Switchlets and Resource-Assured MPLS Networks

May 2000

Richard Mortier, Rebecca Isaacs, and Keir Fraser
Systems Research Group, University of Cambridge Computer Laboratory, UK

Abstract—

MPLS (Multi-Protocol Label Switching) is a technology with the potential to support multiple control systems, each with guaranteed QoS (Quality of Service), on connectionless best-effort networks. However, it does not provide all the capabilities required of a multi-service network. In particular, although resource-assured VPNs (Virtual Private Networks) can be created, there is no provision for *inter-VPN* resource management. Control flexibility is limited because resources must be pinned down to be guaranteed, and best-effort flows in different VPNs compete for the same resources, leading to QoS crosstalk.

The contribution of this paper is an implementation on MPLS of a network control framework that supports inter-VPN resource management. Using resource partitions known as *switchlets*, it allows the creation of multiple VPNs with guaranteed resource allocations, and maintains isolation between these VPNs. Devolved control techniques permit each VPN a customised control system.

We motivate our work by discussing related efforts and example scenarios of effective deployment of our system. The implementation is described and evaluated, and we address interoperability with external IP control systems, in addition to interoperability of data across different layer 2 technologies.

Keywords— **Multi-Protocol Label Switching, Virtual Private Network, switchlet, Tempest, IP, ATM, interoperability**

I. INTRODUCTION

A multi-service network has to meet the diverse QoS requirements of hard real-time, soft real-time, and data traffic in a cost-effective, flexible and efficient manner. Monolithic, general-purpose network control systems have proved to be ill-suited to the task of managing such networks — cases in point are the poor real-time support in IP, and the heavyweight signalling protocols of ATM. Attempts to coerce these control systems into supporting multiple services add complexity, and do not address the long-term problem of currently unknown future services. We believe that in a truly multi-service network the data and control planes must be separate so that multiple control systems may coexist. This enables applications' service requirements to met by the control system best suited to the task. Resource partitioning between control systems at the lowest level is a requirement for the delivery of QoS guarantees to applications.

We have previously proposed the Tempest as a network control framework achieving these aims. We have described our implementation of the Tempest on ATM [1], [2], a technology which naturally lends itself to a clean separation between the network's data and control planes. In this paper we present an implementation of the Tempest over MPLS, and explore interoperability between Tempest and non-Tempest domains within a control system called ALASCA[1].

Our MPLS implementation partitions network resources at the lowest level through differentiated packet forwarding. This allows it to support resource-guaranteed VPNs, each with its own control system, over networks offering neither QoS guarantees nor connection-oriented service, such as Ethernet. Furthermore, the Tempest framework allows the reservation of unused resources for future connections, the reallocation of resources between VPNs, and global resource management between VPNs. The result is a platform also suitable for the realization of other efforts to introduce QoS in networks, such as DiffServ (Differentiated Services) and IntServ (Integrated Services).

Outline of Paper

In the following section we review MPLS and the Tempest, and other related work. In Section III we make the conceptual framework more concrete by describing some example scenarios: support for the IETF's DiffServ and IntServ architectures, provision of differentiated service for requests in a cluster-based web server, and support for aggregate resource reservation in the Internet.

Section IV describes our MPLS implementation for the Linux kernel, detailing how label and resource partitioning is achieved, and discussing how we deal with packets entering the Tempest domain from external sources. We also present ALASCA, a Tempest-style implementation of an LDP (Label Distribution Protocol) which enables connectivity with external systems.

Richard Mortier acknowledges the support of an EPSRC CASE award in association with BT. Email `firstname.lastname@cl.cam.ac.uk`

[1] A Little Autonomous System Control Architecture.

An evaluation of our implementation is given in Section V, and interoperability issues arising from the implementation are discussed in Section VI. Finally, Section VII contains a brief summary and suggestions for future work.

II. Background

A. Multi-Protocol Label Switching

MPLS [3] is a framework for forwarding based on a short, fixed-length label in the packet header. It divorces route determination from the forwarding mechanism, enabling more complex treatment of traffic streams than is possible in current IP networks. Packets may follow paths determined by considerations other than the pure hop-by-hop destination-routed model of IP.

When a packet enters the network, its FEC (Forwarding Equivalence Class) is determined and a label assigned accordingly. This label specifies the path along which the packet will be forwarded; from this point, the network need only perform lightweight label switching operations at each node, until the packet reaches the end of the path. A packet may be classified into a FEC by a variety of means, ranging from simple destination-based classification, analogous to the current IP routing model, to classification based more generally on the packet's headers or content.

The label is either inserted into the layer 2 header if fields are available, or the packet is encapsulated by a special-purpose *shim* header. Switching points may consist of software controlling well-known switch technologies, such as ATM or Frame Relay, or may be custom built to support MPLS. At a switch[2] the label serves to index into a LIB (Label Information Base), a table containing the next forwarding hop and a new label.

Switches construct their LIBs using an LDP, which may be classified according to how they create their LIB entries:

Request-driven Creation based on the messages of a control protocol such as RSVP [4], or traditional ATM signalling.

Topology-driven Creation based on information derived from layer 3 routing protocols, such as BGP, OSPF, or the generic MPLS-LDP [5].

Traffic-driven Creation based on information gathered by monitoring the traffic streams being switched, as with, for example, IP Switching [6].

The separation in MPLS of these three network functions — packet classification, packet forwarding and label distribution — simplifies the data path, and allows greater differentiation between packets. Service differentiation at a range of packet aggregation granularities is supported, as is additional functionality such as traffic engineering. MPLS also provides a platform for building resource-assured VPNs, and hence supporting multiple control systems with resource partitioning.

However, MPLS on its own is not sufficient for a multi-service network. In addition to resources for packet forwarding, such as buffer space and bandwidth, the provision of resource-assured VPNs requires partitioning of the label space. The lack of this facility in MPLS limits the way resource may be guaranteed to each network control system, and the freedom afforded a control system to manipulate its resource allocation. A further problem is that inter-VPN resource management is not supported: since a VPN's resource allocation as a whole is only expressible as the sum of the resource allocations of its paths, resources cannot be guaranteed for the future, other than by pinning down a path — in effect associating each label with a pre-determined QoS.

As an example, consider a scenario where a service provider offers a VPN service, each VPN having a topology and guaranteed resource allocation. Each customer implements internal VPN resource management by assigning various QoS capabilities to paths in their VPN, including one best-effort path. This soaks up the unused resources within a VPN, and so it should be possible to improve service on this path by increasing the size of the resource allocation to its owning VPN. Conversely, if resources are removed from an otherwise unchanged VPN, only the resources available for use by the best-effort path in that VPN should reduce. Other VPNs and their internal best-effort paths should not be affected.

This behaviour can be implemented within MPLS, but requires the LDP to participate in network-wide resource partitioning. We believe that this is infeasible, especially as there may be many LDPs operating independently in the network. The resources associated with a VPN should be under the complete control of the owner of the VPN, and their management should not require co-operation between VPN owners. We consider the capability for both inter- and intra-VPN resource management, as provided by the Tempest, to be essential.

B. The Tempest

The tight coupling of management and control functionality with network hardware leads to a closed and restrictive environment. Due to their monolithic nature, standardised control systems are usually heavyweight and unwieldy, stifling the quick development and deployment of new services. Furthermore, tightly integrating the network

[2] The term switch is henceforth assumed to refer either to a switch, an LSR (Label Switched Router), or a router.

control software and internal network elements makes upgrades and bug fixes difficult and costly.

Open signalling, where control of the network is devolved from the internal network elements to general-purpose workstations, is a partial solution. Switch functionality is encapsulated in an open control interface, accessible by third parties who may be neither the users of the application or network service, nor the manufacturers of the hardware. Although a useful mechanism for the deployment of non-standard network control protocols, it does not provide a satisfactory solution to the problem of inflexible and monolithic control software. By its very nature, a single general-purpose control system cannot always be the best solution in a multi-service network. Operations required by one type of control system may not be suitable for another, and new types of operation may be required as hardware and user demands evolve. At the same time multiple control systems within a network cannot be relied upon to cooperate. This requires the network to provide a mechanism for partitioning between control systems.

The Tempest is a network control framework that provides resource partitioning and open signalling to address these issues. It is based on the *switchlet* concept [7], where the resources of individual switches are subdivided into logically separate partitions. Each switchlet is presented to its owning control system via an open control interface, giving the illusion that the control system is managing an entire switch. This allows multiple control systems to coexist on a single physical network, whilst providing them with fine-grained control of the resources they have been allocated.

Sets of switchlets are combined to form VPNs, each with exclusive access to its share of the network resources. This allows multiple control systems to operate simultaneously, and means that no single system need be prescribed for all users; multiple instances of the same control system may however control separate VPNs. Although general-purpose control systems will suffice for many users, others are able to run service-specific control systems tailored to their individual needs, should they wish.

The creation, deletion and modification of switchlets is managed in the Tempest by the *divider*. One divider operates for each node in the network, and executes off the network hardware, possibly on a general-purpose workstation, and ideally in a resource-controlled environment such as that provided by the Nemesis operating system [8]. It may be co-located with the network node itself, removing the overhead of communicating with the node, but the switching hardware and the partitioning functionality of the divider remain logically distinct. The divider polices

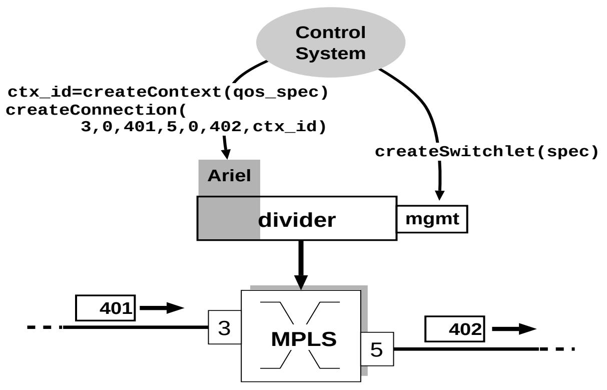

Fig. 1. Control system interactions with the divider management and Ariel interfaces. The packet is switched according to the connection set up through the control interface created via the divider.

invocations on switchlet interfaces to ensure that there is no interference between control systems. In-band policing and shaping mechanisms are used to enforce partitioning in the data path.

Fig. 1 is a schematic showing a control system, Tempest divider, and an MPLS switch. The control system, or an entity acting on its behalf, invokes `createSwitchlet()` on the divider's *management* interface. This results in a switchlet, shown in grey, with a switch control interface made available to the control system. The interface used in the Tempest is called *Ariel* and was developed locally [1], [2]. In the diagram the control system makes two Ariel invocations: `createContext()`, which constructs a description of a particular QoS, and `createConnection()` which creates the specified connection and associates it with the QoS description identified by the context identifier. In the Tempest MPLS implementation, connection establishment results in a suitable LIB entry being created, shown in Fig. 1 by an incoming packet emerging from the designated out-port with the specified out-label.

C. *Related Work*

MPLS is under active development as a set of standards. The IETF MPLS working group [9] is producing standards for the MPLS framework and architecture, and discussing issues of label distribution, encapsulation formats for a variety of layer 2 technologies, inter-operation with existing networks, and operations and maintenance. Provision of IP over ATM has been extensively addressed by a variety of approaches, many of which have been subsumed into the work of the MPLS working group. Of particular relevance is the work addressing provision of VPNs on an IP backbone using MPLS for packet forwarding and BGP for route distribution [10]. Labels are distributed as VPN routes. Within the backbone, routers do not maintain routing tables for every VPN, instead making use of paths established via MPLS. As is the case in the Tempest, label space is partitioned in order to ensure isolation between VPNs. However the concept of general resource partitioning to guarantee QoS to individual VPNs is not addressed.

Open signalling and control has become a widely-accepted mechanism for provision of VPNs in multi-service networks. The XBind project [11] produced a system similar to the Tempest in many ways, but with emphasis on providing abstractions to the service provider and user, rather than partitioning the resources at a low-level, and giving the provider and user complete freedom within their partition.

QoS for the Internet is being addressed by the IETF via its IntServ, and latterly DiffServ, proposals. IntServ has a "classical" view of QoS, providing RSVP as a signalling system for the setup of QoS-assured paths. DiffServ takes a more coarse-grained approach, suggesting that scalability considerations make provision of QoS to aggregates of traffic more reasonable; this allows simple service differentiation to be provided. It relies on policing at the edges and over-provisioning to ensure that service level agreements are met. Both these approaches have merit and are also actively undergoing standardisation. We claim in Section III-A that the system presented in this paper is capable of supporting both IntServ and DiffServ.

III. Motivation

Having described the background and some related efforts, we now motivate our work further by presenting example scenarios where the resource management facilities available within our framework might be exploited.

A. *Traffic Engineering: DiffServ and IntServ*

The resource management facilities of the Tempest can be used to support the IETF's DiffServ [12] and IntServ [13] architectures. The presence of connections with resource guarantees is naturally aligned with the IntServ model, and resource reservation using an RSVP control system has previously been implemented on a Tempest network [1].

DiffServ is supported by allowing FECs to be installed that examine the DiffServ codepoint and assign packets to separate paths within a VPN, or even to separate VPNs, based on the desired class of service. The Tempest allows multiple VPNs to be constructed over the same physical network. Each of these VPNs may support different levels of service for the same codepoint, allowing customers to choose their service provider based on the price/service combinations offered. At the same time, each provider can offer a full range of differentiated services. As a simple example, consider two providers, P_A and P_B, both offering differentiated services over a single Tempest network. P_A may support the expedited forwarding service, while P_B supports the assured forwarding service; both must support the best-effort service. They will both have service level agreements with their customers concerning the proportions of traffic that they expect to carry at the respective service levels.

In a non-Tempest VPN, there is no way to distinguish the best-effort traffic of P_A from the best-effort traffic of P_B. However, in a Tempest-MPLS network, P_A can attempt to offer a higher level of service *for the best-effort traffic*, by requesting a larger VPN from the physical network operator. Thus, whilst service level agreements with both P_A and P_B may state that half the traffic presented will be given the respective higher levels of service, and half will be treated as best-effort, the best-effort traffic presented to P_A should receive a higher level of service than the best-effort traffic presented to P_B. Section V-A demonstrates this type of situation.

B. *Web Server Clusters*

Cluster-based web servers are commonly used to increase request throughput and to provide service differentiation between client requests. Protection between classes of service on the server nodes has previously been addressed [14]. Here we consider providing that same protection within the network — in a geographically distributed cluster, service degradation as a result of network congestion may be as significant as resource starvation at the nodes.

The desired service differentiation can be achieved by creating a VPN for each service class, ensuring that each service receives guaranteed network resources. It is then possible to dynamically adjust the relative resource allocations to VPNs should the pattern of client requests alter.

Ingress routers assign every client request packet to a VPN according to policies specific to the web server. Examples of attributes that may determine which service class a packet belongs to are source address, allowing preferential treatment of certain customers, and information in the HTTP header, allowing differentiation between different types of requests. The control systems of the VPNs then route the packets to the appropriate node in the network, and can coordinate to perform load balancing and ensure fault tolerance by re-routing when appropriate.

C. *Internet Aggregate Reservation*

Service providers multiplex access to the core network for their customers and therefore have information about the current demands customers are placing on the network.

In addition, through pricing mechanisms, they have information about the value customers currently place on network service — estimates of the current utility functions of customers. There will be cases where they wish to provide higher levels of service to customers than current resource competition mechanisms in the core network allow. The framework we present provides an ideal mechanism for them to do so.

A provider purchases a lightweight VPN over the MPLS core. They then run a control system providing IP connectivity, such as ALASCA presented in Section IV-C, over this VPN. This effectively reserves them part of the public Internet, guaranteeing that their customers will receive better service, whilst retaining complete IP connectivity with the rest of the network. This reservation also takes place without requiring detailed traffic engineering knowledge from individual customers, without explicitly knowing customers' utility functions, and without requiring support for end-to-end QoS; customers simply perceive a more lightly loaded Internet. Knowing the current demand enables the provider to translate this higher level of service, and any associated costs, into charges to their customers.

IV. Implementation

Our implementation consists of two distinct parts: a mechanism to deal with IP packets entering the Tempest domain from an external switch, and an implementation of MPLS on Linux providing the necessary low-level resource partitioning. Packet filters are proposed in Section IV-A to deal with the former, and our implementation of the latter is described in Section IV-B. ALASCA, a system enabling interoperability with a non-Tempest domain, is described in Section IV-C.

A. Packet Classification

Before packets can be forwarded through the network, their label, and thus outgoing path, must be decided at the ingress node. We believe that this should be separated into two filtering stages, each packet being processed by at least one packet filter, as depicted in Fig. 2.

The *VPN filter*, owned and installed by the Tempest service provider, identifies the VPN to which a packet belongs. In standard MPLS terms this is FEC assignment with fixed granularity. Performance of this initial filter is critical as it must act on every incoming packet. This is especially true at the border of a domain, where packets belonging to a large number of different VPNs may attempt to enter the Tempest domain. It is therefore in the interests of the service provider to express this filter very simply.

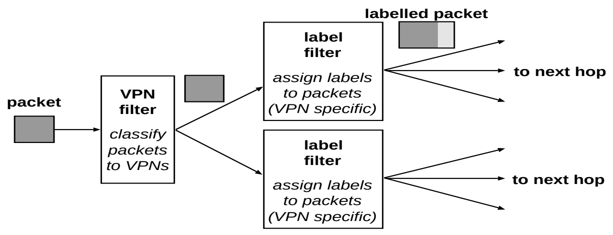

Fig. 2. Filters at the ingress to a Tempest-MPLS domain. Before forwarding, incoming unlabelled packets are first classified to a VPN, and then assigned labels according to a filter installed by the control system of that VPN.

This filter may make use of information contained in the packet header, such as source and destination address, DiffServ codepoint, VLAN tag or even layer 4 or other application specific information. Implementation of this filter would ideally be in hardware, and switching routers that can filter on layer 4 attributes in hardware at wire speed are now available. Since our implementation is on Linux and entirely in software, we use a simple destination address filter. Each VPN is allocated a private address space (an IP subnet), which can be reused as VPNs are created and destroyed.

The purpose of the *label filter* is to allocate labels to packets. This is essentially a routing decision, and therefore determined by the control system owning the VPN. We make use of dynamically installed packet filters, created based on arbitrary policy, and possibly comprising many further sub-filters. Filters for demultiplexing of packets have been used for some time [15], [16]; more recently, interest has grown in filter processing, such as that performed by firewalls and layer 4 switching [17].

We extended our Ariel interface to include a method for installing packet filters: `specializeIngressMapping()`. The control system defines its label assignment filter using the chosen packet filter language, and this is dynamically installed into the MPLS stack at the appropriate point. The presence of this filter is not always necessary, as there may be only one path to choose from. To provide robustness in the face of malicious users and packet filter code, the node on which the filters are installed should provide some form of resource containment. This would prevent the owner of one VPN damaging service to other VPNs by loading packet filters that consume excessive amounts of processor resources.

B. Packet Forwarding

The principle component of the implementation is a Linux kernel module, which operates at the lowest levels of the networking code and processes all incoming MPLS frames. It also passes frames to and from the IP stack at

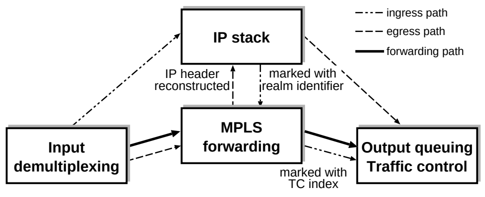

Fig. 3. Paths that a packet may take through the Linux networking architecture at an ingress, egress, or forwarding node.

ingress and egress nodes in a path. A high-level view of the relationship of MPLS with other components of the Linux networking stack is shown in Fig. 3. In this section we describe the operation of this module, which supports both Ethernet and ATM.

Both locally generated IP packets and those arriving from other non-MPLS nodes are passed to the MPLS module after traversing the standard Linux *netfilter* mechanism which allows packet-processing functions to hook packets at key points in the protocol stack. Packet classification is implemented using the ability of these netfilter mechanisms to mark packets with a *realm identifier*. This is stored as part of a routing table entry; deciding whether an IP packet should be placed onto a path thus forms a simple extension of the normal routing process. After passing through the routing code, an *ingress table* is indexed with the packet's realm identifier if it was given one. If a match is found, it tells us to which path the packet belongs, allowing an MPLS frame to be created from the packet. If no match is found, the packet is dropped; if the packet received no realm identifier originally, it is simply routed as normal.

At intermediate switches in a path, MPLS frames are received into the module in two ways. Ethernet frames are received via a previously installed packet handler which passes on all frames of the correct Ethernet data-link type. ATM frames received on MPLS-mapped VPI/VCI pairs are automatically passed on. On receiving an MPLS frame to be label switched, the LIB is accessed, using a combination of the frame's *port* and *label*. The method of determining the port and label for a given frame depends on the network type. For ATM, the incoming label is the VPI/VCI pair on which the frame was received, and the port is the interface on which it was received. To extend the concept of a connection to Ethernet, we define a port to be a *(local interface, remote interface)* pair. The local interface is uniquely identified by its Linux device index, and the remote interface by its six-octet MAC address.

Accessing the LIB with the *(port, label)* pair returns an entry containing information such as the output port, output label, and output QoS details. If a matching entry is found, then the incoming layer 2 headers are stripped from the frame and the outgoing header information prepended. If no match is found, the frame is dropped.

Recent versions of the Linux kernel include support for differentiated packet forwarding based on a flexible traffic control architecture [18]. We make use of this to provide QoS guarantees to paths. The architecture consists of three primary component types: queueing disciplines, classes, and filters. Instances of queueing disciplines are created within classes, and provide the basic mechanism by which QoS is provided. Packets are assigned to classes based on the result of filtering the packet using its TCI (Traffic Classification Index).

We use these mechanisms as follows: a TCI is stored as part of each LIB entry, and the kernel module associates a TCI with each frame as it is switched. When a connection with QoS is created, it causes the creation of a filter, and a class containing an instance of the *token bucket* queueing discipline. When a frame is switched it is filtered on its TCI to place it into the correct queue. If the path provides no QoS guarantees the TCI will be zero and the frame will traverse a low-priority best-effort FIFO.

We do not use the ATM interface's built-in support for QoS as most cards do not fully implement it. Rather, we create a dummy '`atm_mpls`' device for all MPLS-on-ATM frames, and specify best-effort connections to the ATM interface. MPLS-on-ATM frames then pass through the same traffic control code as those destined for Ethernet devices.

When a control system requests an MPLS switchlet, it specifies a label range for each port. Connection setup requests made through the switchlet's open interface are policed by the divider to ensure that only resources within the switchlet's specification are used. Note that the switch control interface is unchanged, except for the addition of methods to handle the ingress and egress of packets, described in the previous section. The interface between the Tempest and the Linux module supports methods to add and remove entries in the LIB, and to create new index filters, token bucket queues, and routing entries with unique realm identifiers.

Our implementation of MPLS was written specifically to support the Tempest and is incomplete with respect to the current IETF draft specifications [3]. In particular, it neither supports label stacking nor includes an LDP; label distribution supporting interoperability is dealt with by ALASCA, described in the following section. The implementation conforms to the drafts in all other respects, and is unique in that it can provide QoS for all paths, and its switching plane operates over both Ethernet and ATM. Section V-A contains a performance evaluation.

C. Label Distribution: ALASCA

In this section we present ALASCA, a control system that translates external routing information into connectivity within and through a Tempest domain. There are two principle component types in ALASCA:

The Domain Manager takes care of internal connectivity and resource management.

A Protocol Manager functions as a translator for external control systems. It interfaces between the external system and the Domain Manager, ensuring that the required connectivity is maintained within the Tempest domain, and advertising relevant routing information to the external system.

Fig. 4 depicts the structure of ALASCA, and the operations performed on receiving notification of a new subnet from an external IP domain. We now describe the components in more detail, and then discuss the advantages of ALASCA over MPLS using a standard LDP to establish connections across domain boundaries.

The Protocol Manager must potentially interface with many different external control systems, such as BGP, OSPF or RSVP. Consequently there are multiple types of Protocol Manager, one per protocol, and a domain may have multiple instantiations of any given type around its borders. A Protocol Manager is instantiated at a domain ingress/egress router, where external control information must be dealt with.

Where multiple VPNs intersect at the same edge node, each may require an instance of a Protocol Manager for the same protocol. These may then wish to peer with the same protocol entity in the external network (e.g. with the same BGP peer). This is made possible by the association of different IP addresses with virtual interfaces for the machine on which the Protocol Managers are running. The external routing entity then perceives the separate VPNs as completely separate autonomous systems, and operation of the protocol can continue as normal[3].

The Domain Manager implements all routing and control functionality for the Tempest domain it manages. It manages the switchlets — forming the VPN — that the control system has been allocated, and uses information from the Protocol Managers to drive routing and connection setup decisions. When new reachability information is received, the Domain Manager attempts to form routes across the VPN, and creates the paths where such routes exist. If this is successful, an egress mapping is installed at the receiving node to allow traffic routed through the newly-created path to be correctly injected into the external network. Finally, an ingress filter is installed at the other edge nodes, as described in Section IV-A. An analogous process takes place when an external peer withdraws a route.

[3]This is true in the case of OSPF and BGP, and any protocol that operates over IP; other routing protocols may require other solutions.

The core functionality of ALASCA — the establishment of connections across a Tempest domain — could also be achieved with the use of a standard LDP. We chose to develop a non-standard system in order to exploit the more advanced functionality that is possible with the Tempest. For example, a Protocol Manager may recover statistics from the edge-switch it is associated with in order to perform traffic-driven label distribution, or to perform admission control for request-driven distribution, such as provided by RSVP, or even for higher-layer protocols such as TCP [19].

Similarly, the Domain Manager may aggregate statistics from the switches in its domain, in order to bill customers, or to provide load information to Protocol Managers, in order to better support policies such as load balancing. A Domain Manager may also implement advanced routing algorithms to provide QoS routing facilities throughout its domain, or to provide route caching and fail-over for robustness.

V. Evaluation

In the first part of this section we show that our implementation performs reasonably by measuring performance in the control plane and in the data plane. We then demonstrate that inter-VPN resource mechanisms function as expected by isolating a best effort flow in one VPN from the effects of resource allocation changes in another. Finally we measure the cost of path setup using ALASCA. Scalability and robustness are discussed in the second part of this section.

A. Baseline Performance

The Tempest architecture is implemented over SPARC machines running Solaris 2.7, and PCs running Linux-2.3.99-pre5. The test network used in this work consists of three FORE ATM switches, a number of ATM video sources and sinks, and three PCs with 100Mbps switched Ethernet, and in two cases 155Mbps ATM interfaces. As described in Section IV-B, our MPLS implementation takes the form of a Linux kernel module and allows paths to be set up linking any combination of Ethernet and ATM interfaces.

To provide baseline performance figures, we measured connection setup/teardown times for the ATM switches using SNMP as the switch control protocol[4], and for the

[4]SNMP is used as the underlying switch control protocol in most

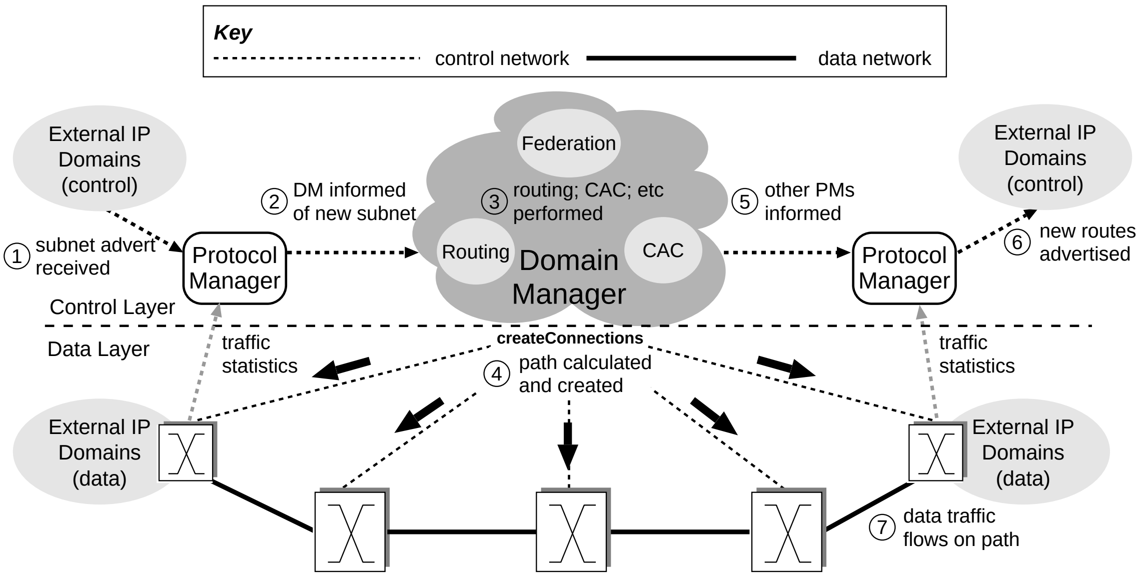

Fig. 4. An overview of the basic operation of ALASCA. The left-hand Protocol Manager receives a subnet advert from the control system in the external IP domain. The pertinent information contained in this is then passed to the Domain Manager, which calculates a route and sets up a path in its VPN. It then informs the other edge Protocol Managers that they may now advertise a route to this new subnet, to their respective external peers. Finally, data traffic may flow along the created path.

MPLS switches. The MPLS times were be significantly faster than for ATM, at around 350μs compared to approximately 7500μs. This difference is largely due to the much faster CPUs of the PCs compared to the SPARCs, and the fact that the divider effectively runs “on-switch” for the MPLS switches rather than “off-switch” as for the ATM switches. Similarly, throughput was measured for a path and found to be almost identical to that without MPLS (UDP transfer attains 93Mbps goodput point-to-point over a path traversing a mixture of Ethernet and ATM links, and 121Mbps goodput point-to-point over a path traversing only ATM links), as was “ping-pong” latency. Although one does not expect significant performance improvements by using label switching with such simple networks and low bandwidths, it is reassuring that performance is not reduced. The table of Fig. 5 shows the QoS implementation performing as expected. The figures are given for data throughput of UDP over connections with contracts as specified; account should therefore be taken of protocol processing and the other overheads involved.

To demonstrate inter-VPN resource allocation, we performed a test where the resource allocation to an entire VPN is altered. The topology is shown in Fig. 6, with two VPNs, both containing a guaranteed connection and two best-effort connections. VPN-A is initially allocated 40Mbps, and its guaranteed connection allocated 25Mbps. VPN-B is initially allocated 10Mbps, and its guaranteed

implementations of the Tempest Ariel interface. A basic GSMP implementation also exists, and work is progressing on support for GSMP v3.

Contract	[min, max] Achieved (Mbps)			
(Mbps)	1 Conn.	5 Conns.	10 Conns.	50 Conns.
50	46.43	–	–	–
25	23.53	–	–	–
10	9.32	[9.26, 9.31]	–	–
5	4.85	[4.63, 4.69]	[4.60, 4.64]	–
2	2.00	[1.93, 1.94]	[1.89, 1.93]	–
1	1.00	[1.00, 1.02]	[0.97, 0.98]	[0.92, 0.94]

Fig. 5. Table showing contracted and achieved bandwidths for MPLS connections created over Ethernet. Values are given for a single connection, and for five, ten, and fifty connections each with a separate contract.

connection allocated 8Mbps. The test then alters the guaranteed connection in VPN-A after 20s, from 25Mbps to 35Mbps. After 40s, the total VPN-B allocation is increased from 10Mbps to 30Mbps, and finally after 60s, the same allocation is decreased to 20Mbps.

The results are given in Fig. 7, showing achieved UDP goodput averaged over 500ms, against time. The initial 20s portion of the graph shows that the guaranteed connections are receiving the bandwidth allocated to them, and the total of the guaranteed and two best-effort connections in each VPN matches the total allocation to each VPN. The portion of the graph between 20s and 40s shows the guaranteed connection in VPN-A receiving its new higher allocation, and the two best-effort connections in VPN-A receiving correspondingly lower allocations; VPN-B is unaffected. The portion of the graph between 40s and 60s shows the allocation to VPN-B being increased from

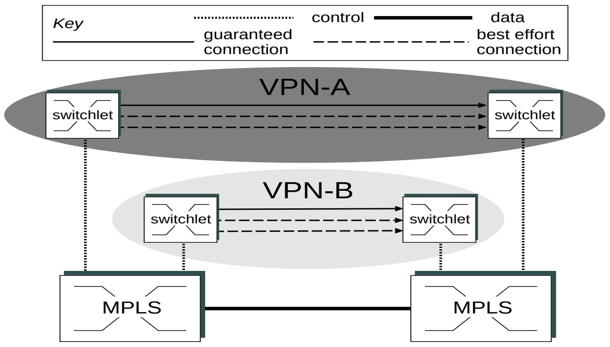

Fig. 6. The topology for the inter-VPN allocation test. VPN-A is initially allocated 40Mbps, and has three connections set up, one a 25Mbps guaranteed connection, the other two best-effort connections. VPN-B is initially allocated 10Mbps, and also has three connections setup, one an 8Mbps guaranteed connection, and the other two best-effort.

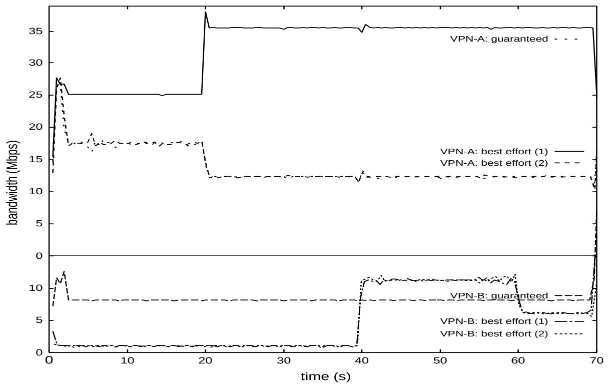

Fig. 7. Results for the inter-VPN allocation test. The topology is given in Fig. 6. After 20s, the allocation to the guaranteed connection in VPN-A was increased from 25Mbps to 35Mbps. After 40s, the total allocation to VPN-B was increased from 10Mbps to 30Mbps, and then after 60s, this allocation was decreased, from 30Mbps to 20Mbps.

10Mbps to 30Mbps. VPN-A is unaffected, as is the guaranteed connection in VPN-B; consequently, the best-effort connections in VPN-B use the newly allocated bandwidth. Similarly, when the allocation to VPN-B is reduced from 30Mbps to 20Mbps after 60s, VPN-A and the guaranteed connection in VPN-B are unaffected, the best-effort connections in VPN-B now reducing to fit within its new allocation.

A prototype of ALASCA, the LDP system, was implemented using a Python interface to the Tempest. Using the topology shown in Fig. 8 we measured the length of time to establish and then withdraw a route across the network. This is a 6-node network, with 2 edge MPLS nodes, 1 internal MPLS node bridging between ATM and Ethernet, and 3 internal ATM switch nodes (two FORE ASX-200s and one FORE ASX-1000). The components of ALASCA ran on three separate Linux PCs, communicating using the Python implementation of OmniORB [20]. The average

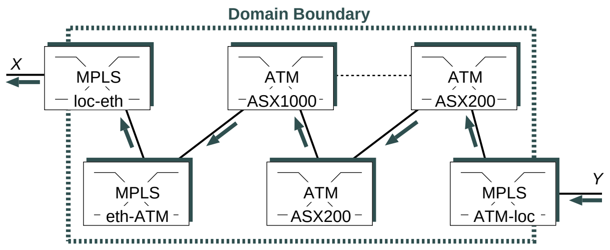

Fig. 8. The test network used to obtain the path setup time. A single test consists of a subnet being advertised at X, a path being created from Y to X, an ingress mapping installed at Y and an egress mapping installed at X, then the mappings being removed and finally the path being removed. This is repeated 100 times.

time from an edge node first receiving notification of a new subnet, the path for the other edge node being calculated and set up, the egress and ingress mappings installed and the route then being withdrawn, and the path correspondingly torn down and ingress and egress mappings removed, was 41ms. Noting that this is a prototype implementation, we believe that this is reasonable.

B. Scalability and Robustness

Whilst it is not possible to fully address scalability in a prototype implementation, we believe that our system should perform adequately in this regard. The lower layers of the MPLS implementation — the label switching functionality added to the Linux kernel — will scale at least as well as the IP routing functionality in the standard Linux distribution. The prototype LDP, ALASCA, is built over a CORBA implementation, using the C++ and Python variants of OmniORB [20]. We believe that it is capable of being extended to control large networks, either by sub-dividing domains as they become too large and then using interoperability techniques discussed in Section VI-A, or by distributing the implementation of the Domain Manager. How large a domain a single Domain Manager may reasonably control depends on the complexity of the protocols being considered, the stability of route information the network (the Internet) with which the ALASCA domain was interoperating, and the method by which the nodes are being controlled.

The current implementation creates a path to the ingress node from each edge node per subnet; in many situations this is likely not to be an appropriate policy for path creation. For example, in highly connected domains, one might rather choose to multiplex traffic from multiple subnets onto single paths, dependent on the requirements expressed by the external protocol. Similarly, it is likely that a full implementation would take advantage of some of the flexibility offered by this scheme, and implement path redundancy and fail-over for robustness. More advanced routing schemes including QoS routing might also be appropriate. Scalability of particular routing protocols is beyond the scope of this paper.

VI. Interoperability

Interoperability between the Tempest and standard ATM control systems has previously been considered [1], [21], [22]. Similarly, the MPLS working group requires that its solutions are deployable within the context of the current Internet. This section considers the problems of interoperability between Tempest and non-ATM systems in two parts — interoperability of control and interoperability of data.

A. Interoperability: Control

Interoperability of control is concerned with translation of connectivity information between domains. At the same time, ease of network management and network efficiency benefits by extending paths as far as possible; unfortunately, extending them across trust boundaries is difficult using standard LDPs. Packets are likely to require reassembly and reclassification at domain edges. Such systems provide no simple mechanism for enabling an adjacent domain to safely control a subset of the resources in the boundary switch.

The basic operation of ALASCA deals with cases where information being provided by external systems is used to provide IP connectivity and even QoS routing within the local domain. ALASCA also allows paths to be extended across domain boundaries through the use of a special Protocol Manager. This negotiates with a corresponding Protocol Manager in a neighbouring domain, and if successful the switchlet on the switch at the interface between the VPNs is shared between the Domain Managers. This means that when connections are created at that switch, they may be continued through the switch into the neighbouring domain. This is depicted in Fig. 9 and contrasts with the requirement for paths to be terminated and their packets reclassified into separate paths at each VPN boundary.

ALASCA provides interoperability with the Internet for applications using IP but running within a VPN. This interoperability is transparent to the user-application; however, applications wishing to make use of the features of the Tempest may do so. They may use a *service-specific control system*, appropriate to the particular application, or class of applications. Such a control system can then achieve interoperability with other networks relatively easily, by reusing the Protocol Managers from ALASCA and either extending or reimplementing the Domain Manager with appropriate admission control, routing or other policies.

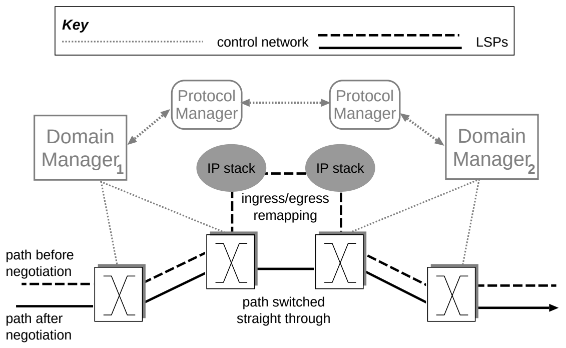

Fig. 9. An example of cooperation between Tempest domains. Traffic uses the upper path before negotiation, requiring intervention from the IP stacks, and the lower fully switched path after negotiation.

B. Interoperability: Data

Interoperability of data refers to the problems associated with the forwarding of data packets into and out of Tempest domains, and between different layer 2 technologies within a Tempest domain. MPLS is intended to support multiple layer 3 protocols over a variety of label switching technologies. The draft standards cover a variety of encapsulations, over ATM, Frame Relay, and PPP for instance, in addition to a generic encapsulation.

Our implementation supports both ATM and Ethernet, and provides an identical control interface for both styles of network. This provides straightforward means to bridge between ATM and Ethernet networks, and can easily be extended to other layer 2 technologies. Since the control interface presented is independent of the layer 2 technology, there is no reason for the control system to be aware of this; it merely controls a set of switchlets.

Mechanisms for dealing with the reception of unlabelled packets into a Tempest domain were dealt with in Section IV-A. Emission of packets from a Tempest domain into another Tempest or non-Tempest MPLS domain is straightforward. If a path has been constructed between the domains, the packets are placed onto this path as usual. When emitting packets into a non-MPLS domain the MPLS control system must be made aware that certain switches will have to strip the label from the packet and construct the correct layer 2 header for a packet of this type. For example, MPLS might be used to carry IPX rather than IP, and if an egress switch is emitting traffic onto an Ethernet, it must construct the correct Ethernet header for IPX.

Our implementation achieves this by the association of a type with the termination of a path. When a path terminates at a port, an *egress mapping* is created which allows the terminating switch to construct the required layer 2 header, perform cell-to-packet reassembly, and so on. The Tempest control interface was extended with the `createEgressMapping()` method to enable this.

VII. Summary and Further Work

We presented an implementation over MPLS of the Tempest, a framework permitting domain-wide resource management between resource-assured VPNs. A new MPLS kernel module for Linux supporting traffic shaping and policing, and operating over both Ethernet and ATM networks was described and evaluated. Issues pertaining to interoperability were explored, and ALASCA, an example control system supporting cross-domain connectivity was described. We showed that our implementation ensures network isolation between VPNs, and that it performs reasonably well.

There is scope for further work, including interoperability, the deployment of the filters at the ingress router, more efficient and accurate support for QoS in Linux, and of QoS provisioning across boundaries. Determining how general the initial VPN assignment filter can be, while still being computationally feasible, is of interest, as are possible optimisations, such as combination of filters. The interoperability and scalability provided by ALASCA has yet to be tested with realistic external traffic patterns, and the negotiation of QoS parameters across domain boundaries has also to be considered.

However, we believe that this combination of MPLS and the Tempest allows infrastructure to be built which supports hard partitioning of the network resources into VPNs. It permits customer-specific signalling systems to run over these VPNs, whilst reducing the scale of the network operator's management problem to the order of the number of VPNs, rather than the number of flows.

VIII. Acknowledgments

We wish to acknowledge the implementors of the Linux and the netfilter code in particular. We would like to thank Paul Barham, Tim Harris, Ian Leslie, Ian Pratt, and Timothy Roscoe for their comments. The MPLS Linux kernel module is available from `http://www.cl.cam.ac.uk/Research/SRG/netos/netx/`.

References

[1] S. Rooney, J.E. van der Merwe, S. Crosby, and I. Leslie, "The Tempest: A framework for safe, resource-assured programmable networks," *IEEE Communications Magazine*, vol. 36, no. 10, pp. 42–53, Oct. 1998.

[2] J.E. van der Merwe, S. Rooney, I. Leslie, and S. Crosby, "The Tempest—a practical framework for network programmability," *IEEE Network Magazine*, vol. 12, no. 3, pp. 20–28, May 1998.

[3] E. Rosen, A. Viswanathan, and R. Callon, "Multiprotocol label switching architecture," Internet Draft, Aug. 1999, Work in progress. Expires Feb 00.

[4] L. Zhang, S. Deering, and D. Estrin, "RSVP: A new resource ReSerVation protocol," *IEEE Network Magazine*, vol. 7, no. 5, Sept. 1993.

[5] B. Jamoussi et al., "Constraint-based LSP setup using LDP," Internet Draft, Sept. 1999, Work in progress. Expires Mar 00.

[6] P. Newman, G. Minshall, and T. Lyon, "IP switching: ATM under IP," *IEEE/ACM Transactions on Networking*, vol. 6, no. 2, pp. 117–129, Apr. 1998.

[7] J.E. van der Merwe and I. Leslie, "Switchlets and dynamic virtual ATM networks," in *Integrated Network Management V*, Aurel Lazar, Roberto Saracco, and Rolf Stadler, Eds., San Diego, USA, May 1997, IFIP & IEEE, pp. 355–368, Chapman & Hall.

[8] I. Leslie, D. McAuley, R. Black, T. Roscoe, P. Barham, D. Evers, R. Fairbairns, and E. Hyden, "The design and implementation of an operating system to support distributed multimedia applications," *IEEE Journal on Selected Areas in Communications*, vol. 14, no. 7, pp. 1280–1297, Sept. 1996.

[9] IETF, "MPLS working group," `http://www.ietf.org/html.charters/mpls-charter.html`, July 2000.

[10] E. Rosen and Y. Rekhter, "BGP/MPLS VPNs," RFC2547, Mar. 1999.

[11] Aurel A. Lazar, "Programming telecommunication networks," *IEEE Network Magazine*, pp. 8–18, September/October 1997.

[12] D. Black, S. Blake, M. Carlson, E. Davies, Z. Wang, and W. Weiss, "An architecture for differentiated services," RFC2475, May 1998.

[13] J. Wroclawski, "The use of RSVP with IETF integrated services," RFC2210, Sept. 1997.

[14] M. Aron, P. Druschel, and W. Zwaenepoel, "Cluster reserves: A mechanism for resource management in cluster-based network servers," in *Proceedings of ACM SigMetrics 2000*, Santa Clara, CA, USA, June 2000, pp. 90–101.

[15] J.C. Mogul, R.F. Rashid, and M.J. Accetta, "The packet filter: An efficient mechanism for user-level network code," in *Proceedings of the 11th Symposium on Operating Systems Principles (ACM SIGOPS)*, Austin, Texas, USA, Nov. 1987.

[16] D.R. Engler and M.F. Kaashoek, "DPF: Fast, flexible message demultiplexing using dynamic code generation," *Computer Communication Review*, vol. 26, no. 4, pp. 53–59, Oct. 1996, Proceedings of SIGCOMM August 1996.

[17] P. Gupta and N. McKeown, "Packet classification on multiple fields," *Computer Communication Review*, vol. 29, no. 4, pp. 147–160, Oct. 1999, Proceedings of SIGCOMM September 1999.

[18] W. Almesberger, J.H. Salim, and A. Kuznetsov, "Differentiated services on Linux," in *GLOBECOM: General Conference*, Dec. 1999, vol. 01b, pp. 831–836.

[19] R. Mortier, I. Pratt, C. Clark, and S. Crosby, "Implicit admission control," *IEEE Journal on Selected Areas in Communications*, Q4 2000, to appear.

[20] AT&T Research Labs, "Omniorb v2 support page," `http://www.uk.research.att.com/omniORB/`, July 1998.

[21] S. Rooney, *The Structure of Open ATM Control Architectures*, Ph.D. thesis, Cambridge University Computer Laboratory, UK, Feb. 1998, Available as Technical Report No. 451.

[22] H. Bos, "Application-specific policies: Beyond the domain boundaries," in *Integrated Network Management VI*, Morris Slo-

man, Subrata Mazumdar, and Emil Lupu, Eds., Boston, USA, May 1999, IFIP & IEEE, pp. 827–840, Chapman & Hall.

APPENDIX

MPLS ON LINUX: USERS' GUIDE

I. INTRODUCTION

Multiprotocol Label Switching (MPLS) is a networking technology that provides an architecture for applying ATM-style *switched paths* to any number of different network technologies (even connectionless ones, such as Ethernet). Further discussion of the MPLS architecture and framework can be found in a number of IETF working drafts.

II. MPLS CONCEPTS, AND HOW TO USE THEM

'`mpls`' is a simple command line utility for creating and deleting MPLS ports, switch paths, ingress mappings, and egress mappings. Each of these is described in more detail, together with some higher-level detail and explanation, in the following sections.

A. Specifying ports

In our MPLS implementation, ports serve a number of purposes:

- As the destination of a switch mapping, they specify which local network interface an MPLS frame should be transmitted on.
- On a broadcast network, such as Ethernet, they also specify the next-hop MAC-layer address.
- Each port has its own label space associated with it. For instance label 200 on port 0 and label 200 on port 1 are *entirely independent*.

There are currently three types of port that can be defined: *local*, *ATM*, and *Ethernet*. We describe each of these separately in turn.

A.1 ATM

These are conceptually the simplest to understand. An ATM port is uniquely specified by a local ATM interface index. When transmitting MPLS frames via an ATM port, the label is stored in the AAL5 encapsulation header as the VPI/VCI pair. The VPI/VCI pair is constructed from a 32-bit MPLS label by mapping the most significant 16 bits of the label to the VPI, and the least significant 16 bits to the VCI.

A.2 Ethernet

Since Ethernet is a broadcast network, a port is uniquely defined by a (local network interface, remote network interface) pair. The local interface is specified by its device name, and a remote interface is defined by its MAC address.

Since the Ethernet link-layer header contains no equivalent of an ATM label field, MPLS frames transmitted on Ethernet have a small *shim header* inserted between the Ethernet header and the network-layer header. The shim header has the format described in the IETF draft specification for MPLS label encapsulation. Note that this format constrains Ethernet labels to 20 bits: the most significant 12 bits of a local label are ignored.

A.3 Local

Local ports are useful as dummy ports to which ingress mappings can be bound. An ingress mapping maps from some protocol-specific set (also known as a forwarding equivalence class, or FEC) of packets to the ingress of an LSP, specified as an *input* port and label. The *output* port and label, and other parameters such as traffic classifier, are looked up in the switching table. See the section on creating ingress mappings for more information.

A.4 Example port creations

The following are some example port creations. In the case of the Ethernet example, note that we must specify both the local interface name and the remote MAC address.

```
mpls ap 0 l                          # Port 0 is Local
mpls ap 1 e eth0 00:10:b5:05:26:0b # Port 1 is Ethernet
mpls ap 2 a 0                        # Port 2 is ATM, interface 0
```

B. *Switch mappings*

If we wished to create a new mapping from port 2, label 20, to port 4, label 35, we would issue the following command:

```
mpls as 2 20 4 35   # 'as' == Add Switch mapping
```

Things become more complicated if we wish to specify some quality-of-service (QoS) constraints on packets traversing an LSP. Recent versions of the Linux kernel support differentiated forwarding of packets based on a flexible architecture of filters, classes and queuing disciplines. In particular, it is possible to filter outgoing packets based on a special *traffic classification* index which can be specified as part of an MPLS switch table entry. The index filter can be used to pass packets to different *traffic classes* based on the value of this index. Each class may be associated with a different queuing discipline supporting differing forwarding constraints (eg. limiting transmission bandwidth).

As an example of this, suppose we wish to limit the above mapping to a sustained traffic rate of 30Mbps, using a token bucket algorithm. The following set of cryptic commands would do the job, assuming port 4 is an Ethernet port bound to local interface `eth0`:

```
tc q a dev eth0 root handle 10: cbq bandwidth 30Mbit avpkt 1500
tc c a dev eth0 parent 10:0 classid 10:1 cbq bandwidth 30Mbit rate 30Mbit maxburst 20 avpkt 1500
tc c a dev eth0 parent 10:1 classid 10:100 cbq bandwidth 30Mbit rate 30Mbit maxburst 20 avpkt 1500
tc q a dev eth0 parent 10:100 tbf limit 65536 burst 2048 rate 30Mbit
tc f a dev eth0 parent 10:0 protocol mpls prio 100 handle 151 tcindex classid 10:100
mpls as 2 20 4 35 151  # Output class index is 151
```

The commands '`tc`' and '`ip`' (which is used later when defining ingress mappings) are both included in Alexei Kuznetsov's iproute2 package. The latest version of the iproute2 package should be downloaded[5] and then patched with the `iproute2-patch` file included with the MPLS for Linux distribution. The patch updates '`tc`' to be MPLS-aware — without it the above example will not work!

C. *Ingress mappings*

NOTE: Currently only IPv4 ingress mappings are supported!

An ingress mapping is required at the first LSR in an LSP. By specifying constraints on higher-level protocol fields, it declares which packets belong to the LSP and will be routed via MPLS rather than the standard routing code.

In the case of IP packets, the existing Linux routing code is used to classify packets. When creating new routing table entries, it is possible to specify a *realm* to which packets matching that entry belong. This realm is stored in a special field of the packet's descriptor [6] where it can be referenced during later stages of packet processing.

So, creating an IP ingress mapping consists of three steps:

- Declare a new routing entry which matches packets which belong to the new LSP. Specify a unique *realm* for packets which match this entry.
- Declare a new MPLS ingress mapping, which specifies which input port and connection packets with the new realm identifier map to.
- Declare a new MPLS switch mapping, which takes the input port and label to the *output* port and label.

For example, suppose we wish to tunnel all packets destined for subnet 128.232.8.0/24 through an LSP, the first hop of which will be on port 2, bound to local interface `eth0`, with label 20. Suppose also that port 0 is a specially-created local port which has label 30 unbound. The following thee commands would be required:

```
mpls ai 100 0 30     # map realm 100 to port 0, label 30
mpls as 0 30 2 20    # map port 0, label 30, to port 2, label 20
ip route add 128.232.8.0/24 dev eth0 realms 100
```

D. *Egress mappings*

NOTE: Currently only IPv4 egress mappings are supported!

An egress mapping is required at the final LSR in an LSP. Such a mapping specifies, for a given input port and label, the protocol stack that packets on that LSP should be handed off to (IPv4, for example). There may also be some number

[5] It's available from `ftp://ftp.inr.ac.ru/ip-routing/iproute2-current.tar.gz`.
[6] The Linux kernel calls these descriptors *skbuff*s

of protocol-specific parameters associated with an egress mapping. In the case of IPv4, the local input interface that we wish to tell the IP stack that the LSP packets are received on must be specified. This is because the Linux fast routing code creates a hash value which includes this interface.

As an example, suppose we wish to pass to the IPv4 stack all packets we receive on port 2 that have label 30. The IP stack should think that the packets were received on local interface `eth0`:

```
mpls ae 2 30 eth0   # Pass port 2, label 30, to IPv4 stack
```

III. Accessing MPLS via netlink sockets

The MPLS module can also be configured directly from your own programs by creating a special *netlink* socket. `mplslib.c` demonstrates how to create such a socket and use it to communicate with the MPLS kernel module. In most cases I would recommend that you use this file, together with `mpls.h`, within your own applications, as it provides a simple and clean alternative to using the arcane netlink interface directly.

For lower-level details, the function `netlink_rcv_skb` in the new kernel file `net/mpls/mpls.c` is the place to look to find precisely how the control messages are interpreted. Each of the netlink command types requires additional information in the command-specific portion of the message. This information is specified in structures which are defined in the new kernel file `include/linux/mpls.h`.

IV. Label Stacking extension

The label stacking extension was added by Phil Quiney `(pquiney@nortelnetworks.com)` to support an in house project that needed it.

The MPLS module is able to support basic label stacking in the following scenarios:

- At ingress the LER is able to push more than one label.
- At an intermediate LSR, the label stack can be popped and an additional label stack pushed.
- At the egress LER the label stack can be completely popped and the packet delivered as normal.
- An additional feature allows an LSR to pop the stack a number of times and use an inner label to forward the packet. This was intended to allow simulation of tunnels where 'penultimate hop pop' by a tunneled through LSR was not available.
- An output label of 0 (zero) for a given port/label has additional meaning when the LSR receives a packet with more than one label. As pushing the zero would be illegal the LSR instead does nothing and effectively performs the 'penultimate hop pop'.

A. Compilation Options

The file `include/linux/mpls.h` has additional compile time flags defined as follows:

- LABEL_STACKING - comment out to remove label stacking support
- LABEL_STACK_DEPTH - set to number of labels you need in the stack. Note that increasing this value will increase the memory overhead of *every* skbuff that is allocated!

Additionally, `net/mpls/mpls.c` has an optional macro, LSTACK_TRC, to enable trace debugging of the label stacking code.

B. Example Label Stack Operations

```
mpls as 1 20 2 36:37  # Push 36 then 37 when switching
mpls ds 1 20          # delete the above
mpls al 2 30 1        # When this label is received pop it
                      # & forward based on label underneath
mpls dl 2 30          # delete the above
mpls as 1 20 2 0      # Pop label 20 & either push 0 if
                      # label is 'bos' or simply forward
                      # if not - thus 'penultimate hop pop'
```

C. Proc file changes

If available the file `/proc/net/mpls` has been changed to display the label stack on switch entries as well as those set to perform the 'pop only' function. I (pquiney) take full responsibility for breaking the neatness of the output....

V. IP Fragmentation

The first attempt at IP Fragmentation was added by Phil Quiney `(pquiney@nortelnetworks.com)` to support an in house project that needed it (yes the same one as above).

An IP Fragmentation mechanism has been implemented which works by ensuring that there is always space for LABEL_STACK_DEPTH shim headers plus the data of the packet to fit in the MTU of the link. This is checked for when the packet has no labels on it which allowed the re-use of an existing kernel function and avoided some complex pointer manipulation. In practice this will mean that it will only be checked for at the ingress (LER). It does assume, however, that the MTU across the MPLS network is the same (or greater) than at the ingress. For 'lab networks', this is unlikely to be a problem.

The code was derived from existing kernel code (function: ip_fragment) which has the advantage of being tested and working. There was a module load problem with an unresolved reference to 'ip_options_fragment'. The current fix is to copy the code for this function verbatim. It would be useful if this was resolved 'properly' as it will need extra work to make sure any further changes to this function in the kernel are also applied to the local copy.

A. Compilation Options

The file `include/linux/mpls.h` has an additional compile time flag IP_FRAGMENT which, if defined, compiles in the code for IP fragmentation. If this is set then an additional message will be displayed when the module is loaded - to the effect that IP Fragmentation is enabled.

VI. Enhanced Ingress Filtering

An enhanced replacement for the Ingress Mappings described above allows packets to be mapped onto an LSP based on the so-called 5-Tuple Match. This match works on combinations of source address, destination address, source port, destination port and protocol. To enable this the kernel should be built with the CONFIG_NET_CLS_TCINDEX flag set. (this is under 'Network Options'/'QoS and/or fair queueing' as 'TC index classifier'). Note that although it offers to be built as a module you should build it in to the kernel. This is because it dosen't work as a module - at least with kernel 2.3.99pre7. The module loaded, the filter could be configured but no packets were matched. The same configuration applied to a 'built in' version works fine.

An extra message appears when the mpls module is loaded which indicates that the 5-Tuple support is present in the code.

A. Sample Configuration

A sample configuration is shown below. The filter does not need all the match entries in order to work, you can for example match on only source and destination address. The value it 'marks' the packet with is used instead of the 'realms' value to 'ip route add'. Setting up the 'realm' is not required if using the 5-Tuple match filter.

```
#!/bin/sh

# Change this to pick up path to 'tc'
TC=/path_to_tc/tc
#change this to be the ingress device
DEVICE=eth1
# Simplify this with a macro
TCP="ip protocol 6 0xff"

# Ingress filter to give a tcindex of 100 - replacing the realms 100
# via 'ip route add' used previously
$TC qdisc add dev $DEVICE handle ffff: ingress
```

```
# Add filter for 5 tuple match
# Note that flowid is in hex 64H = 100
# Match based on source address 10.10.0.100/32
#                destination address 10.11.0.100/32
#                source port 20
#                destination port 1234
#                protocol TCP
$TC filter add dev $DEVICE parent ffff: protocol ip prio 1 u32 \
match ip dst 10.11.0.100/32 \
match ip src 10.10.0.100/32 \
match $TCP \
match ip sport 20 0xffff \
match ip dport 1234 0xffff \
police rate 10Mbit burst 900k drop flowid :64

# View filter with
# $TC filter ls dev eth1 parent ffff:
# Delete filter with
# $TC filter del dev eth1 parent ffff: protocol ip prio 1
```

B. Bogus tc_index values

In testing the 5-Tuple code with tracing switched on it has been noticed that the ’classifier’ represented by the tc_index occasionaly appears to be set to a random non-zero value. This would imply that the skb structure is not ’clean’ - I don’t believe it can be a filter match as when packets match the filter the tc_index is ’as expected’. The filter is the only thing that is supposed to be using the tc_index field as far as I know.

However this could lead to some strange side effects if the bogus value happened to be the same as a value that a filter produces. You will get packets switched to the LSP which shouldn’t be although packet retransmit should cope.