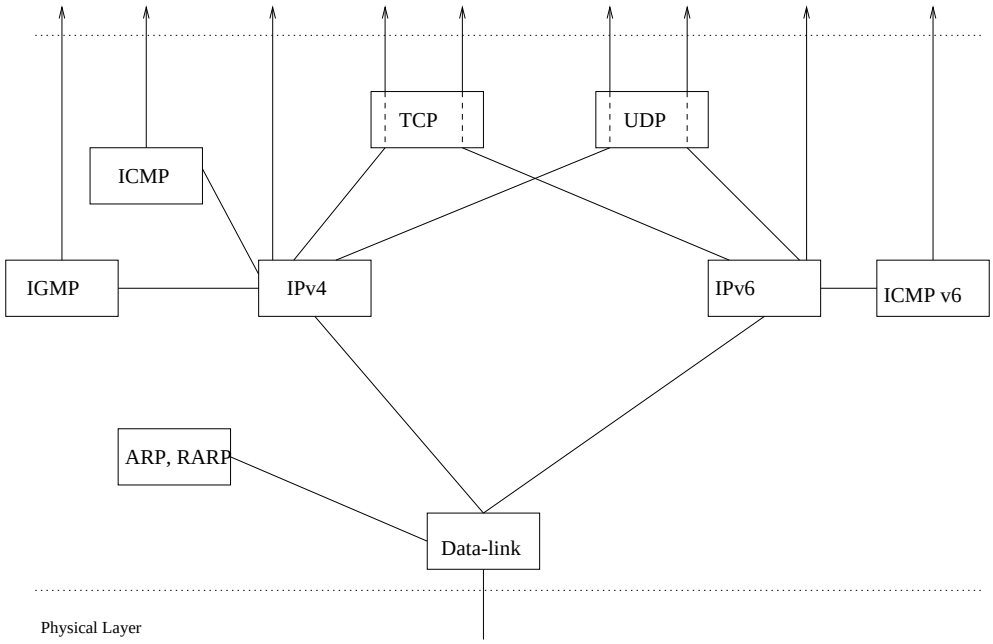

Figure 1.2: Typical Protocol stack present on a router.

as pertaining criterion to a FEC, not only the final destination address, but also other information, as (for example) the Class of Service.

Nevertheless, a particular router will typically consider two packets to be in the same FEC if there is some address prefix X in those router's routing tables such that X is the "longest match" for each packet's destination address. This can be considered true for all the Network Layer's protocols. Indeed both IPv4 and IPv6 and also the OSI model use a searching algorithm that considers how long is the match found.

As the packet traverses the network, each hop in turn reexamines the packet and assigns it to a FEC. In other words each hop searches in its routing table and assigns the packet to a particular FEC based on the destination address of it.

As shown in figure 1.3 in every router the packet goes up again from the Data-Link Layer to the Network Layer, where it is analyzed and then, based on the result of the routing algorithm, forwarded.

Parsing the header of every packet is much more complex, therefore much more time spending, as more complex is the forwarding operation. Indeed, the forwarding opera-

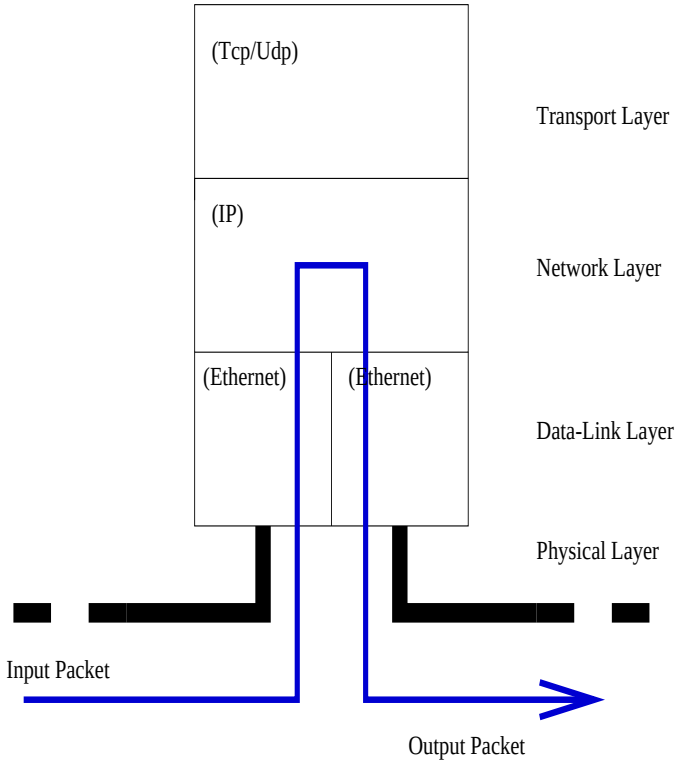

Figure 1.3: Path followed by a packet forwarded by a router.

tion may include several decision concerning the packet, this can be grouped together and called *Traffic Engineering* (from now on indicated with the acronym TE).

TE groups all the operations beyond the simple forwarding of the packet, concerning load balancing, explicit routing (also called source routing), Class of Service, or implementation of additional services that involve the forwarding of packets, for example the Virtual Private Network.

Also in this case, as in the simple forwarding, every router decides, independently from other routers, all the operations necessary to the TE, analyzing several different fields of the packet's header. Therefore on every router part of operations performed on every packet are the same as those performed on the router from which the packet is arrived.

1.2 MPLS Architecture

1.2.1 New Internet Architecture

Introducing the new protocol MPLS (MultiProtocol Label Switching) modify slightly the classic protocol stack, there is a new layer, the MPLS layer precisely, above the Data-Link Layer, but under any other Network Layer protocol. In figure 1.4 is shown how the protocol stack changes adding MPLS.

Note that does not filter entirely the packets exchanged between the Data-Link Layer and the Network Layer. Indeed MPLS forwarding does not replace Network Layer forwarding, but introduce a new kind of forwarding at a lower level; in paragraph 1.2.3 it is furnished an example that explain the aim of this choice. The impact on the pre-existing code is so reduced, indeed it is necessary to introduce only some hooks to make possible for the different modules to interact with the MPLS module.

In the present implementation the MPLS module has been hooked up with IPv4 as Network Layer protocol, and with Ethernet as Data-Link Layer protocol.

MPLS has been projected to be totally independent from Network Layer protocols,

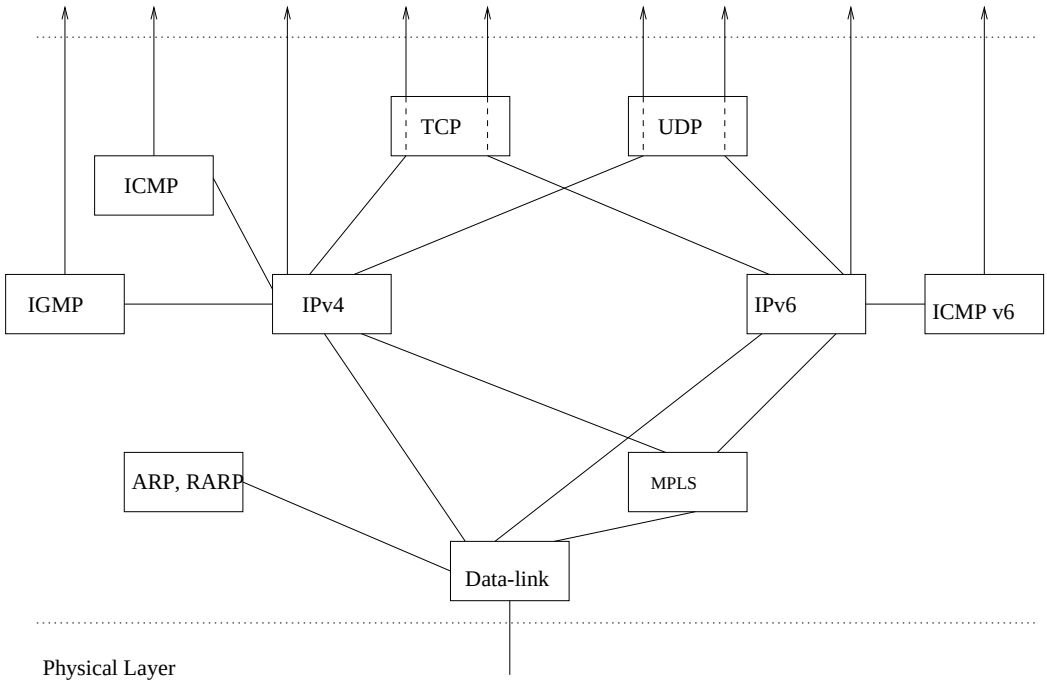

Figure 1.4: Protocol stack after the introduction of MPLS.

in other words on the top of MPLS can run whether IPv4 or IPv6, but also OSI protocols etc. etc.. Note that MPLS is best suited to run on routers, not on end-hosts, this is due because the most vantages that MPLS offers are concerned on forwarding and the related TE. As described in RFCs, routers supporting MPLS are called LSR (Label Switching Router).

In MPLS, the assignment of a particular packet to a particular FEC is done just once, as the packet enters the network. The FEC to which the packet is assigned is encoded as a short fixed length value known as a "label". When a packet is forwarded to its next hop, the label is sent along with it; that is, the packet are "labeled" before they are forwarded.

At subsequent LSR, there is no further analysis of the packet's Network Layer header. Rather, the label is used as an index into a table which specifies the next hop, and a new label. that is going to substitute the one arrived along with the packet. It will be shown as the label is not just used as a linear index like for an array, but it is used in a little bit more complicated way.

In the MPLS forwarding paradigm, once a packet is assigned to a FEC, no further header analysis is done by subsequent routers; all forwarding is driven by the labels. This has a number of advantages over conventional Network Layer forwarding. Furthermore, MPLS forwarding can be accomplished by switches not capable of analyzing the Network Layer headers but capable of doing label lookup and replacement.

Since a packet is assigned to a FEC when it enters the network, the ingress router may use, in determining the assignment, any information it has about the packet, even if that information cannot be gleaned from the Network Layer header. For example, packets arriving on different ports may be assigned to different FECs also if they have the same destination address. Conventional forwarding, on the other hand, can only consider information that travels with the packet in the packet header.

A packet that enters the network at a particular router can be labeled differently than the same packet entering the network at a different router, and as a result forwarding decisions that depend on the ingress router can be easily made. This cannot be done with conventional forwarding, since the identity of a packet's ingress router does not travel with the packet.

The considerations that determine how a packet is assigned to a FEC can become ever more and more complicated, without any impact at all on the routers that merely forward labeled packets. To support TE sometimes is necessary to force a packet to follow a particular route which is explicitly chosen at or before the time the packet enters the network. This technique is used, for example, to make the packet to follow a route different from the one resulting from a normal forwarding hop-by-hop, may be because that path is congested. In the traditional Network Layer protocols, this requires the packet to carry an encoding of its route along with it (source routing). In MPLS, a label can be used to represent the route, so that the identity of the explicit route need not be carried with the packet.

As written above, some routers analyze the Network Layer header of a packet to determine a set of additional services that have been grouped under the name of TE, MPLS allows this information to be fully inferred from the label. In this case, one may

say that the label represents the combination of a FEC and TE information. Therefore, all those information that before were retrieved from various fields of the Network Layer header of the packet, are now inferred from the label of the packet. Furthermore, all the TE related decisions are made just once, when the packet enters the network, and are not repeated on any router.

1.2.2 Labels

A label is a short, fixed length, locally significant identifier which is used to identify a FEC.

The label, which is put on a particular packet, represents the Forwarding Equivalence Class to which that packet is assigned. Most commonly, a packet is assigned to a FEC based (completely or partially) on its Network Layer destination address. However, the label is never an encoding of that address. If Ru and Rd are LSRs, they may agree that when Ru transmits a packet to Rd, Ru will label the packet with a label value L if and only if the packet is a member of a particular FEC F. That is, they can agree to a "binding" between label L and FEC F for packets moving from Ru to Rd. Note that L does not necessarily represent FEC F for any packets other than those which are being sent from Ru to Rd. L is an arbitrary value whose binding to F is local to Ru an Rd. It is the responsibility of each LSR to ensure that it can uniquely interpret its incoming labels.

Let us call *Label Space* the set of all the possible ingress label values, a router may have a unique *Label Space* that include all the possible values of incoming labels, otherwise a router may have several *Label Spaces*, not necessarily disjointed, one for each interface. This is a local choice of every single router, the most important thing is that the router, in the latter case, should be able to discriminate from which interface two packets with the same label value are arrived. In the present work has been choose a unique *Label Space* for all the interfaces present on the host. This choice was done because being this a first implementation of the protocol, the target was to produce a

simple, basic, but also complete version of the protocol itself.

Suppose Ru and Rd have agreed to bind label L to FEC F, for packets sent from Ru to Rd. Then with respect to this binding, Ru is the *Upstream LSR*, and Rd is the *Downstream LSR*. In the MPLS architecture, the decision to bind a particular label L to a particular FEC F is made by the LSR which is downstream with respect of that binding.

Thus labels are assigned from the downstream LSR, and label bindings are distributed from downstream to upstream. Generally is possible to bind some attributes to the assignment between the label L and the FEC F, these attributes can be distributed along with labels to which are bound.

1.2.3 Label Distribution

It comes up from what written above that is necessary a protocol able to provide to the distribution of the label, from the LSR that assigned it to a particular FEC, to the LSR upstream.

A label distribution protocol, will include all initial negotiation that two router have to engage to aware all MPLS capabilities each other. The MPLS architecture does not assume that there is only a single label distribution protocol. Indeed, a number of different label distribution protocols are being standardized. Existing protocols have been extended so that label distribution can be piggy-backed on them[2]. New protocols have also been defined for the explicit purpose of distributing labels[3]. It is obvious that distribution of labels generally introduce a further traffic overhead in the network, partially limited by piggy-backing on existing protocols. Nevertheless is possible that once the labels for a particular path are distributed, routing messages of the higher layers may be reduced.

Labels distribution is a part of the MPLS architecture that is not been considered in

[2]There are works in progress to standardize extensions to both BGP-4 protocol [MPLS-BGP], and RSVP protocol [MPLS-RSVP-TUNNELS].

[3]For more information refer to [MPLS-LDP] or [MPLS-CR-LDP].

the present work, focused instead on packet's storing/retrieving/forwarding procedures.

At this point is possible to better explain why MPLS forwarding does not replace Network Layer forwarding (see figure 1.4). Assume that a new data flow to a certain destination starts flowing on an LSR. Assume that Network Layer routing information are present on the LSR but no label is still assigned to the new data flow. Until the label distribution protocol will not provide a label assignment, it is possible to forward packets in the Network Layer. When a label is provided for that data flow, it is possible to start forwarding packets in the MPLS Layer.

1.2.4 MPLS hierarchy

Label Stack

So far, we have spoken as if a labeled packet carries only a single label. MPLS architecture furnishes a more general model in which a labeled packet carries a number of labels, organized as a last-in, first-out stack. This introduce, as we shall see, the concept of hierarchy, although the processing of a labeled packet is completely independent of the level of hierarchy. The processing is always based on the top label, without regard for the possibility that some number of other labels may have been above it in the past, or that some number of other labels may be below it at present. An unlabeled packet can be thought of as a packet whose label stack has depth 0. See section 4.1 for a detailed description of the label stack organization.

Label Switched Path (LSP)

A "Label Switched Path" (LSP) of level m for a packet as the sequence of routers:

- which begins with an LSR (LSP Ingress) that pushes on a level m label;
- all of whose intermediate LSRs make their forwarding decision by label switching on a level m label;

- which ends (LSP Egress) when forwarding decision is made by label switching on a level m-k label, where k>0, or when a forwarding decision is made by ordinary, non-MPLS forwarding procedures.

A consequence of this is that whenever an LSR pushes a label onto an already labeled packet, it needs to make sure that the new label corresponds to FEC whose LSP Egress is the LSR that assigned the label which is now second in the stack. Anyway this is something concerning labels distribution and therefore not involving this work directly.

Tunnel and LSP

Sometimes a router Ru takes explicit action to cause a particular packet to be delivered to another router Rd, even though Ru and Rd are not consecutive routers on the Hop-by-Hop path for that packet, and Rd is not the packet's ultimate destination. For example, this may be done by encapsulating the packet inside a Network Layer packet whose destination address is the address of Rd itself. This creates a tunnel from Ru to Rd.

It is possible to implement a tunnel as a LSP, and use label switching rather than Network Layer encapsulation to cause the packet to travel through the tunnel. The set of packets which are to be sent through the LSP tunnel constitutes a FEC, and each LSR in the tunnel must assign a label to that FEC. The criteria for assigning a particular packet to an LSP tunnel is a local matter at the tunnel's transmit endpoint.

Hierarchy

Consider an LSP <R1,R2,R3,R4>. Let us suppose that R1 receives unlabeled packet P, and pushes on its label stack the label to cause it to follow this path, and this is in fact the hop-by-hop path. However, let us further suppose that R2 and R3 are not directly connected, but are neighbors by virtue of being the endpoints of an LSP tunnel. So the actual sequence of LSRs traversed by P is <R1, R2, R21, R22, R23, R3, R4>. When P travels from R1 to R2, it will have a label stack of depth 1. R2, switching on the label, determines that P must enter the tunnel. R2 first replaces the incoming label

with a label that is meaningful to R3. Then it pushes on a new label. This level 2 label has a value which is meaningful to R21. Switching is done on the level 2 label by R21, R22, R23. R23, which is the penultimate hop in the R2-R3 tunnel, pops the label stack before forwarding the packet to R3. When R3 sees packet P, P has only a level 1 label, having now exited the tunnel. Since R3 is the penultimate hop in P's level 1 LSP, it pops the label stack, and R4 receives P unlabeled. The label stack mechanism allows LSP tunneling to nest to any depth.

1.2.5 Forwarding in MPLS

Before facing forwarding operation as is defined in the MPLS architecture, let us give some definitions:

NHLFE - The NHLFE (Next Hop Label Forwarding Entry) is a data structure used when forwarding a labeled packet, it contains at least the following information:

1. The packet's next hop.
2. The operation to perform on the packet's label stack; this is one of the following operations:
   (a) Replace the label at the top of the label stack with a specified new label (SWAP);
   (b) Pop the label stack (POP);
   (c) Replace the label at the top of the label stack with a specified new label, and then push one or more specified new labels onto the label stack (PUSH);

It may also contain any other information necessary to correctly forward the packet. In the next chapters is shown how all these information are put together in the basic data structure of the Label Table.

ILM - The ILM (Incoming Label Map) maps each incoming label to a set of NHLFEs. It is used when forwarding packets that arrive as labeled packets. Note that the destination of a forwarding operation may be the router itself. If the ILM maps a particular label to a set of NHLFEs that contains more than one element, exactly one element of the set must be chosen before the packet is forwarded. In the present work this ability is been reduced: ILM maps a label in just one NHLFE, not in a set of NHLFEs.

FTN - The FTN (FEC-to-NHLFE map) maps each FEC to a set of NHLFEs. It is used when forwarding packets that arrive unlabeled, but which are to be labeled before being forwarded. If FTN maps a particular FEC to a set of NHLFEs that contains more than one element, exactly one element of the set must be chosen before the packet is forwarded. As in the ILM case, in the present work this ability is been reduced: FTN maps a FEC in just one NHLFE, not in a set of NHLFEs.

It's now possible to analyze what is a forwarding operation in the MPLS architecture. In order to forward a labeled packet, a router examine the label at the top of the stack. The router, by mean of the ILM, maps the label in a NHLFE. Using the information contained in the selected NHLFE, the router determines where to forward the packet and what operations to perform on the label stack, then it put the packet on the right interface. Thus, a packet that enters an LSR does not reach the Network Layer to be forwarded, but it stops at the "MPLS Layer", as shown in figure 1.5.

In order to forward an unlabeled packet , a router analyze the Network Layer header to determine the FEC which the packet belongs. Then, by means of the FTN function an NHLFE is selected. The information found in the NHLFE, are used by the router to determine in which direction to forward the packet and what operation to perform on the label stack (of course a POP operation is not permitted in this case). Due to the fact that the packet now passes through another software layer, the latency time of the packet in the protocol stack is slightly increased.

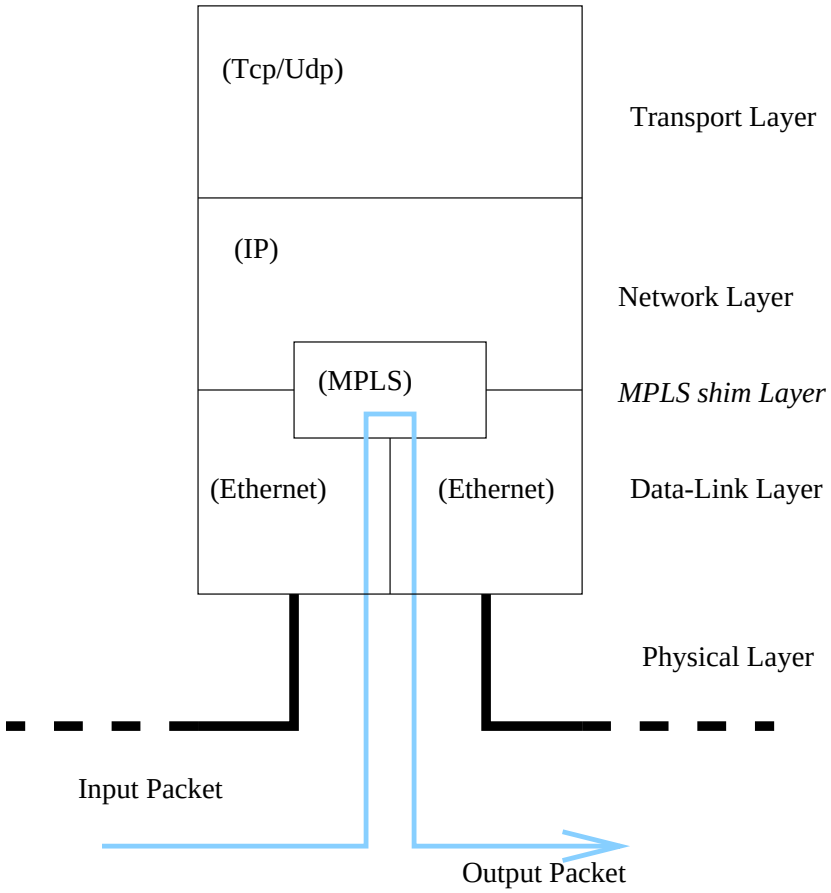

Figure 1.5: Forwarding operation on a LSR.

Note how in both cases the next hop is always taken from the selected NHLFE; this allow to have a next hop different from the one that can be obtained using the normal hop-by-hop routing, as described in 1.2.1 talking about TE and Source Routing. Note that it does not mater how complex are the belongings criteria to a FEC (i.e. if the belongings criteria include TE information) the forwarding operation of a packet consist always in performing an ILM (or an FTN) to select an NHLFE which will contain all the information needed to forward the packet.

Chapter 2

Implementation of the Label Table

2.1 Introduction

The implementation of the Label Table is not started from scratch. Starting point is been the data structures of the NistSwitch's software package[1], named NistSwitch-CAIRN.

Starting with a FreeBSD 3.4 machine, on which was added the Kame package[2], the installation of the NistSwitch-CAIRN package introduce some changes to the kernel of the operating system in order to obtain: The implementation of a label table; the ability of manipulate MPLS packets; and also furnishes a modified version of an RSVP demon, provided with extensions to distribute labels "piggy-backed" on RSVP messages[3].

As mentioned before, this work never face the labels distribution, thus the RSVP demon is not been analyzed and adopted in the present software.

[1] The software package can be retrieved at `http://www.antd.nist.gov/itg/nistswitch/`

[2] The software package Kame `http://www.kame.net` furnishes an implementation of IPv6 and IPsec for FreeBSD machines.

[3] For more detailed information on the argument refer to [MPLS-RSVP-TUNNELS].

Concerning the rest of the NistSwitch package, only the Label Table part is been reused and deeply modified. The part of The NistSwitch software regarding the packets manipulation has not been reused, due to the numerous, little, modifications which the kernel is strewn and that make this solution not compact and hard to improve. The modifications introduced by the software NistSwitch regards also the Kame package. The MPLS layer instead, has been realized from scratch, starting from the native FreeBSD code. This because the target was to collect all of the produced code in few files, trying to reduce the impact on the original kernel. Thus the Kame package is no longer needed.

The developing environment were FreeBSD-4.2 machines, and the language used to develop the code were the C. Due to the stability of the protocol stack's code of FreeBSD, probably the written code works "as is" also with other FreeBSD releases. Anyway, specific test have to be realized for this purpose.

2.2 Main data structure (`LabelTableEntry`)

The Label Table can be thought as a table indexed by a hash function, further will be shown how the index, as a mater of fact, can have two levels, each one accessed by different hash function. Due to the choice of a unique label space for the whole machine, every label is unique within the Label Table.

The Label Table is a data structure allocated in the kernel's memory space; inside this one is been defined a particular memory type, like other big kernel's data structures. The memory type, as shown in figure 2.1, is been defined as `M_LTBUF`.

```
#ifdef _KERNEL
/* Declaration of label table memory type */
MALLOC_DECLARE(M_LTBUF);
#endif /* _KERNEL */
```

Figure 2.1: Definition of memory type where the Label Table will be allocated.

To define a particular memory type for a data structure, or for a class of data structure, is useful for a subsequent control of the amount of memory really allocated, or to measure the memory really allocated to dynamic structures (like the Label Table) using system's utilities like *memstat*.

```
#define         MPLS_LT_STACKSIZE       4

typedef struct _labelTableEntry {
        u_int32_t       lteLabel;
        u_int32_t       lteCoS;
        u_int32_t       lteFlags;
        FecChain        *lteFecList;
        u_int8_t        lteFecCount;
        u_int8_t        lteTTL;
        sa_family_t     lteUpperAF;
        u_int8_t        lteStackDepth;
        union _labelstack {
                MplsShim        NewLabelStack[MPLS_LT_STACKSIZE];
                MplsShim       *NewLabelStackPtr;
        } lteLabelStack;
        u_int32_t       lteStatsOut;
} LabelTableEntry, *LabelTableEntryPtr;
```

Figure 2.2: Main structure of the Label Table.

In figure 2.2 is shown the definition of the main data structure of the Label Table. This data structure contains all the information suggested in [MPLS-ARC] for the NHLFE, indeed this structure is nothing more than a table of NHLFE entries. The meaning of the various fields is described in the subsequent paragraphs.

The `lteLabel` is a 32 bit long field, due to preserve alignment. Only the 20 bits less significant are really used to codify a label as defined in [MPLS-SHIM]. The `lteCoS` field contains all information related to TE, in this work the field is totally ignored, but it is still present for future extensions. The `lteFlags` field contains several different information, particularly it contains all the indications on the operations to perform on the label stack. In table 2.1 are listed all the numeric value and meaning of all possible flags.

Note that the operation related to the `LTE_PUSH` is not corresponding to the one defined in [MPLS-ARC], and described in 1.2.5 paragraph, this because the flag does

Flag	Value	Meaning
LTE_SWAP	0x1	Swap an existing label.
LTE_PUSH	0x2	Push a new label.
LTE_SWAP_PUSH	0x3	Like the previous but the push operation may involve several labels at the same time.
LTE_POP	0x4	Pop label at the top of the stack.
LTE_MODIFING	0x10	The entry is modifying.
LTE_DONE	0x20	Not used.
LTE_MULTICAST	0x200	Not used.

Table 2.1: Possible flag value of the `lteFlags` field.

not involve any swap operation. This choice has been done to simplify forwarding of first time labeled packet that arrive from the Network Layer. The Push operation as defined in [MPLS-ARC] is implemented by the operation corresponding to the `LTE_SWAP_PUSH` flag. Setting this flag is possible to make a swap and then multiple push, that is to introduce more than one label at the top of the stack in the same time.

The last three flags of the table, that is: `LTE_MODIFING`, `LTE_DONE` and `LTE_MULTICAST` are inheritance of the original NistSwitch software. The present work does not consider the multicast traffic, thus the corresponding flag is not used (as a matter of fact also in the NistSwitch software it wasn't in use).

In the NistSwitch code was considered the multi-processor case, thus, due to the possibility of more than one process that might access the Label Table in the same time, it was necessary to signal when an entry were changing and when this was done, so the `LTE_MODIFING` and `LTE_DONE` flags were defined. It were also present some data structures and routines to lock partially the Label Table. These parts have been totally deleted, due to the deep changes made to the original that made those parts not working and thus not usable. The `LTE_MODIFING` flag continue to be set at the beginning of all routines that modify an entry, and cleared at the end of them. Nevertheless, the other routines that access the entries do not check the flag.

`The lteFecList` data structure is a list of `rtentry` structure's type pointers (refer to figure 2.3). It has been depicted that the main belongings criterion to a FEC is the Network Layer destination address of the packet. So, the network layer destination ad-

```
typedef struct fecchain { /* Usefull to chain fecs of a label */
        struct rtentry   *fec;
        struct fecchain  *next;
} FecChain;
```

Figure 2.3: Basic structure of the FEC's list.

dress may be viewed as a very simple FEC. The union of several FEC is a FEC itself, so a list of FECs assigned to single addresses (note that a single address may be also subnet address, not only host addresses) are themselves a FEC. By the `lteFecList` it is possible to access to all the destination addresses belonging to that FEC. In `lteFecCount` there is the number of element in the list. To avoid needless duplication of information already present in other data structures of the kernel, it has been chosen to use pointers to routing tables.

For example, assume there are two data flow with the same destination but a different Class of Service; these data flows can be distinguished by using different incoming labels and different outgoing labels, even if they have the same destination. In this case, if the FEC's information of the destination address were stored directly in the Label Table entry, three copy of the same information were present in the kernel: two copy in the Label Table (the two entries of the two different incoming labels) and a third copy in the routing table. Furthermore, the kernel's code that implements the Data-Link Layer works using routing entries, thus to restrict changes to this code it is useful to be able to furnish this kind of entries. Indeed, in this way ones an MPLS packet is ready it can be just sent; the Data-Link Layer that is not interested in the packet's payload, need just a routing entry to obtain the correct information for the forwarding of the packet. Thus, using a direct access to a routing entry associated to a Label Table entry introduce an efficiency improvement.

The solution adopted is still very general. Usually for every Network Layer protocol is created a different Routing Table, so if X is the number of Network Layer protocols present on the machine, X is also the number of different Routing Table present on the system. Every Routing Table is based on `rtentry`'s type entries, thus any Label Table

entry may refer, by mean of the `lteFecList,` elements contained in different Routing Tables.

It should be clear that in the system will be present always only one Label Table, whatever is the number of Network Layer protocols present. This lead to a very polite and simple architecture. Note, in the end, that through the routing tables is possible to access in a real simple and comfortable way to all the information concerning the forwarding of a packet, particularly those regarding the output interface.

The MPLS architecture assumes that at the top of the label stack of a packet, along with the label, there is always a TTL field (Time-To-Live) for loop protection. Which value must be putted in the TTL field when the packet is labeled for the first time? If the Network Layer protocol, from which the packet arrives, explicitly provides for a TTL value on its header, for example as in the IP case (both IPv4 and IPv6), suffice to copy the value of that field on the top of the stack. On the other hand, if the Network Layer protocol does not explicit provides a TTL value, the one contained in the `lteTTL` field can be copied beside the label at the top of the stack.

To look which is the Network Layer protocol associated to the label suffice to read the value contained in the `lteUpperAF` field.

Labels that have to be pushed on the top of the label stack (except the case of a POP operation) are stored in the data structure `lteLabelStack`. This structure is an `union` type, therefore can be interpreted in two different ways, depending of the value of the `lteStackDepth` field:

1. If `lteStackDepth <= MPLS_LT_STACKSIZE`, then the structure will contains directly the labels that must be added to the label stack.

2. If `lteStackDepth > MPLS_LT_STACKSIZE`, then the structure will contains a pointer to an array which contains the labels that must be added to the label stack.

This choice was made thinking that pushing numerous labels at the same time on the top of the stack is quite rare. Thus, is better to provide space to store directly few

labels, and at the same time to be able to manage particular cases with a high number of labels.

What value is the best choice for `MPLS_LT_STACKSIZE`? It has to be an arrangement between a not too high value, so that not much memory space is wasted, on the other hand, it has to be big enough so that most time the first case is what happens. Due to the lack of statistics, and without any kind of indications, in the present work the value is been fixed to 4. Labels values are stored in "Network Byte Order" so to make just a simple copy of the memory to put these labels at the top of the stack. The "Network Byte Order" is preserved by using the `MplsShim` type and its macros. See paragraph 4.1 for further information about `MplsShim` data structure.

The last field of the data structure analyzed in this paragraph is the `lteStatsOut` field which is a counter of the number of packets sent using that entry. This only for statistics purpose.

In the this data structure are totally missed, and are left for further extensions, all the fields necessary to manage multicast traffic and "explicit routed" traffic.

2.3 Hash Index

As described in the previous sections, exists a hash index, for the Label Table, based on label and used to access the main structure. As should be known, a hash function is characterized by two main features:

- Easy and fast computation.
- Conflicts.

To overwork the advantages of the first point it is compulsory to manage also the second one. Due to this, a pointer is added to the main data structure (see figure 2.4), so that list are generated when conflict arise.

```
typedef struct _labelTable {
        LabelTableEntry         ltEntry;

        struct _labelTable    * ltHashList;
} LabelTable, *LabelTablePtr;
```

Figure 2.4: Structure used to organize the Label Table's entries, that conflict in the hash index, as a list.

2.3.1 First level hash index

let us starting introducing only the first level hash index, leaving out the second level hash index in this first instance. The hash index consists in an array of `MPLS_LT_HASHSIZE` entries of `LabelTableHead` type (see figure 2.5). In this structure there is a pointer, `lthHeader`, that refers the list of all Label Table's entries that conflict on that particular index value, the `lthCount` field contains the number of label that conflict on that particular entry of the hash index (in practice is the number of element of the list).

```
typedef struct _labelTableHead {
        LabelTable             *lthHeader;
        struct _labelTableHead *lthSecondTable;
        u_int32_t               lthCount;
} LabelTableHead, *LabelTableHeadPtr;
```

Figure 2.5: Hash index structure.

The value of `MPLS_LT_HASHSIZE` has been fixed to 256, anyway in the continuation of this report the more general constant `MPLS_LT_HASHSIZE` is used, rather to the particular value, to not restrict the argument. Also if it is possible to change the size of the hash table that forms the index, it is convenient to chose always sizes that are power of 2. In this manner there is a sort of protection against odd results of the hash function, because the result of the function can be masked and reduced on logarithm of base 2 of `MPLS_LT_HASHSIZE` bits. In this way the possible final result is restricted to the range that effectively can be used to access the array that implements the Hash Table. This is putted in evidence by figure 2.6 where is depicted the code of the first level hash function.

```
PRIVATE u_int32_t
lt_label_hash1(u_int32_t first_op, u_int32_t second_op)
{
  u_int32_t answer=0;

  while (first_op) {
        answer += first_op;
        answer *= 69069;
        first_op >>= 8;
  };
  while (second_op) {
        answer += second_op;
        answer *= 69069;
        second_op >>= 8;
  };

  return (answer&(MPLS_LT_HASHSIZE-1)); /* This work well only if
                                  MPLS_LT_HASHSIZE is a power of 2 */

}       /* End of lt_label_hash1(...) */
```

Figure 2.6: First Level Hash function.

The first level hash function is the original one of the NistSwitch package. No changes were introduced, anyway, although its simplicity it is not assured that it is the most effective. May be useful to find a hash function simple as the original one but less heavy to compute, possibly with only one `while` loop inside it. Indeed, as is showed in figure 2.6, there are two successive `while` loops, these sometimes may consume a lot of computation time. In figure 2.7 is depicted how the Label Table is organized when only the first level hash index is present. It is also emphasized the case of one or more conflicts, thus several entries of the Label Table are chained in a list.

To access to a particular entry of the Label Table, the first step is to compute the hash function of the label whose entry is searched, then to access to the corresponding entry of first level Hash Table. If there aren't conflicts directly access to the entry of the Label Table, by means of the first and unique element of the conflict's list, else the conflict's list must be looked up, finding a match between the searched label and the one contained in the `lteLabel` field of the entry under examination.

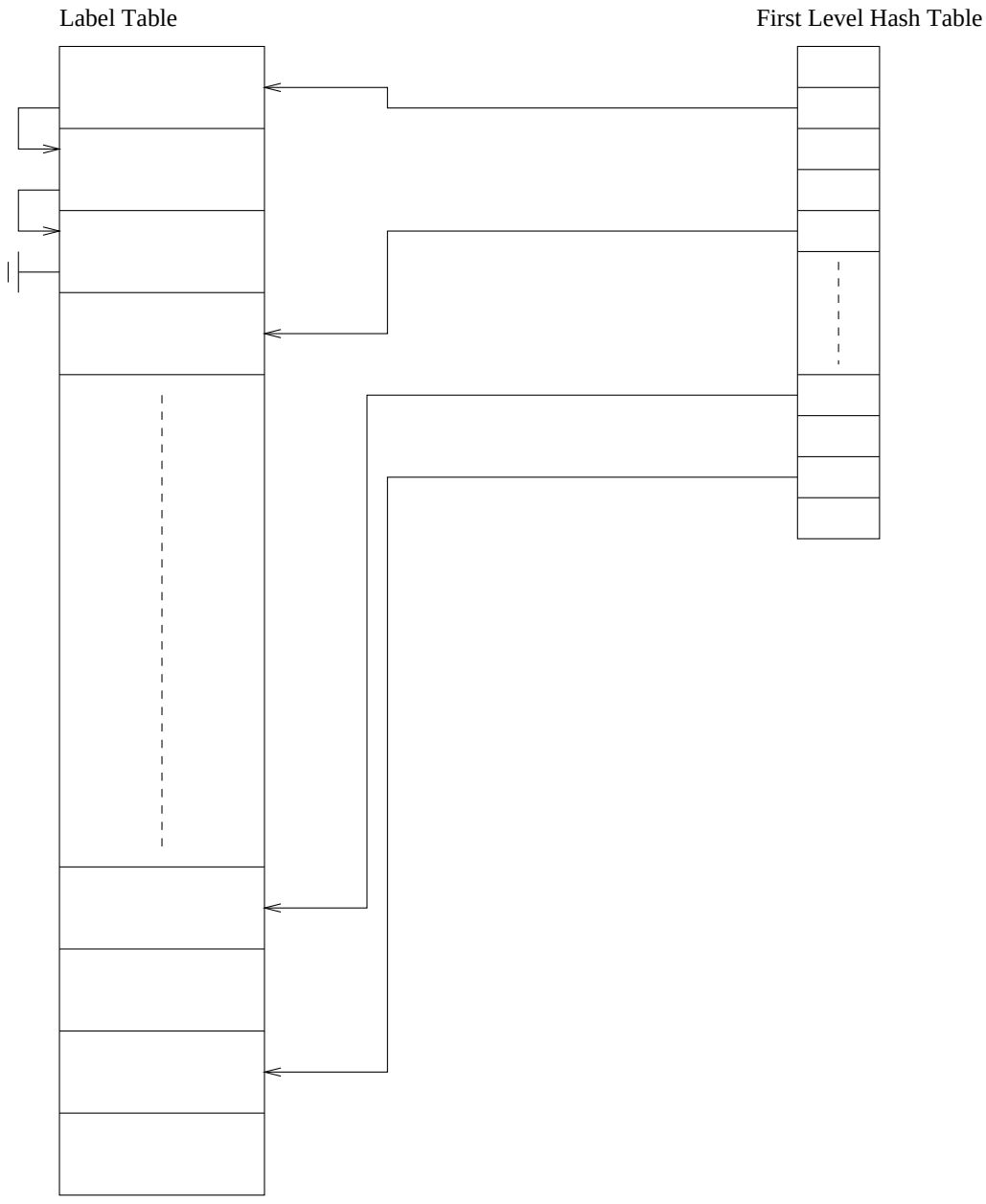

Figure 2.7: Label Table with first level hash index.

2.3.2 Second level hash index

As described in the paragraph above, when there are several conflicts on an entry of the Hash Table, it is necessary to make a sequential look up of the conflict's list. Of course if the conflict's list augment its size that is the number of the conflicts on the same hash entry augment considerably, a sequential look up is computationally time consuming. In an architecture, like MPLS, whose targets are efficiency and speed, this situation is unacceptable.

Due to this, a second level hash index is introduced. Indeed, while conflicts on a single entry of the first level Hash Table increase, may be profitable to make a new look up by a hash index rather than by sequential search on a list. This has been the choice of the NistSwitch package and maintained in this work.

Sudden note that second level hash index is not a static structure, always present for all the entries of the first level Hash Table, rather it exist a second level hash Table only for those entries that reach determined number of conflicts. Thus, the Label Table starts with one level hash index. If while the Label Table is filled, in one of the entry of first level Hash Table the number of conflicts goes over a threshold, then it will dynamically allocated another Hash Table that will play the role of second level index for that entry.

In the present work the constant value `MPLS_LT_MAXCHAIN` controls the max number of conflicts allowed on one entry of the first level Hash Table. Once the number of conflicts reaches this value a second level hash Table is allocated for the entry. The value of `MPLS_LT_MAXCHAIN` has been fixed to 10, how in the original NistSwitch package. Nevertheless, the choice of the correct value should be done by comparing the hash function computation average time and the conflicts list sequential look up average time.

At this point it should be clear the meaning of the pointer `lthSecondTable` shown in figure 2.5 and neglected above. When a second level Hash Table is allocated, by means of this pointer is possible to access the new table. Note that the new second level Hash Table is of the same type as the first level Hash Table, indeed it must support the

same functions as the first level one. The differences between the first level Hash Table and the second level hash Table is that in the latter the `lthSecondTable` pointer would never be used, because there is not a third level index, the former, instead, would not use the `lthHeader` pointer, because there is no conflicts list, being these redistributed on the second level Hash Table.

When a second level Hash Table is introduced , the `lthCount` field slightly modify its meaning, indeed it will contain the number of entries of the Label Table referenced by the second level Hash Table. In figure 2.8 is shown how the Label Table index changes assuming to introduce a second level Hash Table for the entry that in the first level Hash Table in figure 2.7 has got three conflicts.

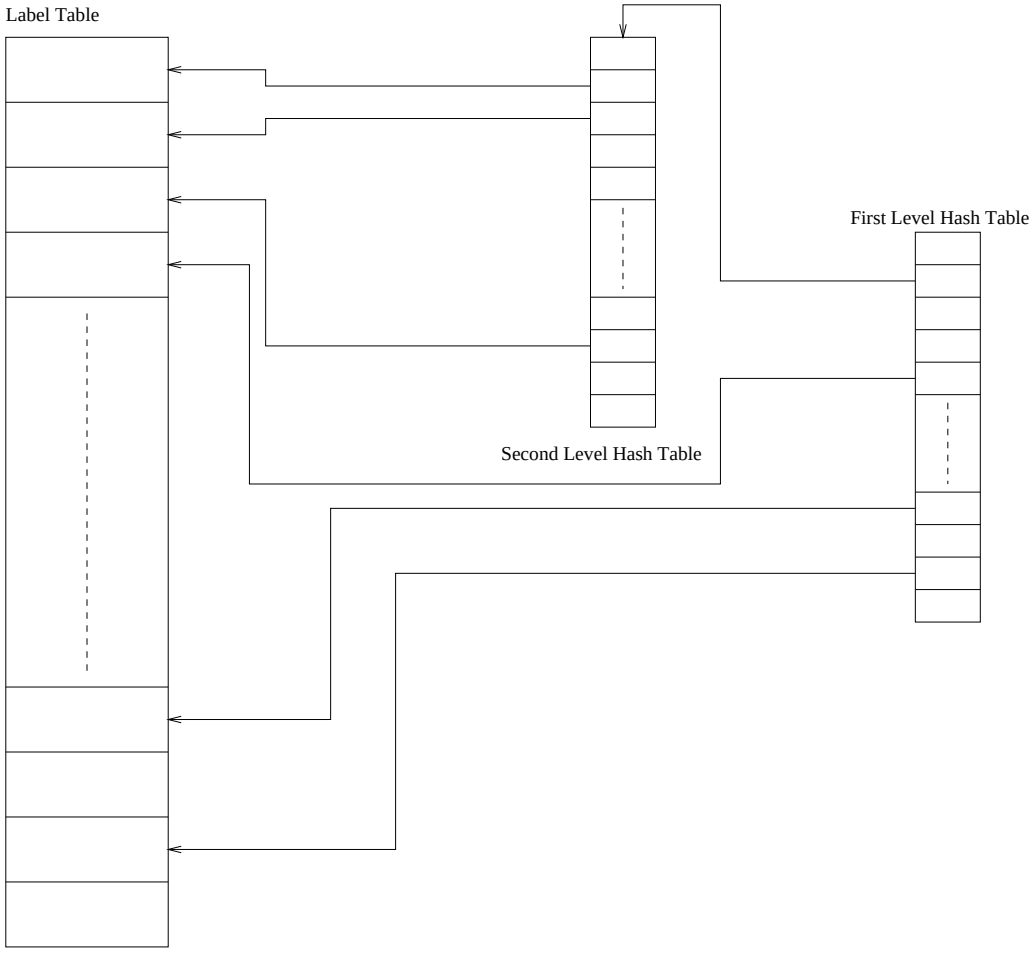

Figure 2.8: Label Table with two levels hash index.

To perform a selection inside a second level Hash Table it is necessary to compute another hash function. Of course it cannot be the same as the first level hash function, because this one has generated the conflicts that lead to introduce a second level Hash Table. In the NistSwitch package's code uses Jenkins's hash function, in the present work the choice is not changed.

Jenkins's hash function[4] is slightly more complex than the first level hash function, so to guarantee more uniformity (i.e. less agglomerate with possible conflicts) in distributing the used entries.

Insofar it has been showed that when the number of conflicts on a first level Hash Table's entry gets over the `MPLS_LT_MAXCHAIN` value, a second level Hash Table is allocated and the conflicting entries are then referenced by this second table which is referenced by the first level Hash Table. Nothing was said about how the situation in figure 2.7 changes to the situation in figure 2.8. Redistribute on two levels is a critic passage, the Label Table must be always in a consistent state. This to avoid problems (for example) while is on redistribution operation arrives an interrupt, perhaps due to an incoming MPLS packet, that starts a process that access to one of the entries that the system is redistributing on two levels. Due to this, before to modify the first level Hash table, a copy of all the Label Table's entries involved in the redistribution is done. The whole operation is done by routine in figure 2.9.

First of all, the routine check if there is enough space in the Label Table to make a copy of all the entries involved in the redistribution, in order to avoid to begin redistribution operation and stop it later due to the lack of space in Label Table. Then a second level Hash Table[5] is allocated. A this point a copy of all conflicting entries is done, putting the right reference in the second level Hash Table. To find a free entry in the Label Table the `lt_alloc()`function is used, this function will be detailed in paragraph 2.4. During the copy, the cache is also updated if necessary, in order to access directly the new copies. The meaning of the for loop containing a call to `Scan_lte_list()`

[4]For more information visit the home page of the original author of the hash function, Bob Jenkins, at URL: `http://burtleburtle.net/bob`

[5]The `lt_addhash()` function simply allocate an array of `MPLS_LT_HASHSIZE` entries whose type is `LabelTableHead` (figure 2.5) and return a pointer to it.

```
PRIVATE int
RedistributeLabel(LabelTableHead *table)
{
  LabelTable *Current,  *newentry;
  LabelTableHead        *second_table;
  u_int32_t             labelhash2;
  FecChain              *finger;
  LabelChain            *temp;
  int                   i;

  if ( lt->lt_blockcount == MPLS_LT_BASESIZE &&
        MPLS_LT_BASESIZE*MPLS_LT_PREALLOC-lt->lt_entrycount\
        <= (table->lthCount +1) ){
        return -1;
  };
  second_table = lt_addhash();
  if (!second_table) {
     /* Should I Panic????? */
     errprint("MPLS [RedistributeLabel]: hash table allocation failled!\n");
     return -1;
  };
  for (Current = table->lthHeader; Current; Current = Current->ltHashList) {
      /* Find a free entry in the label table */
      newentry = lt_alloc();
      if (!newentry) {
          panic("MPLS [RedistributeLabel]: Redistribution failed\n");
      };
      *newentry = *Current; /* Make a copy of the label table entry */
      labelhash2 = lt_label_hash2( Current->ltEntry.lteLabel, 0);

      newentry->ltHashList = second_table[labelhash2].lthHeader;
      second_table[labelhash2].lthHeader = newentry;
      second_table[labelhash2].lthCount++;
      for ( finger = Current->ltEntry.lteFecList; finger; \
                        finger = finger->next) {
            temp = Scan_lte_list( finger->fec, Current);
            if (temp)
                temp->lte = newentry;
      };
  if ( lt_cache.entry[(Current->ltEntry.lteLabel & CACHEMASK)] == Current )
       lt_cache.entry[(Current->ltEntry.lteLabel & CACHEMASK)] = newentry;
  };
  table->lthSecondTable = second_table;
  return 0;
}
```

Figure 2.9: Redistribution Routine of the references on two levels hash index.

function, is described in paragraph 2.6.

Once the copy is completed, and all the references in the second level Hash Table are set, the last step is to install the table, just hooking it to the first level Hash Table. Routines that make the look up through the hash index are able to automatically recognize if there is a second level index, thus, once installed this routine will begin to use the second level index to access the Label Table. No entries are deleted immediately after the redistribution operation, in order to avoid dangling references from the kernel to old copies of the entries.

```
PRIVATE void
ChewLabelChain(LabelTableHead * table)
{
  LabelTable    *Current;

  while (table->lthHeader) {
        Current = table->lthHeader;
        table->lthHeader = Current->ltHashList ;
        lt_free(Current,0);
  };
}
```

Figure 2.10: Routine for the deletion of the old list in the first level index.

Removing the old copies of the entries, in the old conflicts list, is a task of the `ChewLabelChain()` routine, showed in figure 2.10. The `lt_free()` procedure that marks as free the entries of the Label Table will be described in paragraph 2.4. The important question is: when the part of the Label Table that contains the old copies will be freed? That is, when the `ChewLabelChain()` will be called? Every time a new entry is added to the Label Table, it is checked if does exists a second level Hash Table, if present, it is checked is the old copies were deleted, if still presents they are removed.

This solution, also if not strict, it works quite well, considered that once running, adding a label will not be so frequent to create problems.

A consideration must be done concerning the time spent to make operations like redistributing an index entry on two levels, or removing old copies of a first level index. The former seems quite heavy to compute. Does have these operations any consequence

on critic operation such, for example, forwarding an MPLS packet? Both the redistribution procedure and the deleting procedure are called during an operation that adds an entry to the Label Table, operation that comes up as consequence from a message received from the Downstream LSR. Thus they are never called during a critic operation that manage MPLS packets, so they will not affect the performance of the protocol in managing data traffic.

2.4 Dynamic memory allocation of the Label Table

Due to how the whole Label Table is structured, including hash index, the number of entries that can be managed is given by the size of the first level Hash Table multiplied the size of the second level Hash Table. In the present work, this number is nothing else than `MPLS_LT_HASHSIZE` *`MPLS_LT_HASHSIZE`, and due to the fact that `MPLS_LT_HASHSIZE` has been fixed to 256, this means 65536 possible entries. This is not completely true, on any hash index there may be conflicts, so it does not exist an upper limit for the Label table, nevertheless so one can have an idea of what are sizes at stake.

It easy to understand that due to the size of every single entry, and the size of the index tables, there is a big amount of memory occupied allocating statically the whole Label Table and related index, obtaining a big amount of wasted memory. As described above, the second level Hash Tables are not initially allocated, rather they are introduced only when needed by a particular entry.

The main problem is the block of the entries that maintains label's data. To allocate every single entry independently is not the best choice, because to manage the Label Table will become too complex, being every single entry a separated entity in memory. The solution adopted here is to pre-allocate a fixed number of entries as an array of `MPLS_LT_PREALLOC` records. In the present work `MPLS_LT_PREALLOC` has been fixed to 1024. In this way the wasted memory is in the worst case equal to the size of the array.

Once the first array is full, another one with, the same size, can be allocated, and so on while there are filled. To access these blocks it is used an array, where every element is a pointer to a different block of entries.

In this way is possible to start with only one block of entries allocated, and every time is needed another block is allocated and its reference stored in the array above mentioned. The size of this array is given by the `MPLS_LT_BASESIZE` value, set to 256 in this implementation. The max size that the Label Table can now reach, with this structure, is `MPLS_LT_PREALLOC*MPLS_LT_BASESIZE` entries, this means 262144 entries in the present work. This array can be considered static because it must be always present, the same thing is for the first level Hash Table, also every time present.

Nevertheless, at the beginning of this chapter, it has been said that all the data structure concerning the Label Table are allocated in the kernel's memory space, and that memory is marked as `M_LTBUF` type. To obtain this, also the first level Hash Table and the array of references to the blocks of entries, are explicitly allocated at initialization of the Label Table. In figure 2.11 is shown the structure containing all the references to the various structures that forms the Label Table and that are dynamically allocated. Really, also the `LtArea` structure is dynamically allocated, so that it is allocated in the `M_LTBUF` memory type. This solution not introduce further complexity, just another pointer that refers the structure. In the `LtArea` structure in addition to the pointer that refers the first level Hash Table (`lt_labelhash`) and the one that refers the array of blocks of entries allocated (`lt_base`), there are also other fields.

The `lt_range` structure contains two integers meaning the valid label space. Due to the length of 20 bits of the labels, the max range is from 0 to 1048576. But the first 16 values (0 - 15) are reserved and have a particular meaning[6], further on many drafts concerning MPLS it is used a partition of the label space between the machines belonging to the same LAN, this structure is able to store the partition assigned to host where is running. Thus, it is useful to store the valid label range, resulting from the initial negotiation of the Label Distribution Protocol present at higher layers of the

[6]Refer to [MPLS-SHIM] for the detailed definition and the description of the meaning of the first 16 values of the label space.

```
typedef struct labelspace {
        u_int32_t       max_value;
        u_int32_t       min_value;
} LabelSpace;


typedef struct _ltArea {
        LabelTable       *lt_supply;
        LabelTable      **lt_base;
        LabelTableHead   *lt_labelhash;
        LabelSpace      lt_range;

#define   lt_max_label  lt_range.max_value
#define   lt_min_label  lt_range.min_value
        int             lt_mmsd;
        int             lt_blockcount;
        int             lt_entrycount;
} LtArea, *LtAreaPtr;
```

Figure 2.11: Structure containing all the references to the various part of the Label Table.

protocol stack.

The `lt_blockcount` field counts the number of blocks of entries effectively allocated and accessible by `lt_base`. In `lt_entrycount` it is every time maintained the number of used Label Table entries. The `lt_mmsd` field indicate the max size of the label stack that can be putted on a packet. This field is used to check if a labeled packet need to be fragmented, refer to paragraph 4.2.1 for more information about this topic.

When the preallocation mechanism was introduced, nothing was said on how to distinguish the free entries from the occupied one. Indeed, preallocating `MPLS_LT_PREALLOC` entries, but using them once at a time, it is necessary to able to understand which entries are free and which occupied. The adopted solution is to maintain all the free entries in a unique list (referenced by `lt_supply`), using the same pointer (`ltHashList`) used when the entry is occupied, to manage conflicts in the Hash Table .

In this manner to occupy or to free an entry is reduced to just extract or to insert a record from the list of free entries. When the list is empty and there is a request to add an entry to the Label Table, it is only needed to preallocate another block of entries and add all the new entries to the list.

This solution has another advantage: when an entry is freed, no sorting or compacting operations of the Label Table are necessary, because putting the freed entry in the list this may be used like any other free entry.

In figure 2.12 there is the endeavor to represent the comprehensive data structure of the Label Table.

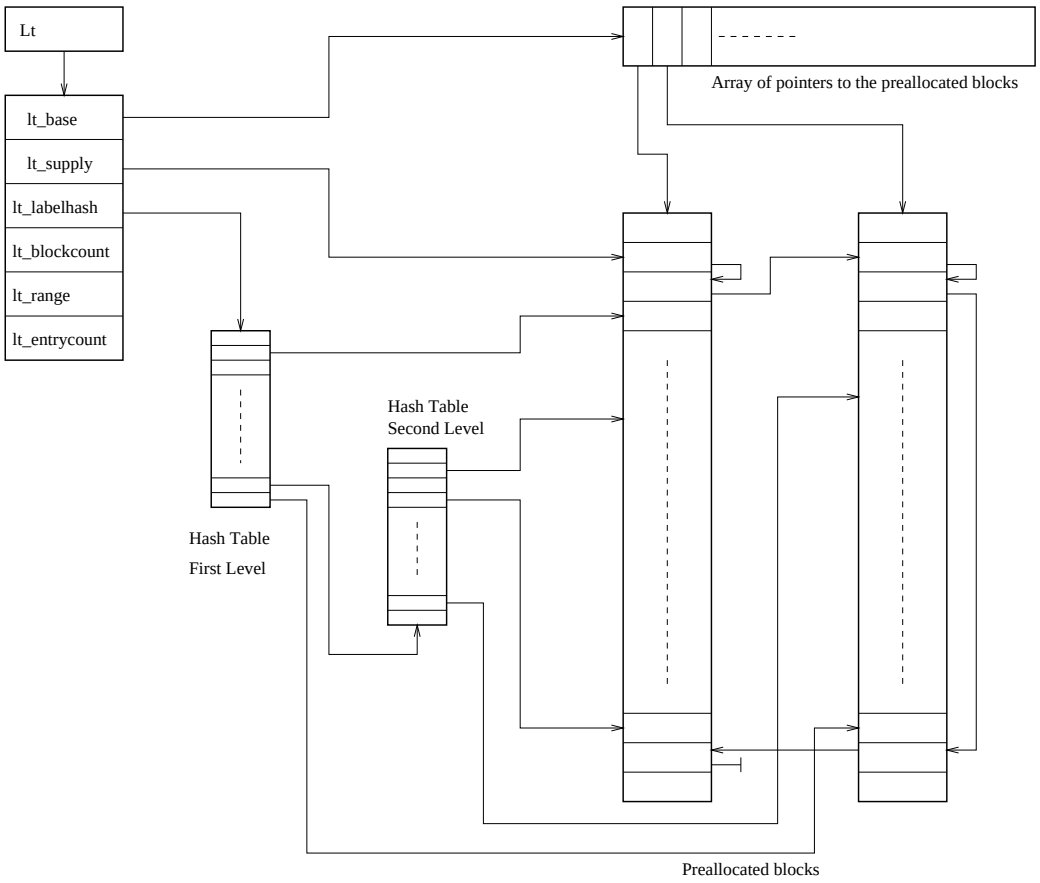

Figure 2.12: Comprehensive structure of the Label Table.

At this point it should be clear the task of the two routines `lt_alloc()` and `lt_free()` mentioned in the paragraph 2.3.2. By practice, they just extract or insert a record in the list of the free entries. The `lt_alloc()` routine provides also to allocate a new block of entries whenever is needed. The `lt_free()` instead never provide to unallocate a block of entries, due to the casual order in which the entries are freed, thus entries belonging to the list are part of different block, making impossible

to unallocated a whole block without any further operation.

Thus, the size of the Label Table is always increasing, never goes back, in this implementation. It is possible to realize a kind of Garbage Collector that groups the free entries in a unique block so to unallocate it when possible, but this is for future improvement of the software.

In the same situation is the hash index. Indeed, once redistribution on two level of a first level Hash Table entry has took place, the second level Hash Table will never be unallocated. Also if the number of entries referred by the second level table should fall under the `MPLS_LT_MAXCHAIN` value, remember that this value is the max number of conflicts allowed, the second level Hash Table is not unallocated, but still used. To modify the routines that manage the hash tables in order to introduce the unallocation of unnecessary second level tables is for further developments of the software.

2.5 Cache

The Label Table is furnished of a cache to make the most frequent access a bit faster. This is very useful, for example, in forward operations. It consist, how is shown in figure 2.13, in a simple array containing `MPLS_LT_CACHESIZE` references to entries of the Label Table. When a look up is performed on the Label Table, the first step is to check the cache, if something is found, then some time has been saved, else a minimum overhead has been introduced.

The access technique of the cache is very simple, the least significant bits of the searched label are used as array index. To use those bits a mask is used, so the cache size must be a power of 2, and the mask must agree with this value. The cache's size does not affect the access time, nevertheless if the cache is too small several labels may collide in the cache, introducing a cache miss at every look up, losing in this way all the advantages introduced by the cache. On the other hand, if the cache is too big, it may result never completely used, and thus wasting memory space.

```
#define MPLS_LT_CACHESIZE        256

#define CACHEMASK       0x000000ff

PRIVATE  struct _cache {
        LabelTable        *entry[MPLS_LT_CACHESIZE];
} lt_cache;

#define c_idx(t)        (t & CACHEMASK)
```

Figure 2.13: Cache of the Label Table.

2.6 Access to the Label Table

In the previous paragraphs, describing the Label Table, it has been described how a label is used to access the table, passing through the hash tables, obtaining a reference to the selected entry. But the system have got a label to use to access the table only when a labeled packet, arrived through one of its interfaces, is processed. Others cases have to be considered.

For packets originated on an MPLS machine, that have to be forwarded on an LSP, is necessary a Label Table's entry containing the stack that has to be added to those packets. But for those packets there is not a label by which perform a look up on the Label Table.

Consider the case of an LSR being a border gateway for an MPLS domain (i.e. an LSR connected to router which is not able to "speak" MPLS), than it will be very common to have unlabeled packets, in input on some interface, that have to be forwarded on an LSP on others interface. Also in this case, packets do not have a label by which perform the look up on the Label Table, to obtain the stack to put at the beginning of the mentioned packets.

Both cases can be reported to the case of Network Layer packets that are passed to the MPLS Layer to be labeled.

Independently of which is the Network Layer protocol the packet that passes to the

MPLS Layer belongs to, it shall exist a routing table with an entry that match the Network Layer destination address of the packet itself. If in the Label Table exists an entry that fits a particular packet, this entry will contains in its FecList a reference to a routing table entry that matches the Network Layer destination address of the packet. So it is sufficient to add a pointer in the routing table entry, so that a reference to the Label Table entry to which is associated can be stored.

In this manner, for packets arriving from a Network Layer protocol, can be performed a look up on the associated Routing Table and by the resulting entry to access the entry of the Label Table containing all the information needed to forward the packet at MPLS level.

The original `rtentry` structure, which is the basic structure of a routing entry and contains the routing information, is been modified for this purpose. It has been added a pointer and a flag, the latter just to signals that there is an LSP for the particular destination which the routing entry refers to.

Thus, for packets arriving from Network Layer, it is performed a look up on the associated Routing Table and then the Label Table is accessed. A routing entry can belongs to different FECList, for example it suffice to have two LSPs those differ only by the Class of Service, because of this the pointer in the routing entry does not refer directly the Label Table, instead it refers a list of pointers to the Label Table, containing the references to all the entries which the routing entry belongs to. In figure 2.14 is shown the record's structure that forms the list. It should be clear now the purpose

```
typedef struct labelchain {
        LabelTable          *lte;
        struct labelchain   *next;
} LabelChain;
```

Figure 2.14: Definition of the reference list to the Label Table.

of the `Scan_lte_list()` routine showed in figure 2.9. `Scan_lte_list()` will find the right element of the list of references to the Label Table associated to the routing entry, in order to update it.

At this point may be understood another of the advantages deriving from using references to the routing table. When a network Layer packet is processed, it is attempted to assign the packet to a FEC, i.e. a destination address that match the destination address of the packet is searched. The packet can be assigned to a FEC that include the whole subnet of the host to which the packet is destinate. This means to be able to recognize when a destination address is included in a subnet address.

The kernel of a Unix system is already able to make this kind of look up operation; indeed routing tables are organized as PATRICIA[7] trees, thus this features of the operating system is used to avoid redundant code. In the kernel's look up routines, if no match is found between the destination address and the routing table entries, the Default Route is returned.

No bindings between default Route and MPLS is permitted in the present work, in order to continue forward the packet at Network Layer when there is not precise information to forward the packet. It has been described how every Label Table entry hash got a label by which perform the access through the hash index. This type of access is used in all the routines that manage the Label Table.

Thus, also to the entries containing information concerning unlabeled packets is necessary to furnish a value to the `lteLabel` field. It has been chose a particular value called `LAB_DEFAULT` and valid for all the entries assigned to packet arriving from the Network Layer.

All the entries with this particular label value are treated in the same particular manner by the procedures that manage the Label Table. Indeed, for these entries it will never performed a look up using the hash index. Thus, all these entries are organized in a unique list accessed through the entry 0 of the first level Hash Table. This list is never redistributed on two levels, because the list will never be scanned for a look up.

The numeric value of `LAB_DEFAULT` has been fixed to 16, the first of the not reserved values, indeed, as described in paragraph 2.4, labels 0-15 are reserved.

Refer to paragraph 3.4, concerning the operations that is possible to perform through

[7]Refer to [STE-2] for a detailed description of the PATRICIA tree data structure.

the MPLS Sockets, for a high level description of the routines that manage the Label Table.

Chapter 3

MPLS Socket Interface

3.1 Introduction

The Label Table structure described in the previous chapter, is rather complex; instead to furnish a set of procedures at user level, by which directly handle the Label Table, it has been chosen to provide the Label Table with a socket interface, similar to the Routing Sockets interface. In the NistSwitch code there is a limited extension of the Routing Socket interface, in order to allow the access to the Label Table. In the present work it has been defined a new class of socket: MPLS Socket. In this way there is more order in the software architecture, Routing Socket are not weighed down, avoiding to mix different kind of operations. Furthermore, with this solution it is simpler to develop demons that implements a label distribution protocol. Also in this way, the present work is uniformed to a general interaction model with the internal structures of

the protocol stack, typically adopted in Unix systems.

3.2 Implementation of the MPLS Sockets

In Unix systems, thus in FreeBSD systems, the Socket Layer is a very general structure, totally independent from the network protocols implemented in the system, and able to interact with them. On the other hand, the Socket Layer offers a uniform access interface to the various protocols. Thus, to implement MPLS Socket it is not necessary to redefine the whole socket's structure adjusting it for MPLS; the Socket interface structure is not changed at all.

The first step is to define a new protocols domain, the MPLS domain, placed side by side with the existing domains. To every domain is assigned a Protocol Family, that in this case is `PF_MPLS`. The definition of this new Protocol family will be used also in the MPLS Layer that handles the MPLS packets and that will be described in the next chapters.

Every domain assigned to a Protocol Family is initialized at system's bootstrap. Also the MPLS domain is initialized at system's bootstrap, also providing to allocate the Label Table in the memory. Inside this new domain, only one protocol is defined, the `MPLSPROTO_LT` protocol, to which is assigned a `SOCK_RAW`[1] socket interface.

In figure 3.1 it is shown the definition of the MPLS domain and the `MPLSPROTO_LT` protocol inside it. In this manner the operating system is able to implement MPLS socket. In figure 3.2 is shown how to create an MPLS socket in any program.

What have to be explicitly defined are the routines that handles the operations requested through MPLS sockets. In the `protosw` structure belonging to the described `MPLSPROTO_LT` protocol, are stored all the references to the routines that implement all the operations that can be requested through the MPLS Socket.

Particularly `mpls_s_output()` has the task to recognize the requested operation, to

[1] For a detailed description about domains, protocols, and their implementation in Unix environment, refer to [STE-2].

```
static struct protosw mplssw[] = {
{ SOCK_RAW,     &mplsdomain,    MPLSPROTO_LT,   PR_ATOMIC|PR_ADDR,
  0,            mpls_s_output,          raw_ctlinput,   0,
  0,
  raw_init,     0,              0,              0,
  &mpls_usrreqs
}
};

static struct domain mplsdomain =
  { PF_MPLS, "mpls", mpls_init, 0, 0,
    mplssw, &mplssw[sizeof(mplssw)/sizeof(mplssw[0])] };

DOMAIN_SET(mpls);
```

Figure 3.1: Definition MPLS domain and MPLSPROTO_LT protocol.

```
int sockfd;

sockfd = socket(PF_MPLS, SOCK_RAW, 0);
```

Figure 3.2: Creation of a MPLS socket in user's code.

extract the data structures from the received message, and to call the right routines, and to send back the message. Note that, how in routing socket, also in MPLS Socket when a message is received through a particular socket, it is sent back toward all opened MPLS Sockets.

Like any other operation, also the operations requested through an MPLS Socket can produce an error. Thus, some new error code are defined, specific for the Label Table, and that can be sent back through the MPLS Sockets. In table 3.1 are listed the new error codes defined along with their numeric value and their meanings.

The code for the routines that handle the MPLS Sockets are produced starting from the code of the routines that handles operation of the Routing Socket. Note the mentioned code, is free for copy and for modification, but it is also protected by the *University of California, Berkeley*[2] copyright .

[2]This product includes software developed by the University of California, Berkeley and its contributors.

Error	Code	Meaning
ELTFULL	87	Label Table is full
EFECNOTPRESENT	88	FEC is not present in the Routing Table
ELTNOTCON	89	Label Table not consistent
EADDFAIL	90	New entry insert operation failed
ENOLABVAL	91	Label not valid
ENOLABPRESENT	92	Label not present
ENOFLAGVALID	93	Flags not valid
EUAFNOTVALID	94	Upper layer Address Family not valid

Table 3.1: System's error concerning the Label Table structure.

3.3 The `lt_msghdr` data structure

The exchange of data through the MPLS Sockets is the same as in the Routing Socket, by messages. An MPLS message consist in a fixed length header followed by a variable length set of data structures, depending on the requesting operation. The fixed length header `lt_msghdr` is shown in figure 3.3.

```
struct lt_msghdr {
        u_short ltm_msglen;     /* to skip over non-understood messages */
        u_char  ltm_version;    /* future binary compatibility */
        u_char  ltm_type;       /* message type */
        int     ltm_fields;     /* bitmask identifying fields in msg */
        pid_t   ltm_pid;        /* identify sender */
        int     ltm_seq;        /* for sender to identify action */
        int     ltm_errno;      /* why failed */
        int     ltm_flags;      /* flags */
};

#define LTM_VERSION     0       /* Up the ante and ignore older versions */
```

Figure 3.3: Header present in every message sent/received through MPLS socket.

The first field of the structure shows the length of the whole message, including the data structures queued to the header. The second field is used to point out to which version of the MPLS Socket the message belongs to. Being this absolutely the first implementation of this solution, the version has been fixed to 0. This field will be useful when new kind of message will be introduced or will be modified the existing messages, indeed modifying the version, the kernel is sudden able to discern if it can process the received message or not.

The `ltm_type` field indicates the type of message, i.e. the operation that will be performed; the possible operations will be described in paragraph 3.4. The `ltm_fields` field indicates which data structures are queued to the header. Every bit of this field indicates, if set, of a precise data structure. In table 3.2 is showed the meaning of every single bit, and the data structure assigned to it. The only exception concerns the `lt_labelstackinfo` data structure.

Flag	**Value**	**Meaning**	**Associated data structure**
LTA_INABEL	0x1	Incoming Label	u_int32_t
LTA_COS	0x2	TE related information	u_int32_t
LTA_FLAGS	0x4	Flag	u_int32_t
LTA_STACK	0x8	Label Stack	struct lt_labelstackinfo
LTA_DST	0x10	Destination Address	struct sockaddr
LTA_NETMASK	0x20	Subnet mask	struct sockaddr
LTA_LRANGE	0x40	Valid label range	struct labelspace
LTA_PRINTBUF	0x80	Pointer to the print buffer	char*
LTA_PRINTLINES	0x100	Number of lines to print	int*

Table 3.2: Flags of the `ltm_fields` field.

This data structure, whose declaration is showed in figure 3.4, contains the label stack that will be putted at the beginning of every packet, its size and the TTL value, whose meaning has been shown in paragraph 2.2. The `ltm_pid` and `ltm_seq` fields

```
struct lt_labelstackinfo {
        u_int16_t       ttl;
        u_int16_t       depth;
        MplsShim        Stack[0];
};
```

Figure 3.4: Structure `lt_labelstackinfo`.

are needed to identify respectively the process that sent the message and the message's sequence. Like Routing Socket, MPLS Socket send the answer to a message in broadcast to all the open MPLS Socket's, by means of the last two described fields, every process can understand if the received message is an answer to a message sent by it, and particularly to which. Finally, the `ltm_flags` field contains all needed flags.

Once a message is sent through an MPLS Socket, the code that handles the messages gathers the data structures queued to the message header, in unique data structure, whose declaration is showed in figure 3.5. This data structure is used as a parameter in the main routines that access/handles the Label Table, and has the advantage to furnish a unique access to all the possible data structures that can be exchanged through the MPLS Socket.

```
struct lt_fieldsinfo {
        int                          lti_fields;
        u_int32_t                    lti_inlab;
        u_int32_t                    lti_CoS;
        u_int32_t                    lti_flags;
        struct lt_labelstackinfo *   lti_stack;
        struct sockaddr          *   lti_dst;
        struct sockaddr          *   lti_netmask;
        struct labelspace        *   lti_range;
        char                     *   lti_bufptr;
        int                      *   lti_lines;
};
```

Figure 3.5: Structure `lt_fieldsinfo`.

While all the data structures are gathered, some integrity checks are also made, for example it is checked if all the data structures necessary to perform the requested operation are present. Then the routines that handles the Label Table and that implements the requested operation are called. The results of the operations, if there is any, are sent back along with the broadcast message sent to all open MPLS Socket.

3.4 Operations through MPLS sockets

Let analyze the various operations permitted through the MPLS Socket. In table 3.3 are showed all the possible operations. In the next paragraphs, describing the various operations will be showed the routine, that access the Label Table, that implements the requested operation.

Message Type	Value	Operation
LTM_GETLRANGE	0x1	Returns the current label range
LTM_SETLRANGE	0x2	Modifies the label range
LTM_ADD_ENTRY	0x3	Adds a label to the Label Table
LTM_ADD_DLENTRY	0x4	Adds a LAB_DEFAULT entry to the Label Table
LTM_REMOVE_ENTRY	0x5	Removes a Label Table entry
LTM_REMOVE_DLENTRY	0x6	Removes a LAB_DEFAULT entry from Label Table
LTM_ADD_FEC	0x7	Adds a FEC to an entry of the Label Table
LTM_REMOVE_FEC	0x8	Removes a FEC from an entry of the Label Table
LTM_GET_ENTRY	0x9	Returns the wholes information concerning an entry of the Label Table
LTM_GET_DLENTRY	0xa	Returns the wholes information concerning a LAB_DEFAULT entry of the Label Table
LTM_PRINT	0xc	Copies the whole Label Table in a buffer

Table 3.3: Operations through the MPLS Sockets.

3.4.1 `LTM_GETLRANGE`

Through this operation is possible to obtain the current valid label range of the system. It has been described in paragraph 2.4 that not all label's value are allowed inside the system. No data structures are needed to be sent along with the operation request, thus the value of the `ltm_fields` field will be null. To this message will be queued a data structure of type `labelspace` containing the requested information, the `LTA_LRANGE` flag of the message header is set, then all is sent to the higher levels, out on all the open MPLS Socket. The routine that implements the operation is `lt_getrange()`, that simply returns the `LabelSpace` structure of the `ltArea` structure of the Label Table. The `mpls_s_output()` routine copies the values in the message sent back on the sockets.

3.4.2 `LTM_SETLRANGE`

Is the counter-party of the previous operation. By this operation is possible to modify the current valid label range. When the Label Table is initialized at bootstrap time, the valid range of tha system is set to the bigger range allowed, that is 17-1048576 (remember that values ranging from 0 to 15 are reserved, while the value 16 is used

as LAB_DEFAULT). In the message, besides the header, there is the labelspace data structure, containing the new label range. Thus the LTA_LRANGE flag is set. The message is forwarded unchanged toward the open MPLS sockets.

The update operation is implemented by the routine lt_setrange(). The routine is able to adjust the furnished values, that is, if the new upper limit is higher than 1048576 then the upper limit is set to the latter value, likewise if the new lower limit is less than 17, then the lower limit is set to the latter value. No update is made if the new lower limit is higher than the new upper limit.

3.4.3 LTM_ADD_ENTRY & LTM_ADD_DLENTRY

The LTM_ADD_ENTRY operation, adds a new entry to the Label Table. All the information concerning this new entry have to be queued to the message header, this information are: incoming label, Class of Service, Label Table Entry's flags, label stack to attach to the packets, FEC associated to the entry and consisting in two sockaddr structures containing the destination address and its netmask (the latter is optional).

All the corresponding flags have to be set in the message header, furthermore, the order which the data structures are queued to the message header is meaningful, particularly right order is the same as the information are listed above.

This can be expressed as general rule for the MPLS Socket: every time several data structures are present in a message, the order that have to be followed to queue the structures is the one obtained running through the table 3.2 from the top to the bottom, adding the corresponding data structure if needed in the current operation.

Note that sockaddr structures used to describe a FEC are general and can contain an address (both a host address or a subnet address) of any Network Layer protocol, if considered just as a bit sequence. Thus, this solution let the present software to be open to any Network Layer protocol.

The LTM_ADD_DLENTRY operation, is very similar to the preceding one, it adds a new entry whose label value is the LAB_DEFAULT. Thus, in the information queued to

the message header no label is present, in the rest part is equal to the `LTM_ADD_ENTRY` operation. The routine that implements both operations is the `lt_add()`, rather complex, because it must be able to manage several different situations. Really, for the `LTM_ADD_DLENTRY` operation, the first step is to call the `lt_add_dl()` routine, that adds the `LAB_DEFAULT` in the furnished data and then the `lt_add()`is called. In figure 3.6 is depicted the flow-chart of the routine. It is important to note that if the label and the FEC of the entry that is going to be added are already present in the Label Table, the insert operation will be changed in an update operation. On the other hand, if only the label is present, but not the FEC, the latter is added to the FecList.

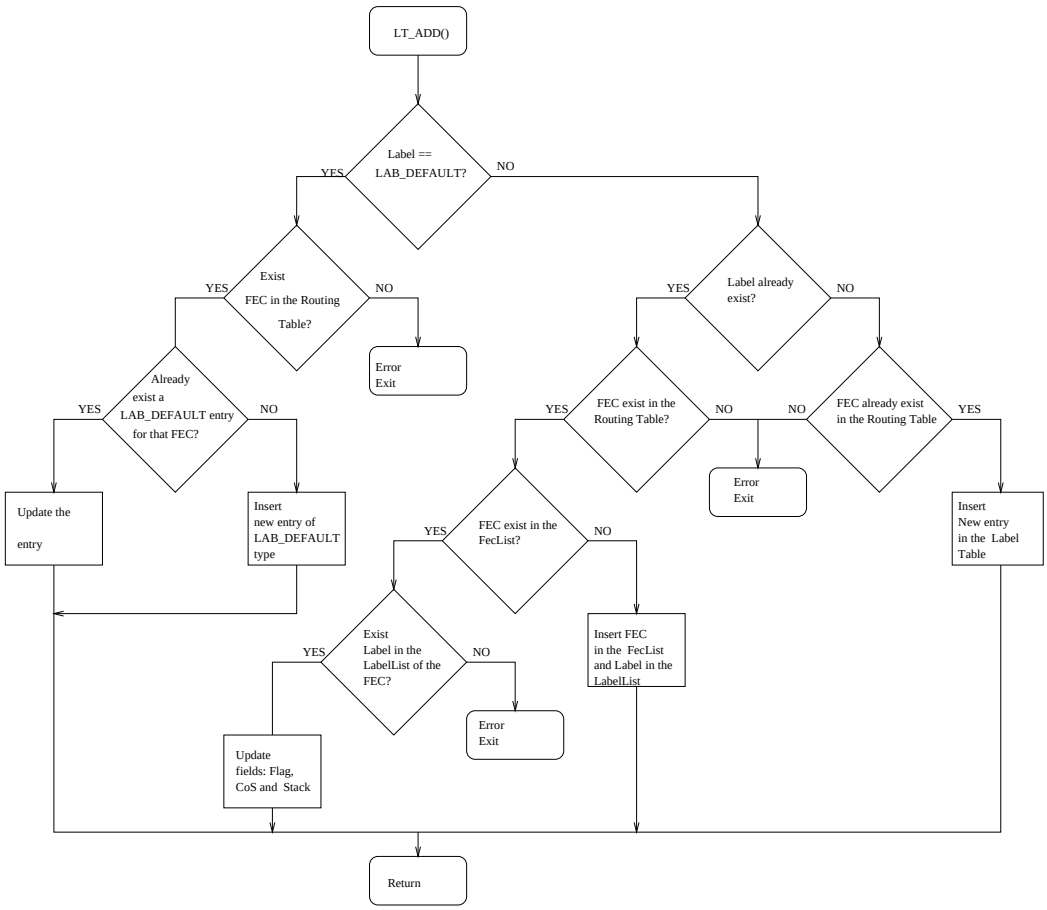

Figure 3.6: Flow chart lt_add() routine.

3.4.4 LTM_REMOVE_ENTRY

This operation allows to remove an entry of the Label Table, starting simply from an incoming label. Indeed, the only information that has to be queued to the message header sent through the MPLS Socket, is the incoming label by which perform the look up to find the entry to remove.

The `LTM_REMOVE_ENTRY` operation is implemented by the `lt_rm_lab()` routine. The routine, first check that the provided label is not a `LAB_DEFAULT`, because this value is not permitted in this operation, then it performs a look up based on the label, to find the entry to remove. To remove the found entry from the Label Table, is a task of the `lt_rm()` routine. The routine have also the task to remove all the references from the various routing tables to the particular entry, leaving the Label Table in a consistent state.

3.4.5 LTM_REMOVE_DLENTRY

The `LTM_REMOVE_DLENTRY` operation is similar to the previous one that removes Label Table entries, while this one removes an entry that has the `LAB_DEFAULT` value as label. In the previous chapters it has been described how in the Label Table can be present several entries with `LAB_DEFAULT` label, thus to find the entry that has to be removed the process is the same as described in paragraph 2.6, performing the look up based on the destination address.

Thus, to perform this operation is necessary to provide a destination address, so in the message sent through the MPLS Socket it must be queued to the header a `sockaddr` structure, containing the destination address, and optionally, another `sockaddr` structure containing the netmask.

The routine that implements this operation is `lt_rm_deflab()`, that as a first step performs a look up of the provided destination address in the appropriate routing table. Note that the `sockaddr` structure contains a field with the Address Family of the address, thus the Network Layer to which the address belongs. By this information, the

look up routine is able to access the right routing table of the Network Layer protocol of the address.

The next step of the routine is the look up of the `LAB_DEFAULT` entry reference in the list of all the references to the entries of the Label Table to which the FEC belongs. Third and last step is to call the `lt_rm()` routine that provides to the effective removal of the entry of the Label Table.

3.4.6 `LTM_ADD_FEC`

This operation allows to add further FEC to the FecList of an existing entry. The data structures to queue to the header of the message sent through the MPLS socket are: the label used find the right entry to which the FEC must be added, the destination address that forms the FEC, optionally equipped with a netmask.

The `lt_add_fec()` routine implements the operation in objects, performing a look up on the Label Table based on the provided label and a look up on the right routing table based on the provided destination address. Once the references to the Label Table entry and routing table entry are obtained, the list of FECs and the list of references to the Label Table are updated.

3.4.7 `LTM_REMOVE_FEC`

This operation is the counter-party of the preceding operation, to which is very similar. The data structures to queue to the message header that is sent through the MPLS socket are the same. Further the routine that implements this operation, the `lt_rm_fec()` follows the same scheme of the `lt_add_fec()`routine, obviously with the right changes updating the references lists.

The only big difference is that if the FEC to remove is the only one present for that entry, then the whole Label Table entry is removed.

3.4.8 LTM_GET_ENTRY

The LTM_GET_ENTRY operation provides in the broadcast answer to the received message, chosen all the information related to an entry, chosen by the incoming label. To the message sent through the MPLS socket it must be queued the label by which perform the look up.

The operation is implemented by the `lt_getentry()` routine, which by means of the `lt_find_by_label()` routine obtains a reference to the searched entry. The `lt_find_by_label()` routine performs a look up on the Label Table, accessing the hash index of the table. The routine that handles the messages arrived through the MPLS socket, the `mpls_s_output()`, provides to queue all the data to the received message, updating the corresponding flags of the header. Then the message is sent back, as already described in previous sections, to all open MPLS socket.

3.4.9 LTM_GET_DLENTRY

This operation is very similar to the previous; the only difference is that the entry whose information are required is of LAB_DEFAULT type. This means that the message sent through the MPLS socket have to carry a FEC, compounded by a destination address and optionally a netmask.

The operation is implemented by the `lt_getdlentry()` routine, which first perform a look up on the right routing table, depending on the provided FEC, then extract from the list of references to the Label Table the one that point to the LAB_DEFAULT entry, finally returns the result of the look up operation.

Also in this case the `mpls_s_output()` routine provides to complete the message with the lacking information and send it back to all open MPLS socket.

3.4.10 LTM_PRINT

This operation provides a dump of all the Label Table. Each entry is copied in a printable format and putted in a buffer to which is provided a reference by the request message sent through an MPLS socket. Along with the pointer to the buffer used to make the dump of the data, the message contains also an integer, used to obtain the number of lines contained in the buffer.

If the pointer of the incoming message has a NULL value, the operation is anyway performed, but nothing is copied in any buffer, but the integer value, at the end of the operation, will contain the same value as it will be obtained performing a normal print operation (i.e. providing a non NULL value). This feature is useful to calculate the how big has to be the print buffer that must be allocated.

Indeed, the routine that implements the operation does not make any integrity check on the buffer, it suppose that the buffer is big enough to contains the whole print. With the described feature instead, is possible to make a first request setting the pointer to NULL, obtaining the number of lines that will be produced, thus to allocated enough space and then make a real print request.

In this way is possible to allocate a buffer whose size is the number of printable lines multiplied the max size of one line on the screen (generally 80). Of course in this way the buffer is oversized, but there is no risk that the print operation not overflows the buffer limit, soiling memory not belonging to the buffer. To perform the operation in object the `lt_print()` routine is called, that scans the whole Label Table.

The real print operation in the buffer is made by the `lt_print_entry()` routine, which, in the present implementation understands only FECs of IPv4 type, translating all the addresses in the typical *dot notation* of that protocol:*"aaa.bbb.ccc.ddd"*.

Extensions that made the `lt_print_entry()` routine able to translate in a printable format also other kinds of network Layer addresses are for future expansions of the code.

In table 3.4 are listed all the prototype of the routines called by `mpls_s_output()`, which realize the various operations possible through MPLS Sockets. These routines represents the low level access interface to the Label Table.

`void lt_setrange( LabelRange );`
`LabelSpace* lt_getrange( void );`
`LabelTable* lt_add( LabelTableEntry* );`
`LabelTable* lt_add_dl( LabelTableEntry* );`
`LabelTable* lt_add_fec( LabelTableEntry* );`
`void lt_print( char*, int* );`
`int lt_rm_lab( u_int32_t );`
`int lt_rm_deflab( struct sockaddr*, struct sockaddr* );`
`int lt_rm_fec( struct lt_fieldsinfo* );`
`LabelTableEntry* lt_getentry( u_int32_t );`
`LabelTableEntry* lt_getdlentry( struct sockaddr*, struct sockaddr* );`

Table 3.4: Low level access routine to the Label Table.

Chapter 4

MPLS Layer in the Protocol Stack

4.1 Codification of the Label Stack

How described in the previous sections of the present document, the label stack is formed by a set of "label stack entries" (figure 4.4). The label stack entries are putted in a packet after the Data-Link Layer header, but before of any Network Layer header. As defined in [MPLS-SHIM], each entry is a 32 bit sequence, split in several fields, how is showed in figure 4.1. Let us give a description of the various fields:

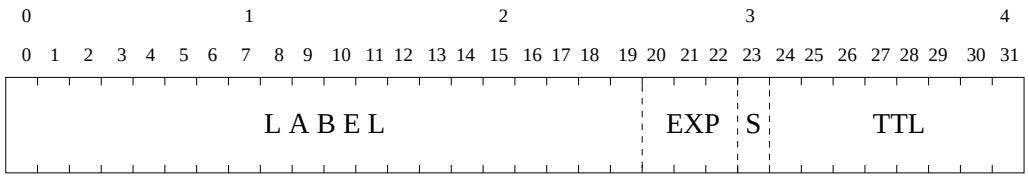

Figure 4.1: Structure of a Label Stack Entry.

1. - Label, 20 bits. Is the real label of the entry.

2. - Exp, experimental use, 3 bits. This field is completely not use din this implementation. The IETF (Internet Engineering Task Force) has reserved the use of this bits to experimental purpose, without defining further their function.

3. - S, bottom of the stack, 1 bit. If set, this bit means that the entry is at the bottom of the label stack.

4. - TTL, Time-To-Live, 8 bits. Used to bound wasting effects of possible loops, its meaning is similar as the homonymous field present in the IPv4 protocol.[1]

When a labeled packet is received, the value of the label at the top of the stack is used to access the Label Table (figure 4.2). The result of the last operation is used to retrieve the following information:

1. - Next host toward which the packet has to be forwarded.

2. - The operations to perform on the label stack, whose description is in the previous sections but that will be completed in the next paragraphs.

The packet once it has been processed by MPLS, can be forwarded on the network if necessary. We talk of "Label Swapping" when procedures 1 and 2 are used in order to forward the packet.

As mentioned in the previous chapters, some label values have a particular meaning (i.e. are reserved):

- The value of 0 represents the *"IPv4 Explicit NULL Label"* . This value is allowed only at the bottom of the stack. When present it means that the label stack has to be removed, and the forwarding operations have to be performed based on the IPv4 header of the packet.

- The value of 1 represents the *"Router Alert Label"*. This value is allowed anywhere in the label stack but not at the bottom of it. When a received packet contains

[1] Refer to [STE-1] for a detailed description of the TTL field in IPv4.

Label Name	Value
LAB_IP4_EXPLICIT_NULL	0
LAB_ROUTER_ALERT	1
LAB_IP6_EXPLICIT_NULL	2
LAB_IMPLICIT_NULL	3
LAB_RESERVED4	4
LAB_RESERVED5	5
LAB_RESERVED6	6
LAB_RESERVED7	7
LAB_RESERVED8	8
LAB_RESERVED9	9
LAB_RESERVED10	10
LAB_RESERVED11	11
LAB_RESERVED12	12
LAB_RESERVED13	13
LAB_RESERVED14	14
LAB_RESERVED15	15
LAB_DEFAULT	16
LAB_LOWHARDLIMIT	17
LAB_UPHARDLIMIT	0x0FFFFF

Table 4.1: Reserved label's values.

this label value at the top of the stack, it is delivered to a local software module for processing. Remember that in a router the forwarding operations are implemented in hardware; this label value tells to the router that the operation that has to be performed on the packet is slightly more complex than a simple forwarding operation, thus the packet has to be processed by a software module able to handle such kind of operations. The real operation is determined by the next label in the stack. However, if a further forwarding is done, the *Router Alert Label* value has to be re-inserted at the top of the stack before the forwarding operation. The use of this label is similar to the use of the "Router Alert Option" in IP packets ([STE-1]).

- A value of 2 represents the *"IPv6 Explicit NULL Label"*. This value is legal only at the bottom of the stack. When present it means that the label stack has to be removed, and the forwarding operations have to be performed based on the IPv6

header of the packet.

- A value of 3 represents the *"Implicit NULL Label"*. This is a label value that an LSR can assign and distribute, but that never can be present in the label stack. When an LSR would otherwise replace the label at the top of the stack with a new label, but the new label is *Implicit NULL,* the LSR will pop the stack instead of doing the replacement. This situation may happens for example in the penultimate LSR of an LSP. If the POP operation is not performed by the penultimate LSR, then the egress LSR of an LSP will perform two look up operation, a first one to check the label at the top of the stack, once the POP is done, a second look up is performed to check the label that became the new top of the stack (or if no other labels are present, the packet is delivered to the Network Layer that check the corresponding header of the packet). Instead, performing the penultimate LSR popping, the egress LSR of an LSP have to check only one label (or only the network Layer header). For a detailed description of this feature, refer to *"Penultimate Hop Popping"* in [MPLS-ARC].

- Values that range from 4 to 15 are reserved by IETF for future use.

Any other label value can be used to access the Label Table (figure 4.2). In the table 4.1 are resumed all the label value with a particular meaning. Note that the last two value of the list (`LAB_LOWHARDLIMIT & LAB_UPHARDLIMIT`) are not reserved values but they are just the range limit of the usable labels.

In the next sections the IPv4 protocol is used in place of a generic Network Layer protocol, because, as mentioned in the previous chapters, it is the only one that already is able to interact with MPLS in the present implementation.

4.2 Label Stack Encapsulation

The label stack is encapsulated in every sent packet, after the Data-Link Layer header and before the Network Layer header. In the present work, a packet that is putted in

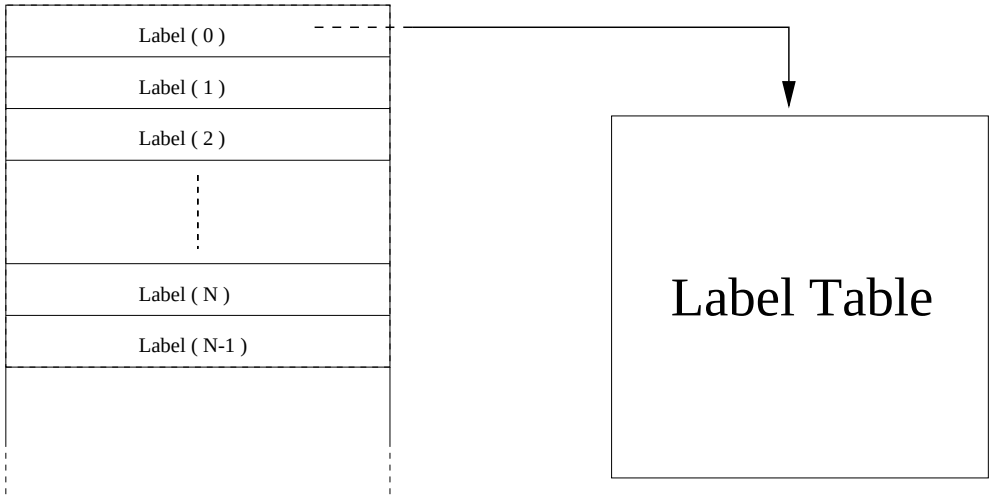

Figure 4.2: Access to the Label Table.

the net has the structure showed in figure 4.3.

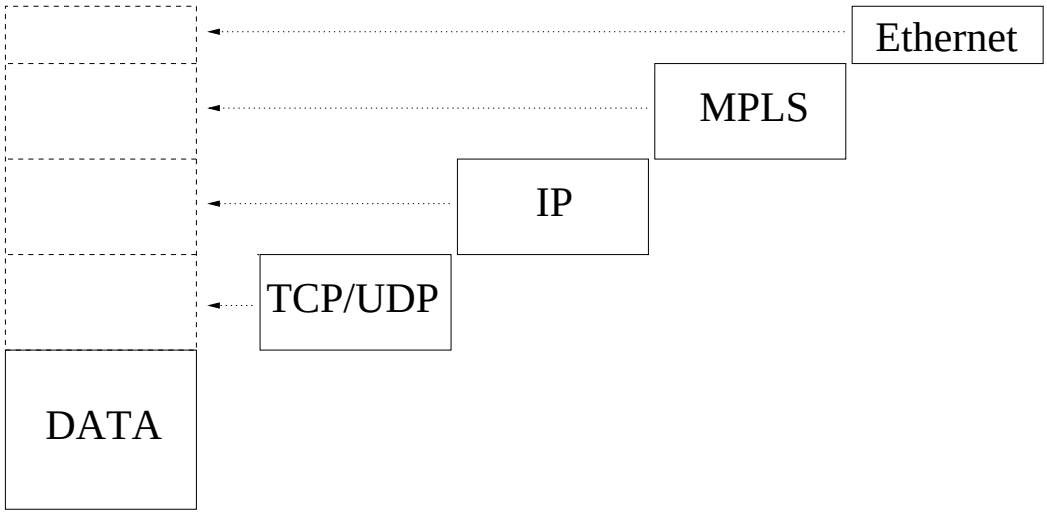

Figure 4.3: Shim Header encapsulation.

Exists a *"Shim Header"* between the Data-Link header and the Network Layer header. Once a packet is forwarded with a Shim header, all the successive forwarding operations are performed based of this new header (refer to figure 1.5), the Network Layer no longer involved in forwarding (except particular cases that are described further). The general structure of the Shim header is showed in figure 4.4.

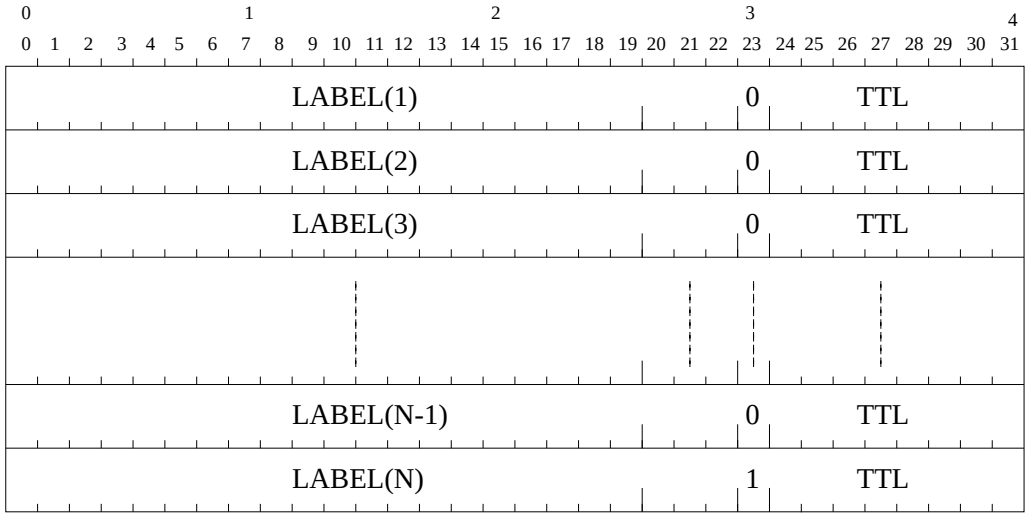

Figure 4.4: Label Stack structure.

Note that the S bit is only set at the bottom of the stack. It will be described how the meaningful TTL value is the one at the top of the stack. When the last label is removed, the packet is forwarded by using the network Layer header.

In the next sections it is assumed that an LSR is able to:

a) receive an unlabeled IPv4 datagram

b) add a label stack to the datagram, and

c) perform the forwarding of the labeled packet

where IPv4 datagram means an IPv4 packet along with its header. Two important problems have to be faced after introducing the new layer in the protocol stack:

- Fragmentation
- Process of the TTL field

4.2.1 Fragmentation

Each interface has a max permitted length, MTU (Maximum Transmission Unit), that a packet must respect in order to be forwarded by the interface. This limit is typically due to hardware factors that lead to a no reliable transmission if data unit is longer than the limit. If an Ethernet interface is used the MTU is, including the MPLS header, 1500 bytes. What may happens is that receiving an unlabeled packet and adding the label stack to it, its total length will now exceed the MTU of the interface through which it has to be forwarded. Otherwise, what can happens is that a received labeled packet exceed the MTU of the interface on which has to be forwarded, due to the insertion of more labels at the top of the stack. The former case can be easily solved, the latter need some other consideration.

before to go on, let us introduce some useful terms:

- Frame Payload: Is the of a Data-Link frame, excluding any Data-Link Layer headers or trailers.
- Conventional Maximum Frame Payload Size: Is the maximum Frame Payload size allowed by data link standards. For example, the Conventional Maximum Frame Payload Size for Ethernet is 1500 bytes.
- True Maximum Frame Payload Size: Is the maximum size frame payload which can be sent and received properly by the interface hardware. For example, it is believed that most Ethernet equipment could correctly send and receive packets carrying a payload of 1504 or perhaps even 1508 bytes at least, as long as the Ethernet header does not have an 802.1Q or 802.1p field.
- Effective Maximum Frame Payload Size for Labeled Packets: This is either the Conventional Maximum Frame Payload Size or the True Maximum Frame Payload Size, depending on the capabilities of the equipment on the data link and the size of the Data-Link header being used.

- Initially Labeled IPv4 Datagram: It is a received unlabeled IPv4 Datagram to which a label stack is added before forwarding it.
- Previously Labeled IPv4 Datagram: Is an IPv4 Datagram which had already been labeled before it was received by a particular LSR.

The Initially Labeled Ipv4 Datagram size is a parameter that can be set by an LSR to a no negative value. If it is set to a null value then it has no effect, on the other hand, if it is set to a positive value, greater than 0, it is used in several different ways.

Particularly if:

1. An unlabeled IPv4 datagram is received, and
2. that datagram does not have the DF[2] bit set in its Ipv4 header (note that an IPv4 Datagram has the DF bit not set when it is the only fragment or the last of a set of fragment belonging all to the same packet), and
3. that datagram needs to be labeled before being forwarded, and
4. the size of the datagram (before labeling) exceeds the value of the parameter,

then

1. the datagram must be broken into fragments, each of whose size is no greater than the value of the parameter, and
2. each fragment must be labeled and then forwarded.

For example, if this configuration parameter is set to a value of 1488, then any unlabeled IPv4 datagram containing more than 1488 bytes will be fragmented before being labeled. Each fragment will be capable of being carried on a 1500-byte Data-Link, without further fragmentation, even if as many as three labels are pushed onto its label stack. Setting this parameter to a non-zero value allows one to eliminate all fragmentation of Previously Labeled IPv4 Datagrams, but it may cause some unnecessary fragmentation

[2]For a detailed description of the meaning of this bit refer to [STE-1].

of Initially Labeled IPv4 Datagrams. In the present work, this parameter is statically defined, but this not excludes that it may be controlled by the Label Distribution Protocol in future releases. Particularly, in the `ltArea` data structure, there is the `lt_mmsd` parameter that contains the maximum allowed size that the label stack can reach, this value (multiplied by 4, the size expressed in byte of every label stack entry) is deducted from the MTU of the output interface to obtain the maximum size of an Initially Labeled IPv4 Datagram.

When are labeled IPv4 Datagrams too big? A labeled IPv4 datagram whose size exceeds the Conventional Maximum Frame payload Size of the Data-Link over which it is to be forwarded may be considered to be too big. A labeled IPv4 datagram whose size exceeds the True Maximum Frame Payload Size of the Data-Link over which it is to be forwarded must be considered to be too big.

A labeled IPv4 datagram which is not too big must be transmitted without fragmentation.

If a labeled IPv4 datagram is too big, and the DF bit is not set in its IPv4 header, then the LSR may silently discard the datagram. If the LSR chooses not to discard a labeled IPv4 datagram which is too big, or if the DF bit is set in that datagram, then it must execute the following algorithm:

1. Strip off the label stack entries to obtain the IP datagram.

2. Let N be the number of bytes in the label stack, if the IPv4 datagram does not have the DF bit set in its IPv4 header:

   (a) concert it into fragments, each of which must be at least N bytes less than the Effective Maximum Frame Payload Size

   (b) Prepend each fragment with the same label header that would have been on the original datagram had fragmentation not been necessary, and the forward the fragments

3. If the IP datagram has the DF bit set in its IPv4 header:

(a) The datagram must not be forwarded

(b) If possible, transmit the ICMP Destination Unreachable Message to the source of the discarded datagram.

Thus, if in a forwarding operation, there is no space for the MPLS header in the packet, it has to be discarded or fragmented. If fragmentation take place, this will cause forwarding operation to take more time. Anyway, these are not common cases, thus they can also be not considered, discarding silently the packet. The best solution is to use a Path MTU Discovery mechanism, but it is not implemented in the present work.

4.2.2 TTL Field

Before to go on, let us introduce some definitions:

- The *incoming TTL* of a labeled packet is defined to be the value of the TTL field of the top label stack entry when the packet is received.
- The *outgoing TTL* of a labeled packet is defined to be the larger of:
  - one less than the incoming TTL,
  - 0

If the outgoing TTL of a labeled packet is 0, then the labeled packet must not be further forwarded. The packet's lifetime in the network is considered to have expired.

Depending on the label value in the label stack entry, the packet may be simply discarded, or it may be passed to the appropriate "ordinary" Network Layer for error processing.

Note that the outgoing TTL value is a function solely of the incoming TTL value, and is independent of whether any labels are pushed or popped before forwarding. There is no significance to the value of the TTL field in any label stack entry which is not at the top of the stack.

In the present implementation, that is hooked only to IPv4 as Network Layer protocol, when a initially labeled packet, that is arriving from Network Layer, the TTL field

of the entry at the top of the stack is set to the same value contained in the TTL field of IPv4 header of the packet (figure 4.5). It is assumed the if the TTL field of IPv4 header has to be decremented, as a result of processing IPv4 header, this is already done.

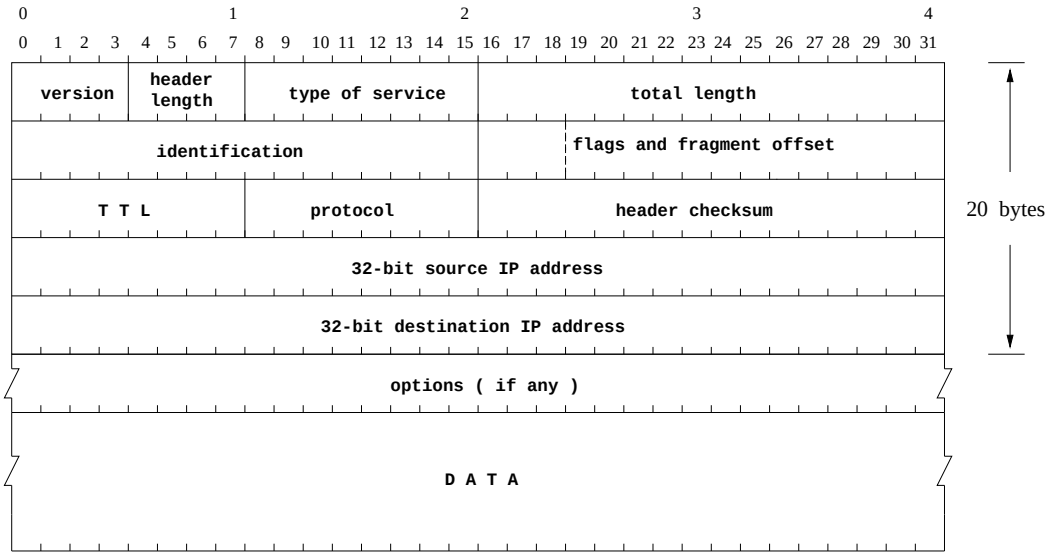

Figure 4.5: IPv4 Datagram.

Once the last label stack entry is removed from a packet, as result of a POP operation, then the value of the IPv4 TTL field should be replaced with the outgoing TTL value, this is the solution adopted in this implementation. Of course, after the update of the TTL field the checksum of IPv4 header must be recalculated.

When a packet traverses an LSP, it should get out of the LSP with the same TTL value as it would have if it traversed the same hop sequence without using MPLS switching.

There may be situations where a network administration prefers to decrement the IPv4 TTL by one as it traverses an MPLS MPLS domain, instead of decrementing the IPv4 TTL by the number of LSP hops within the domain. This is the reason why a packet traversing an MPLS domain *"should"* decremented by the number of hops traversed by the packet.

4.3 Forwarding operations

Let us describe, once again, the operations performed on an incoming packet of a host that does not run MPLS. When a packet is received, the procedure the handles the input of the interface by which the packet is arrived, provides to put the packet in the right Network Layer input queue (every Network Layer protocol has a different input queue). Let us consider only the IPv4 case, the packet is putted in its input queue, then a software interrupt is generated. Since, hardware interrupts have a higher priority than software interrupts, many packets can be queued before the software interrupt really starts.

Once it starts, the `ipintr()` function removes the packets from the `ipintrq` queue until it becomes empty, passing all the packets to the `ip_input()`routine, that handles all the input operations concerning IPv4. On the destination host, IPv4 reassembles packets in datagrams delivered directly to the higher layer (no queues are present passing from Network Layer to the higher Transport Layer).

On the other hand, if packet does not have reached its destination, `ip_input()` delivers the packets to the `ip_forward()` routine is the host is configured to work as router (later will be described how these operations can become faster). Both the Transport Layer protocols and `ip_forward()`, delivers the output packets to the `ip_output()`routine, that completes the IPv4 header, chose an output interface, fragments the packets if necessary and then the resulting packets are delivered to the output interface that provides to forward them on the network (figure 4.6).

Introducing the MPLS Layer the architecture changes, how is shown in figure 4.7.

Labeled packets, arriving from the network are delivered to the MPLS Layer, through a queue system, how for the Network Layer. MPLS Layer, process the packet and then packets are delivered to an output interface if they have to be forwarded, else, if the label stack becomes empty, packets are delivered to IPv4.

A packet outgoing from the `ip_output()`routine, if necessary, i.e. if MPLS is required, is delivered to the output routine of MPLS, which provides to insert the label

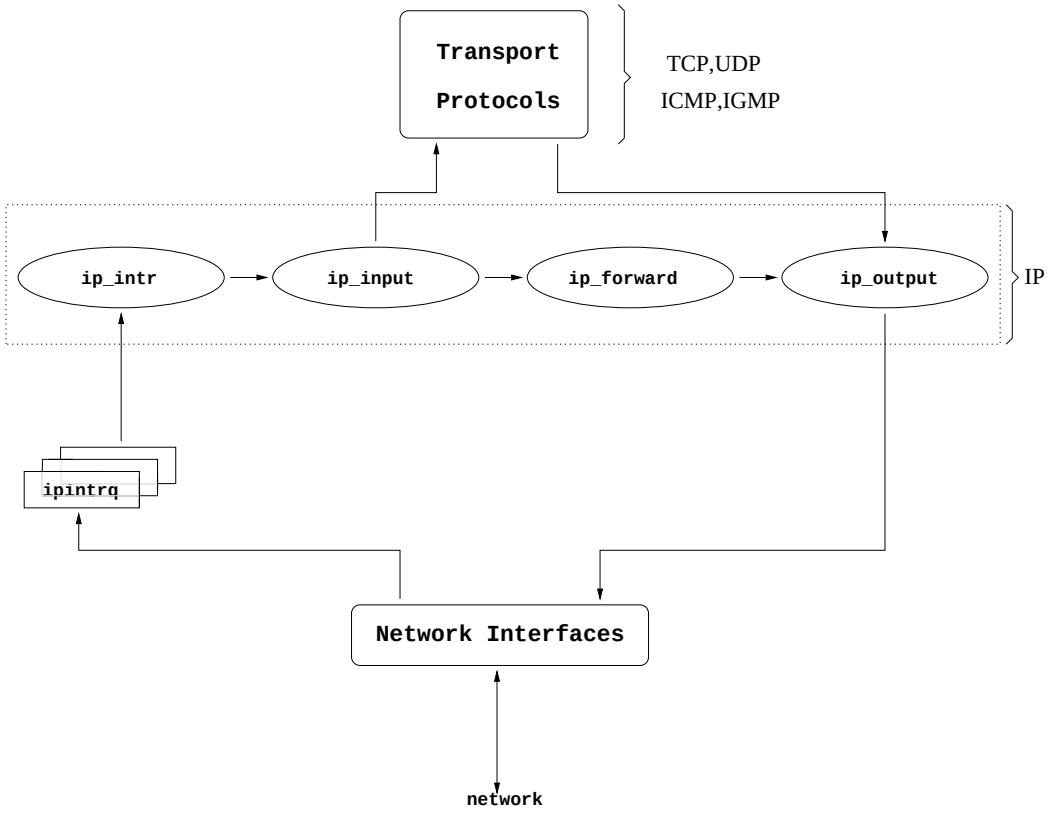

Figure 4.6: Packet's Network Layer process (particularly IPv4).

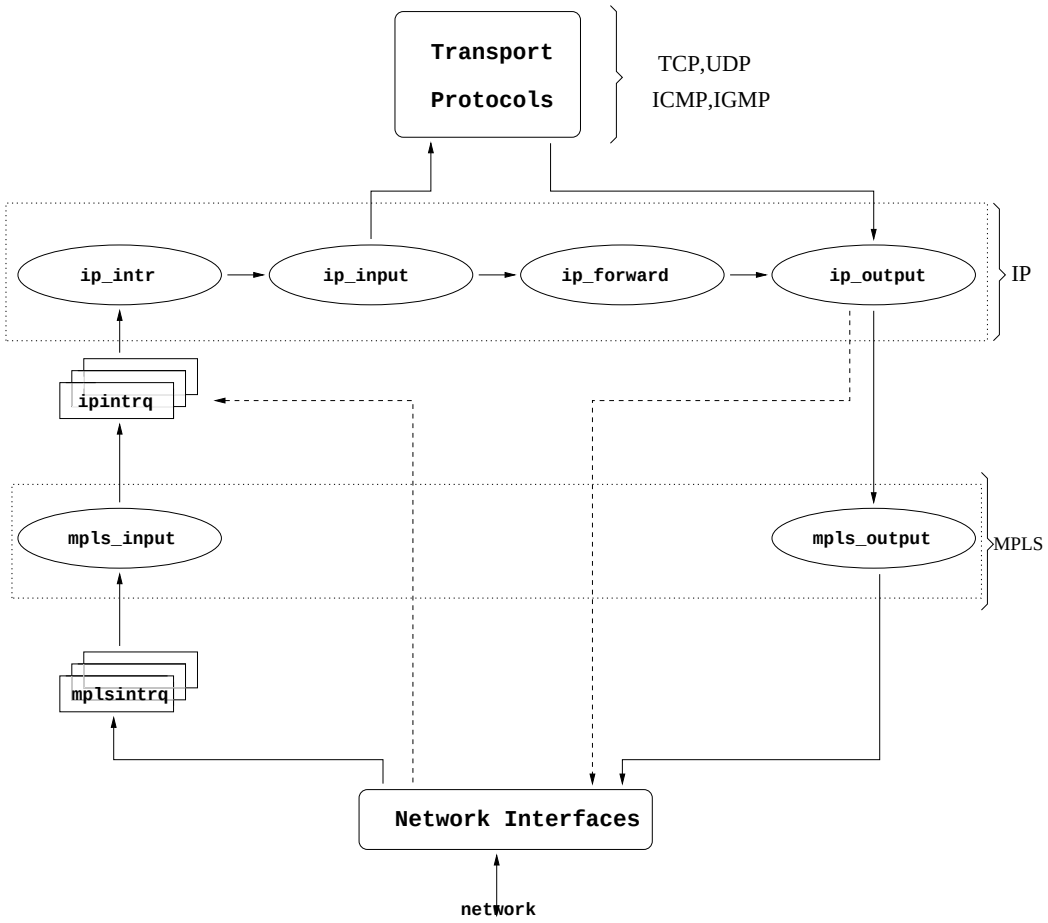

Figure 4.7: Packet's process after introduction of MPLS.

stack at the top of the packet and then puts the packet on the output interface.

The Ethernet encapsulation contains also a *type* field, which indicates the kind of payload, thus the protocol to which the packet has to be delivered once it is received (figure 4.8 and 4.9). When the Ethernet output routine is called, a parameter indicates the family to which belongs the protocol the made the call, depending on this information the type field is opportunely set. In this manner the Ethernet input routine, to which the packet arrives, is able to deliver the packet to the right protocol. Along with MPLS a new protocol family is been introduced, the AF_MPLS family, to which belongs to possible Ethernet type values:

- ETHERTYPE_UMPLS for MPLS unicast packets
- ETHERTYPE_MMPLS for MPLS multicast packets

Remark that multicast traffic is not been considered in the present implementation.

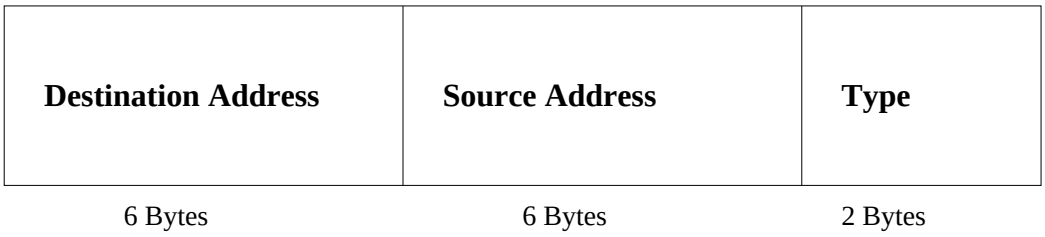

Figure 4.8: Ethernet Header Format.

4.3.1 Input operations

Let us consider IPv4, an unlabeled packet arriving from the net, is queued by `ether_input()` in the IPv4's input queue, and a software interrupt is generated. When the interrupt is handled, the packet is removed from the queue and [processed by `ip_input()` (figure 4.6). To speed up the forwarding operation of packets those are not addressed to the particular host, the `ipflow_fastforward()`routine used. This routine is directly called (i.e. not by means of a software interrupt), and the packet delivered to it without using a queue, in this way the call operation is very fast. The `ipflow_fastforward()`

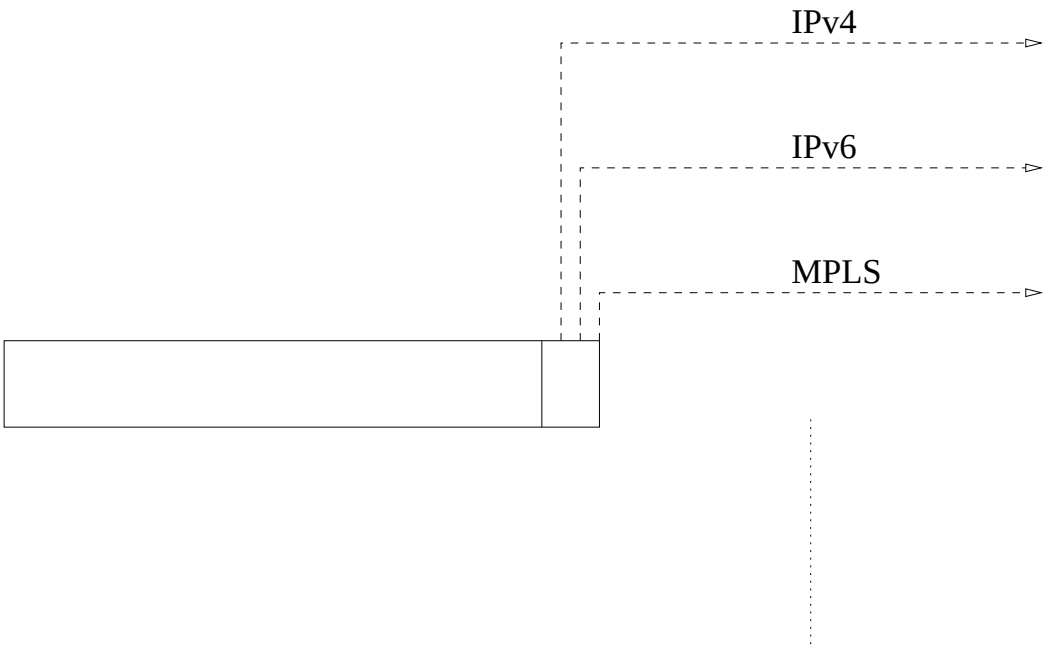

Figure 4.9: Ethernet Packet Demultiplexing.

routine uses a small cache containing the story of the last few received packets, and is very simple and fast respect the `ip_input()` routine, simple checking if the routing information can be retrieved in the cache, instead of accessing the routing table. If the needed information are in the cache, the routine success, and suddenly delivers the packet to the output interface. When this is not the case, the routine fails and the packet is putted in the input queue of IPv4, as described above (figure 4.10).

If MPLS is present, the labeled packets are passed to a routine similar to the fast_forwarding of IPv4. A labeled packet that traverses an MPLS domain, passes over several LSR, the most frequent operation performed on the packet is the SWAP operation. The PUSH and POP operations are performed few times, typically at the ingress and egress LSRs. The `mpls_fastswap()`routine optimized to perform only SWAP operations, directly called by the interface input routine, reduces the average latency time of a packet that has to be just forwarded. If the operation to perform on the labeled packet is not a Swap operation, the packet is delivered to the `mpls_input()` routine (figure 4.11).

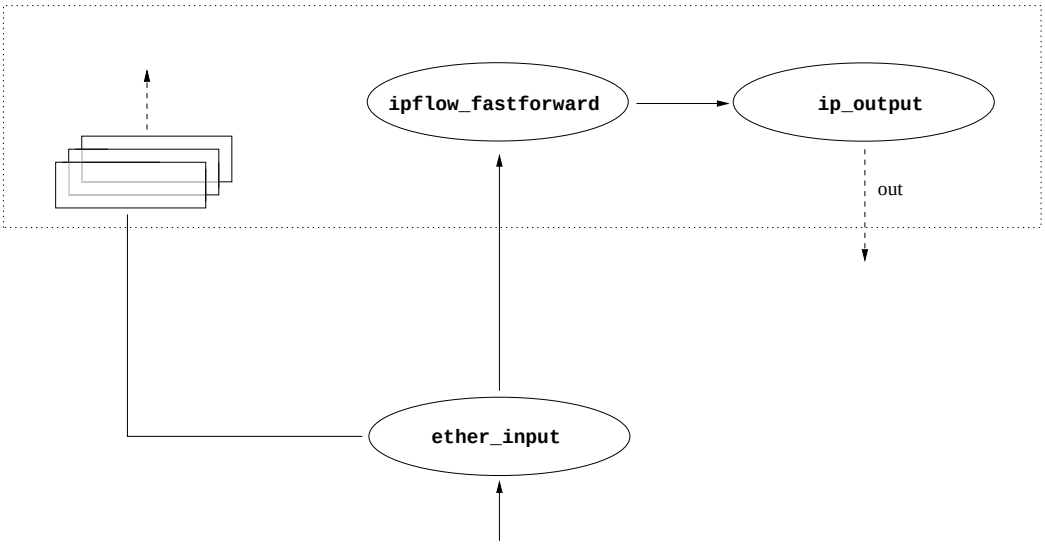

Figure 4.10: IPv4 fast forwarding.

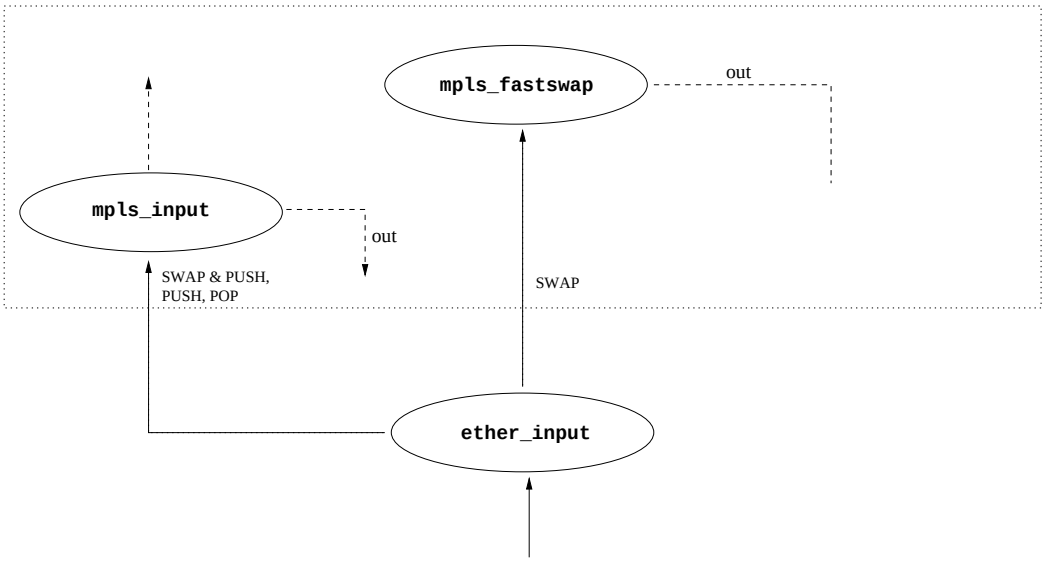

Figure 4.11: MPLS fast swap.

This routine handles all the operations that `mpls_fastswap()`is not able to handle:

- PUSH, POP, SWAP & PUSH operations on the label stack.
- Special label management.
- Error conditions (for example expired TTL).

A packet processed by the `mpls_input()` routine, can be delivered to different Network Layer protocols (only IPv4 case really implemented). This can be due to:

- The packet has reached its final destination.
- The packet has at the top of the label stack the value *"IPv4 Explicit NULL Label"* or *"IPv6 Explicit NULL Label"*.
- The label that has to be replaced is an *"Implicit NULL Label"*
- The incoming TTL value is null.
- A pop operation is performed and the label stack becomes empty.

The `mpls_input()` is called by `ether_input()` by means of the software interrupt mechanism, only if the `mpls_fastswap()` routine fails.

4.3.2 Output operations

Both the output routines of IPv4 and MPLS, once they have processed a packet, deliver it to the output routine of the interface, that provide to forward it on the net (figure 4.12).

If the packet has to be labeled, the `ip_output()` routine provides to deliver the packet to the `mpls_output()` routine that inserts a label stack and then forwards the packet to the output interface. If the packet does not have to be labeled, the `ip_output()` routine delivers the packet directly to the output interface, not involving the MPLS Layer. Remark that if a packet is not labeled , its forwarding at the next

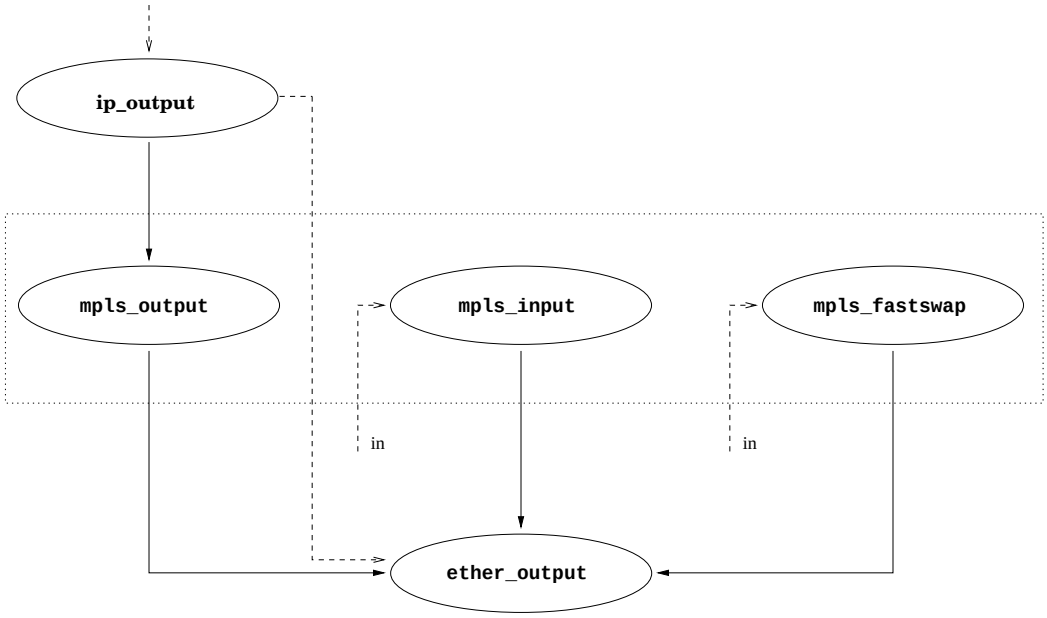

Figure 4.12: Packets output.

hop is done at Network level (the input routine of the input interface of the next hop will recognize that the packet pertains to IPv4 protocol). A packet that is delivered to `ip_output()` can arrive from :

- the Transport Layer , or
- from the `ip_forward()` routine (figure 4.7).

When a labeled packet arrives from the net, the label at the top of the stack is used to access the Label Table, so to retrieve the operation to perform on the label stack of the label stack and the next hop toward the packet must be forwarded. A packet that arrives from the IPv4 Layer, is not labeled, therefore the Label Table is not directly accessed, but through the routing table, how described in section 2.6 *"Access to the Label Table"*.

4.4 Utilized data structures

Each label stack entry (figure 4.1), is an MplsShim structure, whose definition is shown in figure 4.13. In table 4.2 are listed the Macros defined in order to access this structure.

```
typedef struct _shim_fields {         /* For single field access */
#if BYTE_ORDER == LITTLE_ENDIAN
       u_int32_t shimLabel_h:16,/* first 16 bits */
       shimS:1,/* bottom of stack */
       shimExp:3,/* Experimental */
       shimLabel_l:4,/* last 4 bits of label */
       shimTTL:8;/* time to live */

#elif BYTE_ORDER == BIG_ENDIAN
       u_int32_t shimLabel:20,/* actual label */
       shimExp:3,/* Experimental */
       shimS:1,/* bottom of stack */
       shimTTL:8;/* time to live */
#else
#error  "Please fix <asm/byteorder.h>"
#endif
} Mpls_Shim_Fields;

typedef union _shim {
       u_int32_t  Shim;/* for when a 32-bit long is more convenient */
Mpls_Shim_Fields  Shim_F;
} MplsShim;
```

Figure 4.13: MplsShim definition.

These Macros allow to handle the entries of label stack, without worry about the byte order used by the system to store data, the entries are maintained always in Network Byte Order.

The `mpls_output()` routine, receives from the Network Layer, unlabeled packets to which a label stack is added if there is one assigned to the Fec to which the packets belongs (packets pertaining to the same Fec have the same label stack). Once the packets are labeled, they are delivered to the output interface that provides to forward them on the net. If in the Label Table there is not any entry related to the Fec to which the packet belongs, the packet is delivered to the output interface unlabeled, and Network Layer routing is performed in the next hops, the MPLS Layer has no effects

Macro Name	Operation
MplsSetLabel (a, label)	Assigns the value of *label* to the shimLabel field of *a*
MplsGetLabel(a)	Returns the value contained in the shimLabel field of *a*
MplsSetS(a)	Sets the shimS field of *a*
MplsClearS(a)	Clears the shimS field of *a*
MplsGetS(a)	Returns value contained in the shimS field of *a*
MplsSetTTL(a, ttl)	Assigns the value of *ttl* to the shimTTL field of *a*
MplsGetTTL(a)	Returns the value contained in the shimTTL field of *a*
MplsSetExp(a, exp)	Assigns the value of *exp* to the shimExp field of *a*
MplsGetExp(a)	Returns the value contained in the shimExp field of *a*

Table 4.2: Macros to access the MplsShim data structure.

in this case.

In figure 4.14 is shown a high level flow-chart, that describes how the `mpls_input()` routine works.

When MPLS drops a packet? A packet is dropped only when its label at the top of the stack is not present in the Label Table, or it is not a legal value (table 4.1). These situations are due to hardware errors or label distribution protocol errors. Packets with an expired TTL are not dropped but delivered to the higher level protocol which handles the error condition (the higher level may also decide to drop the packet).

To a host can never arrive an unlabeled packet belonging to MPLS (i.e. with the Ethernet type field set to an MPLS value), because once the last label entry is removed from a packet it is forwarded at Network level. This is the case when a packet leaves an MPLS domain. Nevertheless, a packet that leaves an MPLS domain can meet another MPLS domain traversing the network, thus it can be labeled again and forwarded again using MPLS switching.

To send data using MPLS on an end host, the user have to open a TCP or UDP socket and to set the MPLS socket option. This option has been added to let the user decide himself whether to use the MPLS or not in forwarding his packet. This guarantees that the packets do not leave the host using MPLS switching, but does not ensure that traversing the net the packets will be not labeled. The MPLS socket option has only a local significance.

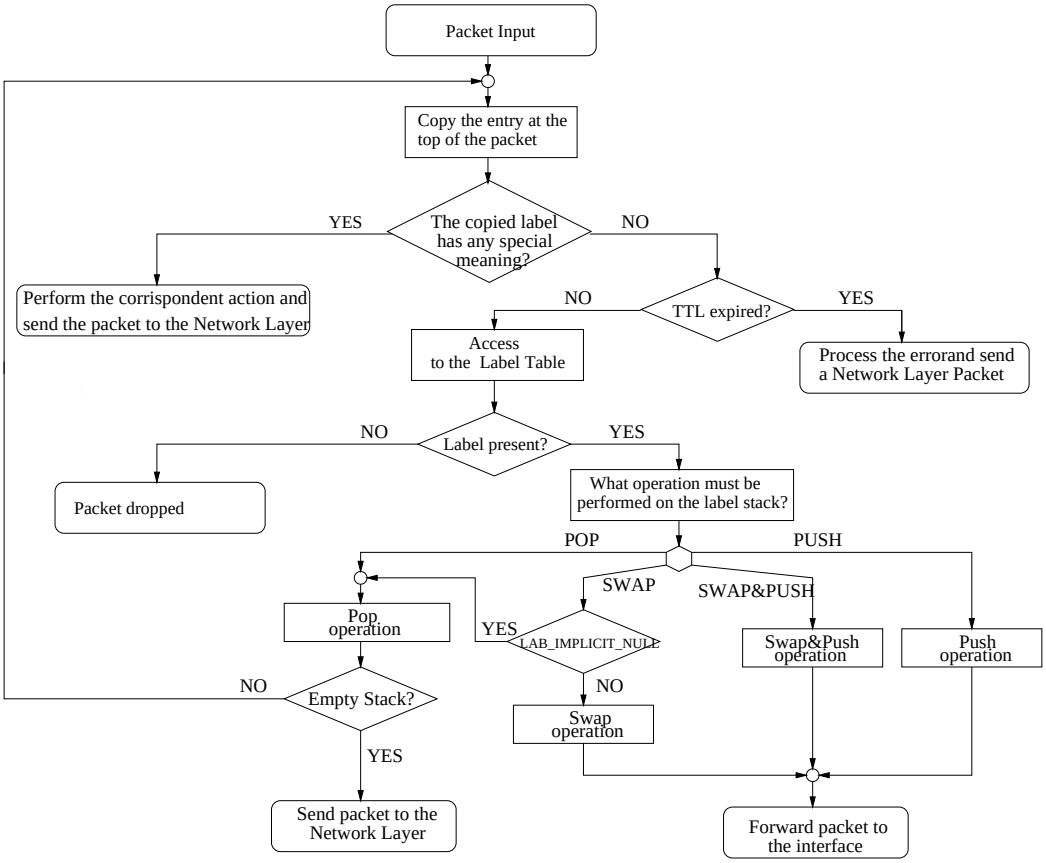

Figure 4.14: Flow-chart of the `mpls_input()` routine.

The kernel's files modified, and the changes brought in order following the specification described in this chapter are listed in table 4.3.

File	**Modified Routines**	**Description**
/sys/netinet/tcp_output.c	`tcp_output()`	If MPLS is enabled it has to inform `ip_output()` that the socket by which the packets are sent uses MPLS.
/sys/netinet/ip_output.c	`ip_output()`	If MPLS is enabled, and exist an LSP, it have to deliver the packets to `mpls_output()`.
/sys/netinet/ip_input.c		If a forwarding operation is performed, and the `RTF_MPLS` bit of the routing entry is set, the `ip_output()` have to deliver the packet to the MPLS Layer.
/sys/net/if_ethersubr.c	`ether_output()`,`ether_input()`	The routines have to be able to handle also MPLS packets.
/sys/sys/uipc_socket.c	`sosetopt()`,`sogetopt()`	Respectively sets and returns the socket's options.
/sys/sys/socket.h		The Address Family `AF_MPLS`, the Protocol Family `PF_MPLS` and the `SO_MPLS` socket option are defined.

Table 4.3: Modified kernel's files.

Chapter 5

Testing

Our tests have taken place on an isolated LAN (figure 5.1), formed by two end hosts (A and C) and a host configured to work as router (C).

The primary target was to check the operating state of an MPLS network; secondary target was to have a measure, as tests result, of how much is the speed up obtained using the MPLS label switching instead of traditional Network Layer IPv4 forwarding.

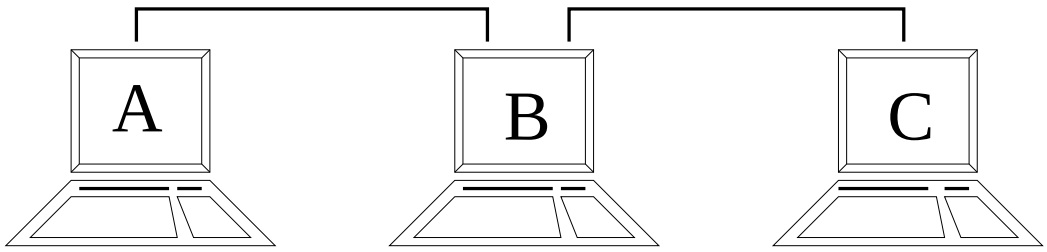

Figure 5.1: LAN used for the test.

In our tests, it was not considered the effective forwarding time, but the latency time of the packet in the layers higher than the Data-Link Layer, that is the latency time of the packet respectively in the IPv4 layer and in the MPLS layer. The effective forwarding time is given by the latency time to which is added the time spent by the input and output routines of the Data-Link Layer to process the packet. As a final measure, it was taken the average time of sufficient high number of packets.

5.1 Testing technique specifications

To measure the latency time for both IPv4 and MPLS, the time-stamps were sampled in the same points for both cases, and as much as possible on the boundary that divide the Data-Link Layer from the upper layer protocol.

Particularly, as showed in figure 5.2, a first timestamp is fixed exiting the `ether_input()` routine (T1), while a second times-tamp is fixed at the beginning of the `ether_output()` routine (T2). The latency time is just the difference between the two values (T2 - T1). To fix the time-stamps the `rdtsc()` function is been used; the function returns the value of a particular counter register of the processor, which is increased at each clock tick. The estimated latency time is given by the number of clock's tick (T2- T1) multiplied the time that elapses between two consecutives clock's ticks (in a CPU at 400 Mhz this is 1/(400*1000*1000) sec.).

Due to the poor significance of a test on only one packet, a high number of packets have been considered, stopping the test once the measure on a new packet does not introduce relevant changes on the average value.

All the time-stamps are printed on a log file, using the `log()` function and extracting the average value in a second moment.

The following configuration have been tested:

- IPv4 Forwarding using the fast forwarding option: In this manner the packets are not queued to the input queue of IPv4, instead are directly processed by the fast forwarding routine.

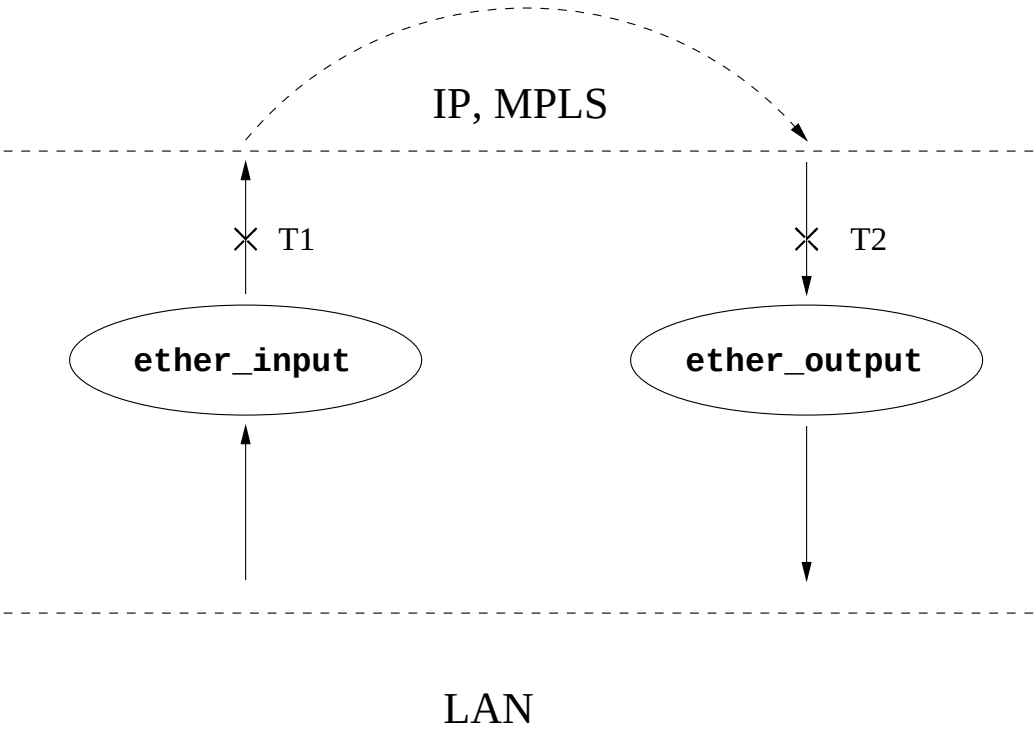

Figure 5.2: Times-tamp measurement points.

- MPLS Forwarding without cache: Packets are forwarded using label switching, but not using the cache to access the Label Table.
- MPLS Forwarding using the cache: Packets are forwarded using label switching using the cache to access the Label Table.

For the tests has been used the `tcpblast`[1] program. This program permits to transfer packets on a TCP or UDP connection, by installing a demon on one of the end host and starting the program on the other end host, further is possible to decide some parameters like the number of packets to transmit. The original version of the program has been patched so to enable the MPLS option.

[1] The program is included in the FreeBSD ports.

5.2 Result

Tests have been made extracting the average time of 1500 packets of 1024 bytes of data. In figure 5.3are showed the results.

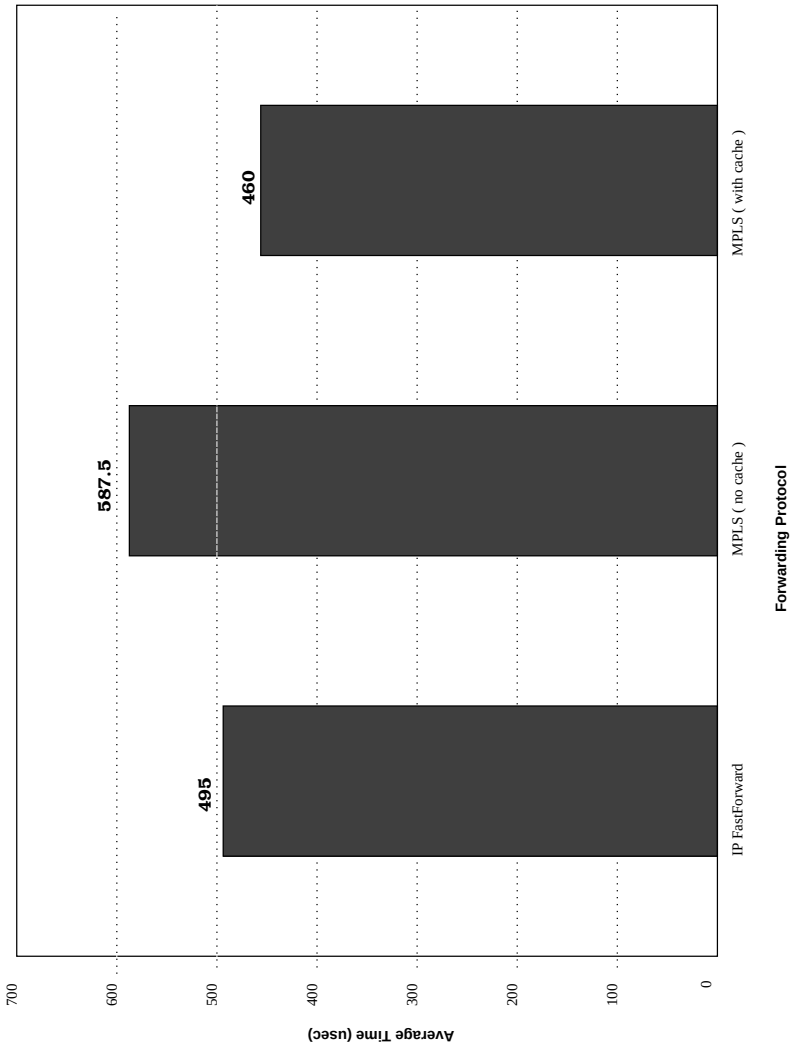

Figure 5.3: Packet's latency above Data-Link Layer.

In the figure one can remark that:

- Using the cache, performance is slightly better.

- There is not a big difference between MPLS and IPv4 using the fast forwarding option, nevertheless MPLS is a bit faster.

Referring to figure 5.1, the host on which the tests are done is the one that is configured as router (B). Particularly using MPLS, and assumed that on the host C the tcpblast demon is running and on the host A the tcpblast program is launched, at MPLS level the following operations are performed on the various hosts:

- Hosts A and C on each sent packet performs a PUSH operation of a label, and on each received packet performs a POP operation.

- The host B on each received packet performs a label SWAP operation and then forwards the packet on the network.

The MPLS's input queue is never used on host B, because the swapping operations are done by the `mpls_fastswap()`routine.

Note that in our tests, the label stack of every packet never exceed the depth of 1.

5.3 Testing procedure specifications

Give a look, now, how the testing procedures have been realized, particularly how to realize an MPLS network. All what follows is part of the kernel's patch realized by us.

A user that wish to send data using MPLS must:

1. Enable the MPLS in the kernel.

2. To set the MPLS option on the socket by which he will send the data.

A user that wish to receives data on his host using MPLS must:

1. Enable the MPLS in the kernel.

To Enable the MPLS in the kernel it is necessary to compile a new kernel once the MPLS option is been inserted in the kernel's configuration file. The MPLS option (`SO_MPLS`)of a socket an be easily set using the `setsockopt()` user level routine.

Once the previous step are done, if labels are available in the Label Table, packets sent labeled (see figure 4.12).

The Label Table is filled by the label distribution protocol, that provides to distribute the labels to all neighbors LSR. For our tests, no label distribution protocol is been used, instead a simple program that opens a MPLS socket and permits to add label to the Label Table is been produced.

5.4 Conclusions

The tests performed on the present work have showed that router that use MPLS are not so faster than the ones that not use MPLS. This can lead one to say that MPLS is not so convenient, if he think that the only target of MPLS is to go faster. But the advantages of the MPLS protocol goes over the simple forwarding time gain.

MPLS can include several information concerning a packet in one fixed length value: a label. Furthermore, MPLS is protocol independent, this means that it is designed to work over any Data-Link Layer protocol and under any Network Layer protocol.

Maybe the advantage of the MPLS protocol can be appreciated on ATM networks, where the Virtual Path concept fits well the LSP concept (for more information refer to [MPLS-ATM]).

Grazie

Profondo.... ai nostri genitori che ci hanno dato la possibilità di arrivare alla laurea.

Caloroso... alle nostre famiglie che ci hanno sempre sostenuto.

Affettuoso... a tutti gli amici e le persone che ci vogliono bene e ci sopportano.

Particolare... al Prof. Luigi Rizzo, per tutti gli stimoli che ci ha fornito in questi anni di studio.

Luca D'avico

Luigi Iannone

" Empty your mind, be formless, shapeless, like water. Now you put water into a cup, it becomes the cup. You put water into a bottle, it becomes the bottle. You put it in a teapot, it becomes the teapot. Now water can flow, or it can crash! Be water my friend....."

(Lee Jun Fan)

List of Figures

List of Tables

Bibliography

[MPLS-BGP] *"Carrying Label Information in BGP-4"*, Rekhter, Rosen: Work in Progress.

[MPLS-RSVP-TUNNELS] *"Extensions to RSVP for LSP Tunnels"*, Awduche, Berger, Gan, Li, Swallow, Srinvasan: Work in Progress.

[MPLS-LDP] *"LDP Specification"*, Andersson, Doolan, Feldman, Fredette, Thomas: RFC 3036, Jannuary 2001.

[MPLS-CR-LDP] *"Constraint-Based LSP Setup using LDP"*, Jamoussi: Work in Progress.

[MPLS-ARC] *"Multiprotocol Label Switching Architecture"*, Rosen, Viswanathan, Callon: RFC 3031, Jannuary 2001.

[MPLS-SHIM] *"MPLS Label Stack Encoding"*, Rosen, Tappan, Fedorkow, Rekther, Farinacci, Li, Conta: RFC 3032, Jannuary 2001.

[MPLS-ATM] *"MPLS using LDP and ATM VC switching"*, Davie, Lawrence, McCloghrie, Rekhter, Rosen, Swallow, Doolan: RFC 3035, January 2001.

[STE-1] *"TCP/IP Illustrated: Vol. 1 The Protocols"*, Stevens, Wright: Addison Wesley, 1993.

[STE-2] *"TCP/IP Illustrated: Vol. 2 The implementation"*, Stevens, Wright: Addison Wesley, 1995.