Click for Measurement

Eddie Kohler

UCLA Computer Science Department Technical Report TR060010, February 2006

Abstract

The Click modular router was designed to forward packets, but some of its strengths—modular design, speed, and scalability—are well suited for measurement tasks as well. We present simple and efficient Click elements that process traces and live packet data, and configurations and tools that combine those elements in useful ways.

1 Introduction & Related Work

The mechanics of Internet measurement and analysis—writing and running the code or scripts necessary to accomplish some measurement goal—remains surprisingly painful. There are a lot of reasons for this, including an infinite variety of analyses, sheer data volume, and the details of packet format and protocol parsing. These requirements are at odds. For example, scripting languages make it easy to support a wide variety of analyses; but simple Perl scripts that process `tcpdump`'s textual output tend to be slow and to ignore protocol details. Entire continuous-query systems have been designed from the ground up to address these issues [3].

This paper instead advocates an ad-hoc approach that ends up being surprisingly powerful: Build network analyses using a router forwarding path—specifically, the Click modular router [8].

Click was designed not for measurement, but as a high-performance software forwarding path. The components originally distributed with Click focused on forwarding and other active tasks (for example, decrementing IP packets' time-to-live). But routing and measurement both involve slinging lots of packets around, a task at which Click excels. Click's infrastructure for building efficient, modular forwarding paths naturally supports efficient, modular trace analyses, given the right components; and those components are easy and natural to write. Minimal extensibility mechanisms, such as a limited per-packet annotation area, have large reach. Components used at user level for trace analysis can migrate into a kernel forwarding path when desired, avoiding the high overhead of copying packets to user level. Click's modular component architecture makes these efficient configurations relatively easy to write and read.

Compared to `libpcap`-based analyses, Click makes it easier to write reusable analysis fragments. It's also easy to handle multiple data sources in a single configuration. Compared to perhaps the most common trace manipulation methodology—namely, scripts that analyze ASCII `tcpdump` output—Click analyses are faster, and often just as readable and easier to make correct. In addition, new components let Click generate an ASCII trace format more consistent and manipulable than `tcpdump` itself.

In terms of measurement tools, Click is most similar to Fisk and Varghese's data flow-based `smacqq` tool [3], on which some of Click's elements are based (*AggregateFirst* and *AggregateLast*, §4.1). Click is less limited in some ways (its dataflow graphs can branch) and more limited in important other ways (it doesn't support an arbitrary dynamic type system). Its infrastructure is somewhat faster and it offers a smooth path to kernel deployment, should that be necessary; it also provides a wider collection of file reading elements. The most salient distinction between the systems lies in their different attitudes towards data types. `Smacq` modules are polymorphic, operating on many data types; it is operation-oriented, not data-oriented. In contrast, Click, partially because of its routing heritage, focuses entirely on *one* compound data type, the packet. Depending on taste, this packet-oriented focus can make it easier to conceptualize analyses. This paper's contribution is to show how much can be done within a constrained, lean data model. Compared to other modular networking systems [6, 11], Click's basic abstractions tend to be more flexible [4]. Unlike Scout, for example, Click enforces no overall processing structure. This paper demonstrates some unintended, but pleasant, consequences of this flexibility.

2 Click

Click routers consist of components called *elements* plugged together into *configurations*. Elements process packets in various ways—creating them, modifying them, classifying them into different paths, and so forth. Packets flow from element to element along the edges of a configuration graph. Example elements include "*FromDevice(eth1)*", which reads and emits packets from network device `eth1`, and "*Discard*", which drops any packets it receives. Here's a simple router configuration file using those elements:

```
FromDevice(eth1) -> Discard;
```

This file, like any Click configuration, uses a simple declarative language. Compiler-like optimization and analysis passes can transform Click-language files to improve performance [9].

Connections between elements can use either *push* or *pull processing*. In push processing, packets are actively pushed forward through the graph. In pull processing, packet *requests* move *backwards* through the graph.

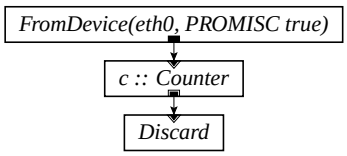

Figure 1: Counting packets from the *eth0* device.

Pull processing models packet transfer as an "upcall": downstream elements call upwards to retrieve a packet. The combination of push and pull can model complex control flow patterns, including explicit queues.

A Click driver can run inside the Linux kernel, or at user level on any Unix-like OS. Most element source files can be compiled for either driver. Click kernel configurations can run at or close to the limits of conventional PC buses [2, 8, 9]. For example, an optimized Click IP router can forward 740,000 minimum-size packets a second over Gigabit Ethernet on a 1.6 GHz Athlon MP with 64-bit/66 MHz PCI [9].

3 SIMPLE ANALYSES

Figure 1 shows one of the simplest analyses possible: a Click configuration that counts all packets arriving on device *eth0*, then throws those packets away. The "*PROMISC true*" argument says that the *eth0* device should be put into promiscuous mode. This configuration runs equally well at user level or inside a patched Linux or FreeBSD kernel. At user level, it extracts packets from the OS via *libpcap* or a raw packet socket—an operation not without overhead. In a patched kernel, it steals packets directly from the relevant device queues, avoiding all packet copies and scheduling delay. Furthermore, for certain network cards, a *PollDevice* element can be used instead of *FromDevice*. This increases performance at high input rates via polling, which eliminates all interrupt and programmed-I/O overhead as well as any receive livelock [10].

User and kernel behavior will differ in one important way: User-level configurations *augment* normal network processing, so the kernel still processes every packet; but kernel configurations *replace* normal network processing, so this configuration will effectively disconnect the host machine from its *eth0* card. This may or may not be what you want. Replacing *Discard* with a *ToHost* element will pass all packets to the kernel network stack for normal processing.

3.1 Extracting results

Click elements export the statistics they collect through *handlers*, or functions whose arguments and return values are textual strings. This text interface makes handler interactions relatively easy to script. For example, this user-level configuration, based on Figure 1, counts packets for 10 seconds and prints the result:

```
FromDevice(...) -> c :: Counter -> Discard;
DriverManager(wait 10s, save c.count -, stop)
```

The *DriverManager* element does not process any packets; instead, it directs Click's progress using a high-level script. This script runs Click for 10 seconds, then calls element *c*'s *count* handler and writes the result to standard output. The *count* handler simply returns the packet count seen by *Counter* as a decimal string.

Data-dependent behavior can be implemented by calling handlers contigent on particular events. For example, this configuration reports how long it takes to receive 1000 packets:

```
FromDevice(...)
   -> Counter(COUNT_CALL 1000 stop)
   -> tr :: TimeRange -> Discard;
DriverManager(wait, save tr.interval -, stop)
```

The *Counter* element calls the "*stop*" handler after receiving the 1000th packet. The *DriverManager* has been paused at the the "*wait*" instruction, waiting for this call; it then prints the value of *tr*'s *interval* handler (which returns the difference in seconds between the last and first timestamps the element has seen), and stops. This level of scripting makes easy analyses easy to write; to support more complex control flow patterns, a programmer would generally write corresponding C++ code as an element.

Handlers are directly accessible as well. For kernel configurations, handlers appear as files in a `/proc`-like file system; extracting the *c* element's count is as simple as `cat /click/c/count`. For user-level configurations, the *ControlSocket* element lets other programs connect to a TCP or UNIX socket and read handler values.

3.2 Classification

Figure 1 counts *all* packets, but most analyses focus on a subset instead. Click provides many elements that can help with this classification task. The generic *Classifier* element examines arbitrary bytes of packet data. The IP-specific *IPClassifier* element accepts a variant of the Berkeley Packet Filter language, as in *IPClassifier(tcp syn && src port 80)*. *Classifier* and *IPClassifier* classify packets by traversing binary decision trees. These trees are extensively optimized to eliminate redundant checks, making classification fast; for even faster results, the *click-fastclassifier* tool can dynamically generate the corresponding machine code [9]. Other elements implement more specific tasks; *CheckIPHeader*, for example, classifies packets into those with valid IP headers and those without, and *RandomSample* samples each packet with some probability.

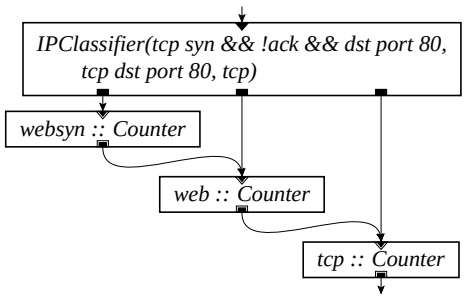

Figure 2: Counting overlapping kinds of traffic with a single classifier and dataflow.

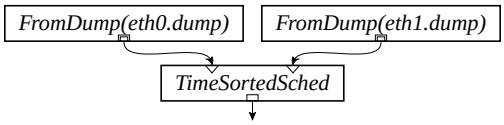

Figure 3: Merging two overlapping `pcap` files into a single packet stream sorted by timestamp.

Figure 2 demonstrates a particularly useful classification pattern, combining independent analyses into one configuration. This configuration counts three overlapping categories of packet: (1) TCP SYN packets sent to port 80, (2) any TCP packets set to port 80, and (3) all TCP packets. Instead of classifying the packet stream three times, the user has built the overlapping categories into the configuration's data flow: every packet in category (1) is also in category (2), so the configuration sends packets from the category-(1) counter directly into the category-(2) counter. This query optimization structure avoids redundant checks and is easily automated.

3.3 Sources

User-level Click configurations can read not only from live network devices, but also from a wide variety of trace formats. For example, the following configuration prints the number of packets in a *pcap* dump file:

```
FromDump(x.trace, STOP true)
   -> c :: Counter -> Discard;
DriverManager(wait, save c.count -, stop)
```

The *FromDump* element reads packet data from the named file, constructs the corresponding Click packet structures, and emits them downstream. *FromDump* uses efficient memory-mapped I/O to reduce copies and memory allocation; each packet is actually a pointer into a shared immutable buffer. The "*STOP true*" argument calls the "*stop*" handler when the dump file runs out of packets. As before, this wakes up *DriverManager*, which outputs the count to standard output.

Normally, *FromDump* emits packets as fast as it can read them from the file; this behavior mirrors that of network sources. However, it can also act in passive or *pull mode*, where it returns packets only in response to requests. Pull mode, plus Click's support for packet scheduling, allows the easy construction of unusual configurations. For example, Figure 3 uses pull mode and an odd packet scheduler to merge two independent dump files into a single stream of packets sorted by timestamp. The *TimeSortedSched* element maintains a one-packet-long queue for each of its inputs; on receiving a pull request, it responds with the earliest packet in any of the queues, then refills that queue with an upstream pull request. Previously whole programs have been written to do this merge.

Click can also read NLANR trace files (*FromNLANRDump*), DAG trace files generated by Endace tools (*FromDAGDump*), traces from Mark Allman's *cap* tool [1] (*FromCapDump*), textual NetFlow summary files (*FromNetFlowSummaryDump*), and even textual `tcpdump` output (*FromTcpdump*), and will transparently decompress compressed files when necessary. Some of these formats can be written, with *ToDump* et al. Users can also construct arbitrary test streams using Click's packet generation elements.

3.4 Packet modification

Click's wide range of packet modification elements is also available for measurement. For example, the *IPReassembler* element can be used to efficiently reassemble packets, simplifying tasks such as TCP normalization [5]; as it's designed for in-forwarding-path use, *IPReassembler* carefully avoids resource exhaustion attacks. The *AnonymizeIPAddr* element anonymizes IP packets' addresses, using a *tcpdpriv*-style prefix-preserving anonymization function, either at user level or in the kernel; *EraseIPPayload* extends this anonymization by removing any packet payload.

3.5 Discussion

In Linux kernel configurations, polling via *PollDevice* can take precedence over all user processes. Processes will not starve, but they can slow down significantly. This can cause problems for measurement; for example, any user-level `tcpdump` process might not be scheduled frequently enough. To mitigate this, an optional adaptive scheduler can limit Click's maximum CPU share to a user-defined fraction.

4 BIGGER ANALYSES

Click of course supports analyses more complex than counting packets. This section describes several, beginning with the *aggregate* elements that count packets based on an aggregate annotation, and moving on to elements for analyzing TCP.

4.1 Aggregates

Click was designed for routing packets, not arbitrary analyses, and it derives much of its speed from lean data structure design. Even routers need to annotate packets with information, however; for example, they must remember packets' source interfaces so as to generate appropriate redirect messages. Therefore, each Click packet provides limited *annotation space*: 24 bytes, plus a small set of general annotations (a timestamp, a pointer to the network header, a destination IP address, and so forth). This space turns out to suffice for many analyses as well.

A particularly common analysis task is to count the number of packets seen for each possible value of some field or property—for example, to count the number of packets sent to each /8 network, or the number of packets sent per IP protocol. In Click, this is carried out using the *aggregate annotation*. This four-byte annotation contains the packet's *aggregate number*, a generic packet identifier. Some elements set the aggregate annotation based on packet data, while others count the number of packets with each aggregate annotation value or otherwise manipulate aggregates. For example, this configuration counts the number of packets seen per destination /8:

```
... -> AggregateIP(ip dst/8)
   -> ac :: AggregateCounter -> Discard;
DriverManager(wait, write ac.write_text_file -,
      stop)
```

The *write_text_file* handler call causes *AggregateCounter* to write a file such as this to standard output:

```
!num_nonzero 2
128 1
134 1
```

This file indicates that *AggregateCounter* saw two packets, one heading to network 128.0.0.0/8 and the other to 134.0.0.0/8. "`!num_nonzero`" reports the number of aggregates with nonzero counts. *AggregateCounter* uses an efficient prefix tree data structure to store its counts, and can write packed binary as well as textual output.

The aggregate annotation can be set in several ways. *AggregateIP* sets it based on the (possibly partial) value of some TCP/IP header field. *AggregateLength* sets it based on packet length. Particularly powerful is *AggregateIPFlows*, which sets the annotation based on TCP or UDP flow identity: two packets that are part of the same end-to-end flow will receive the same aggregate annotation. *AggregateIPFlows* watches TCP flows for FIN handshakes and correct RSTs, thus separating distinct flows with the same address/port 4-tuple, and includes configurable timeouts for UDP (and TCP). It correctly handles fragments, assigning them the correct aggregate annotation once the TCP/UDP header can be determined, and ICMP errors, assigning them to the relevant flows. It also localizes this intelligence: only *AggregateIPFlows* needs to know the semantics of flow completion.

Besides *AggregateCounter*, the aggregate annotation is used by several elements. *AggregateFilter* classifies packets based on their aggregate annotations. *AggregateFirst* emits the first packet seen for each aggregate annotation, while *AggregateLast* emits the last packet seen. *AggregateLast* also marks each packet with the first and last timestamps observed, and the total number of packets and bytes seen with the aggregate value. Also noteworthy is *AggregateCounter*'s *counts_pdf* handler, which transforms aggregate counter data into a scaled PDF of the original counts. Thus, if exactly five aggregates in the old *AggregateCounter* had counts of 42, then the new *AggregateCounter*'s value for aggregate 42 will equal five.

Putting it all together, consider the following:

```
... -> AggregateIPFlows
   -> AggregateFirst
   -> AggregateIP(ip dst/8)
   -> ac :: AggregateCounter -> ...
DriverManager(wait, write ac.counts_pdf,
   write ac.write_pdf_file -, stop)
```

The first element aggregates packets by flow identifier. The second drops all but one packet per flow identifier. The third resets the aggregate annotation to the top 8 bits of the destination address, and *ac* counts the results. Thus, the *ac* element counts how many flows are present in the trace per destination /8 network. The handlers in the *DriverManager* element transform those counts into a PDF, which is written to standard output. Output like

```
100 0.42968
128 0.125  ...
```

would indicate that in this trace, 43% of /8 networks contained exactly 100 flows; 12.5% of /8 networks contained exactly 128 flows; and so forth. Thus, the aggregate elements naturally combine to calculate an interesting measurement result.

While the aggregate annotation is sufficient for many purposes, it is limited to 32 bits in length. For IPv6 addresses, for example, this will not suffice; and switching to a larger annotation would require changing many elements. This is one instance where `smacq`-style generic types would be a win.

4.2 TCP analysis

Aggregate elements are quite effective for simple analyses, but less so for more complex ones, such as available bandwidth analysis for TCP flows. These analyses require fairly complex control flow poorly expressed by data flow between simple modules. Thus, Click's TCP analysis elements tend to be more coarse grained, and introduce extensibility mechanisms of their own.

The central TCP analysis element is called *TCPCollector*. This element simply collects a detailed record of every TCP flow it sees, including a digest of each packet. (TCP flows are defined by aggregate annotations, so *TCPCollector* usually appears downstream of an *AggregateIPFlows* element.) Packet digests include sequence number, acknowledgement and SACK, packet number, TCP flags, IP ID, and timestamp information, as well as a set of informational flags: Was the packet a window probe? Did it fill the receive window? Was the packet in sequence order, or not? Does it appear to be a duplicate or a (possibly partial) retransmission? These digests are stored on doubly linked lists. There are also per-flow digests, which record initial and final sequence numbers and timestamps, maximum sequence numbers, packet counts, window scaling, MTU, and so forth, and maintain pointers to the packet digests. Total memory usage is 48 bytes per packet plus 172 bytes per connection.

TCPCollector does little analysis of its own, aside from setting sequence-ordering flags. Its output is an XML file giving important properties of each connection:

```
<?xml version='1.0' standalone='yes'?>
<trace file='<stdin>'>
<flow aggregate='1' src='146.164.69.8' sport='33397'
        dst='192.150.187.11' dport='80'
        begin='1028667433.955909' duration='131.647561'
        filepos='24'>
  <stream dir='0' ndata='3' nack='1508' beginseq='1543502210'
        seqlen='748' sentsackok='yes'> </stream>
  <stream dir='1' ndata='2487' nack='0' beginseq='2831743689'
        seqlen='3548305'> </stream> </flow> </trace>
```

Note the `filepos` attribute, which specifies where in the trace file the first packet on this flow appears. A trace reader can use `filepos` to skip ahead to the relevant portion of a large trace; Click's *FromDump* element, for example, accepts a *FILEPOS* argument.

Besides this summary information, *TCPCollector* can also generate XML containing summaries of each data packet, or lists of interarrival times. But its real power comes from its extensibility. Other elements can hook up to *TCPCollector* to store extra information in each packet or stream digest, and to add their own per-stream XML tags to *TCPCollector*'s output. For example, the *MultiQ* element hooks up to *TCPCollector* to provide one or more bottleneck capacity estimates for each observed significant TCP flow, based on packet interarrival times. Capacity estimates are based on the MultiQ algorithm [7]. *MultiQ* checks each TCP flow's MTU and size to determine significance, then calculates interarrival times from the packet digests, runs the algorithm, and reports the results as part of a `<multiq_capacity>` XML tag. It can also calculate interarrival times from an input packet stream, but this works only if the input stream is one flow; associating with *TCPCollector* makes it easy to work on multiple flows. The division of labor is simple: *AggregateIPFlows* keeps track of the details of flow association, *TCPCollector* remembers per-flow state, and *MultiQ* concentrates on its capacity estimation algorithm. Moving to Click, from an earlier Python-based tool, let us process raw traces, support multiple flows, and analyze flows about an order of magnitude faster, while keeping the MultiQ-specific code about the same length (and this is not counting several external tools that the Click version no longer requires).

Besides *MultiQ*, Click also supports a prototype loss inference tool called *TCPMystery* which makes much more extensive use of packet sequence number data.

TCPCollector's memory usage is bounded by the number of currently active connections, and thus by the packet stream and *AggregateIPFlows*'s timeout for completed TCP connections. For fast traces, this can get large; when processing a 440 MB, 5.7 million packet DAG-formatted packet trace, representing 5 minutes of traffic on dual gigabit Ethernet links, *TCPCollector* occupied a maximum of 275 MB of memory (on a dual 2 GHz PowerPC with 512 MB of memory). It took just 24 seconds to process this trace, however. For comparison, a Click configuration that simply read the trace, throwing all its packets away, took about 9 seconds, roughly the same as `gzcat trace | wc -c`. If memory becomes a problem, the user can simply run Click multiple times, sampling disjoint subsets of flows with a *Classifier*-like element each time.

5 APPLICATIONS

It is certainly possible, and even easy, to write measurement tools and analyses directly as Click scripts. After running your tenth or eleventh trivial variant of the same configuration, however, a more automated tool seems attractive. Click naturally supports applications that parse command line options, generate a corresponding Click script, and then run it. Most application logic boils down to the selection and combination of elements. If more logic is required, the Click library and elements can be linked with the application directly; the application can provide its own elements or handlers, facilitating two-way communication.

The most widely used Click measurement application has a seemingly-simple specification. *Ipsumdump* reads IP packets from one or more data sources, then summarizes those packets into a textual file, one packet per line. Unlike `tcpdump`, with its complex and evolving output, *ipsumdump* gives the user full control over what information appears on each line, and its output was designed for easy processing by scripts.

The *ipsumdump* tool simply creates and runs a configuration containing a *ToIPSummaryDump* element. For example, this configuration:

```
FromDump(a.dump, FORCE_IP true, STOP true)
```

```
-> ToIPSummaryDump(out.txt, CONTENTS
        src dst proto sport dport);
```

might write this to `out.txt`:

```
!IPSummaryDump 1.1 ...
!data ip_src ip_dst proto sport dport
192.150.187.20 192.150.187.34 T 22 3243
206.71.111.206 192.150.187.11 T 4044 80
192.150.187.11 206.71.111.206 T 80 4044 ...
```

Other supported fields include timestamp, IP and TCP options, sequence numbers, and so forth. Additional fields are very easy to add. And there is a corresponding source element as well: *FromIPSummaryDump* reads *ipsumdump* files and emits packets with the specified characteristics.

Ipsumdump has proven a surprisingly useful tool. For measurement or tool designers, such as the authors of Bro [12], its output is a human- and script-friendly *lingua franca*. Adding requested functionality, such as collating multiple packet sources (*TimeSortedSched*), random sampling (*RandomSample*), or writing `pcap`-formatted dump files (*ToDump*), has usually proven trivial. The elements have seen other uses as well. Hand-written *ipsumdump* files, changed into packets via *FromIPSummaryDump*, make natural regression test input for packet processing functionality. Even better, *FromIPSummaryDump* can turn *non*-packets into packets, allowing broader application of Click measurement elements. For example, here is a Click script that changes a file of interarrival times in seconds, one per line, into a sequence of packets with those values as timestamps, then calculates the implied capacity using *MultiQ*:

```
FromIPSummaryDump(file.txt,
       CONTENTS timestamp, STOP true)
   -> MultiQ(RAW_TIMESTAMP true) -> Discard;
```

An interesting variant, *ToIPFlowDumps*, splits a packet stream into *many* summary files, one per aggregate annotation (i.e., one per TCP flow). Careful caching and file descriptor usage lets *ToIPFlowDumps* explode a packet stream into tens of thousands of single-flow files, optionally stored in many directories, at file system speeds.

Other tools generate aggregate count files for arbitrarily-defined aggregates (*ipaggcreate*) or manipulate aggregate counts in various ways, including calculating wavelet energy (*aggmanip*).

6 Conclusion

The code described in this paper is all freely available. See `http://pdos.csail.mit.edu/click/`.

This material is based in part upon work supported by the National Science Foundation under Grant No. 0230921. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

References

[1] Mark Allman. Measuring end-to-end bulk transfer capacity. In *Proc. 1st Internet Measurement Workshop (IMW '01)*, San Francisco, California, November 2001.

[2] Benjie Chen and Robert Morris. Flexible control of parallelism in a multiprocessor PC router. In *Proc. 2001 USENIX Annual Technical Conference (USENIX '01)*, June 2001.

[3] Mike Fisk and George Varghese. Agile and scalable analysis of network events. In *Proc. 2nd Internet Measurement Workshop*, November 2002.

[4] Yitzchak Gottlieb and Larry Peterson. A comparative study of extensible routers. In *Proc. 5th International Conference on Open Architectures and Network Programming (OPENARCH '02)*, pages 51–62, New York, New York, June 2002.

[5] Mark Handley, Vern Paxson, and Christian Kreibich. Network intrusion detection: Evasion, traffic normalization, and end-to-end protocol semantics. In *Proc. 10th USENIX Security Symposium (Security '01)*, Washington, DC, August 2001.

[6] N. C. Hutchinson and L. L. Peterson. The *x*-kernel: an architecture for implementing network protocols. *IEEE Transactions on Software Engineering*, 17(1):64–76, January 1991.

[7] Sachin Katti, Dina Katabi, Charles Blake, Eddie Kohler, and Jacob Strauss. MultiQ: Automated detection of multiple bottleneck capacities along a path. In *Proc. Internet Measurement Conference (IMC) 2004*, Taormina, Sicily, October 2004.

[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] Eddie Kohler, Robert Morris, and Benjie Chen. Programming language optimizations for modular router configurations. In *Proc. 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-X)*, pages 251–263, San Jose, California, October 2002.

[10] Jeffrey C. Mogul and K. K. Ramakrishnan. Eliminating receive livelock in an interrupt-driven kernel. *ACM Transactions on Computer Systems*, 15(3):217–252, August 1997.

[11] David Mosberger and Larry L. Peterson. Making paths explicit in the Scout operating system. In *Proc. 2nd Symposium on Operating Systems Design and Implementation (OSDI '96)*, pages 153–167, October 1996.

[12] Vern Paxson. Bro: A system for detecting network intruders in real-time. *Computer Networks*, 31(23–24):2435–63, December 1999.