A Prototype MPLS Packet Trailers Processor

KEVIN MITCHELL
Agilent Laboratories

A scheme for monitoring per-hop characteristics of MPLS LSPs was recently proposed in [Mitchell 2004]. The paper described the trailer packets, and the trailer processor functionality required to support them, at a fairly superficial level. This made it difficult to assess the feasibility of supporting this technique in small devices embedded in line cards or GBICs. This paper goes some way towards remedying this situation by describing the implementation of a prototype trailer processor.

Key Words and Phrases: MPLS, packet trailers, network monitoring, OSS

1. INTRODUCTION

After a slow start, MPLS is becoming widely deployed in the Internet, particularly in QoS-critical environments requiring some form of traffic engineering. This creates an associated need for monitoring the performance of MPLS tunnels. We can monitor end-to-end packet loss, delay and jitter for a tunnel using approaches adapted from the IP environment. However, when a problem arises we would like to pin down the culprit more precisely. This requires, at the very least, knowing which label-switched path (LSP) is currently being used to support this tunnel. Ideally, we would also like to know the per-hop performance statistics for the LSP. Collecting statistics at this fine level of granularity, in a cost-effective fashion, is a challenging task. Probing the MPLS MIBs can only give us some of this information, and will introduce a heavy load on the network. Approaches such as NetFlow suffer from similar problems. Using traditional passive probes to collect such data is expensive, particularly when they are deployed pervasively.

[Mitchell 2004] proposed an active approach, injecting packets called MPLS trailers into the LSP to collect the performance details in an efficient fashion. MPLS is particularly well-suited to the use of active measurements. We can inject a monitoring packet at the ingress, be sure it is treated identically to all the other packets in the LSP as it traverses the path, and then be able to automatically separate out the measurement packet at the egress, or penultimate hop. The trailers proposal takes this a step further, proposing the use of the label stack to indicate the presence of a monitoring, or trailer, packet. A suitably aware router, on detection of such a packet, would append performance statistics to the packet as it passed by. If all MPLS routers were aware of such trailers then we would have an efficient way of gathering per-hop data across an LSP. We would start by injecting a trailer packet with an empty payload at the ingress. On delivery to the egress, the packet would contain a record of which router interfaces had been traversed, along with timestamps, packet and byte counts. These packets could then be forwarded to a monitoring station, using the usual IP forwarding mechanism, for further analysis.

In this document we use the term *trailer processor* (TP) to denote the device that appends measurement details to trailer packets. Ideally, we would like these processors to be situated as close to the "edge" of the router as possible; immediately prior to any queues on packet receipt, and after all queues when transmitting a packet. This suggests the TPs would be best implemented in hardware or firmware on the linecards. Unfortunately, Agilent does not manufacture routers, and so we must devise alternative strategies for retrofitting such measurement functionality to existing devices. The *Smart GBIC* proposal is one such attempt. A GBIC, short for GigaBit Interface Converter, is a transceiver that converts serial electric signals to serial optical signals and vice versa. GBICs are hot-swappable, allowing a device to be adapted for either optical or copper applications. A smart GBIC augments a traditional GBIC with additional functionality to perform monitoring and measurement activities. An SGBIC is particular attractive from the trailers perspective as it is situated at the very edge of the device. Clearly, the processing and memory capabilities of an SGBIC would be very limited. This raises a number of questions concerning the feasibility of supporting the packet trailers proposal in such a small form factor. In this paper we describe a prototype of a trailer processor that attempts to answer at least some of these questions.

Author's address: K. Mitchell, Agilent Labs Scotland, South Queensferry, Scotland EH30 9TG

1.1 The Prototype

With neither access to the internals of a router, or a smart GBIC, we are forced to develop the trailer processor prototype as an external device, placed inline between a router interface and the rest of the network. This situation is illustrated in Figure 1. Note that we view the trailer processor, at least logically, as being part of the router it is attached to. This creates an asymmetry when the TP is connected between two routers, R_1 and R_2, using point-to-point links. One of these routers, R_1 say, will "own" this processor, where the choice depends on which port of the TP a router is connected to. The asymmetry becomes apparent when the TP needs to generate packets. It is not visible to any of the devices on the network, and uses the MAC and IP addresses of the host router when constructing new packets. In the prototype a third interface is used to debug the probe. It has an IP address, acquired via DHCP, but this interface is not an essential part of the device. In particular, it is not used to transmit or receive packets as part of the core functionality of the trailer processor.

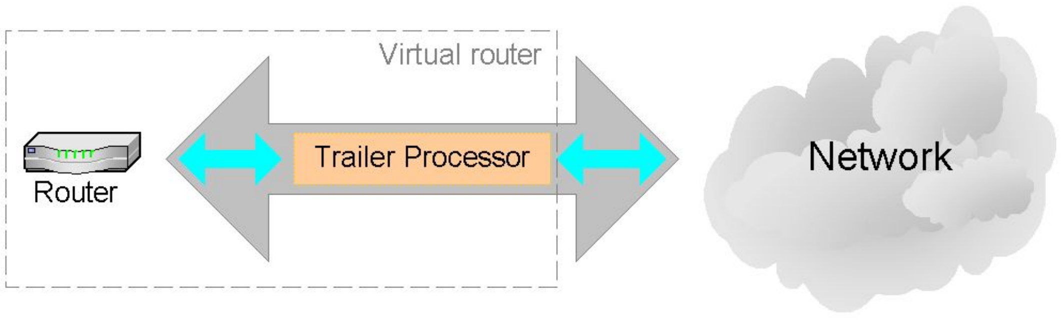


Fig. 1. Trailer processor connections

1.2 Click

To allow the prototype to be connected to the network transparently, and to provide a reasonable level of performance, we cannot just run an application in "user mode". We could modify the networking code in a Linux kernel to add support for trailers, and to support the required level of anonymity. However, for prototyping purposes, Click[Kohler et al. 2000] provides an interesting alternative. It allows us to achieve a good balance between packet forwarding speed and development time. Quoting from the abstract in the Click paper, "A Click router is assembled from packet processing modules called *elements*. Individual elements implement simple router functions like packet classification, queuing, scheduling, and interfacing with network devices. A router configuration is a directed graph with elements at the vertices; packets flow along the edges of the graph." The prototype trailer processor implementation consists of a small number of custom click elements, together with a Click script to build the required packet-processing pipeline.

The rest of this document is structured as follows. Section 2 describes the state we need to maintain for this application. Section 3 discusses the management of this state, and Section 4 defines the format of the trailer packets themselves. The new Click elements, and the configuration that exploits them, are presented in Section 5, followed by the conclusions and suggestions for future work.

2. TRAILER STATE

Trailer processors must be small and cheap. Furthermore, unless deployed in cooperation with other devices on a line card, they must maintain their own state. We assume that there will only be a relatively small amount of memory available for such devices. In the context of Smart GBICs, an upper memory limit of 128KB for applications has been quoted. This complicates the design of a trailer processor considerably. We have no control over the number of LSPs traversing the TP. Furthermore, some of these may use multiple EXP settings in the MPLS shim headers to split the path into multiple channels, each requiring their own counters due to the differing QoS treatments they may receive. In an extreme case, we may have thousands of LSPs, each using all eight EXP settings. Furthermore, to allow a monitoring station to interpret the trailer packets correctly, we may also need to keep track of the associated tunnel details, and the mapping from label to tunnel. Allocating byte and packet counters for all eight possible EXP settings, together with the tunnel details for each label, would only allow us to track a few hundred LSPs in each direction. Whilst this may be sufficient for an edge router, it may not be adequate for a core device. This suggests we may need to aggressively reclaim storage when LSPs expire, or are used infrequently, and adapt our monitoring strategy depending on our current memory utilization.

If we had access to the MPLS ingress router software then the ingress routers could be modified to generate periodic trailer packets for each active LSP. However, in the context of the Smart GBIC incarnation of the packet trailers approach, we must assume we do not have access to these devices; the trailer processors themselves must be responsible for the generation of these packets. Ideally, we would like one TP per LSP to be responsible for the generation of trailer packets, namely the TP currently closest to the ingress of the LSP. All other trailer processors along the path merely append their trailer entries to these packets as they pass by. Whilst the TP responsible for initiating the trailer packets for an LSP may need to maintain the tunnel details for this LSP, the tunnel details are not essential for the downstream nodes. This suggests that we may need to maintain tunnel details for some entries but not others.

In summary, a trailer processor needs to keep track of labels, counters and tunnels, but we can't necessarily keep track of everything if we want to support a large number of LSPs. Furthermore, at one extreme we may need to maintain a label, tunnel, and eight "counters" for an LSP. At other extreme, just the label and one "counter" may be sufficient if the EXP bits are unused and the TP is downstream of the trailer generation point. Although, on average, we might expect a similar number of LSPs to be traversing in each direction, we have no guarantees that this will be the case. Indeed, in certain scenarios there will not be such symmetry. This analysis suggests the need for a flexible storage structure, where all data is fetched from a common pool.

We start with the structure used to represent label entries, as this forms the scaffolding that holds the other entries. We have noted that not all entries need to maintain details of the tunnel they are associated with, so the tunnel details should not be stored in the label entry. The number of counters in use varies between LSPs, and also potentially over time. The counters should therefore be stored separately as well, leading to the model illustrated in Figure 2.

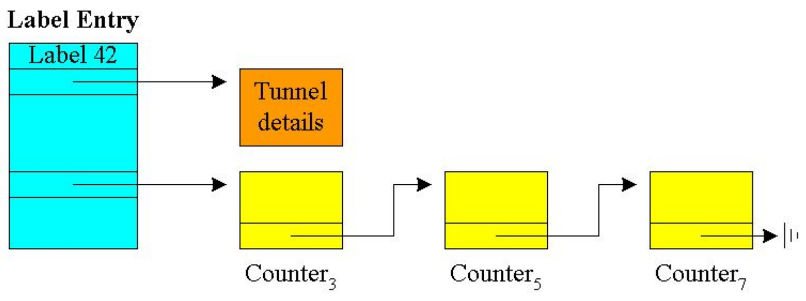


Fig. 2. Label entries, tunnel details and counters

On reception of an MPLS packet, we need to retrieve the corresponding label entry, assuming there is one. This suggests the need for a hash table to map from labels to their entries, in the form of indexes into the pool table. We use chaining to deal with hashing clashes; all the label entries with the same hash value are chained together using a field within each label entry, as illustrated in Figure 3.

Given the variability in the traffic we may have to observe, it would seem prudent to allocate label, tunnel and counter entries from the same pool. This would allow us to monitor a small number of labels, using a large number of counters, in one scenario, whilst supporting a large number of labels with a small number of counters in another. It also makes sense to share the pool between the incoming and outgoing traffic. There may not be the same amount of traffic in each direction, and we wish to avoid the situation where one pool is exhausted whilst the other pool is underused. If the data structures representing labels, tunnels and counters are of differing sizes this raises the risk of fragmentation. We could, of course, add a storage compaction mechanism to the system. However, a simpler approach, and one arguably more suitable for something like a Smart GBIC, uses the same size "cell" for each of the main structures. Of course there is a price to pay for avoiding fragmentation in this way, namely the need to set the cell size to the size of the maximum structure we want to allocate from the pool. Note that it is not essential to store all state in this pool, only that which is allocated as part of the LSP-tracking process. So, for example, the TP maintains

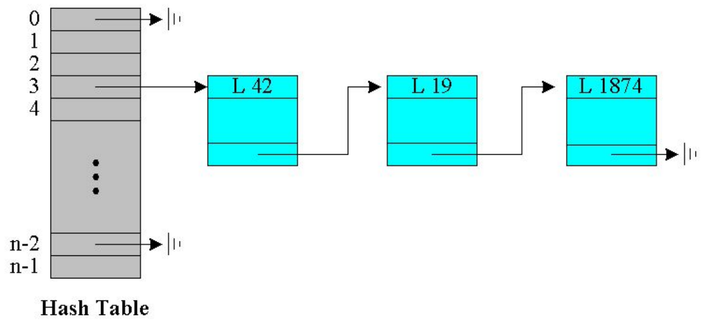


Fig. 3. Chained hash table for label entries

a cache of MAC addresses for adjacent routers. The bounds on the size of this table are not related to the LSP entries, are relatively small, and so we can store these entries in their own table. By judicious use of such "external" tables, we are able to store details of labels, tunnels and counters using a cell size of just twelve bytes. We would require ten cells to store an LSP using all eight EXP values, together with its tunnel details. At the other extreme, in some cases we can still add useful information to a trailer packet, i.e. interface and timing details, with no cells allocated for the LSP.

A uniform cell size greatly simplifies storage allocation as the currently unallocated cells can be simply chained together in a free list, illustrated in Figure 4. To save space we store pointers, for example to counter and tunnel entries, as indexes into the pool table. We use 15-bit fields for these references, thus supporting a maximum pool size of approximately 400KB. We do not use the first entry in the pool, allowing us to use 0 to indicate a "null" pointer.

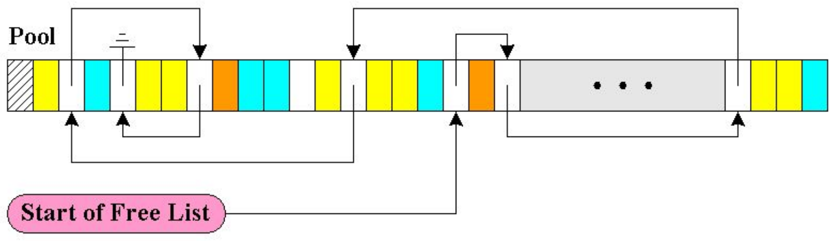


Fig. 4. The freelist

2.1 The Label Entry structure

The structure of each label entry is shown in Figure 5. It follows the layout suggested earlier. However, a few of the fields deserve some additional comments. We have already observed that if we detect we are the most upstream TP for an LSP then we should be responsible for initiating the generation of trailer packets for the LSP. In such cases the `initiate_trailers` field will be set to 1. Furthermore, the `trailer_due` field will record how many seconds remain until we must generate the next trailer. However, this field has a duel purpose. If we are not initiating trailers then the field records how long we must wait until we can take over the role of trailer initiator. Trailers are generated asynchronously, and so we need to know what MAC address to use for the next hop. The `mac_index` field is an index into a table of MAC addresses providing this information. We discuss this aspect of the processor later, in Section 5.2.

2.2 The Counter Entry structure

The structure of the counter entries is illustrated in Figure 6. We allow the byte and packet counts to overflow by wrapping. However, this will occur sufficiently infrequently, relative to the generation of trailer packets, that the monitoring station should be able to detect and correct for such occurrences. The purpose of the `mark` bit is explained in Section 3. An LSP may use multiple settings of the EXP bits to distinguish between different subflows. Where memory permits, it is important to track the performance of each subflow

```
struct LabelEntry {
  unsigned label: 20;              // The label for this entry
  unsigned expires: 12;            // Time to expiry, in seconds
  unsigned outgoing: 1;            // Outgoing or incoming entry
  unsigned tunnel_index: 15;       // Link to optional tunnel details
  unsigned initiate_trailers: 1;   // Are we generating trailers?
  unsigned first_counter_index: 15; // Link to counters list
  unsigned trailer_due: 8;         // Time to next trailer
  unsigned lock: 1;                // Used to gain exclusive access
  unsigned mac_index: 7;           // Link into MAC table
  unsigned mark: 1;                // Garbage collection bit
  unsigned next_index: 15;         // Next entry in hash list
};
```

Fig. 5. The LabelEntry structure

separately. After all, the whole point of making such distinctions in an LSP is to allow the routers along the path to provide different levels of treatment to each subflow. This will alter the QoS experienced by each subflow, and we would like to determine the level of service received in each case. For this reason we may need to associate multiple counter entries with each label entry, with new counters being created in response to the observed traffic. All the counters for a label are linked together via the `next_index` field.

```
struct CounterEntry {
  uint32_t packet_count;      // The packet count for this sub-flow.
  uint32_t lo_byte_count;     // Bottom 32 bits of the byte count.
  unsigned hi_byte_count: 13; // Top 13 bits of the byte count.
  unsigned exp: 3;            // The EXP setting for this counter.
  unsigned mark: 1;           // Used for garbage collection.
  unsigned next_index: 15;    // The next entry in the chain, or 0.
};
```

Fig. 6. The CounterEntry structure

2.3 The Tunnel Entry structure

An LSP is associated with a tunnel, i.e. a connection between a source and destination. A tunnel ID allows multiple tunnels between these endpoints. To allow a tunnel to be implemented by multiple LSPs, either for load balancing, or during a transition between two LSPs, a tunnel entry also includes an LSP ID. Signaling, e.g. via RSVP/TE, creates an association between labels and tunnel entries for the current node. Monitoring stations are primarily interested in tunnels, i.e. which LSP is currently supporting a tunnel, and the performance and route of this LSP.

Figure 7 shows the structure of a tunnel entry. At present we assume IPv4 addresses are used for the source and destination. This is not necessarily the case though. RSVP/TE, for example, can use IPv6 addresses in the signaling packets. What should we do in such cases, as such addresses will clearly not fit into our existing cells? We do not want to extend the size of the tunnel entries, particularly due to the knock-on effect on the other structures. We could chain multiple entries together to make additional space. However, due to the size of IPv6 addresses, this is not ideal either. We could follow the approach used for neighbor MAC addresses, maintaining them in a separate table, and storing a reference to them in the tunnel entries. Unfortunately it is harder to bound the size of this table, and a large table will take space away from the main pool storage. A more interesting alternative is to hash the IPv6 addresses into four bytes, and use the same tunnel structure for both IPv4 and IPv6 addresses. A monitoring station can determine the expected endpoints from topology information, and/or information obtained from the ingress routers via MIBs. The chances of two IPv6 tunnels hashing to the same values, with the same Tunnel ID are sufficiently small that this approach is likely to be acceptable in practice.

3. STORAGE MANAGEMENT

Memory is a precious commodity in a small embedded device like a Smart GBIC. Furthermore, these devices will need to function for months at a time without user intervention. To cope in such an environment it

```
struct TunnelEntry {
  uint32_t src; // IPv4 or hashed IPv6 source address
  uint32_t dst; // IPv4 or hashed IPv6 destination address
  uint16_t tunnel_id;
  uint16_t lsp_id;
};
```

Fig. 7. The TunnelEntry structure

is critical they manage their storage use carefully. Cells must be reclaimed when no longer required, and the system must be able to adapt its behavior as memory becomes scarcer. We use a simple mark-sweep garbage collector to reclaim cells, and an adaptive mechanism for allocating cells. The exact mechanisms used depend on the cell type, and so we consider each of these in turn.

3.1 Garbage collection

An MPLS tunnel may be long-lived. Furthermore, the ingress router may use any of the eight EXP-bit settings for packets destined for this tunnel. The trailer processors automatically adapt to the use of different EXP settings by creating new counters as required. However, we do not want to use storage for a counter that is no longer in use. This suggests the need for a storage reclamation process. The strategy adopted in the prototype is very simple. Each counter entry has a single "mark" bit. A global variable in the trailer processor, `current_mark`, keeps track of the current mark bit setting for the processor. When a new counter entry is allocated from the free list the mark field is initialized with the current value of this variable. The mark bit in the counter is also set to this value whenever the counter is updated. Consider the situation where all the counters currently have the value b as their mark bit; we show how this situation can arise shortly. We now set the current mark bit to $\neg b$. From this point on all new counters will have their mark bit set to this value. Similarly, any existing counters that are updated as a result of receiving MPLS packets will also have their mark bit reset to this value. After waiting for a small period, a few tens of seconds for example, we inspect all the counters. Any that still have the mark bit set to b have clearly not been used recently, and so can be reclaimed. Furthermore, at the end of the scavenging period we know that all the counters now have $\neg b$ as their mark bit. We can therefore toggle the mark bit, and start a new garbage collection cycle, with $\neg b$ instead of b. Recycling a counter that is still in use is clearly a disruptive operation that we would like to avoid where possible. This suggests that the waiting period between collections should initially be quite long, allowing long gaps between packets with the same EXP settings. However, this period should be gradually reduced as the memory utilization increases.

Labels themselves also need to be garbage collected. The entries are introduced as a result of observing MPLS signaling traffic. The current prototype uses an RSVP-TE decoder, but an approach based on CR/LDP would function in a similar fashion. The RSVP RESV packets contain a lease time for the tunnel. This indicates the length of time between refresh messages. However, the tunnel should not expire if no refresh is received within this period as RSVP uses UDP, an unreliable transport protocol. The RSVP RFC suggests waiting for $(K + 0.5) * 1.5 * R$ seconds, where R is the refresh time and K is 3 for a hop with a small loss rate, before concluding the tunnel has expired. We call the expression $3.5 * 1.5 * R$ the lease time for this LSP. It is tempting to just use the lease time as the basis for expiring label entries. However, there may be situations where a large number of LSPs are idle. When memory is in short supply we want to ensure that these entries do not prevent us from monitoring tunnels that are more active. We therefore introduce an adaptive expiry mechanism. Each label entry is given an expiry time, i.e. a number of seconds before it can be reclaimed. A global variable maintains the current maximum expiry time, and this value gets reduced as the utilization level increases. Whenever a signaling packet is received, we reset the expiry time to the minimum of the RSVP lease time and the current maximum lease time. We also reset the lease time to the current maximum value whenever an MPLS packet for this LSP arrives. What does this achieve? Initially, when memory is plentiful, the label entry will only be expired if there is no data traffic or signaling traffic for longer than the lease time for the LSP. This should only happen if the tunnel has been dropped without sending a TEARDOWN message. As memory utilization increases then the maximum lease time will drop below that allowed by the signaling traffic. At this point the data packets for this LSP must be arriving sufficiently fast to keep the entry alive; the presence of the signaling traffic alone is not sufficient. This mechanism therefore gives preference to those LSPs that are actively in use.

3.2 Adaptive allocation

A garbage collection strategy can help us avoid wasting cells. However, even with the best garbage collector, we may reach the stage where we run out of memory. Whilst we clearly cannot prevent this situation from occurring, we can ensure that the trailer processor continues functioning in this state. Furthermore, in some cases we may be able to slow down the onset of this condition. To see why, consider the tunnel entries as a concrete example. These are only required by the TP initiating trailer packets for this LSP. The downstream TPs for this LSP can therefore save space by not storing the tunnel details. However, upstream failures, or memory exhaustion, may result in a downstream device having to take over the generation of trailer packets. If this occurs, and there is no associated tunnel entry, we may have to wait until we see the next signaling exchange before we can construct a tunnel entry. In the intervening period our trailer headers (see Section 4.1) will be empty. This analysis suggests the following strategy. When memory is plentiful, we should store tunnel entries for all label entries, allowing a speedy transition to a trailer initiator if required. However, as resources become scarcer we should only maintain tunnel details for trailer initiators. The inconvenience of having to wait for the tunnel details during a role change is more than offset by the reduced memory requirements.

To better understand the scope for reducing memory requirements when space is scarce, it is worth enumerating the different states a TP can be in when an MPLS packet arrives.

- There is no label entry for this label. In this case we can update a trailer packet as it passes by with interface and timing details, but must return 0 for the packet and byte counts. We won't be able to initiate trailer generation for this LSP as we have no way of tracking the time since the last upstream trailer packet was observed.
- There is a label entry, but no tunnel or count entries. If there are no counts then we cannot add them to the trailers, as in the previous case. However, we can recognize whether we should be generating trailers or not. If we do find we need to start generating trailers then we will not be able to fill in the trailer header with the details of the tunnel. Is there any advantage to generating such trailers? Suppose we are keeping historical records of the data in the monitoring station. At a later point we might be able to start adding tunnel details to subsequent trailers , either because a slot gets freed up, or there was already a free slot and we were waiting for the next signaling packet. In that case the monitoring station may be able to look backwards through the history, and be able to infer what the tunnel information must have been for the earlier packets.
- There is a label entry, with counts but no tunnel entry. This will allow us to fill in trailer entries as they pass by, but we will only be able to initiate trailers with blank headers. The count details may be incomplete for some EXP values, and so some trailer entries will still have 0 for the counts.
- A label entry, with associated tunnel entry but no, or partial, counts. This is similar to the previous case, but if we have to initiate trailers then we can fill in an accurate tunnel header.
- A label entry, with tunnel entry and full counts. We can support the full functionality of the TP in this case.

This analysis suggests that if we are in the position where we are initiating trailers for an LSP then it may be preferable to allocate a tunnel cell, even if it means scavenging one of the existing counter cells. Without tunnel details the counters will be meaningless in the long term. There is also a case for just keeping an aggregate count, rather than a per-EXP count, in extreme cases to save space. However, given the very different treatments each subflow can experience, such aggregated measurements may have limited use in many settings.

3.3 The collection task

The Click-based prototype uses a timer to interleave the garbage collection steps with the main packet-processing task. The GC task needs to repeatedly iterate over all the label entries, decrementing the lease times and reclaiming old counters. We define a segment size, N, and then view the hash table as consisting of N consecutive chunks. One chunk of entries is processed each time the timer is triggered, which is once every $1/N$ seconds. Thus all the label entries will be visited once per second. If we also make the reasonable assumption that the labels will be spread randomly over the table then the processing task should be spread fairly evenly over this period. The timing of this cycle isn't terribly accurate, but is sufficient to enforce lease times.

4. TRAILER PACKETS

A trailer packet consists of an MPLS shim for the LSP under test, a second MPLS shim to tag the packet as a trailer packet, followed by IP/UDP headers, a trailer header, and then zero or more trailer entries. This layout is illustrated in Figure 8. The IP destination address is obtained from the Click configuration. In a real deployment we assume that some other mechanism would be used to provide this information. For example, we could arrange for packets containing configuration details to be flooded across the MPLS network, with a copy of each packet traversing every link. A combination of multicast, plus a suitable choice of egress points for the network topology, could achieve this. The trailer processors would then merely have to recognize these packets as they went past. Although the TPs are transparent to the IP layer, they could still be individually addressable, and hence configurable by the management station, by using fields within the configuration packets.

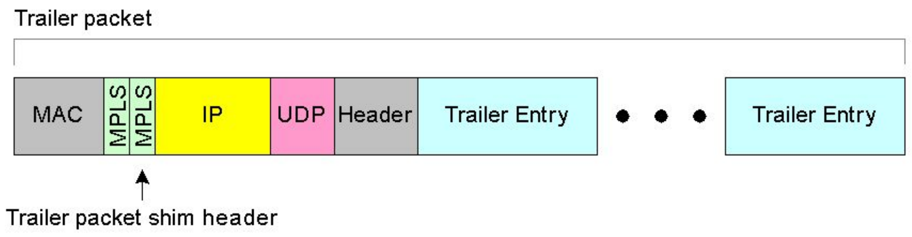


Fig. 8. The trailer packet layout

4.1 The trailer header

Monitoring stations need details of the label to tunnel mapping. Trailer entries refer to labels, but an operator is primarily interested in the performance of MPLS tunnels. Without a mapping between these, the trailer information is of limited utility. The information we require can be gleaned from the signaling traffic. What is the best way of conveying this information to the management station? We could just send a copy of all signaling packets, but this is wasteful. Most of the information in an RSVP packet is of no relevance to the trailers application, although of course other applications might be interested in such details. There is also the question of which TP should forward the packets; there is a danger that we compound the problem by sending multiple copies of each RSVP/TE packet. Many, perhaps most, RSVP/TE messages will merely be refreshing the existing state. We could reduce the amount of traffic we generate by only sending updates to the management station when the label to tunnel binding changes, and by batching up such notifications. However, even this approach is problematic. Ideally we do not want to establish a TCP connection with the management station, particularly as the TP is supposed to be invisible to the outside world. Unfortunately, the obvious alternative, UDP, is an unreliable protocol. A lost packet, particularly when only sending batched change information, may cause severe problems for the monitoring station. A simpler approach, that avoids all these problems, includes the tunnel details with the trailer packets themselves. This introduces a relatively small overhead, and has the advantage that if the packet is received we will know the context in which to interpret it.[1] Figure 9 shows the layout of the trailer header in the prototype.

4.2 The trailer entry

The raison d'être of a trailer processor is to append trailer entries to trailer packets. What should a trailer entry contain? We want to know where we are, what time it is, and one or more counts recording the position in the flow. Each of these deserves further elaboration. The prototype TP should be invisible, and is logically viewed as being part of an adjacent router interface. The "where" should therefore be the MAC address of the router interface, *not* the address of any of the interfaces in the TP. We shall say more about timing in Section 6; at this stage we just note the requirement for access to a reasonably accurate clock in the TP. We can measure our current position in the packet stream in terms of packets or bytes. We store

[1]Actually this is not always true, as observed previously. If free memory is exhausted we may not have the tunnel details for the label. But we still send the trailers as we may be able to make sense of them later.

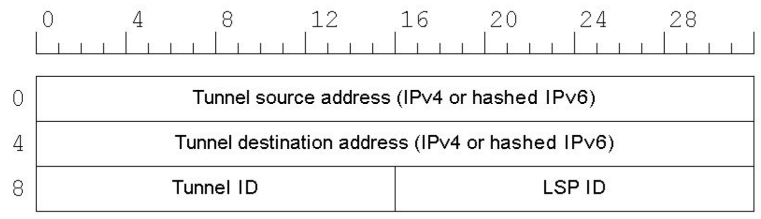

0
4
8
12
16
20
24
28
0
4
8
Tunnel source address (IPv4 or hashed IPv6)
Tunnel destination address (IPv4 or hashed IPv6)
Tunnel ID
LSP ID

Fig. 9. Trailer header layout

both, so we can get a clearer idea of the potential impact of any losses. The position is intended to be a relative notion. By this we mean that the counts can be used to determine the volume of traffic that has past this point since the last trailer processor, for example. This assumes the counters haven't been reset, i.e. garbage collected and then reallocated, in the intervening period. The counters are not intended to record the absolute position in the flow, in the sense of the number of packets and bytes since the LSP was established. Given the nature of the TP, with counters being allocated and deallocated over time, it would not be possible to track position at this fine level of detail, and it is not necessary to do so for our intended purpose. Comparing counts from the same location at different points in time is fine, as long as we allow for resets and overflows. Comparing counts from different locations, however, will produce meaningless results.

As mentioned earlier, the EXP settings may have a profound effect on the level of service experienced by an LSP subflow. We generate trailer packets for each EXP setting in use, and so this value needs to be included in the trailer packet. We could store this value in the trailer header. However, to allow for the possibility of remarking this value as the packet traverses the LSP, we record it in each trailer entry, along with the label in use at this point in the path. We would not usually expect the EXP setting to change in this way. However, LSPs are allowed to merge, and there may be scenarios where it is natural to remark at the merge point. Figure 10 shows the layout of the trailer entries in the prototype.

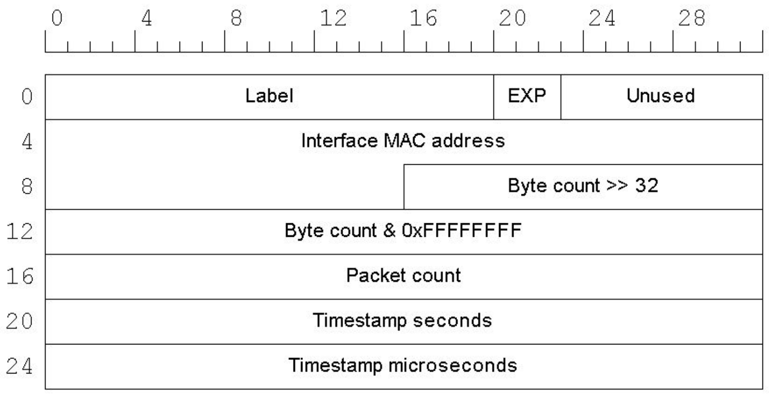


Fig. 10. Trailer entry layout

4.3 Header updates

An IPv4 header includes a total length field, and this value will change as a trailer packet transits the TP. Unfortunately, this will invalidate the existing header checksum. Worse still, if we are using UDP checksums then we will also need to recompute the checksum for the IP pseudo header, the UDP header, and trailer payload aggregate as well. We do not want to buffer up the whole trailer packet before starting to output it, particularly in an SGBIC setting, as this would introduce an unacceptable delay. Fortunately, there is a standard technique for incrementally updating checksum values, for example in response to decrementing the TTL field, that can be adapted to our needs. For example, in the case of the IP total length field, we know how this will change without waiting for the whole packet, and so we can use the old value, and the old checksum, to compute the new checksum value. The UDP checksum is slightly trickier. We could just disable UDP checksums, on the grounds that the monitoring information isn't "mission-critical". If we assume that the information will be regularly refreshed, then outliers due to corrupted data could be easily removed at the monitoring station. However, if we do want to use UDP checksums then it is still not necessary to wait until we have seen the whole packet. As soon as we see the MPLS header we can start to fetch the counters from the trailer state. Having constructed the trailer entry data, we can compute the change to the existing UDP checksum to incorporate the new data. At this point we can then start to write out the packet data, even if the entire packet has not yet been received, this reducing the buffering delay.

5. CLICK TRAILER PROCESSOR ELEMENTS

Click provides a large number of general-purpose elements that can be combined in many different ways to produce routers, firewalls and similar devices. However, as our previous discussions have noted, the TP application requires the manipulation of specialized data structures. These, in turn, require specialized Click elements to manipulate them. Of course, where possible, we would like to use existing Click elements to minimize the prototyping effort. The current prototype uses four new elements. The *CacheMAC* element is responsible for caching the MAC address of the adjacent router. The *MPLSLinkState* element manages the main data structures associated with the TP functionality. The *RSVPTunnelMonitor* element is responsible for tracking the RSVP/TE signaling traffic. Finally, the *MPLS* element processes MPLS packets, appending trailer elements where necessary. We now describe these elements in more detail, followed by the Click script that ties them all together.

5.1 The CacheMAC element

The sole purpose of the *CacheMAC* element is to store the MAC address of the adjacent router, together with its IPv4 address. It performs this simple task by listening to ARP responses from the router. There is a disadvantage to this strategy; ARP timeouts may be very long on some routers, in the order of hours. However, given the trailer processors are likely to be deployed for a long time, waiting to see an ARP packet before generating any trailers isn't likely to be a problem. Furthermore, it is trivial to prompt an ARP refresh if required.

Given the potential disadvantage of waiting for ARP responses, it is worth exploring why we need this information in the first place. The need for the MAC address should be clear. When we construct a new trailer packet we need to fill in the appropriate MAC addresses for source and destination. If the packet is sent on the outgoing link, away from the router, then the address found by the *CacheMAC* element is used as the source address. On an incoming packet it is used as the destination address. On some links we could extract the MAC address from any packet flowing past, but multicast packets, for example, complicate the situation. Waiting for an ARP packet, whilst slower, has the virtues of simplicity and accuracy.

Unlike the MAC case, we do not, strictly speaking, always need the IP address of the adjacent router. We could use a fictional source address in the IPv4 header for the trailers. However, whilst the source address doesn't usually affect the route taken by a packet, firewalls and similar devices may block traffic with a source address that matches, or fails, certain criteria. Using the valid address of the adjacent router, whilst not guaranteeing such problems will not occur, can at least make the problem easier to diagnose.

5.2 The MPLSLinkState element

The *MPLSLinkState* element contains the persistent state for the processor, in particular the hash table and storage pool. The *MPLS* and *RSVPTunnelMonitor* elements update the state in this element as new packets arrive. The *MPLSLinkState* element is also responsible for generating trailer packets, and uses two output ports for this purpose, one for each direction.

Previously we have mentioned, in passing, that a trailer processor must sometimes take responsibility for injecting trailer packets into an LSP. How does a TP decide to take over this role? And what triggers the generation of a trailer packet? These questions are related. A trailer processor takes over the role of initiator when it infers there is unlikely to be an upstream TP for this LSP. It does this by looking for incoming trailer packets, or the absence of them. Nevertherless, to make an informed decision, it needs to know what events can trigger the production of a trailer packet, and therefore how long to wait before making a decision.

Consider the situation where we have somehow decided we must start generating trailer packets for an LSP. What strategy should we use to trigger the production of each packet? We could use the reception of the MPLS packets themselves, for example creating a new trailer packet after receiving N packets for this (sub)flow, for some N. This approach is not ideal. For example, if an LSP is currently idle, with no MPLS packets flowing across it, then we will see no trailers at all. A monitoring station may have many reasons for wanting to observe the (potential) performance of such LSPs. We can avoid this particular situation by using the reception of an RSVP/TE refresh message, for example, to also trigger a trailer packet. At the other extreme, a heavily utilized LSP may generate many trailers per second. Whilst we can still guarantee that the trailers will only use a small proportion of the overall bandwidth, there is perhaps little use in tracking the performance of an LSP at that fine level of detail, not to mention the load it would impose on a monitoring station.

Given that an operator might only be interested in observing changes at the granularity of a few seconds, for example, it makes more sense to trigger trailer production using time, rather than traffic volumes. This is the approach adopted in the current prototype. At present, the rate is set globally, as part of the configuration of the TP. In a real deployment we would want to support other mechanisms for setting this value, for example by eavesdropping on other traffic.

There is a hidden complication associated with generating trailers by time rather than rate. Remember that a trailer packet requires both a source and destination MAC address for the next hop before it can be injected into the network. The CacheMAC element only provides one of these addresses; the particular one depends on the direction of the packet. If an MPLS or signaling packet triggers the generation of the trailer then the triggering packet contains the other MAC address we need to construct the trailer packet. However, if the trailers are generated asynchronously, as a result of a timer expiring, then we have no other packet to act as a "template". Although we assume the TP is connected directly to its host router, via a cross-over cable, the outgoing link could be connected to multiple routers, via a switch for example. The desired next-hop address may depend on the particular LSP in such cases. Fortunately the number of potential addresses is likely to be fairly small, a few tens at most, and so we can allocate a small table to store them. The label entry contains an index into this table, the `mac_index` field, and this index is used to retrieve the second MAC address when constructing the trailer packets. And where do these addresses come from in the first place? The RSVP signaling messages trigger the production of label entries, and the signaling packets contain the desired MAC addresses in their Ethernet headers.

5.3 The RSVPTunnelMonitor element

The *RSVPTunnelMonitor* element is responsible for decoding RSVP/TE signaling traffic, and using the information obtained to update the shared state in the *MPLSLinkState* element. Most of the details present in the RSVP traffic are of no interest to this particular application, and so the decoding task is not as complex as the protocol specification might suggest. The PATH message traverses the LSP, or planned LSP, from source to destination. It is the returned RESV message that interests us, containing as it does the label and lease time. For this reason the LSP is actually established in the opposite direction to the path containing the *RSVPTunnelMonitor* element. As mentioned previously, we only need to store the tunnel information for label entries currently initiating trailer packets. However, the element attempts to store this information for all label entries, relying on the adaptive allocation mechanism failing when resources are stretched.

5.4 The MPLS element

The *MPLS* element, as its name suggests, is responsible for processing MPLS packets. The element must first determine whether this is a trailer packet or not. If it is then the element appends a trailer entry to the end of the packet before forwarding it. If it is not a trailer packet then the element retrieves the appropriate counter from the *MPLSLinkState* element, creating it if necessary, and then updates it. Note that the trailer packets themselves are not recorded in the counts.

5.5 The TrailerProcessor configuration

An abstract view of the Click configuration is presented in Figure 11. The solid lines indicate the direction of packet flow. Where there are multiple outputs for an element a packet will only flow along one of these paths, the choice being made by the element itself. The dashed lines indicate flow of information, and the direction of this flow. So, for example, the *RSVPTunnelMonitor* passes information to the *MPLSLinkState* element, whereas the *MPLS* element sends and receives information from this element. Appendix A contains the configuration itself. The processing steps involved in processing a packet are almost identical, irrespective of whether the packet is leaving the router, or arriving at it. The only asymmetry arises because of the *CacheMAC* element. This is attached on the link coming from the router. It is the placement of this element that dictates which router a TP is associated with when connected in place of a point-to-point link.

A trailer processor needs to inject trailer packets into the flow. Furthermore, the values of the counters should accurately reflect the position of this packet in the flow. It is not sufficient to construct a complete trailer packet and then queue it for later insertion into the main flow; by the time the packet is eventually output the counter values may be incorrect. The Click configuration avoids this situation by inserting the trailer packets upstream of the *MPLS* element. They initially contain only a header, relying on the *MPLS* element to attach a trailer entry, just as if the packet had originated upstream of this device. This guarantees the counts in the entry match the position in the packet stream.

Whenever two packet streams merge there is a choice as to how to perform the merge. The prototype currently uses a priority-scheduling element for this task, giving priority to the main packet stream. The effect of this approach is to merge the new trailer packets into the stream only when there is a gap in the main flow. These prenascent trailer packets are stored in a fixed size queue and so trailers could be dropped when the load is very high. It is not disastrous if this happens. Nevertheless, it may make sense to replace the priority queue by a weighted fair queue so that we do not lose all trailers when the network starts to fail.

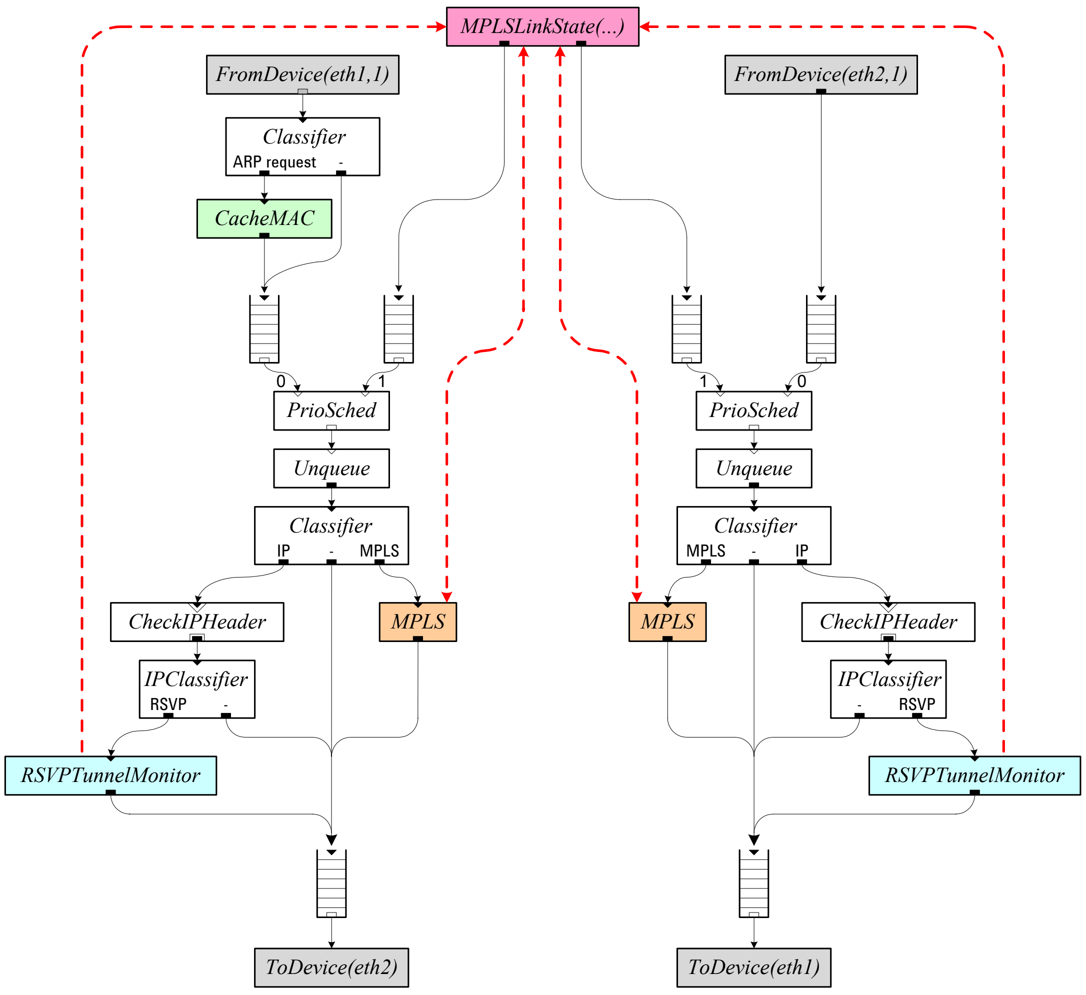


Fig. 11. Click configuration

6. MONITORING STATION

There is little point in generating trailer packets unless we also have one or more monitoring stations capable of interpreting this data, and drawing valuable inferences from it. The reception of a single trailer packet for a tunnel allows us to display the path, or one of the paths, currently being used to support this tunnel. It also gives us some timing details, but we will consider these shortly. What it does not do is give us any idea of the losses along this path. Remember that it will, in general, be meaningless to compare counts from different processors, even if they are associated with the same LSP. So to say anything about loss rates we need to receive two or more trailers for a tunnel. So let us suppose a monitoring station receives two successive trailer packets for the same MPLS tunnel, with the following sequences of trailer entries

$$\langle i_1, c_1, t_1\rangle, \langle i_2, c_2, t_2\rangle, \langle i_3, c_3, t_3\rangle, ..., \langle i_n, c_n, t_n\rangle$$

and

$$\langle i_1, c_1', t_1'\rangle, \langle i_2, c_2', t_2'\rangle, \langle i_3, c_3', t_3'\rangle, ..., \langle i_n, c_n', t_n'\rangle$$

Here we are considering the case of a stable tunnel, and so the paths have stayed the same, as indicated by the interface components, i_1, i_2 etc. matching in the two packets. To estimate the current packet loss between the first and second trailer processors we must compute the value of $(c_1' - c_1) - (c_2' - c_2)$.[2]

A monitoring station must also be prepared to expect the unexpected. When devices fail, and the tunnel gets rerouted, we may find that the paths in consecutive trailer packets are not so readily comparable. This implies that for brief periods we will not be able to report a sensible loss figure for every hop. Similarly we may get misleading counts when a fast-reroute starts and stops, or space is temporarily exhausted, triggering a counter reset. LSP merging may also cause additional scope for confusion, where more packets are flowing across links downstream of a merge point than upstream of it.

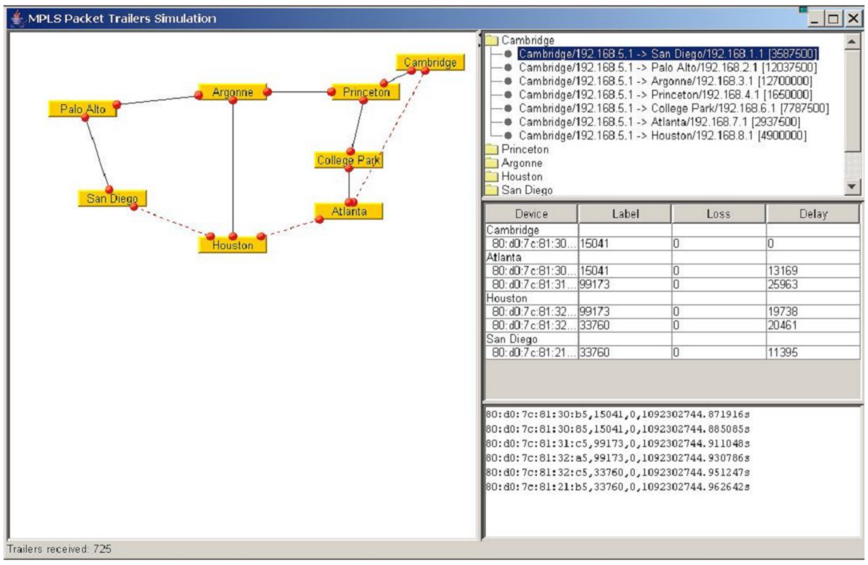
Computing the per-hop delay has its own set of problems. If all the clocks were synchronized then we could just subtract t_1' from t_2' to compute the delay over that leg. However, in practice they won't all be synchronized. It is worth distinguishing between inter-router delays, the delays introduced on the links between routers, and intra-router delays, the delays introduced by the routers themselves. In some settings the inter-router delays will not be affected by network load, and can therefore be estimated in advance. The variability in delay and loss due to network load consists almost entirely of the intra-router delays in these cases. The position of the trailer processors in the network, particularly in their SGBIC or linecard incarnations, gives us a good starting point for calculating accurate intra-router delays. Furthermore, it may be much easier to synchronize all the clocks in the TPs attached to a single router than it is synchronize remote clocks. For example, within a machine room we may be able to use simple WiFi beacon signals to achieve the required degree of synchronization, given suitable decoding hardware in the TP.

In some settings we may therefore be able to approximate, with a large degree of confidence, the end-to-end delay by combining the intra-router delays with our fixed estimates of the inter-router delays. However, such an approach will not always be applicable. In some scenarios we may have access to other mechanisms for achieving accurate time synchronization, for example a GPS feed or IEEE 1588. But in the general case we need to cater for sets of TPs with varying degrees of known synchronization between them. We would like to be able to group sets of TP devices, where all devices within the same group would have their clocks synchronized with each other. The monitoring station would then show not only the delay across each hop, or pair of devices, which may be fairly meaningless in general, but also some indication of which measurements could be relied upon. So, for example, the delay across two entries for the same router might be displayed differently to the delays between two routers. Perhaps where we have no knowledge of the degree of synchronization between two clocks we wouldn't even display the computed delay at all. For the typical case, where there is no synchronization between clocks, then sharp changes in the difference are probably better to highlight than absolute values. For example, we might not be able to calculate the true delay between the two trailer insertion points. But if we compare $t_2 - t_1$ with $t_2' - t_1'$, and keep doing this for each consecutive pair, we'll see changes due to clock drift. Hopefully large increases in delay, due to router overload, would be distinguishable from the gradual change over time due to drift, and such changes could be highlighted in the monitoring station. The trailer processors themselves also introduce a delay. In the case of the prototype this delay will be noticeable. However, in an SGBIC incarnation of a TP the delay introduced by the appending of a trailer as a packet passed by would hopefully be negligible compared to the intra-router delay, and could therefore be ignored.

With only a single trailer processor prototype it is difficult to construct even a simple mockup of a monitoring station. The power of the approach relies on tracking a tunnel across multiple devices. The scenarios that can be constructed with a single monitoring point are very limited. To allow work on the monitoring station, in parallel with the development of the trailer processor, a simple simulator was constructed. The simulator is seeded with a router topology, and a set of tunnels. It then generates a simulated set of trailer packets that are fed to a monitoring station for visualization. The virtual links

[2] The expression is slightly more complex than this as we also have to consider the situation where one or more counters overflows before the generation of the second trailer.

and routers can be disabled, allowing the user to see how a real system might respond to such events, and the trailer packets that would be produced. Figure 12 shows a screenshot of the simulator. The left pane presents the router topology, whilst the top-right pane gives a tree-view of the current tunnels, ordered by ingress. Having selecting a tunnel, the system displays the trailer entries received for this tunnel in a decoded form in the middle pane, and the raw form in the bottom pane.[3] The path currently being used for the tunnel, as inferred from the trailer data, is highlighted in the topology view.



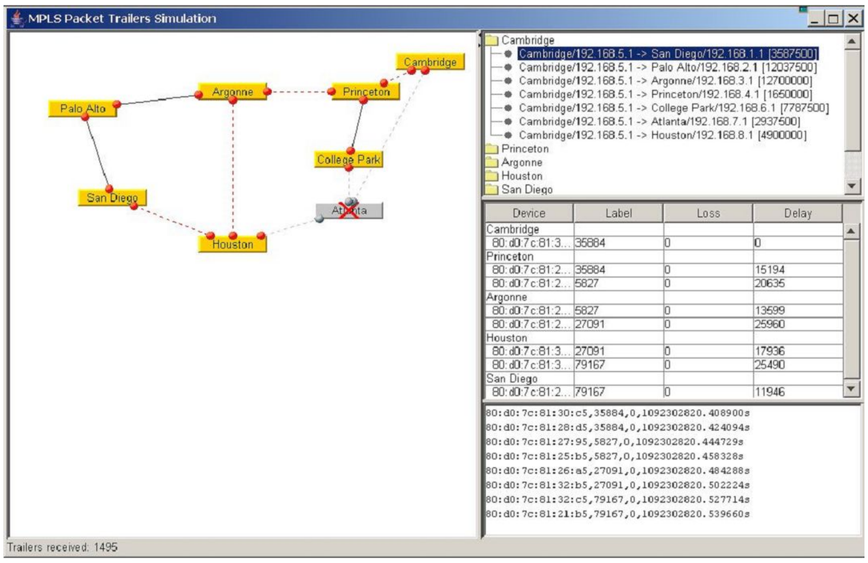
```
Cambridge
  Cambridge/192.168.5.1 -> San Diego/192.168.1.1 [3587500]
  Cambridge/192.168.5.1 -> Palo Alto/192.168.2.1 [12037500]
  Cambridge/192.168.5.1 -> Argonne/192.168.3.1 [12700000]
  Cambridge/192.168.5.1 -> Princeton/192.168.4.1 [1650000]
  Cambridge/192.168.5.1 -> College Park/192.168.6.1 [7787500]
  Cambridge/192.168.5.1 -> Atlanta/192.168.7.1 [2937500]
  Cambridge/192.168.5.1 -> Houston/192.168.8.1 [4900000]
Princeton
Argonne
Houston
San Diego
```

Device	Label	Loss	Delay
Cambridge			
80:d0:7c:81:30...	15041	0	0
Atlanta			
80:d0:7c:81:30...	15041	0	13169
80:d0:7c:81:31...	99173	0	25963
Houston			
80:d0:7c:81:32...	99173	0	19738
80:d0:7c:81:32...	33760	0	20461
San Diego			
80:d0:7c:81:21...	33760	0	11395

```
80:d0:7c:81:30:b5,15041,0,1092302744.871916s
80:d0:7c:81:30:85,15041,0,1092302744.885085s
80:d0:7c:81:31:c5,99173,0,1092302744.911048s
80:d0:7c:81:32:a5,99173,0,1092302744.930786s
80:d0:7c:81:32:c5,33760,0,1092302744.951247s
80:d0:7c:81:21:b5,33760,0,1092302744.962642s
```

Fig. 12. A monitoring station simulation

Figure 13 illustrates what happens when a link fails. The trailer entries received after the failure show the new path for the tunnel. If the trailer processors are only partially deployed then we may not be able to determine the complete path currently being used. The monitoring station prototype deals with such cases by highlighting partial routes, making it clear what is known and what is just conjecture.



```
Cambridge
  Cambridge/192.168.5.1 -> San Diego/192.168.1.1 [3587500]
  Cambridge/192.168.5.1 -> Palo Alto/192.168.2.1 [12037500]
  Cambridge/192.168.5.1 -> Argonne/192.168.3.1 [12700000]
  Cambridge/192.168.5.1 -> Princeton/192.168.4.1 [1650000]
  Cambridge/192.168.5.1 -> College Park/192.168.6.1 [7787500]
  Cambridge/192.168.5.1 -> Atlanta/192.168.7.1 [2937500]
  Cambridge/192.168.5.1 -> Houston/192.168.8.1 [4900000]
Princeton
Argonne
Houston
San Diego
```

Device	Label	Loss	Delay
Cambridge			
80:d0:7c:81:3...	35884	0	0
Princeton			
80:d0:7c:81:2...	35884	0	15194
80:d0:7c:81:2...	5827	0	20635
Argonne			
80:d0:7c:81:2...	5827	0	13599
80:d0:7c:81:2...	27091	0	25960
Houston			
80:d0:7c:81:3...	27091	0	17936
80:d0:7c:81:3...	79167	0	25490
San Diego			
80:d0:7c:81:2...	79167	0	11946

```
80:d0:7c:81:30:c5,35884,0,1092302820.408900s
80:d0:7c:81:28:d5,35884,0,1092302820.424094s
80:d0:7c:81:27:95,5827,0,1092302820.444729s
80:d0:7c:81:25:b5,5827,0,1092302820.458328s
80:d0:7c:81:26:a5,27091,0,1092302820.484288s
80:d0:7c:81:32:b5,27091,0,1092302820.502224s
80:d0:7c:81:32:c5,79167,0,1092302820.527714s
80:d0:7c:81:21:b5,79167,0,1092302820.539660s
```

Fig. 13. Rerouting due to node failure

[3]These screenshots are for an older version of the prototype; the raw trailer data in these views is no longer accurate.

6.1 Trailer forwarding

In our discussions we have given the impression that the trailer packets are sent to the monitoring station. As the final destination of a trailer packet is determined by the destination IP address when the packet is formed, this suggests that each TP would need to be aware of the monitoring station address. However, there may be many different monitoring stations requiring access to the details provided by the packet trailer mechanism. We could devise complex schemes, involving periodic reconfiguration of the probes, multicast groups, and other mechanisms, to address this problem. We could use a fixed synthetic address, in conjunction with static routes in the egress routers, to forward these packets. If penultimate hop popping (PHP) is in use then we would also need static routes in the penultimate routers as well. For the prototype we adopt a much simpler approach. We specify the destination address in the TP configuration script. However, this address runs a simple packet-forwarding service. Monitoring stations, for example the prototype described earlier, subscribe to this service when then start up, and periodically thereafter. When the packet forwarder receives a trailer packet it forwards it to all current subscribers, optionally remapping the UDP port if required.

7. CONCLUSIONS AND FUTURE WORK

The current trailer processor prototype has explored many of the issues surrounding the design of such a device. It has been tested in a small test network, with minimal traffic, and functions well. However, it would need to be tested on a much more heavily loaded network before we could be confident the storage management strategies performed as expected. Some networks use other signaling protocols, for example CR/LDP, to establish their MPLS tunnels. The prototype would need to be extended to handle this protocol. Whilst the information conveyed by the two protocols is similar, CR/LDP relies on TCP sessions to carry the tunnel information, making it slightly more tedious to eavesdrop on these conversations. The current monitoring station loads in topology information from a static file. If deployed in more complex networks it would be preferable to use a live feed of this information, for example of the kind provided by the IGP Detective[Lehane 2002].

The current implementation uses single-threaded code to process the packets, a byproduct of using Click. Clearly an SGBIC implementation would be able to perform many of these steps in parallel. In anticipation of this, lock bits have been reserved in the label entries and the hash table.

We have been assuming that trailer processors would be deployed pervasively across the network. Coverage gaps cause obvious loss of information in the trailer entries, which must be reflected in the output from the monitoring stations. In some cases we may be able to infer information about the state of the hidden bits of the network from the side-effects on the visible parts. Can this intuition be made more precise?

We currently use label 0 to indicate a trailer packet. How safe is this choice? To avoid inadvertently misclassifying a packet as a trailer packet, it may be safer to pick a more unique label. We could statically reserve a label for this purpose in all the MPLS routers by suitably configuring them. A better long-term solution would be to submit a proposal to use one of the unused reserved labels for this purpose. If the approach proved useful, there might also be a case for adding additional options to RSVP/TE to allow various aspects of the trailer generation process to be controlled more precisely. Tunnel priority, to give preference to particular tunnels when storage is in short supply, would be one such example. Trailer generation frequency would be another one. Until that time, we could encode some of these properties in other ways. For example, tunnels have names, and we could pick some convention to indicate which tunnels we wanted to monitor. Tunnels also have priorities that are used to control things like LSP preemption. Although not currently tracked, we could use the tunnel priorities to influence the adaptive allocation scheme.

Deploying large numbers of trailer processor prototypes, to explore some of the scaling issues further, is clearly problematic due to the cost and space required. There is one option that might be worth considering in this context. Linksys' WRT54G router is a fairly ordinary, but cheap, 802.11g wireless router combined with a four-port 10/100 Ethernet switch. What makes this device interesting is that there are numerous Linux-based alternative firmware downloads available for the device. Whilst unlikely to support high packet rates, such a device could potentially provide a route for deploying trailer processors, and similar prototypes, cheaply and pervasively in a test network. We might be able to configure the four ports as two "inline" trailer processors. Furthermore, it may be possible to use the wireless link to synchronize the clocks on all such devices within a machine room, in addition to using this link to configure the devices from a management station.

In conclusion, this paper has described the development of a simple prototype of the MPLS trailer processor concept. It has explored many of the pitfalls and design choices associated with this task, and suggested a number of avenues for future work.

A. THE CLICK CONFIGURATION

```
elementclass TrailerWire { $from, $to, $link, $outgoing |
  classifier :: Classifier(
    12/8847 /* MPLS packets */,
    12/0800 /* IP packets */,
    - /* everything else */);

  ipclassifier :: IPClassifier(
    ip proto 46, /* RSVP */
    - /* everything else */);

  input -> SimpleQueue -> psched::PrioSched -> Unqueue -> classifier;
  input[1] -> trailers::SimpleQueue(20) -> [1]psched;

  outputqueue::SimpleQueue -> ToDevice($to);

  classifier[0] -> mpls::MPLS($link, $outgoing) -> outputqueue;
  classifier[1] -> CheckIPHeader(14) -> ipclassifier;
  classifier[2] -> outputqueue;

  ipclassifier[0]
    -> rsvp::RSVPTunnelMonitor($link, $outgoing)
    -> outputqueue;
  ipclassifier[1] -> outputqueue;
}

elementclass MPLSLink { $name, $from, $to, $monitoring_address |
  link :: MPLSLinkState($name/addr, $monitoring_address, 10);

  outgoing :: TrailerWire($from, $to, $name/link, true);
  incoming :: TrailerWire($to, $from, $name/link, false);

  link[0] -> [1]outgoing;
  link[1] -> [1]incoming;

  arpreqclassifier :: Classifier(
    12/0806 20/0001 /* ARP requests */,
    - /* everything else */);
  arpreqclassifier[0] -> addr::CacheMAC -> outgoing;
  arpreqclassifier[1] -> outgoing;

  // The reads need to be promiscuous as the probe is transparent to the
  // rest of the network, and so no packets will be addressed to it.
  FromDevice($from,1) -> arpreqclassifier;
  FromDevice($to,1) -> incoming;
}

link::MPLSLink("link", eth1, eth2, 130.30.167.153);
```

REFERENCES

Kohler, E., Morris, R., Chen, B., Jannotti, J., and Kaashoek, M. F. 2000. The click modular router. *ACM Transactions on Computer Systems 18,* 3 (August), 263–297.

Lehane, A. 2002. The IGP Detective. Tech. Rep. AGL-2002-9, Agilent Labs.

Mitchell, K. 2004. Computing packet loss and delay using mpls trailers. Tech. Rep. AGL-2004-2, Agilent Labs.