A splitting infrastructure for load balancing and security in an MPLS network

Stefano Avallone, Vittorio Manetti, Marina Mariano and Simon Pietro Romano
COMICS Lab, Dipartimento di Informatica e Sistemistica
Università di Napoli Federico II, Via Claudio 21, 80125 Napoli, Italy
Email: {stavallo, vittorio.manetti, spromano}@unina.it, mac.mariano@gmail.com

***Abstract*— Several multi-path routing algorithms have been recently proposed to achieve load balancing and increase security. However, the functionalities required to split traffic flows over multiple paths have not been standardized yet. Multi-Protocol Label Switching (MPLS) offers a suitable environment to implement multi-path routing algorithms, as multiple parallel Label Switched Paths (LSPs) may be established to carry each a portion of a traffic flow. This paper focuses on a traffic splitting mechanism designed for MPLS networks. We propose a set of new operations to be performed by the edge routers and illustrate them with reference to a popular open source implementation of the MPLS stack. We implemented the new operations and give a demonstration of their functionality using an experimental testbed.**

I. Introduction

Recently, several multi-path routing algorithms have been proposed aimed to both increase the security level in case of malicious attacks and to best utilize the network resources through load balancing. Such algorithms typically rely on the possibility to explicitly route flows, which is available, e.g., in MPLS networks. However, MPLS lacks mechanisms to split an incoming flow over multiple LSPs. Indeed, the IETF standard [1] provides that a forwarding equivalence class (FEC) may be associated with multiple outgoing labels, but does not specify how this association can be realized. In this paper we present an engineering approach to the solution of such a problem by implementing and evaluating a technique for per-packet load balancing in MPLS networks.

In order to fulfil the main goals we had in mind when designing our splitting technique, we have to deal with the two following major aspects: (i) the splitting criterion adopted; (ii) the specific features of the alternative paths to be selected. With reference to the first point, it looks clear that an appropriate splitting criterion needs to be implemented at the MPLS edge routers if we want to offer an adequate protection against potential security threats associated with the possibility that third parties intercept our traffic in some point of the network. Indeed, the idea of splitting traffic on a per-flow basis, while interesting from the load balancing perspective, would prove completely useless in the above depicted scenario. Hence, our approach relies on a per-packet splitting technique: packets arriving in sequence at an edge router are assigned to different paths inside the MPLS cloud (Fig. 1). Thanks to this choice, the information content that can be inferred by tampering with a single flow is almost negligible. A further advantage of

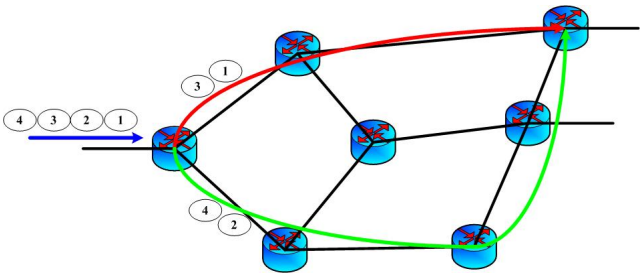

Fig. 1. Per-packet splitting

the proposed approach resides in the observation that all the information needed to implement the splitting procedure (as well as the associated reconstruction at the egress side of the network) is confined at the edges of the MPLS cloud. The overall process is completely transparent to the core routers, which just keep on performing the standard label-swapping procedure. If our mechanism is put into place, a potential intruder would have to successfully accomplish all of the the following tasks: gain access to traffic flowing across each and every network path in the MPLS cloud; be aware of the specific splitting criterion adopted; be capable of identifying the set of LSPs selected to carry a specific traffic aggregate from the ingress to the egress edges of the cloud.

We remark that the proposed mechanism can improve security by appropriately dispersing information content across the network. There is no way to look at it as an alternative to data encryption. Indeed, we believe it is possible to boost the security level of a network by using both encryption and a splitting technique like the one we are proposing in this paper.

Coming to the second aspect mentioned above, it looks clear that in an ideal situation all of the paths selected for the transport of the packets pertaining to a single flow that has been split at the ingress of the MPLS cloud should share the same properties in terms of parameters such as residual bandwidth, delay, packet loss, path length. The choice of paths characterized by significantly differing features definitely entails issues such as asymmetric balancing of the traffic load or out of order packet delivery.

The rest of the paper is structured as follows. Section II reviews some related work. Section III introduces the software package implementing MPLS in Linux which is the basis of our work. Section IV illustrates the splitting mechanism,

while Section V details its implementation. Section VI presents the reordering mechanism. Section VII reviews the step to configure the splitting of a traffic aggregate in an MPLS network. Section 8 concludes our work.

II. Related work

Many research works have recently dealt with the issue of implementing load balancing techniques in MPLS networks. Our approach presents many facets which let it clearly depart from such works, most of which rely on load balancing in order to improve network resiliency to congestion conditions. MATE [2], for example, proposes a model for traffic dispersion along auxiliary paths that have been selected in such a way as to guarantee QoS requirements related to incoming traffic. The model presented in [3] works in a quite similar fashion. DYBLA (DYnamic Load Balancing Algorithm [4]) introduces the original idea of splitting traffic among multiple paths right after having released the congested main path. In [5], an interesting model is presented, aimed at balancing traffic through a mapping process between a flow and a path depending on the characteristics both of the path itself and of the traffic to be forwarded.

Our model definitely differs from the aforementioned approaches aimed at tackling congestion issues (or more in general at better exploiting network resources), since it has been conceived at the outset as a means to improve the security level of the data sent across the splitting paths. With respect to this point, a related approach has recently been proposed in [6], which describes a mechanism based on traffic dispersion across multiple paths to solve the issue of potential eavesdropping. The mentioned model envisages that a hijacking cost is associated with each link. For a specific communication session, a packet dispersion model is elaborated in such a way as to ensure that the cost associated with eavesdropping is higher than the maximum hijacking budget available to the attacker.

Some other related works have been designed with the goal of serving traffic classes with predefined features. In [7], a load balancing model for MPLS networks is presented with special reference to voice traffic support. The focus of the paper is on the formulation of an appropriate multi-path routing algorithm.

Indeed, only few works propose load balancing solutions based on a per-packet approach. In [8] a path selection mechanism is devised, together with a per-packet splitting procedure based on a hash function applied to information carried inside network packets. In [9] a different per-packet splitting technique is presented. Once again, such mechanism aims to solve congestion issues in MPLS networks.

The following related papers are all based on a per-flow load balancing approach. A common feature of all such works is represented by the high level of abstraction of the proposals. In general, the proposed models and algorithms are not designed for a specific network architecture; hence, they do not go in any detail with respect to implementation. Except for [10], which proposes a load balancing mechanism for connectionless networks whose effectiveness is assessed through Linux-based emulations, all other papers just rely on simulation studies. As an example, in [11] some simulation results are presented in order to compare the effectiveness of the application of load balancing techniques both with and without splitting.

With our approach, the primary goal is to perform real world experimentations aimed at demonstrating the effectiveness of the application of a per-packet splitting technique in an actual operational scenario. Based on this assumption, we delve into the details of the solution design to the point that the information needed to implement the splitting mechanism in a concrete platform is provided. In the light of the above consideration, we will not deal in this paper with issues such as the dynamic selection of the most suitable splitting paths or the automated configuration of the MPLS cloud. In some cases, we may assume that, by using paths sharing similar properties, packets keep on being forwarded in the right order even in the presence of splitting. Though, this assumption does not apply to all situations. Hence, we decided to implement a reordering mechanism at the MPLS level (i.e. at the egress of the MPLS cloud), so to make splitting completely transparent to the upper layers of the protocol stack. Finally, we would also like to remark that our splitting technique does not put any limitation on the number of alternative paths that can be chosen to forward a single traffic aggregate entering the MPLS network, as it happens in some research proposals that can be found in the literature (see, for example, [12] in which the authors propose to associate each flow with a pair of paths used, respectively, as the primary and secondary routes).

III. MPLS-Linux

The splitting technique we propose has been actually implemented by modifying the code developed within the mpls-linux project (http://sourceforge.net/projects/mpls-linux). This project provides a patch for the latest Linux kernel series (2.6.x) to add a full functional MPLS stack. A userspace utility is also provided to manually configure label switched paths. To this end, a set of instructions have been defined as follows:

- ***push*** : adds an MPLS header to the packet with a specified label;
- ***set*** : sends a packet down to the data link layer, thus causing it to be transmitted to the next hop router;
- ***pop*** : removes an MPLS header from the packet;
- ***fwd*** : causes a packet to be processed according to the instructions of a specified NHLFE entry, as will be explained shortly;
- ***dlv*** : sends a packet to the IP stack for further processing;

The mpls-linux patch introduces into the kernel the tables MPLS uses to forward data, as defined in [1]:

- FTN (FEC To NHLFE)];
- ILM (Incoming Label Map);
- NHLFE (Next Hop Label Forwarding Entry);

FTN

The FTN table is made of a number of entries, each of which refers to a particular Forwarding Equivalence Class (FEC).

Thus, the FTN is looked up when forwarding an unlabeled packet. Each entry contains a **fwd** instruction pointing to an NHLFE entry describing how to process packets belonging to that FEC.

ILM

The ILM table is looked up when forwarding labeled packets, i.e. packets containing an MPLS header. The entries of the ILM table are indexed by a key whose value is defined as follows:

$$key = type \cdot 2^{30} + label \cdot 2^{10} + labelspace$$

where:

- *type* identifies the layer-2 technology (Ethernet, ATM or Frame Relay);
- *label* is the label included in the MPLS header;
- *labelspace* is the space of label associated with the interface the packet has been received from.

The above key is coded on 4 bytes. Each entry of the ILM contains a **fwd** instruction pointing to an NHLFE entry describing how to process packets matching that key.

NHLFE

An entry of the NHLFE table is pointed to by an entry of the FTN, an entry of the ILM or another entry of the NHLFE and describes how to process a packet. Each entry of the NHLFE table is indexed by a key, too. However, here the key value is not related to any label and is set from a counter.

To illustrate how these tables are used in forwarding packets, we look at the operation performed by different routers in an MPLS network:

Ingress LER (Label Edge Router)

An ingress LER receives unlabeled packets, adds an MPLS header and makes them continue their travel along an appropriate LSP. The first step is thus to classify each packet into a forwarding equivalence class (FEC) and look the FEC up in the FTN table. As a result, we obtain a key to access the entry of the NHLFE table which specifies the label to be inserted into the MPLS header. Then, the packet is sent to the next-hop router.

LSR (Label Switching Router)

An LSR receives labeled packets, swaps the label and sends them to another LSR. The label in the MPLS header is used to calculate the key to access an entry of the ILM table (as shown above). Such an entry usually contains a **pop** instruction to remove the MPLS header and a **fwd** instruction pointing to an entry of the NHLFE table. This entry will typically instruct how to add a new MPLS header and specify the next-hop LSR.

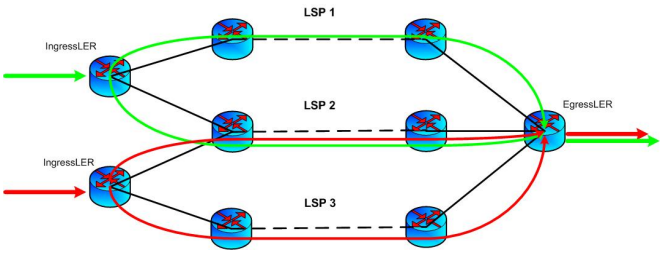

Fig. 2. Reordering packets

Egress LER

An egress LER receives labeled packets, removes the MPLS header and pass them on to the IP layer. The label in the MPLS header is used to calculate the key to access an entry of the ILM table. Such an entry usually contains a **pop** instruction to remove the MPLS header and a **dlv** instruction to deliver the packet to the IP layer, which is in charge of routing the packet out of the MPLS network.

IV. Reordering packets

Splitting the packets belonging to a flow over multiple paths may cause the packets to arrive at the egress LER out of order. It is known that such event cannot happen if all the packets are sent along the same LSP. Since the ordered delivery of packets is a desirable feature, we provide a reordering mechanisms in order to emulate the behavior of an MPLS network with no splitting capabilities. It is clear that the reordering of packets takes place at the egress LER. However, the ingress LER has to put some extra information into the packets to allow a correct reordering of packets. In this section we identify the information required to allow for appropriate reassembly of the flow at an egress LER and how to convey such information.

We underline that the packet reordering can complement our splitting technique, but it is not mandatory. The reordering mechanism clearly imposes a non-negligible overhead on the ingress and egress LERs. However, it may prevent delays due to TCP reconstructing the original data flow. A performance evaluation of our reordering mechanism is presented in Section VIII.

The MPLS label stacking capability can be exploited to carry some extra information. Indeed, the ingress LER can insert an additional MPLS header containing such information. At every hop, only the outer MPLS label is swapped and processed by the forwarding process. The inner label is carried unmodified till the egress LER where it is appropriately processed.

To allow for the reordering of packets at the egress LER, the most straightforward solution relies on a counter whose value is incremented and attached to a packet when it crosses the splitter ingress node. The edge routers responsible for the reordering procedure interpret such value as a sequence number and are thus able to reconstruct the correct sequence of packets representing the original traffic.

However, the adoption of a sequence number to be associated with all flows entering the MPLS network through the

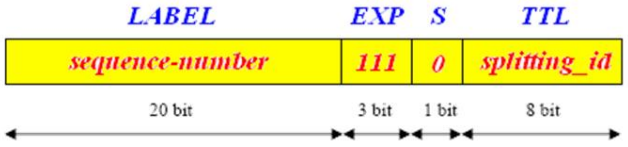

Fig. 3. A "splitting" MPLS header

same ingress LER does not work in case the splitting paths can be shared. Let us consider the case where two flows belonging to different equivalence classes share a common splitting path (Fig. 2). The core LSRs falling along the shared path just perform standard label swapping operations, hence they do not elaborate splitting information contained inside the packet. If the above mentioned flows also have the final destination in common, then the egress LER must be able to associate each packet with the particular traffic aggregate to which it belongs.

To allow for a correct reassembly of the flows in the context depicted above, we need to introduce a novel parameter used to uniquely identify traffic belonging to a well defined FEC. We call such parameter splitting-id. A splitting-id uniquely identifies a traffic aggregate whose packets would be forwarded, in the absence of splitting, along the same path.

Thus, in case reordering must be performed at the egress LER, every single packet has to carry both the sequence number and the splitting-id. We propose to encode such information in an additional (inner) MPLS header in the following way (Fig. 3):

- Label : sequence-number;
- EXP (Experimental Use): contains a binary string (e.g. 111) used to identify the label information as a splitting parameter rather than as a standard labelswapping identifier;
- S (Stack): must be always zero, to indicate that the label in question is the last one in the lable stack;
- TTL (Time To Live): such field is actually interpreted as a splitting-id, which means that it does not have to be decremented when the tagged packet crosses a router.

In the following sections we detail the implementation of the splitting technique, stressing the steps necessary for the reordering to take place. Since the core routers perform their normal operation (label swapping), we focus on the packet processing at the ingress and egress LERs.

V. Processing at the ingress LER

The splitting mechanism we propose is based on a specific feature of the MPLS protocol, namely the possibility to associate a single FEC (Forwarding Equivalence Class) with multiple NHLFEs (Next Hop Label Forwarding Entry). A FEC identifies a class of flows asking for the same treatment in terms of forwarding: flows belonging to the same FEC are actually considered as a single entity in the MPLS cloud. Clearly, the possibility to associate a single FEC with multiple entries in the NHLFE table can be seen as a chance to manage flows on a per-packet basis, since different packets of the flow can be forwarded along different LSPs (Label Switched Path).

The splitting mechanism thus necessitates the following functionalities:

- possibility to associate a single FEC with multiple NHLFE entries;
- inserting a sequence number and a splitting-id in the MPLS header in case an ordered delivery at the egress LER is required;
- forwarding of packets belonging to the same FEC to different LSPs.

Associating a FEC with a single NHLFE entry is equivalent to impose that packets belonging to the same equivalence class will be routed along the same LSP. Thus, providing a means to associate a FEC with multiple NHLFE entries is fundamental to make the splitting possible. We propose to perform such an operation as illustrated in Fig. 4. The basic concept is quite simple: with respect to the common procedure performed in an ingress LER (see Section III), here we introduce an intermediate step consisting of the processing of an additional NHLFE entry. The FTN entry corresponding to the FEC of a given packet still points to an entry of the NHLFE. However, such an entry contains the special instructions we added and in turn points to a set of NHLFE entries. These last entries actually contain the **push** and **set** instructions to add an MPLS header and send the packet to the next hop. The new instructions to be inserted into the intermediate NHLFE entry are **ipush** (increment and push) and **mfwd** (***multiple*** forward). The first instruction is in charge of managing the parameters needed to implement the reordering mechanism, so it present only if needed. The **ipush** instruction has to perform two tasks: (i) recognition of the flow the packet belongs to, and (ii) update of the sequence number and the splitting id. Then, a splitting MPLS header (as shown in Fig. 3) is added to the stack of headers. The **mfwd** instruction actually splits a flow by associating the intermediate NHLFE entry with other NHLFE entries. According to the adopted splitting policy, different NHLFE entries are selected for different packets, thus splitting the flow over multiple LSPs.

An ingress LER splitting a flow over multiple LSPs performs the following operations:

1) The FEC the packet belongs to is used to access the FTN table in order to determine the key of the intermediate NHLFE entry;
2) In case we want to emulate the transmission along a single LSP, an **ipush** instruction is present in the intermediate NHLFE entry and it must be processed. As a consequence, a splitting MPLS header is added. In any case, the intermediate NHLFE entry contains an **mfwd** instruction which bounds the current FEC to a set of NHLFE entries (each corresponding to a different LSP). For each packet to be forwarded, one NHLFE entry in such set is selected based on the splitting policy (i.e., round robin);
3) The instructions contained in the selected NHLFE entry

are processed. These includes adding a swapping MPLS header and sending the packet to the next hop.

The code related to **ipush** and **mfwd** has been inserted in linux/net/mpls/mpls_opcode.c. The **ipush** instruction is quite similar to **push**, so it has been been possible to reuse some code. Parameters required by these instructions are provided by an user-space utility, the **'mpls'** utility, which we needed to modify as well.

We remark that at the time of this writing no mechanisms are available to build and to destroy splitting paths in a dynamic way. Though, it is possible to exploit a user level tool named **'mpls'** (available in the iproute package) that provides instructions for the configuration of the MPLS tables entries, and hence of static LSPs. We have modified this tool in order to provide a splitting paths configuration mechanism. In the near future, we have in mind to work on the design and implementation of an automated policy-based configuration tool capable to carry out configuration of the MPLS cloud.

VI. Processing at the Egress LER

In case a flow is split and the packet reordering is not needed, the egress LER must perform no additional operation. Thus, this section describes the configuration of the egress LER in case the ordered delivery of packets is required.

When designing such mechanism, once again we decided to leave unchanged the already available MPLS functions (i.e. those related to label swapping, label popping and packet delivery to the higher layers), while introducing brand new functionality specifically conceived for the splitting/ reordering procedure.

Coming to the details, we impose that all packets associated with the same FEC (and thus characterized by the same splitting-id), once at the egress LER are forwarded to the same entry in the ILM (Incoming Label Map). Such entry contains the sequence of instructions needed to correctly reconstruct the original flow. Given the above constraint of using at the egress LER special instructions for packets belonging to flows that have been split, it looks clear that upon configuration of the MPLS cloud special entries must be added to edge routers tables. Such tables will then contain both standard label-swapping entries and experimental splitting entries.

The management of packet reordering in the described solution is based on the new functionality offered by MPLS layer in version 1.946 and later, including the mpls protocol driver mpls4. This module works between MPLS layer and IP layer, in particular representing the entry point to IPv4 layer. In the solution, the reordering mechanism has been put at this level, making the reordering mechanism transparent to all users. Moreover it uses the efficient and general methods of Linux system to manage data packets and manipulate them.

Making the reordering mechanism transparent is possible by using the usually instructions related to an ILM entry: if a packet arrives with several MPLS labels, all of them are looked up step by step. Each label hooks an ILM entry and the corresponding instructions are executed. At to bottom of the stack, a DLV instruction is executed and the packet is sent to the upper layer. Indeed, the delivery instruction uses the central structure of network implementation, the so-called socket buffer, which represents a packet during its entire processing lifetime in the kernel. In the socket buffer they are defined hooks dedicated to manage the flow control requiring their access to mpls4 module (field skb→dst→ output is a pointer to mpls_local_delivery), matching to dst_entry of latest ILM entry. With few modifications to the function mpls_opcode_peek it is possible to pass data that are required to protocol driver to reorder packets. This information has been coded, respectively, inside the TTL, LABEL and EXP fields, and they are passed inside control-buffer field of socket buffer, in a similar way to MPLS layer and IP layer. In current design, the only action performed by mpls_local_delivery is returning the control to IP layer, to process the packets to be delivered locally. If the reorder is required, that is the experimental bits are setted up to 111, the function mpls_local_deliver() passes IP packets to mpls_reorder(). The packets are then managed in a packet-cache, until the packets arrive in a right order, so that all the packets stored in the list can be delivered to the local machine. The packet-cache consists of a table hashed by Splitting-id and refer to an ip-queue structures. Each of these ip-queue structures store the packets waiting the right packet in the sequence. The individual socket buffers related to the same flow are linked in a list. All packets of a flow must to be ordered in the same sequence as they occur in the original flow. If an error occurred and the maximum wait time for the packet has expired, all the packets will be discarded. Note that timeout for similar protocols, such as IP fragment, is usually set to 30 seconds. The correct sequence has to be re-established starting from the next packet in the buffer. The following cases have to be considered:

- $(S_N + 1) = LABEL$: the received packet has arrived in the correct sequence. In such case, we proceed by simply invoking the ip_rcv() function on the packet and then increasing the value of the sequence number variable (which will now contain information about the latest received packet);
- $(S_N + 1) < LABEL$: the packet received is subsequent to the expected one. In such case, we put it in a temporary buffer and perform a check on the other packets previously stored in the buffer. For each packets, the following options are possible:
  - $(S_N + 1) = LABEL$: the packet in question conforms to the right sequence. In such case we invoke the ip_rcv() function on the packet and then increase the sequence number variable. Finally, we move to the next packet in the buffer;
  - $(S_N + 1) < LABEL$: we just move to the next packet stored in the buffer;

VII. Configuring the MPLS cloud for splitting

We summarize in the following the sequence of actions to be performed in order to configure the nodes of an MPLS cloud with splitting capabilities.

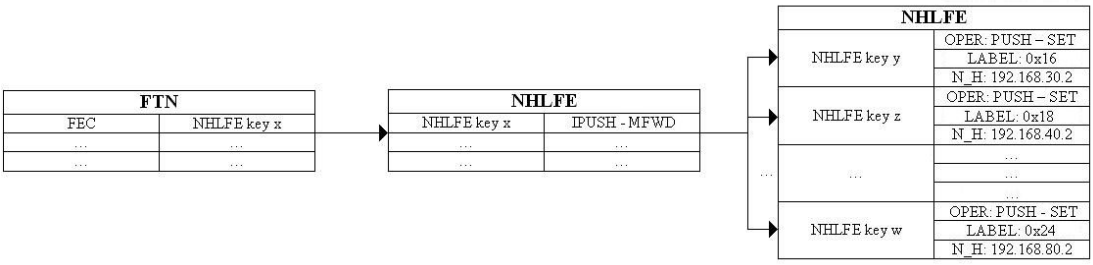

Fig. 4. Splitting technique

1) The first step is to identify the forwarding equivalence class of the traffic aggregate we intend to split. A traffic aggregate is composed of a set of distinct flows having in common the same pair ingress LER/egress LER. In case the packet reordering is needed, we must associate the traffic aggregate with the corresponding identifier (splitting id);
2) A set of LSPs between the ingress LER and the egress LER are to be selected. The traffic aggregate will be splitted among the selected LSPs;
3) The MPLS tables of the nodes along the selected LSPs must be configured accordingly:
   - **ILER** : For each LSP, an NHLFE entry containing the **push** and **set** instructions is to be added. The key associated with these entries must be passed as parameters of the **mfwd** instruction present in the intermediate NHLFE entry. This last entry is pointed to by the entry of the FTN associated with the FEC of the traffic aggregate. In case the packet reordering is required, an **ipush** instruction must precede the **mfwd** instruction in order to add the splitting header.
   - **LSR** : No particular operation is required on the core nodes. They are only required to perform the label swapping mechanism.
   - **ELER**: In case the packet reordering is required, we need to configure an ILM entry containing the following instructions: **pop**, **reorder**, **dlv**.

VIII. Experimental results

In this section we present the results of some experiments conducted in a real testbed to show the effectiveness of our approach. As we explained, in order to perform per-packet splitting according to our approach, it is only necessary to perform some additional operations on the MPLS edge routers. We focus our attention on the splitter ingress node. The main goal of the experimental trials is to prove that the splitting mechanism causes a lower overhead and consequently a lower usage of computational and storage resources when the reordering mechanism is disabled. We present a comparison between the classical label-switching mechanism on a single LSP and the splitting mechanism on several splitting paths, both with and without the reordering mechanism. The trials

TABLE I
8 FLOWS

label-switching	splitting w reord	splitting w/o reord	function
0.00260	0.00590	0.00370	mpls_output2
0,00190	0,00290	0,00180	mpls_set_nexthop2
0,00200	0,00210	0,00170	mpls_send
0,00190	0,00170	0,00150	mpls_push
0,00110	0,00140	0,00100	mpls_output
0,00110	0,00130	0,00120	mpls_out_op_set
0,00073	0,00110	0,00071	mpls_output_shim
0,00032	0,00060	0,00049	mpls_p_f_by_ethtype
0,00048	0,00055	0,00067	ipt_mpls
0,00030	0,00015	0,00030	mpls_op_push
0,00003	0,00014	0,00003	mpls4_get_ttl
0	0,00215	0,00144	mpls_op_mfwd
0	0,00260	0	mpls_op_ipush
0,01246	0,02259	0,01454	TOT

are carried out on a real testbed consisting of eight nodes. Each node is equipped with a Pentium 4 Xeon processor, with hyper-treading capability and an estimated speed like 3,392.47 MHz. The Linux-MPLS patch (version 1.950) enhanced with our modifications was applied to all the nodes.

In order to perform these experiments, we have selected a monitor (Oprofile) for evaluating the computational resources used by both user-level and kernel-level functionalities. This monitor produces a report showing the number of CPU cycles each function uses (in terms of percentage, related to the measurement interval).

The experiments consist in configuring the LSP along which the data flows are forwarded, generating UDP traffic (by using the D-ITG traffic generator) on the Ingress LER (splitter node), and evaluating the computational load on the splitter node (by using the Oprofile monitor).

We compare the performance of the following three mechanism: label-switching on a single LSP, splitting over 7 LSP with reordering, and splitting over 7 LSP without reordering. Every mechanism is implemented using 8, 12, and 16 data flows forwarded across the MPLS cloud.

The tables show the cost of every single function/instruction

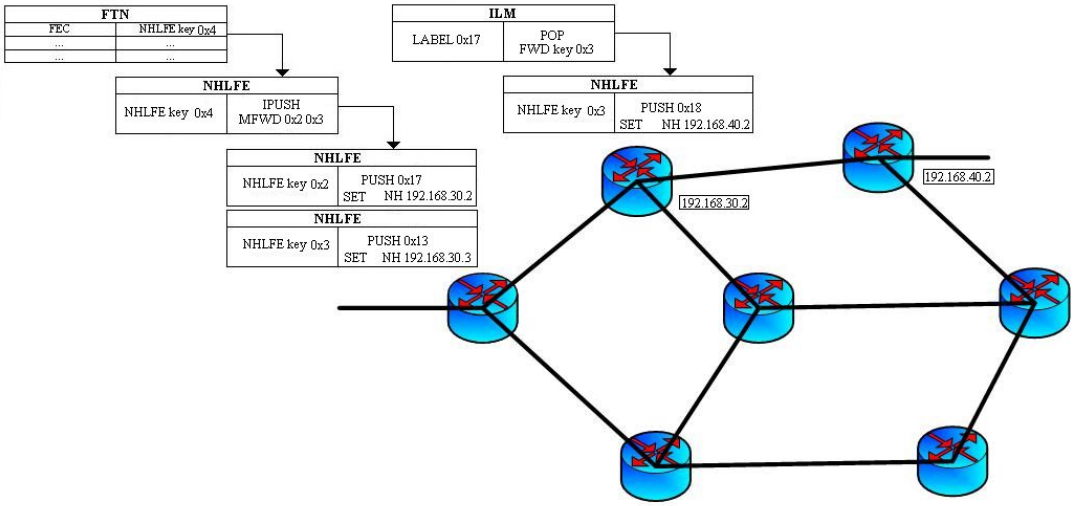

Fig. 5. Configuration

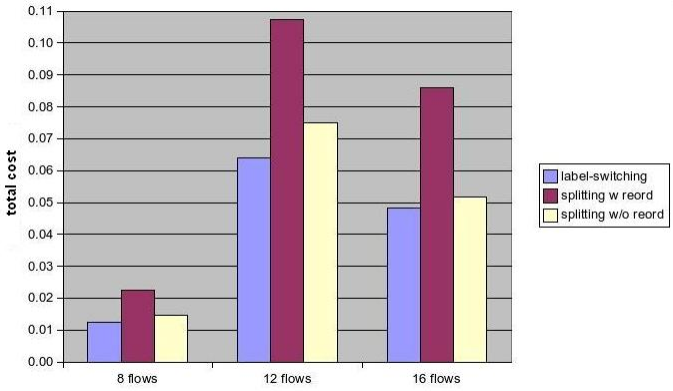

Fig. 6. Performance evalution

TABLE II
12 FLOWS

label-switching	splitting w reord	splitting w/o reord	function
0,01390	0,02510	0,01960	mpls_output2
0,00730	0,00970	0,00700	mpls_set_nexthop2
0,00910	0,01030	0,00780	mpls_send
0,00970	0,01320	0,00920	mpls_push
0,00740	0,00730	0,00650	mpls_output
0,00560	0,00620	0,00600	mpls_out_op_set
0,00410	0,00590	0,00410	mpls_output_shim
0,00240	0,00330	0,00220	mpls_p_f_by_ethtype
0,00250	0,00270	0,00190	ipt_mpls
0,00190	0,00170	0,00140	mpls_op_push
0,00008	0,00048	0,00014	mpls4_get_ttl
0	0,00940	0,00910	mpls_op_mfwd
0	0,01200	0	mpls_op_ipush
0,06398	0,10728	0,07494	TOT

TABLE III
16 FLOWS

label-switching	splitting w reord	splitting w/o reord	function
0,01100	0,02060	0,01370	mpls_output2
0,00490	0,00910	0,00460	mpls_set_nexthop2
0,00710	0,00850	0,00580	mpls_send
0,00720	0,00970	0,00590	mpls_push
0,00560	0,00520	0,00450	mpls_output
0,00460	0,00490	0,00400	mpls_out_op_set
0,00310	0,00500	0,00270	mpls_output_shim
0,00170	0,00250	0,00140	mpls_p_f_by_ethtype
0,00170	0,00230	0,00140	ipt_mpls
0,00110	0,00089	0,00075	mpls_op_push
0,00013	0,00024	0,00006	mpls4_get_ttl
0	0,00940	0,00910	mpls_op_mfwd
0	0,01020	0	mpls_op_ipush
0,04813	0,08613	0,05171	TOT

of the MPLS stack for each of the three configurations under test and for the three scenarios we considered (8, 12 and 16 flows). Such a cost refers to the ratio (multiplied by 100) of the number of CPU cycles required to execute every single function to the number of CPU cycles required to execute all the functions in the sample interval.

The set of MPLS functions clearly comprises those we introduced for the splitting mechanism, which are indicated in the tables as mpls_op_mfwd and mpls_op_ipush.

Fig. 6 shows for every scenario and every configuration the sum of the costs of all the functions. Such value, which we refer to as total cost, is also reported in the last row of each table.

We can draw that the splitting mechanism involves a very low overhead with respect to the classical label-switching mechanism, and, as we could expect, the reordering mechanism requires some computational load.

The results indicate that the total cost shows a very similar behavior for all the three configurations. It is worth to note that, given the approach used to determine the computational load of each function, it is not appropriate to compare the numeric values for the three scenarios. It is clear indeed that each of such scenarios is characterized by a different load of the CPU of the splitter node, relating to which we compute the cost of the MPLS functions. Also, the CPU load due to the traffic generator varies with the number of flows involved.

The tables also show that the mpls_output2 function (which depends on the number of time we access the NHLFE table) exhibit a similar behavior for all the three scenarios. The implementation of the splitting mechanism causes a certain increase in the cost of such a function, especially if we use the reordering mechanism. This is clearly due to the increased number of time we access the NHLFE table of the ingress LER when the splitting is enabled.

IX. Conclusion and future work

In this paper we presented an approach to improve network security and resource utilization in MPLS networks. The mechanism we proposed is based on the idea of splitting flows on a per-packet basis at the ingress of an MPLS cloud, while reassembling them in the right sequence before they leave the egress LER. Our main contribution resides in having devised an engineering solution that has been implemented in a real-world operational scenario. Indeed, the solution itself is independent of the specific technology chosen for its implementation, since it relies ont the introduction of some new functions which complement the standard MPLS way of operating. Though, we also presented in the paper an actual implementation of the splitting mechanism realized by appropriately modifying the available MPLS-Linux experimental stack. As to packet reordering, we are currently working on refining our first prototype implementation. Our future work will be organized along three main directions: (i) complete the implementation of the reordering mechanism; (ii) perform a thorough experimental campaign aimed at both assessing the validity of our approach and the performance of the newly created MPLS modules; (iii) work on the management part, by providing an automated policy-based configuration tool capable to carry out configuration of the MPLS cloud in a straightforward fashion.

References

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

[2] A. Elwalid, C. Jin, S. Low, and I. Widjaja, "MATE: MPLS Adaptive Traffic Engineering," in *Proc. of IEEE INFOCOM 2001*, vol. 3, 2001, pp. 1300–1309.

[3] M. Heusse and A. Gravey, "A Routing and Resource Preservation Strategy for Traffic Engineering in Communication Networks," in *Proc. of 18th ITC*, September 2003.

[4] E. Salvadori and R. Battiti, "A load balancing scheme for congestion control in MPLS networks," in *Proc. of ISCC 2003*, July 2003, pp. 951–956.

[5] B. Cui, Z. Yang, and W. Ding, "A Load Balancing Alghorithm Supportino QoS for Traffic Engineering in MPLS Networks," in *Proc. of The Fourth International Conference on Computer and Information Technology (CIT'04)*, 2004.

[6] H. Zlatokrilov and H. Levy, "Session Privacy Enhancement by Traffic Dispersion," in *Proc. of IEEE INFOCOM 2006*, April 2006.

[7] B. Yang, C. Casetti, and M. Gerla, "Stateless Load Balancing over Multiple MPLS Paths," available at `http://gregorio.stanford.edu`.

[8] X. Hesselbach, R. Fabregat, B. Baran, Y. Doloso, F. Solano, and M. Huerta, "Hashing based traffic partitioning in a multicast-multipath MPLS network model," in *Proc. of IFIP/ACM Latin America Networking Conference 2005 (LANC 2005)*, 2005.

[9] A. Dana, A. K. Zadeh, and M. J. Akhlaghinia, "Performance Evaluation of a Load Scheme in MPLS Network," in *Proc. of Communication Systems and Applications 2005*, 2005.

[10] H. T. Kaur and S. Kalyanaraman, "A Connectionless Approach to Intra- and Inter-Domain Traffic Engineering," in *Proc. of 2nd New York Metro Area Networking Workshop*, September 2002.

[11] J. Wang, "Load Balancing in Hop-by-Hop Routing with and without traffic splitting," University of Illinois at Urbana-Champaign, Tech. Rep., October 2003.

[12] N. Akar, I. Hokelek, M. Atik, and E. Karasan, "A reordering-free multipath traffic engineering architecture for DiffServ-MPLS networks," in *Proc. of 3rd IEEE Workshop on IP Operations & Management (IPOM2003)*, October 2003, pp. 107–113.