MPLS on Linux: Users' Guide

Keir Fraser (`kaf24@cl.cam.ac.uk`), Phil Quiney

September 4, 2000

1 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.

2 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.

2.1 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.

2.1.1 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.

2.1.2 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.

2.1.3 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.

2.1.4 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
```

2.2 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[1] 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!

2.3 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 [2] 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:

[1] It's available from `ftp://ftp.inr.ac.ru/ip-routing/iproute2-current.tar.gz`.

[2] The Linux kernel calls these descriptors *skbuffs*

```
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
```

2.4 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 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
```

3 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`.

4 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'.

4.1 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.

4.2 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'
```

4.3 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....

5 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.

5.1 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.

6 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.

6.1 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
```

6.2 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.