Optimizing Reconfigurable Measurement Probes

Kevin Mitchell
Agilent Labs Scotland
South Queensferry
Scotland EH30 9TG
kevin_mitchell@agilent.com

Abstract

This paper describes an experiment that uses the Click modular router framework[8] as part of a reconfigurable probe for monitoring computer networks. After describing the context in which the probe operates we discuss the need for reconfiguration, and motivate why the Click configuration language provides an interesting trade-off between modularity, processing speed and security. To improve performance further we illustrate how some inefficiencies that can arise in this domain can be eliminated by transformations of the Click configurations. Finally we discuss some deficiencies of the Click approach, and suggest areas for future work.

Keywords

MPLS, probe, network, measurement, Click

I. Introduction

Consider the problem of trying to observe and characterise the packets flowing across a router interface within the core of a high-speed network. We might wish to do this in order to support traffic engineering or accounting applications for example. Obtaining the stream of packets in a cost-effective fashion is not trivial. Using optical taps to attach an external probe to each interface is expensive and physically intrusive. In some cases we can use SPAN ports or multicast techniques to extract the traffic of interest, feeding it to one or more monitoring interfaces/probes. Embedded probes eavesdropping on the traffic flowing through a router line-card might also provide a more pervasive solution in the longer term. But no matter how the packet stream is obtained we are still left with the problem of analysing the resulting packets at line speed.

The simplest form of analysis might just break the stream into flows, counting the number of packets and bytes associated with each flow. The flows could be defined in many different ways, e.g. source/port-destination/port tuples or MPLS label switched paths (LSPs). Many packet-processing engines already extract such flow information from packets, particularly in a QoS-enabled network. In such cases there may be few technical problems involved in building a probe that runs at line-rate, although cost is still an issue. In the case of an embedded probe we may even be able to use the existing packet processor to perform some of the work for us. But for some analysis tasks we need to delve deeper into the packet headers, performing work that is not necessary for pure packet forwarding. In such cases it becomes a lot more difficult to perform this work at line speed, even when the classification tasks are fixed. When the scope of the analysis tasks is more open-ended then a software-based approach seems essential, but would be hopelessly slow on many high-speed links.

A hybrid approach based on sampling provides an attractive alternative to a hardware-only solution. By decoupling the analysis task from the raw packet rate we can exploit a range of processing techniques with varying costs. The price we pay for this flexibility is the introduction of some bounded statistical uncertainty into the results we obtain. In this approach we use hardware to split the packets into flows, maintaining accurate byte and packet counts for each flow. The hardware then samples packets, with the sampling rate set independently for each flow, and passes the sampled packets to the software for further analysis. To allow the probe to keep up with the fastest routers within the network we must ensure that the flow definitions we use are no more complex than those used by the routers themselves for QoS differentiation. In an embedded solution we might even be able to use the flow ID computed by the adjacent packet processor to avoid duplicate work.

Note that for very high-speed networks the definition of flow may have to be rather courser-grained than that achievable using something like SRL[3] or FPL[12]. In some cases we can post-process the sampled

packets to perform a finer-grained analysis of the micro-flows, but we will only have statistical estimates of properties like packet counts for these microflows. Fortunately techniques like RFC[6] should enable us to perform reasonably fine-grained classification at high speed.

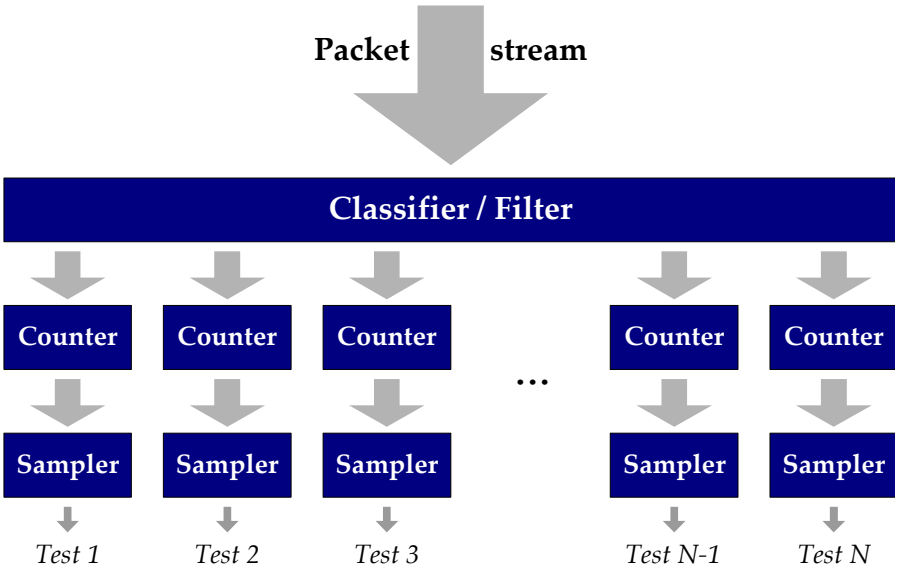


Fig. 1: Packet Sampler

The sampling model is illustrated in Figure 1. Conceptually a classifier splits the stream into some number of flows based on the contents of the packet headers. The nature of the flows will depend on the kind of traffic present in the stream, and the properties we wish to characterize. Each flow then passes through a counter that accumulates packet and byte counts for the flow before reaching a packet sampler. Each sampler has a current sampling rate and this is used to determine what proportion of the incoming packets can pass through the sampler, using a stratified random approach. The sampling rates are chosen based on the packet and byte rates encountered on each flow, and the time complexity of the analysis we wish to perform on these packets. For flows with low packet volumes the sampling rate might be set to 1, indicating that no sampling occurs, and therefore no statistical uncertainty will be introduced into the results obtained for this flow. In order to prevent the software from getting overloaded we assume the sampling rate for each flow may be varied. As the packet rate increases the sampling rate decreases, and as more tests are added for new flows we may also need to reduce the sampling rate for existing flows. Obviously such changes have an impact on the errors introduced by the sampling process, creating challenges for both the software and theory involved in computing error bounds for the test results.

The presence of the packet counters complicates the model slightly. Without these elements the classifier could simply partition the input stream into some number of flows, with a sampling rate of 0 then being used to filter out those streams that are of no current interest to us. However, as the number of available counters may be less than the total number of potential flows, we assume the classifier also plays a filtering role as well. In practice we might want the sampler to be slightly more complex. For example, we typically don't require the complete contents of each sampled packet and so the sampler might support a truncation function. If our probes are also calculating two-point measurement of loss and delay using test packet injection and interception then the sampler must be aware of such packets to avoid filtering them out and also support packet time-stamping.

The physical manifestation of such a sampler outputs a single stream of sampled packets, each tagged by the corresponding flow ID. We wish to perform some measurements on the flows represented by these annotated packet streams in order to characterise the traffic they contain. The exact nature of the characterisation process is not fixed. For example, an operator trying to traffic engineer an MPLS network might be interested in the top N protocols flowing through a label switched path, whereas a statistical-based billing application might be more interested in the top N source and destination ASs using a link. We could obviously build

multiple probes, one for each intended use, but a parameterizable scheme would obviously be more desirable. However, we must be careful not to trade off too much speed for flexability. Although the sampler decouples the raw packet rate from the speed of the software, we still want a fast test framework in order to maximise the sampling rates, and hence the accuracy of our results. If all the tests are run on a shared resource then we also need fast processing to maximise the number of tests that can be active simultaneously.

The rest of this paper explores one approach to the design of such a test framework based on the Click Modular Router[8].

II. Reconfigurable Tests

The sampled packet data can be characterised and aggregated in many different ways depending on the intended usage of the results. This suggests that the test components must be reconfigurable at run-time, adapting to the current needs of the result consumers. If we only need to support a fixed repertoire of tests then the reconfiguration process simply involves passing the index of the test we wish to run on a given packet stream. But what if we do not know, in advance, all the tests we might need to run, or else they can be combined in ways that make a simple "shopping list" approach intractable? In such cases we require a more flexible approach to test deployment. The most powerful and flexible strategy involves incrementally loading code into the probe. For example, for each new test we might dynamically download a library representing the code for a C++ test class[9]. Although such an approach is likely to be efficient, at least for single tests on a flow, it also has some disadvantages. The size of each downloaded test may be quite large and security is likely to be a concern, particularly in the case of an embedded probe. The risk is perhaps more psychological than real given that the code is not running on the packet forwarding path. But the results produced by the probe may be sensitive and so, at the very least, tests would have to be authenticated by the probe. Even with such authentication it will be very easy to crash such a probe, particularly if the code is generated manually for each test rather than via a translation phase from a safer domain-specific language. Sandboxing techniques can help isolate tests from each other, and can also be very efficient [4], but this still won't prevent a test from consuming too many resources.

Dynamic class loading techniques can yield high processing speeds for individual tests. But what if we want to deploy multiple tests on the same flow. Each test will typically drill down inside the packet header until it finds the field(s) of interest. When multiple tests are applied to the same packet then some of this "top-level" processing may be identical in some of the tests. To maximise the performance of the probe we should be able to factor out such common classification tasks. In the case of a single client wishing to run multiple tests on a flow then one could imagine deploying a compound test class that encapsulates the functionality of all of the required tasks. This compound test could be optimised to factor out the common classification steps. However, adding a new task to an existing set of tasks running on a flow may require a bulky download. Furthermore, if multiple clients are allowed to deploy tests on the same flow then it becomes more difficult to coordinate such optimisations.

At the other extreme we could use a general-purpose scripting language, such as JavaScript[5] or Tcl[10], to describe the tests. In this scenario there would be a single implementation of the test component running in the probe, parameterised by a high-level description of each test. This approach can be very flexible but the large interpretive overhead clearly limits the sampling rates we could use. This overhead can be reduced, to some extent, by running a compilation phase on the probe, but this creates its own problems due to the processing overhead of this task. Java, and JIT technology, illustrate this approach. What we would really like is a test description language that is more domain-specific than a general purpose language like Java, or Tcl, but with an efficient translation into a run-time representation with low interpretive overhead. Furthermore, we would like the test descriptions to be compact, composable, and with a bounded run-time. In this paper we propose an approach based on the Click modular router[8] and illustrate some of the advantages and disadvantages of using this technology.

III. The Click Router

Click was originally intended as a toolkit for building flexible and configurable routers. The routers are assembled from packet processing modules called elements. A large number of elements come as standard with the Click environment, performing such common tasks as packet classification, queuing and scheduling. A router configuration then consists of a directed graph with elements at the vertices, with packets flowing across the edges of the graph. Click configurations are written in a simple declarative language, allowing easy syntactic manipulation by tools such as configuration optimizers. Although the idea of modular routers is not new, finding the right level of abstraction to achieve high performance and flexibility is difficult. If the building blocks are very fine-grained, with expensive packet transfers between blocks, then performance will be poor. Creating monolithic blocks can reclaim this performance, but at the expense of flexibility. Click has explored a particular region within this spectrum and has demonstrated impressive packet forwarding speeds whilst retaining a high degree of modularity. Click router configurations run in the context of a driver, either at user level or in the Linux kernel. The kernel driver offers increased performance, but also allows new configurations to be "hot deployed". Some Click elements maintain local state, and hot reconfiguration allows changes to the current configuration whilst preserving state common to the old and new configurations.

Although primarily intended for describing routers, and siblings such as firewalls, many of the Click elements can also be used to analyse the output of our sampler. Tests need to classify packets based on various criteria, analyse packet headers, maintain packet counts, and so on. Some requirements are not met by existing Click elements, such as maintaining "top-N" statistics for various properties of interest[7], but these are simple to write. The approach explored in this paper represents each test by a partial Click configuration. The configurations for all the active tests are then combined, together with some supporting infrastructure to classify and dispatch incoming packets to the appropriate test. The resulting configuration can then be deployed on a probe connected to the hardware sampler. We argue that this produces a very modular solution, whilst maintaining fast processing speeds. The high-level language used to describe Click configurations is also amenable to automated analysis. We show how this property can be used to combine and optimize test descriptions. The language is sufficiently constrained that it should also be possible to deduce various performance and security guarantees for the tests given a fixed set of elements as building blocks.

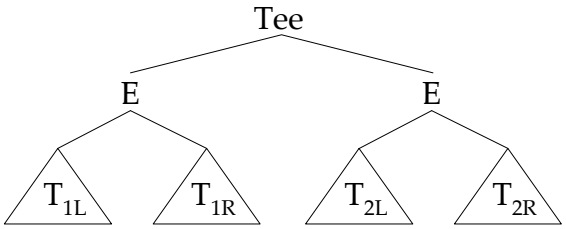
IV. Click Test Descriptions

We could describe each different kind of test using a separate Click element class. However, such course-grained elements wouldn't fully exploit the modular power of Click. Furthermore, as new test requirements arose, we would continually need to develop new Click elements to support these, and the code for the new elements would then have to be deployed on the probes. We would eventually end up with something not dissimilar to the dynamic class loading approach, with all the problems that entails. It also makes it difficult to leverage any commonality between tests, a point we will return to shortly. At the other extreme, using very fine-grained elements to describe our tests will lead to bulky test descriptions, and inefficient processing. We are faced with a delicate balancing act. Although still work-in-progress, our prototype probe currently uses existing Click elements for header analysis and packet classification, but adds a small number of domain-specific elements for parameterised "top-N" counters, aggregation elements and Linux message queues. Each individual test typically requires two or three Click elements. However, some of these elements can be shared between tests; as more tests are deployed on the same LSP each additional test typically requires only one or two extra elements to be added. These counts exclude the Click elements required to filter out packets for the LSPs of interest. The appendix provides more details of this process.

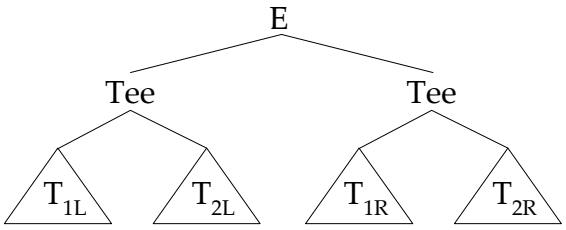
Suppose we want to run multiple tests on a single packet stream. This might be at the request of a single consumer or multiple consumers could also be interested in the same flow. In either case all the tests will be sent the same sequence of packets, i.e. they will all be using the same sampling rate. If we have two Click test descriptions, T_1 and T_2, we can combine these into a single description, suitable for attaching to a packet stream, by using the `Tee` element. This element sends a copy of each incoming packet to each output port. Although the resulting description behaves correctly, running both tests on each incoming packet, it may not be the most efficient description for this compound task. The `Tee` element uses a copy-on-write strategy to

avoid copying the packet payload where possible. If a downstream element attempts to modify the packet contents then a copy will be made if necessary. Such state is held in a separate header containing, amongst other things, a pointer to the packet data, and this header *is* copied by the `Tee` element. None of our test descriptions currently modify the packet contents, and so uses of the `Tee` element would not lead to any packet copying. However, they will result in copying of the Click header which we would like to avoid if possible.

There is a second source of inefficiency that can frequently arise where multiple tests are deployed on the same flow. Consider the case of two tests containing identical top-level elements.[1]



In some cases we can rewrite this description into the following semantically equivalent one:



What have we achieved with this transformation? First, the processing associated with element E is now shared between the two tests. But there is a second advantage to be gained by pushing the `Tee` elements closer to the leaves. Consider the case where E is of the form `Classifier(`X`,`Y`)` and T_{1R} and T_{2L} are both `Discard` elements. Clearly any branch of a `Tee` component connected to a `Discard` element can be dropped. Furthermore, a `Tee` element with a single child can be replaced by the child itself. In this simple example the transformations would remove the `Tee` component completely. Click comes with a pattern replacement tool, called *click-xform*, that consists of a generic search-and-replace engine for configuration graphs. Such tools can be used to automate the rewritings illustrated above.

Unfortunately such transformations are not always valid in the Click environment. Elements like `Classifier` are stateless, but consider the case where E is some kind of counter element for example. Passing two copies of the packet through two separate instances of the element will certainly not yield the same result as passing a single copy of the packet through a single instance of the component. For the "built-in" Click elements we can clearly hard-wire suitable rules for use by *click-xform*. But we really need a more declarative way of determining whether an element can be treated functionally.

There is another property of Click elements that can break the simple transformations outlined above. Each Click element, on start-up, can cache state about its context. For example, an element could locate another element by searching "upstream" within the graph. Having located the required element it can cache a pointer to this object in its local state. Subsequently the cached pointer can be used to call methods on the upstream element when processing packets. Although this feature can be very useful in some cases, it also creates havoc with any general-purpose transformation framework. For example, although unlikely to occur in practice, an element could locate its upstream partner by using a fixed path length, rather than just searching upstream until an element of the required type is found. Clearly any transformation that alters the path distances between elements could potentially break such code. This is another example where a more declarative description of the element's behaviour, in this case its context requirements, would help

[1]For simplicity we display test subgraphs as trees, but they may consist of DAGs in practice.

a transformation system. Indeed one might be able to replace such initialization code entirely, with the framework using the declarative information to locate the required context components and passing them directly to the component at initialization time.

V. Sequential Composition

Although we can sometimes push `Tee` elements past common elements, and such optimizations may occasionally remove the `Tee` elements entirely, in most cases we will not be so fortunate. Test configurations will typically have a single entry point and one or more `Discard` elements at the fringes to soak up any packets injected into the subgraph. Packets will also not be altered as they traverse the graph. In some cases it is tempting to strip bytes from the front of the packet to ease the analysis, using the `Strip` element, but this can usually be avoided by using packet offsets. Consider two tests running on a packet stream, as illustrated in Figure 2 a).

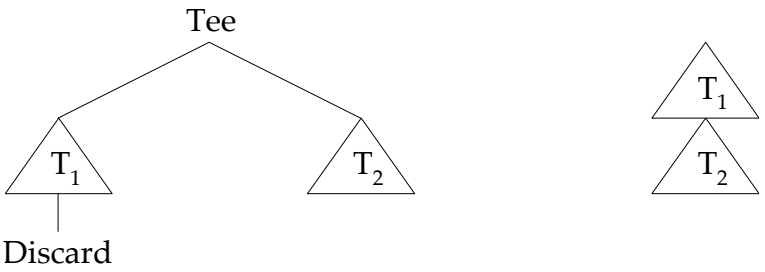


Fig. 2: a) Before transformation b) After transformation

If we can deduce that any packet entering the test will eventually be passed, unchanged, to the `Discard` element shown in the figure, then we may be able to sequentially compose the two tests, avoiding the need for the Tee element. The validity of this transformation depends on the context requirements of T_2. We require that either T_2 has no context requirements, or else the mechanism used to locate its context elements is not affected by the interposition of T_1 in its ancestor path. Such optimizations provide further evidence of the need for declarative context dependencies.

The situation is usually more complex than this as there may be many paths through T_1, each leading to a `Discard` element, or some other element that discards packets. The branches will typically be generated by variants of the `Classifier` element so each packet will only follow a single path through the tree. In such cases we need to check that every packet reaches a single `Discard` element at the fringe of the tree. We then remove these elements and replace them by links to the start of T_2.

VI. Sharing Tests

There is often a need to deploy the same test on multiple flows, and we would like to share parts of the Click graph in such cases. Unfortunately some of the elements in a test description have state, for example counting the number of packets of a particular "kind". Such elements prevent test subgraphs from being shared between flows. One solution is to decouple this state from the elements themselves, making them more functional. This would increase sharing but at the expense of carrying more state around.

An alternative strategy uses Click annotations. As mentioned previously, a Click packet consists of the actual packet data and a small packet header. The packet header includes a pointer to the packet data, allowing multiple headers to share the same packet. The header also contains a number of annotations, such as the time the packet arrived, the offset of the IP payload etc. Although there is currently no way to dynamically add a new kind of annotation, a "user annotation" is provided to allow the user to attach an arbitrary marking to a packet. The `Paint` method can be used to mark a packet. We can rewrite our tests so that the counters contain a set of counts, indexed by the user annotation. This process is illustrated in Figure 3. We need to be careful when adding new tests in such an approach. When altering the `Classifier` expression we must preserve the association between annotation and test if the hotconfig mechanism is to preserve state correctly, i.e. the Paint annotation values are not simply the child index.

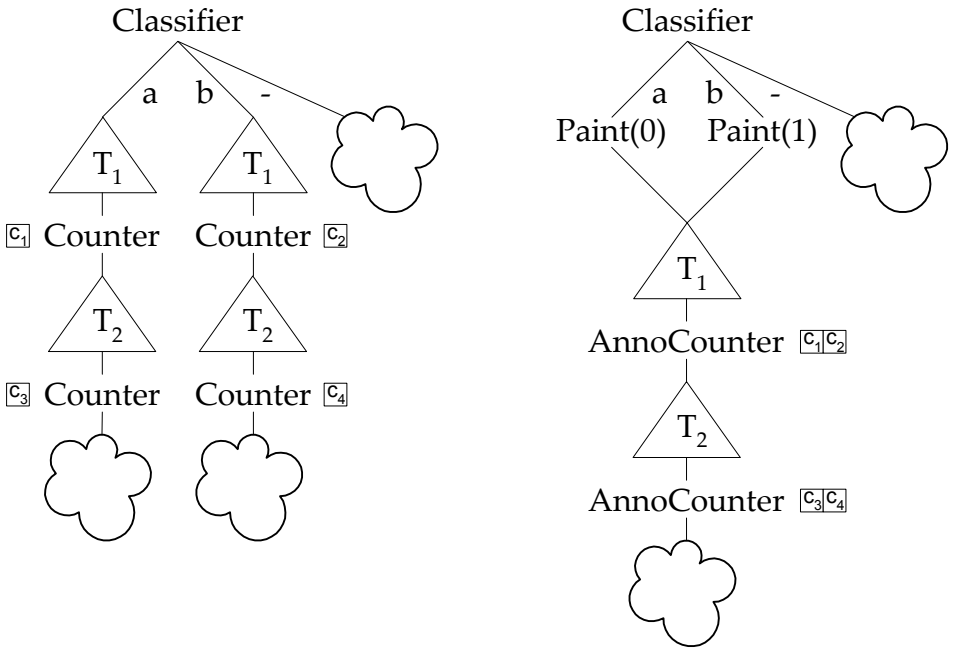


Fig. 3: a) Simple counters b) Annotation-aware counters

VII. Dynamic Reconfiguration

Tests need to be deployed and removed incrementally. Individual tests are represented by Click subgraphs, and the current state of the test environment by a large Click graph. Adding a new test therefore involves splicing a subgraph into an existing graph. Context dependencies complicate this operation as altering the graph may invalidate cached element pointers previously obtained by context lookups. This is unlikely to happen in our case as each test is independent of its neighbours. But without a declarative description of context dependencies we cannot check this automatically in a general framework. If the Click driver had access to such information then it could just splice in the new subgraph. The context specification would then be used to determine which elements might have stale cached state that needs updating, in a similar fashion to incrementally updating an attribute grammar when the state of a node changes[11].

Unfortunately Click doesn't support such incremental updates. Click descriptions are usually fairly static. Typically one would deploy a configuration corresponding to a particular kind of router and then leave it running for a while. Such scenarios do not require fine-grained incremental updates of the Click graph. Deploying a new configuration in Click, when using the user-level driver, requires restarting the Click process. This has the side-effect of discarding any state associated with the currently running description. Such an approach would be disastrous in the case of adding a new test as it would reset all the state associated with any existing tests. Fortunately when Click is running as a Linux kernel module it supports a "hotconfig" mechanism. When loading the new configuration any element that has the same name as an element in the old description can be initialized with the current state of the old element. Although obviously not as efficient as a truly incremental solution, our prototype probe currently uses the hotconfig mechanism to incrementally add and remove tests. For small test networks the resulting performance is acceptable, but such an approach is unlikely to scale very well when there are hundreds of tests active simultaneously.

VIII. Current Implementation

The Click framework is being used as part of a prototype MPLS monitoring system currently under development within Agilent Labs. An FPGA-based sampler extracts packets from a Gigabit Ethernet link and passes them to a probe consisting of a Linux box running Click. The packet stream contains sampled packet headers, with the sampling rates being variable on a per-LSP basis, together with RSVP-TE signalling traffic. The probe is controlled via a CORBA interface allowing tests, in the form of Click configurations, to be incrementally added and removed.

In addition to using Click for passive measurements, the probe also injects packets into the network to measure loss and delay across an LSP. The sampler is configured to allow such test packets to pass through the sampler unhindered. Without such configuration these packets would be subject to sampling, just like all other packets, which is obviously undesirable. The sampler is also configured to allow certain signalling traffic, e.g. RSVP-TE packets, to bypass the sampling mechanism.

The two-point measurements use the IPPM methodology[1], [2]. Using "active" packets to calculate loss and delay, rather than the alternative of passive correlation, works particularly well in high-speed MPLS networks. For a given measurement rate the percentage overhead of active packets decreases as network speed increases. Furthermore we can be sure that the active packets follow the same route as other packets within the LSP, and are treated identically by intermediate routers. Finally, the label stack mechanism allows us to detect and discard or forward the active packets in the egress router without perturbing the rest of the LSP stream.

We run Click as a Linux kernel module, partly for the increased processing speed, but mainly because of its support for hot deployment of new Click descriptions. Although Click is used to process the passive tests deployed on the LSPs, some tasks are best handled in other threads, for example processing the signalling traffic. A `ToMsgQueue` element was written to allow selected packets to be sent to a user-level thread via Linux message queues. This mechanism is currently used for handling RSVP-TE traffic and the recovery of injected packets.

IX. Conclusions

In this paper we have illustrated how the Click framework can be used to implement reconfigurable measurement probes. Click configurations provide an interesting point in the spectrum between fine-grained and course-grained test descriptions. They provide a reasonable level of modularity, coupled with high performance. We have some control over the performance and security of the tests, particularly if we fix the set of Click elements supported by a probe. We can also provide more flexibility by allowing new Click elements to be loaded dynamically, although this is not currently supported in the kernel-mode driver.

The Click configuration language allows some optimizations to be automated but these tend to be very element specific, and are unlikely to scale well as new elements are added. Ideally we would like to use a more general rewriting/proof framework, but this will require additional declarative information to be provided by the elements themselves.

Our use of Click is obviously rather specialized. Can any parts of the Click framework be simplified in this setting? This question might be particularly important if we wanted to use this approach in an embedded solution. The use of the hotconfig mechanism is also unlikely to scale well and it would be interesting to explore more incremental solutions to this problem.

Appendix

I. Click Elements

A number of specialized Click elements were developed as part of the prototype MPLS monitoring project. A brief description of each of these elements now follows.

ActivePFilter Filters out active packets from the packet stream. Active packets are embedded within an MPLS shim header, have a distinguished label at the bottom of the label stack, with an easily recognisable packet payload. Active packets are forwarded on output 1; all other packets are forwarded on output 0. The element maintains counts of active packets for use by downstream elements.

AggregateCounter Passes packets unchanged from its input to its output, maintaining statistics information about packet counts and sizes. This element is used in conjunction with the Reporter element.
CheckEASEHeader Each sampled packet is sent to the probe in the form of an SNMP trap. The traps have an EASE header prefixed to the front of each sample providing additional information about the sampled packet. The EASE header includes

- The time at which the packet was sampled.
- The number of octets sampled from this packet.
- The sampler interface on which the packet was received.
- The current sampling rate for the sampler interface associated with this packet.
- The number of times when packets destined to be sampled were dropped due to lack of resources.
- The total size of the sampled packet.
- The total number of packets, including error packets, seen on this interface.

The CheckEASEHeader element retrieves and caches these fields from the EASE header for use by downstream elements.
CheckMPLSHeader Input packets should start with a valid MPLS shim header. The element caches the values associated with this header before forwarding the packet. It is typically used in conjunction with the `Reporter` element.
CheckSampledIPHeader This element is identical to the `CheckIPHeader` element except it does not check the packet length. The probe is only passed a prefix of each sampled packet and so `CheckIPHeader` would reject all sampled packets larger than the prefix size.
MPLSEncap Encapsulates each incoming packet in an MPLS shim header.
PrintEASEAttributes Displays the EASE attributes of a sampled packet. The element should be placed downstream of a `CheckEASEHeader` element.
PrintReport Prints the contents of packets generated by the `Reporter` element.
Reporter At graph initialization time this element discovers all instances of `PassiveCounter` between two points in the Click graph, where these points are provided as arguments to the element. The element then periodically gathers results from these `PassiveCounter` elements and outputs packets containing the concatenation of these results. Examples of `PassiveCounter`s include `AggregateCounter`, `CheckMPLSHeader` and `TopNCounter` elements.
StripEASEHeader Skips past the EASE header for a sampled packet. The sample offset can vary due to the use of ASN.1 encoding in SNMP traps. The `CheckEASEHeader` element computes the size of the EASE header for each sampled packet and this value is then used by the `StripEASEHeader` element.
ToMsgQueue Sends each incoming packet to Linux in the form of an IPC message. The message queue and type can be specified as arguments to the element and the packet arrival time can optionally be included in the message contents. This element behaves like a packet sink. It's primary use is to support out-of-band packet processing. The probe uses such elements to monitor the RSVP-TE signalling traffic and to process active packets injected into the network to measure loss and delay.
TopNCounter Passes packets unchanged from its input to its output, maintaining "top-N" statistics information about packet counts and sizes. The element is parameterised on a field of interest, for example the IP protocol or the TCP source port, using a field offset and length. If a list of field values is also supplied then the element compares each packet against these values and maintains packet and byte counts for each associated bucket. An alternative mode of operation replaces the list of values by a simple count. In this case the element maintains counts for the top N values observed for the field. The algorithm used, [7], trades off accuracy for storage space to avoid allocating counters for every different field value observed. The probability of misranking a field value can be estimated, and in practice this can be made very small for the values that occur most frequently. This element is typically used in conjunction with the `Reporter` element.

The probe constructs a fresh Click description whenever a test is added or removed. Some of this description, in particular the top-level analysis of each packet received from the sampler, remains constant and is simply loaded from a file. Other definitions vary depending on whether Click is running in user or kernel mode. The network addresses and the code to implement each of the deployed tests is also generated dynamically. A simplified version of such a description is presented below, excluding details of the tests themselves. A

single LSP, with label 100, is being monitored from sampler interface 3. Although this document has provided insufficient background to understand many of the details in the example, it should give a flavour of what the Click descriptions look like. Figure 4 illustrates the MPLS sampler packet classification task at a more abstract level.

```
elementclass Punt Discard;

elementclass ARPHandler Discard;

AddressInfo(
  gateway 156.141.110.1 0:2:fd:a0:8c:c,
  probe   156.141.110.109 0:d0:b7:e3:f2:b);

elementclass Sampler { FromDevice(eth3, 1) -> output }

<< elementclass definitions to support the currently deployed tests go here >>

// Declarations for channel on sampler interface 3
// -----------------------------------------

elementclass INTERFACE_3_LSP_100 { $probe, $test |
  << The click code corresponding to the deployed tests for this LSP goes here >>
  input -> ...
  ... -> output
}

ShimClass_3 :: Classifier(
  0/00064000%FFFFF000, // MPLS label 100
  -);

ToGateway::EtherEncap(0x0800, probe, gateway) -> Queue -> ToDevice(eth0);

ShimClass_3[0] -> INTERFACE_3_LSP_100(0x00000000636e8d9c,0x000f424d)
               -> UDPIPEncap(...) // Address of consumer of these test results
               -> ToGateway;
ShimClass_3[1] -> Discard;


// Declarations for channel on sampler interface 2
// -----------------------------------------

ShimClass_2 :: Discard;


// Common handling code embedded from suffix file
// ----------------------------------------------

Sampler -> EtherTypeClass::Classifier(12/0800, 12/0806 20/0001, 12/8847, -);

EtherTypeClass[1] -> ARPHandler;
EtherTypeClass[2] -> Discard; // Non-sampled active packets
EtherTypeClass[3] -> Punt;

// Check the IP header.
AlignmentInfo(CheckIP 4 0);
EtherTypeClass[0] -> CheckIP::CheckIPHeader(,14);

// Just drop invalid IP packets
CheckIP[1] -> Discard;

// We have received a valid IP packet so classify it
CheckIP[0] -> IPClass :: IPClassifier(dst udp port 162, ip proto 46, -);

// RSVP messages will be forwarded to the probe's message queue for processing.
IPClass[1] -> ToMsgQueue(1001);

// Anything other than a trap or RSVP will be punted for now.
IPClass[2] -> Punt;

IPClass[0]
  -> StripIPHeader()
  -> CheckUDPHeader()
```

```
  -> Strip(8) // Strip UDP header
  -> Ease :: CheckEASEHeader();


elementclass MplsClass
{
  // An MPLS classifier separates out MPLS packets from the rest based on
  // the ethertype
  MplsClass :: Classifier(12/8847, -);

  input -> StripEASEHeader->MplsClass;

  MplsClass[0]  // We have an MPLS packet so
    -> Strip(14)        // strip ether header
    -> afilter::ActivePFilter(99)
    -> [0]output;       // pass on non-active packets.

  afilter[1]
    -> Unstrip(14)             // re-construct ether header
    -> ToMsgQueue(1009,8497);  // pass to probe via message queue
       // with time stamp anotation

  MplsClass[1] -> Punt; // Just punt unlabelled packets
}

Ease[0] -> Discard;
Ease[1] -> MplsClass -> ShimClass_2;
Ease[2] -> MplsClass -> ShimClass_3;
Ease[3] -> Discard;
```

References

[1] G. Almes, S. Kalidindi, and M. Zekauskas. A one-way delay metric for IPPM. Technical Report RFC 2679, Internet Engineering Task Force, September 1999.

[2] G. Almes, S. Kalidindi, and M. Zekauskas. A one-way packet loss metric for IPPM. Technical Report RFC 2680, Internet Engineering Task Force, September 1999.

[3] N. Brownlee. SRL: A language for describing traffic flows and specifying actions for flow groups. Technical Report RFC 2723, Internet Engineering Task Force, October 1999.

[4] Tzi cker Chiueh, Ganesh Venkitachalam, and Prashant Pradhan. Integrating segmentation and paging protection for safe, efficient and transparent software extensions. In *Symposium on Operating Systems Principles*, pages 140–153, 1999.

[5] D. Flanagan. *JavaScript: The Definitive Guide*. O'Reilly and Associates, second edition, 1996.

[6] Pankaj Gupta and Nick McKeown. Packet classification on multiple fields. In *SIGCOMM*, pages 147–160, 1999.

[7] Jonathan Jedwab, Peter Phaal, and Bob Pinna. Traffic estimation for the largest sources on a network, using packet sampling with limited storage. Technical Report HPL-92-35, HP Laboratories Bristol, March 1992.

[8] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. The click modular router. *ACM Transactions on Computer Systems*, 18(3):263–297, August 2000.

[9] J. Norton. Dynamic class loading for C++. *Linux Journal*, pages 166–172, May 2000.

[10] J.K. Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley, Reading, Mass., 1994.

[11] G. Ramalingam and Thomas Reps. A categorized bibliography on incremental computation. In *Conference record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Charleston, South Carolina*, pages 502–510, 1993.

[12] E. Rothfus. The case for a classification language. Agere white paper, September 1999.

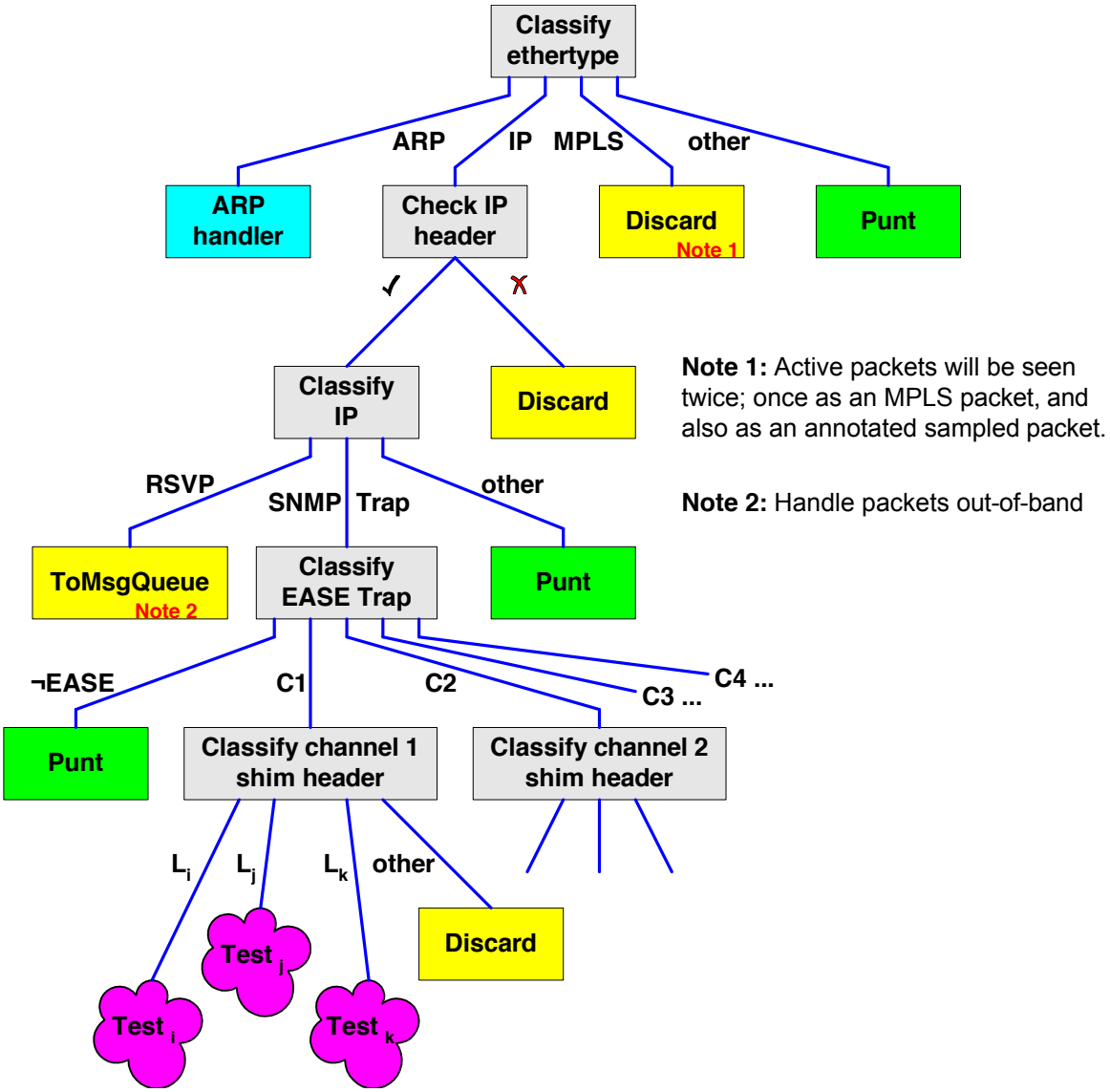


Fig. 4: MPLS sampler packet classification