17th ACM Symposium on Operating Systems Principles (SOSP '99)
A reformatted version was published as *Operating Systems Review* **34**(5):217–231, December 1999

The Click modular router

Robert Morris, Eddie Kohler, John Jannotti, and M. Frans Kaashoek
MIT Laboratory for Computer Science
{*rtm, eddietwo, jj, kaashoek*}*@lcs.mit.edu*

Abstract

Click is a new software architecture for building flexible and configurable routers. A Click router is assembled from packet processing modules called elements. *Individual elements implement simple router functions like packet classification, queueing, scheduling, and interfacing with network devices. Complete configurations are built by connecting elements into a graph; packets flow along the graph's edges. Several features make individual elements more powerful and complex configurations easier to write, including* pull processing, *which models packet flow driven by transmitting interfaces, and* flow-based router context, *which helps an element locate other interesting elements.*

We demonstrate several working configurations, including an IP router and an Ethernet bridge. These configurations are modular—the IP router has 16 elements on the forwarding path—and easy to extend by adding additional elements, which we demonstrate with augmented configurations. On commodity PC hardware running Linux, the Click IP router can forward 64-byte packets at 73,000 packets per second, just 10% slower than Linux alone.

1 Introduction

Routers are increasingly expected to do more than route packets. Boundary routers, which lie on the borders between organizations, must often prioritize traffic, translate network addresses, tunnel or filter packets, or act as firewalls, among other things. Furthermore, fundamental router policies like packet dropping are still under active research [5, 11, 13], and initiatives like Differentiated Services [3] are bringing the need for flexibility closer to the core of the Internet.

Unfortunately, most routers have closed, static, and inflexible designs. Network administrators may be able to turn router functions on or off, but they cannot easily specify or even identify the interactions of different functions. Furthermore, network administrators and third party software vendors cannot easily implement new functions. Extensions require access to software interfaces in the router's forwarding path, but these often don't exist, don't exist at the right point, or aren't published.

This paper presents Click, a flexible, modular software architecture for building routers. Click's building blocks are packet processing modules called *elements*. To build a router configuration, the user connects a collection of elements into a graph; packets move from element to element along the graph's edges. To extend a configuration, the user can write new elements or compose existing ones in new ways, much as UNIX allows one to build complex applications directly or by composing simpler ones using pipes.

Two specific features add power to this simple architecture. *Pull processing* models packet motion driven by transmitting interfaces and makes packet schedulers easy to compose, and *flow-based router context* examines the router graph to help an element locate other interesting elements. We present an element in Section 4.2 that, using these features, implements four variants of the random early detection dropping policy (RED) [11]—RED, RED over multiple queues, weighted RED, and drop-from-front RED—depending on its context in the router. This would be difficult or impossible to achieve in previous modular networking systems [12, 18, 25].

We have implemented this architecture on general-purpose hardware (which is cheap and has good performance) as an extension to Linux. A Click IP router running on a 450 MHz Pentium III can forward 73,000 64-byte packets per second, and can forward 250-byte packets (the average size seen on WAN links [28]) at 100 megabits per second.

In the next sections, we describe Click's architecture in detail, including the language used to describe configurations (Section 2), present a functioning Click IP router

This research was supported by a National Science Foundation (NSF) Young Investigator Award and the Defense Advanced Research Projects Agency (DARPA) and Rome Laboratory under agreement number F30602-97-2-0288. In addition, Eddie Kohler was supported by a National Science Foundation Graduate Research Fellowship.

SOSP-17 12/1999 Kiawah Island, SC

(Section 3), and outline some useful router extensions as implemented in Click (Section 4). After summarizing our implementation (Section 5), we evaluate its performance on some of the presented routers (Section 6). Finally, we describe related work (Section 7) and summarize our conclusions (Section 8).

2 Architecture

A Click router configuration is a directed graph whose nodes are called *elements*. A single element represents a unit of router processing. An edge, or *connection*, between two elements represents a possible path for packet transfer. This graph resembles a flowchart, except that connections represent packet flow, not control flow, and elements are actual objects that may maintain private state. Inside a running router, each element is a C++ object and connections are pointers to elements. The overhead of passing a packet along a connection is a single virtual function call.

The most important properties of an element are:

- *Element class*. Like objects in an object-oriented program, each element has a class that determines its behavior.
- *Input and output ports*. Ports are the endpoints of connections between elements. An element can have any number of input or output ports, which can have different semantic meanings (a normal and an error output, for example).
- *Configuration string*. Some element classes support additional arguments, used to initialize per-element state and fine-tune element behavior. The configuration string contains these arguments.

Figure 1 shows how we diagram these properties for a single element, *Tee(2)*. '*Tee*' is the element class; a *Tee* copies each packet it receives from its single input port, sending one copy to each output port. (The packet data is not copied: Click packets are copy-on-write.) Configuration strings are enclosed in parentheses: the '2' in '*Tee(2)*' is a configuration string that *Tee* interprets as a request for two outputs.

Every action performed by a Click router's software is encapsulated in an element, from device reading and writing to queueing, routing table lookups, and counting packets. The user determines what a Click router does by choosing the elements to be used and the connections among them. Figure 2 shows a sample router that counts incoming packets, then throws them all away.

Click provides two kinds of connections between elements, *push* and *pull*. In a push connection, the upstream element hands a packet to the downstream element; in a pull connection, the downstream element asks the upstream element to return a packet. Each kind of handoff is implemented as a virtual function call. Packet arrival usually initiates push processing, which stops when an element discards the packet or stores it for later. Output interfaces initiate pull processing when they are ready to send a packet; processing flows backwards through the graph until an element yields up a packet. Pull elements can simply and explicitly represent decisions that should occur at packet transmission time, such as packet scheduling.

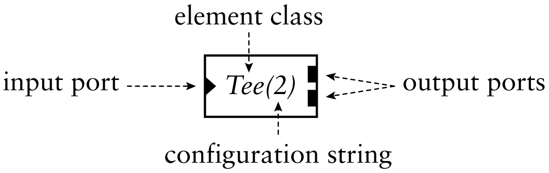

Figure 1: A sample element. Triangular ports are inputs and rectangular ports are outputs.

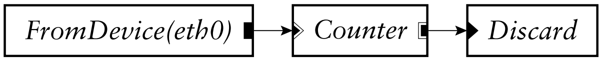

Figure 2: A router configuration that throws away all packets.

The rest of this section discusses the Click architecture in more detail, including push and pull processing, flow-based router context, the implementation of an element, and the Click language for specifying router configurations.

2.1 Control flow and queues

When an element receives a packet from a push connection, it must store it, discard it, or forward it to another element for more processing. Most elements forward packets by calling the next element's `push` function. Since packet handoff is just a virtual function call, a Click CPU scheduler could not stop packet processing at arbitrary points—elements must cooperatively choose to stop processing.

Packet storage must be implemented by the element itself; unlike some systems [18, 25], Click elements do not have implicit queues on their input and output ports, or the associated performance and complexity costs. Instead, Click queues are explicit objects, implemented by a separate element (*Queue*). This enables valuable configurations that are difficult to arrange otherwise—for example, a single queue feeding multiple interfaces, or a queue feeding a traffic shaper on the way to an interface. *Queue* is the most common element that stops packet processing, giving the system a chance to schedule different work: it enqueues packets it receives rather

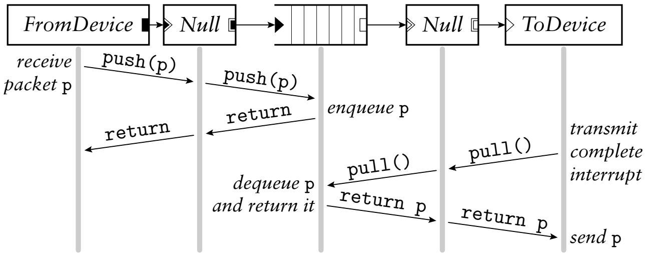

Figure 3: Push and pull control flow. This diagram shows functions called as a packet moves through a simple router. The central element is a *Queue*. During the push, control flow moves forward through the element graph starting at the receiving interface; during the pull, control flow moves *backward* through the graph, starting at the *transmitting* interface. The packet p always moves forward.

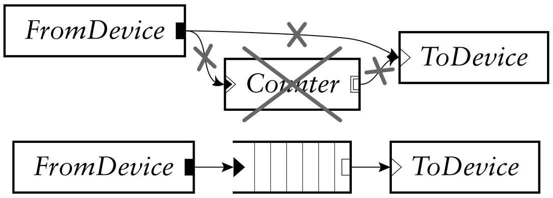

Figure 4: Some invariant violations. The top configuration has four errors: (1) *FromDevice*'s push output connects to *ToDevice*'s pull input; (2) more than one connection to *FromDevice*'s push output; (3) more than one connection to *ToDevice*'s pull input; and (4) the agnostic element *Counter* is in a mixed push/pull context. By contrast, the bottom configuration is legal. In a properly configured router, the port colors on either end of each connection will match.

than passing them on. Thus, the placement of *Queue*s in a configuration determines that configuration's execution profile. If a user wants to carefully manage packet scheduling as soon as packets enter the system, she will want *Queue*s early in the graph.

2.2 Push and pull processing

Push and pull are duals of one another: the upstream end of a connection initiates a push call, while the downstream end initiates a pull call. Together, push and pull allow the appropriate end of a connection to initiate packet transfer, solving several router control flow problems. For example, packet scheduling decisions—choosing which queue to ask for a packet—are easily expressed as composable pull elements, as we show in Section 4.1. As another example, the system should not send packets to a busy transmitting interface. If it did, the interface would have to store the packet, and the router would lose the ability to affect it later (to throw it away, to modify its precedence, and so forth). This restriction can be simply expressed by giving the transmitting interface a pull input; then the interface is in control of packet transfer, and can ask for packets only when it's ready.

Figure 3 shows how this works in a simple router. In our configuration diagrams, black ports are push and white ports are pull. This particular configuration has two *Null* elements, one push and one pull. Like many elements, *Null* is *agnostic*, meaning it can work as either push or pull depending on its context in the router. Agnostic ports are shown in diagrams as push or pull ports with a double outline.

The following invariants hold for all correctly configured routers: Push outputs must be connected to push inputs, and pull outputs must be connected to pull inputs. Each agnostic port must be used as push or pull exclusively; furthermore, if packets can flow within an element between an agnostic input and an agnostic output, both ports must be used in the same way (either push or pull). Finally, push outputs and pull inputs must be connected exactly once. (This ensures that each packet handoff—pushing to an output port or pulling from an input port—has a unique destination.) These invariants are automatically checked by the system during router initialization. Figure 4 demonstrates violations of each of them.

The invariants are designed to catch intuitively invalid configurations. For example, in Figure 4, the connection in the figure from *FromDevice* to *ToDevice* is disallowed by the invariants because *FromDevice*'s output is push while *ToDevice*'s input is pull. But this connection should be illegal: if it remained, *ToDevice* might receive packets when it was not ready to send them. The *Queue* element, which converts from push to pull, is also intuitively necessary to provide the temporary packet storage required.

Every push call in a running router passes an actual packet object, but pull calls can return a null pointer if no packet is ready. In this case, the pulling element must arrange to wake up when it makes sense to try again. This can be done element-specifically—using a timer, for example—but Click also includes a generic mechanism called *packet-upstream notification*. During initialization, each *Queue* uses flow-based router context (described in more detail below) to find the elements downstream of it that are interested in packet-upstream. When the *Queue* becomes nonempty, it notifies these elements of a packet-upstream event; they will soon react by retrying the pull. The combination of pull processing and packet-upstream notification resembles Clark's upcalls and arming calls [7].

2.3 Flow-based router context

Sometimes an element must find other elements that might not be directly connected to it. For example, a *Queue* must find the elements downstream of it that are interested in packet-upstream notification; these might

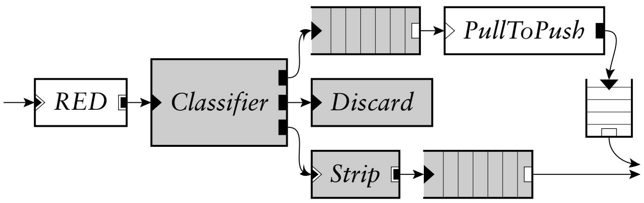

Figure 5: The elements downstream of *RED*, found by flow-based router context with a filter that stops at *Queue*s. The downstream elements are colored grey.

be directly connected to the *Queue*, or they might be separated from it by arbitrarily many elements. They are related to the *Queue* not by direct connection, but by its transitive closure, *packet flow*.

The Click architecture can provide any element with packet flow information for the whole router, which we call *flow-based router context*. For example, an element can find the elements downstream of its first output, or the elements upstream of its second input. These questions have a well-defined answer even in the presence of cycles in the router configuration.

The flow-based router context algorithms accept an optional filter that limits the search. If the filter matches an element on a downstream search, then nothing downstream of that element will be returned (unless it is reachable on another path), and similarly for upstream searches. Filters can match arbitrary element classes and interfaces, so searches can be stopped at *Queue*s (and subtypes of *Queue*) or at any element implementing a hypothetical *Queue*like interface. Figure 5 shows how this works. With these filters and flow-based router context, an element can find nearby elements that are known to implement a specific interface; it can then manipulate their exported variables and methods, gaining access to information like queue lengths, interface addresses, and so on.

2.4 Implementation

We implement elements as C++ objects. Each element class corresponds to a C++ subclass of `Element`, which has on the order of 20 virtual functions. `Element` provides reasonable default implementations for many of these, allowing most subclasses to get away with overriding six of them or less. Only two virtual functions are used during router operation, namely `push` and `pull`; the others are used for identification, push and pull specification, configuration, initialization, and statistics.

Subclasses of `Element` are easy to write, so we expect users will have no problem writing new element classes as needed. In fact, the complete implementation of a simple working element class (*Null*, which passes packets from its single input to its single output unchanged) takes less than 20 lines of code; see Figure 6. Most elements define functions for parsing configuration strings and initialization in addition to those in Figure 6, and take about 120 lines of code.

```
class NullElement : public Element {
 public:
  NullElement()
           { add_input(); add_output(); }
  const char *class_name() const
           { return "Null"; }
  PushOrPull default_processing() const
           { return AGNOSTIC; }
  NullElement *clone() const
           { return new NullElement; }
  void push(int port_number, Packet *p)
           { output(0).push(p); }
  Packet *pull(int port_number)
           { return input(0).pull(); }
};
```

Figure 6: The complete implementation of a do-nothing element.

2.5 Language

Click configurations are written in a simple textual language with two important constructs: *declarations* and *connections*. A declaration says that an element should be created; connections specify how those elements should be connected. Syntactic sugar allows a user to elide declarations and piggyback connections for readability. The syntax is easy enough to learn from an example; Figure 7 uses it to define a trivial router.

Configuration strings are opaque to the language. They are sent uninterpreted to the elements themselves, which are free to use them however they like. Most of the elements we have written treat configuration strings as comma-separated argument lists, using a common library to parse data like integers and IP addresses.

The language contains constructs that allow users to define new element classes by composing existing ones. Thus, any user can create a library of personalized element classes; for example, a user could define *MyQueue* to be a *Queue* followed by a *Shaper*, and use *MyQueue* as if it was a Click primitive. These new classes, called *compound elements*, are strictly compile-time constructs: at run time, a compound element has exactly the same representation as the corresponding collection of simple elements. Thus, compound elements have no additional run-time overhead.

Router configurations in the Click language can be optimized using a preprocessor based on pattern matching. The optimizer reads a router configuration and a

```
# a trivial router that drops everything
src :: FromDevice(eth0);
ctr :: Counter;
sink :: Discard;
src -> ctr;
ctr -> sink;

# the same, with anonymous elements
FromDevice(eth0) -> Counter -> Discard;
```

Figure 7: The trivial router of Figure 2 specified in two ways.

file describing element patterns and their replacements; it replaces patterns in the configuration until no more changes can be made, then writes out the new configuration. We plan to write other preprocessors, including one that checks configurations using a static type system. This would prevent users from sending Ethernet packets to elements that expect IP packets, for example. Currently, Click configurations are not type checked, except for the push and pull invariants described above.

3 An IP router

This section shows how a real router configuration—an IP router that forwards unicast packets in nearly full compliance with the standards [1, 23, 24]—can be written in Click. Figure 8 shows a two-interface Click IP router configuration. (The reader may want to refer to Figure 9, a glossary of Click elements used in Figure 8 and elsewhere in the paper.) The rest of this section describes the IP router in more detail. Section 4 shows how to extend this router by changing its scheduling and queueing behavior, and Section 6 evaluates its performance.

The IP forwarding tasks that are most natural in Click are those that involve only local information. For example, *DecIPTTL* decides if a packet's TTL has expired. If it has, it emits the packet on its second output (usually connected to an *ICMPError* element); if the TTL is still valid, *DecIPTTL* decrements it, updates the packet's checksum, and emits the packet on its first output. These actions depend only on the packet's contents; they don't interact with decisions made elsewhere except as expressed in the packet's path through the element graph. Such self-contained elements compose easily—for example, one could connect *DecIPTTL*'s "expired" output to a *Discard* to avoid generating ICMP errors, or insert an element that limits the rate at which errors are generated.

Some forwarding tasks require that information about a packet be calculated in one place and used in another. Click uses *annotations* to carry such information along. (An annotation is a piece of information attached to a packet that isn't part of the packet data.) The annotations used in the IP router include:

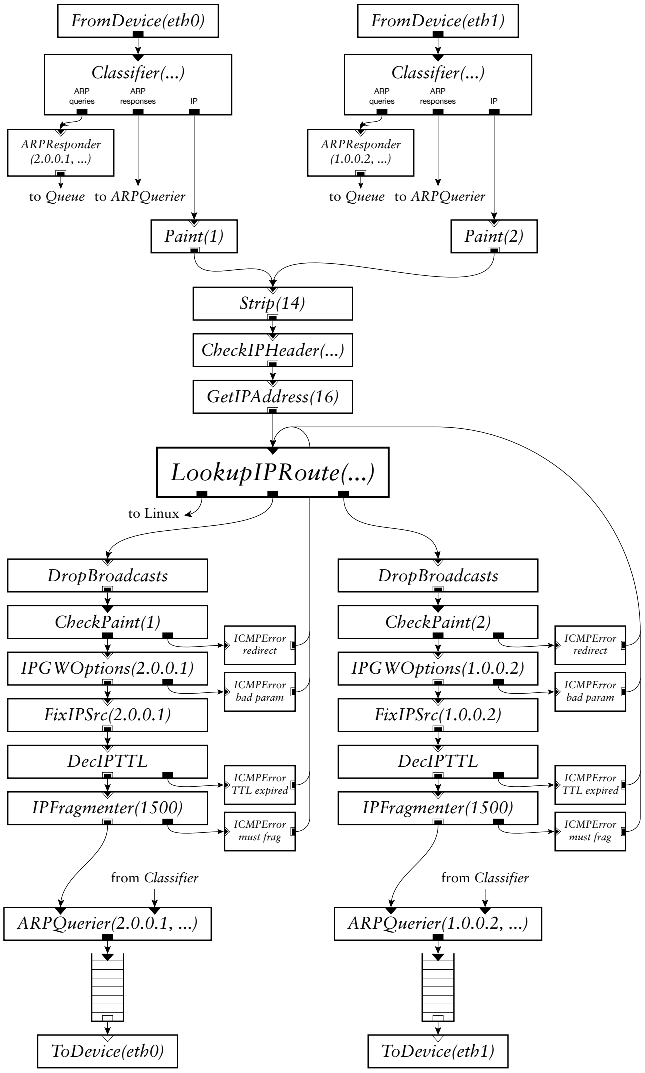

Figure 8: The IP router configuration.

Element	Description
ARPQuerier(...)	Encapsulates IP packets in Ethernet headers using ARP; 2nd input processes ARP responses
ARPResponder(x y)	Responds to ARP queries for IP address *x* with static Ethernet address *y*
CheckIPHeader(...)	Discards packets with invalid IP length or checksum fields
CheckPaint(p)	Sends packets with paint annotation = *p* to both outputs; otherwise just to first
Classifier(...)	Checks packet data against classifiers; sends packet to output for 1st classifier that matched
DecIPTTL	Decrements IP packets' time-to-live; sends to 2nd output iff TTL has expired
Discard	Discards all packets
DropBroadcasts	Discards packets that arrived as link-level broadcasts
EtherSpanTree(...)	Implements the IEEE 802.1d spanning tree algorithm for Ethernet switches
EtherSwitch	Learning, forwarding Ethernet switch
FixIPSrc(addr)	Sets the IP header's source field to *addr* if the Fix IP Source annotation is set
FromDevice(device)	Outputs packets when they arrive from a Linux device driver
GetIPAddress(...)	Copies the destination address from the IP header to the destination address annotation
HashDemux(...)	Sends packet to one of *n* outputs, chosen by a hash of specified packet contents
ICMPError(type, code)	Encapsulates IP packets in ICMP error packets, sets Fix IP Source annotation
IPEncap(...)	Encapsulates packets in a statically specified IP header
IPFragmenter(mtu)	Fragments IP packets larger than *mtu*; too-large packets with DF flag set go to 2nd output
IPGWOptions	Handles IP Record Route, Timestamp options; packets with invalid options go to 2nd output
LookupIPRoute	Looks up the destination annotation in a static routing table, choosing the output and setting the annotation based on the result
Meter(r)	Sends packets to 1st output if recent input rate averages < *r*, 2nd output otherwise
Paint(p)	Sets the paint annotation to *p*
PrioSched	Pulls a packet from one of *n* inputs; lower numbered inputs have priority
Queue(n)	Stores at most *n* packets in a queue
RED(...)	Drops packets probabilistically according to the Random Early Detection algorithm
RoundRobinSched	Pulls a packet from one of *n* inputs, chosen by round-robin
SetIPDSCP(c)	Sets the IP header's diffserv code point field to *c*
Shaper(n)	Simple pull traffic shaper: allows average of *n* packets per second
Strip(n)	Deletes packets' first *n* bytes
Suppressor	Optionally drops packets arriving on particular inputs
Tee(n)	Sends each packet to all *n* outputs
ToDevice(device)	Hands packets to a Linux device driver for transmission
ToLinux	Hands packets to Linux's default network input software

Figure 9: Element glossary.

- **Destination address.** Elements that deal with a packet's destination address use this annotation rather than the IP header field, allowing several such elements to be chained together. *GetIPAddress* copies the destination field from the IP header to the annotation, *LookupIPRoute* replaces the annotation with the next-hop gateway's address, and *ARPQuerier* maps the annotation to the next-hop Ethernet address.

- **Paint.** The *Paint* element marks a packet with an integer "color". *CheckPaint* emits every packet on its first output, and a copy of any packet with a given color on its second output. The IP router uses paint to decide whether a packet is leaving the same interface on which it arrived, and thus should prompt an ICMP redirect.

- **Link-level broadcast flag.** *FromDevice* sets this flag on packets that arrived as link-level broadcasts. The IP router uses *DropBroadcast* to drop such packets if they are about to be forwarded, but not if they are destined for the router itself.

- **ICMP Parameter Problem pointer.** This is set by *IPGWOptions* on erroneous packets to specify the bad IP header byte, and used by *ICMPError* when constructing an error message.

- **Fix IP Source flag.** The IP source address of an ICMP error packet must be the address of the interface on which the error is sent. *ICMPError* can't predict this interface, so it uses a default address and sets the Fix IP Source annotation. After the ICMP packet has been routed towards a particular interface, a *FixIP-*

Src on that path will see the flag, insert the correct source address, and recompute the IP checksum.

In a few cases elements require information of an inconveniently global nature. A router usually has a separate IP address on each attached network, and each network usually has a separate IP broadcast address. All of these addresses need to be known at multiple points in the Click configuration: *LookupIPRoute* needs to know how to decide if a packet is destined to the router itself, *CheckIPHeader* must discard a packet with any of the IP broadcast addresses as source address, *ICMPError* must suppress responses to IP broadcasts, and *IPGWOptions* must be able to recognize any of the router's addresses in an IP Timestamp option. Each of these elements takes the complete list of addresses as part of its configuration string, but ideally they would derive the list automatically using flow-based router context.

Some of the elements in Figure 8 require more explanation. *CheckIPHeader* checks the validity of the IP length fields, the IP source address, and the IP checksum. *IPGWOptions* processes just the Record Route and Timestamp options, since the source route options should be processed only on packets addressed to the router. *IPFragmenter* normally fragments packets larger than the configured MTU, but sends unfragmentable too-large packets to an error output instead. An *ICMPError* element encapsulates most input packets in an ICMP error message and outputs the result; it drops broadcasts, ICMP errors, fragments, and source-routed packets.

4 Extensions

This section presents Click configuration fragments that implement several useful router extensions. We have written elements that support RFC 2507-compatible IP header compression and decompression, IP security, communication with wireless radios, tunneling, and many other specialized routing tasks, but this section focuses on scheduling and dropping policies, queueing requirements, and Differentiated Services—and one non-IP router, an Ethernet switch. The last subsection concludes the discussion by presenting some of Click's architectural limitations.

4.1 Scheduling

With pull processing, a packet scheduler can be implemented in Click as a single element that maintains only local knowledge of the router configuration. Packet scheduling is a kind of multiplexing—a scheduler decides how a number of packet sources (usually queues) will share a single output channel—and a Click scheduler is a pull element with multiple inputs and one output. It reacts to requests for packets by choosing one of its inputs, pulling a packet from it, and returning it. (If the chosen input has no packets ready, the scheduler will usually try other inputs.)

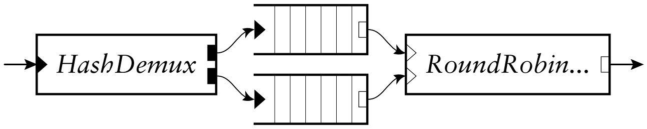

Figure 10: A virtual queue implementing Stochastic Fairness Queueing.

We have implemented two scheduler elements, *RoundRobinSched* and *PrioSched*. *RoundRobinSched* pulls from its inputs in round-robin order, returning the first packet it finds (or no packet, if no input has a packet ready). It always starts pulling on the input cyclically following the last successful pull. *PrioSched* (for *prio*rity *sched*uler) always tries its first input, then its second, and so forth, returning the first packet it gets.

Both *Queue*s and scheduling elements have a single pull output, so to an element downstream, *Queue*s and schedulers are indistinguishable. We can exploit this property to build *virtual queues*, compound elements that look exactly like queues from the outside but implement more complex behavior than FIFO queueing. Figure 10 shows a virtual queue that implements a version of Stochastic Fairness Queueing [15]: packets are hashed by flow identifier into one of several queues that are scheduled round-robin, providing some isolation between competing flows.

4.2 Dropping policies

The *Queue* element implements a simple dropping policy, namely a configurable maximum length beyond which all packets are dropped. More complex drop policies can be created by combining *Queue*s with other elements. For example, we implement random early detection [11] as an independent *RED* element containing only drop decision code. *RED* bases its decisions on queue lengths—specifically, the lengths of the nearest downstream *Queue*s, which it finds using flow-based router context. For example, in Figure 5 above, *RED* will include the grey *Queue*s in its queue length calculation.

If there is more than one downstream *Queue*, *RED* adds all their lengths together before performing the drop calculation. This simple generalization allows the user to create useful RED variants like RED over multiple queues by rearranging the configuration. Other variants like weighted RED [5], where packets are dropped with different probabilities depending on their priority, also naturally follow from modular *RED* elements (see Figure 11). In addition, the *RED* element can be positioned *after* the queue; in this case, it is a pull element and looks

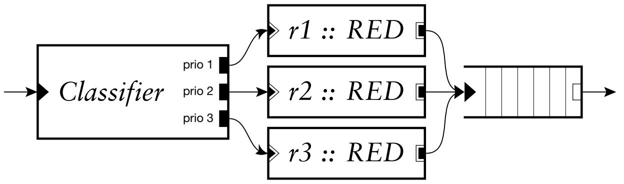

Figure 11: Weighted RED. The three *RED* elements can have different RED parameters, allowing packets with different priorities to be dropped with different probabilities when the router is under stress.

for upstream rather than downstream queues. This results in a strategy like drop-from-front [13], which reports congestion back to senders more quickly than the usual drop-from-tail.

4.3 Complex queueing

Imagine a router with the following requirements:

- two parallel T1 links to a backbone, between which traffic should be load-balanced;
- division of traffic into two priority levels;
- fairness among the connections within each priority level;
- RED dropping driven by the total number of packets queued.

Figure 12 shows how to build this combination in Click. Other router platforms provide these features individually, and perhaps in certain predefined combinations; in Click, since the configuration consists of simple elements composed together, many other configurations could be built by rearrangement or by choosing different elements.

4.4 Differentiated Services

The Differentiated Services architecture [3] specifies mechanisms for border and core routers to jointly manage aggregate traffic streams. Diffserv border routers classify and tag packets according to traffic type, and ensure that traffic enters the network no faster than allowed. Core routers queue and schedule packets based on their tags. The diffserv architecture envisions flexible combinations of classification, tagging, shaping, dropping, queuing, and scheduling functions. These components naturally correspond to Click elements, and building them as elements gives the router administrator full control over how they are arranged. For example, Figure 13 shows a Click configuration corresponding closely to Figure 4 (“An Example Traffic Conditioning Block”) in Bernet et al [2].

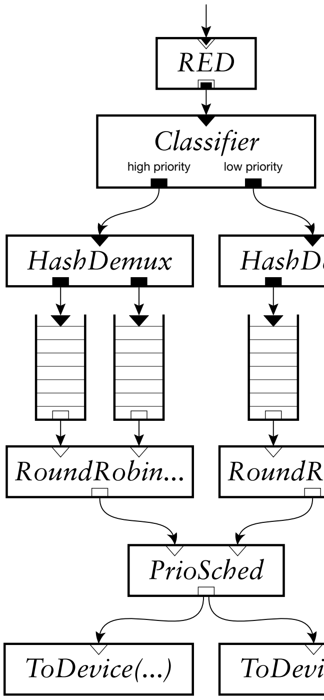

Figure 12: A complex combination of dropping, queueing, and scheduling. The *Classifier* prioritizes input packets into two virtual queues, each of which implements stochastic fair queueing (see Figure 10). *PrioSched* implements priority scheduling on the virtual queues, preferring packets from the left. The router is driving two equivalent T1 lines that pull packets from the same sources, providing a form of load balancing. Finally, *RED*, at the top, implements random early drop over all four *Queue*s.

This configuration separates incoming traffic into 4 streams, based on the IP Differentiated Services Code Point (DSCP) [20]. The first three streams are rate-limited, while the fourth represents normal best-effort delivery. The rate-limited streams are given priority over the normal stream. From left to right in Figure 13, the streams are (1) limited by dropping—whenever more than 7500 packets per second are being sent on average, the stream is dropped; (2) shaped—at most 10,000 packets per second are allowed through the *Shaper*, and any excess packets are queued; and (3) limited by reclassification—whenever more than 12,500 packets per second are being sent, the stream is reclassified as best-effort delivery and sent into the lower priority queue.

4.5 Ethernet switch

The Click system is flexible enough to handle applications other than IP routing. For example, Figure 14 shows a functional Click configuration for an IEEE 802.1d-compliant Ethernet switch. It acts as a learn-

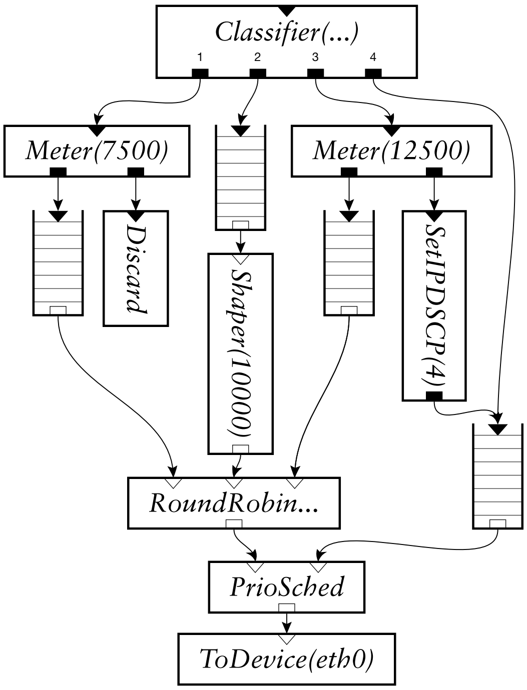

Figure 13: A sample traffic conditioning block. *Meter*s and *Shaper*s measure traffic rates; they are available in varieties that measure bytes per second or packets per second. This example uses packets per second. 1, 2, 3, and 4 represent DSCP values.

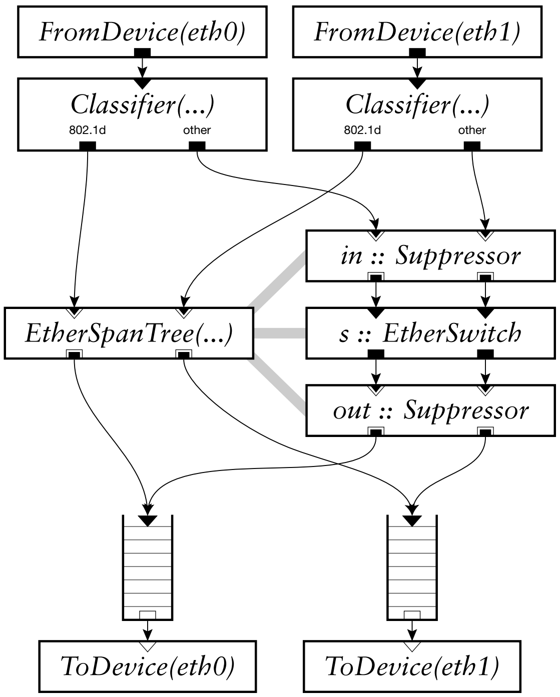

Figure 14: The Ethernet switch configuration.

ing bridge and participates with other 802.1d-compliant bridges to determine a spanning tree for the network, eliminating cycles in the LAN graph. The central element, *EtherSwitch*, can be used alone as a simple, functional learning bridge. The other infrastructure in the figure—*EtherSpanTree* and the two *Suppressor*s—is necessary only to avoid cycles when multiple bridges are used in a LAN.

EtherSpanTree implements the IEEE 802.1d protocol for constructing a LAN-wide spanning tree. At a given switch, forwarding only occurs among the ports that lie on the spanning tree. *EtherSpanTree* controls the learning and forwarding behavior of *EtherSwitch* using two generic *Suppressor* elements. *Suppressor* normally forwards packets from each input to the corresponding output, but it exports methods to suppress and unsuppress individual ports; packets arriving on a suppressed port are dropped. *EtherSpanTree* uses the *Suppressor*s to prevent the *EtherSwitch* from learning from or forwarding to inappropriate ports. The relevant *Suppressor*s cannot be found using flow-based router context, so the user must currently specify the *Suppressor*s by name in *EtherSpanTree*'s configuration string.

4.6 Limitations

A Click user will generally prefer small elements like *DecIPTTL* to large ones like *EtherSpanTree*, since small elements can be rearranged to create arbitrary configurations. However, Click's reliance on packet flow as an organizational principle means that small elements are not appropriate for all problems. Particularly, large elements are required when control or data flow doesn't match the flow of packets: the control flow required to process a protocol like 802.1d is too complex to split into elements.

This also makes it difficult to implement shared objects that don't participate in packet forwarding, such as routing tables. In the configurations shown in this paper, each routing table is encapsulated in a single packet-forwarding element, which is its sole user. We plan to investigate other ways to accommodate shared objects, perhaps by using something like Scout's typed ports [18].

We have not yet fully investigated how to schedule CPU time among competing push and pull paths, a problem that arises whenever multiple devices simultaneously receive or are ready to send packets. Currently, Linux handles much of this scheduling, and the work list described in the next section controls the rest. Eventually all of it should be controlled by a single mechanism.

5 Implementation

This section describes details of the Click implementation, including how Click coexists with a Linux kernel. The implementation consists of about 17,000 non-blank lines of C++ code. The code compiles into about 145,000 bytes of i386 instructions in the form of a loadable Linux kernel module. (Click can also be compiled as a user-level program that communicates with the network using BPF [14].) A simple element's `push` or `pull` function compiles into a few dozen i386 instructions.

5.1 System components

A running Click router contains five important object classes: elements, a router, packets, timers, and a work list.

- **Elements.** The system contains an element object for each element in the current configuration, as well as prototype objects for every kind of primitive element that could be used.

- **Router.** The single router object collects information relevant to a given router configuration, and is mostly used at initialization time. It configures the elements, checks that connections are valid, and puts the router on line. The router breaks the initialization process into stages, making it possible to allow cyclic configurations without enforcing any initialization order on the graph. In the early stages, elements can set object variables, add and remove ports, and change whether they are push or pull. In later stages, they can check their connections and query flow-based router context. Errors can be reported at any stage.

  The most complex part of initialization is dealing with push and pull. The router checks the invariants and assigns agnostic ports their final push-or-pull status in a single step. Agnostic ports cause the problem: global context is necessary to determine what an agnostic port should be, since arbitrary numbers of agnostic elements can be strung together. If the router decides that one of a string of agnostic elements is push, that constraint must propagate through the entire string.

- **Packets.** Click packet data is copy-on-write—when copying a packet, the system copies the packet header but not the data. Annotations are stored in the packet header in a fixed static order; there is currently no way to dynamically add a new kind of annotation. In the Linux kernel, Click packet objects are equivalent to `sk_buffs` (Linux's packet abstraction).

- **Timers.** Some elements use timers to keep track of periodic events. In the Linux kernel, Linux timer queues are used, which on Intel PCs have .01-second resolution.

- **Work list.** A lightweight work list can be used to schedule Click elements for later processing. It is effectively a simple, single-priority CPU scheduler, and is run after every 8th input packet or whenever there are no more input packets. *Queue*s and *Shaper*s currently use the work list to delay packet-upstream notification (Section 2.2). This improves i-cache performance: under high load, 8 packets will be enqueued before the work list is run and pull processing begins.

5.2 Linux kernel environment

The Linux networking code passes control to Click at one of three points: when a packet arrives, when a network interface becomes ready to send another packet, or when a timer expires. Small changes to the kernel were necessary to gain access to packet arrival and interface-ready events. In all cases Linux runs Click code in a bottom-half handler; bottom halves execute functions that are too substantial to run during an interrupt, but are not naturally associated with any user process. Linux ensures that at most one bottom half is active at a time, so element code need not be reentrant. Interrupts ordinarily take precedence over bottom halves, which always take precedence over user processes. This organization follows Linux's own networking code (allowing a fair comparison), but has performance implications detailed in Section 6. We plan to implement a polling architecture for future work.

When a Linux network device receives a packet, the device hardware copies the packet into a Linux packet buffer and interrupts. The Linux device interrupt code appends the buffer to an input queue of packets waiting to be processed, then allocates a buffer for the next packet and wakes up the bottom half. When a Click router is online, this bottom half passes packets from the input queue directly to the appropriate *FromDevice* element, bypassing normal Linux network processing. The *FromDevice* then pushes each packet through the element graph. The push processing typically stops when the packet is enqueued at a *Queue*.

At some point an output hardware device will interrupt to indicate that it can send more packets. The Linux interrupt code wakes up the bottom half, which calls the appropriate *ToDevice* element. The *ToDevice* initiates a `pull` call which makes its way to the *Queue*. The *ToDevice* passes the pulled packet directly to the Linux device driver's output routine, avoiding Linux's output queues.

The Click kernel module uses Linux's /proc filesystem to communicate with user processes. To bring a router online, you create a configuration description in the Click language and write it to /proc/click/config. Reading this file returns the current configuration, and writing subsequent descriptions causes the configuration to change on the fly. When a router is active, a directory is created under /proc/click for each element in its configuration. Elements can easily add read and write access points to their directories; we use this interface to provide access to statistics like packet counts and queue lengths, and to make parameters like maximum queue lengths and RED probabilities reconfigurable at run time.

6 Evaluation

Click's performance goals are to forward packets quickly enough to keep typical access links busy, to impose a low cost for incremental additions to configurations, and to correctly implement complex behaviors like packet scheduling. This section demonstrates that Click meets these goals.

6.1 Experimental setup

The experimental setup consists of three Intel PCs running Linux 2.2.10: a source host, the router being tested, and a destination host. The router has two 100 Mbit Ethernet cards connected, by point-to-point links, to the source and destination hosts. During a test, the source generates an even flow of UDP packets addressed to the destination; the router is expected to get them there.

The router hardware is a 450 MHz Intel Pentium III CPU, an Intel 440BX PCI chip set, 256 megabytes of SDRAM, and two DEC 21140 100 Mbit PCI Ethernet controllers. The Pentium III has a 16 KB L1 instruction cache, a 16 KB L1 data cache, and a 512 KB L2 unified cache. The source host has a 300 MHz Pentium II CPU and a DEC 21140 Ethernet controller. The destination host has a 200 MHz PentiumPro CPU and an Intel EtherExpress 10/100 Ethernet controller. The source-to-router and router-to-destination links are point-to-point full-duplex 100 Mbit Ethernet.

The source host generates UDP packets directly from the kernel to avoid the expense of system calls. It produces packets at specified rates using busy loops, and can generate up to 130,000 64-byte packets per second. The destination host counts and discards the source's UDP packets at interrupt time in the device driver and can receive up to 130,000 64-byte packets per second. The 64 bytes include Ethernet, IP, and UDP headers. When the 64-bit preamble and 96-bit inter-frame gap are added, a 100 Mbit Ethernet link can carry up to 148,800 such packets per second.

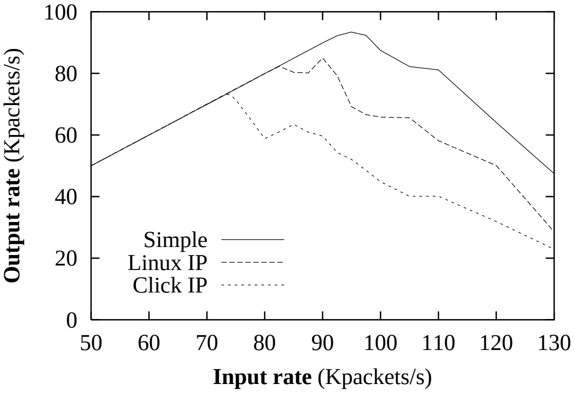

Figure 15: Forwarding rate as a function of input rate for 64-byte packets. An ideal router that forwarded every packet would appear as a straight line $y = x$. The Simple plot is the measured performance of a Click configuration that does no processing other than to emit each input packet. The Linux plot shows the performance of a standard Linux IP router. The Click plot shows the performance of the Click IP configuration in Figure 8.

6.2 Forwarding rates

We characterize performance by measuring the rate at which a router can forward 64-byte packets over a range of input rates. A plot of input and output rates indicates both the maximum loss-free forwarding rate and the router's behavior under overload.

Figure 15 shows the results. An ideal router would emit every input packet regardless of input rate, corresponding to the line $y = x$. The line marked Click shows the performance of the Click IP configuration in Figure 8. Click forwards all packets for input rates up to 73,000 packets per second. Input rates above that exhibit receive livelock [17]: an increasing amount of CPU time is spent in input interrupt processing, leaving less and less time to forward packets. Figure 15 shows that the Linux 2.2.10 IP forwarding system exhibits the same behavior under overload, though Linux is faster than Click. The line marked Simple shows the performance of a Click configuration that forwards input directly to output with no intervening processing.

Figure 16 shows the effect of packet size on forwarding rate. Each point is the maximum over all possible input rates of the router's throughput for packets of the indicated Ethernet frame size. For packet sizes of 250 bytes or larger, both Linux and Click are limited only by the 100 Mbit Ethernet. For smaller sizes the per-packet CPU overhead limits the rate.

An otherwise idle Click IP router forwards 64-byte packets with a one-way latency of 33 microseconds. This number was calculated by measuring the round-trip ping time through the router, subtracting the round-trip ping

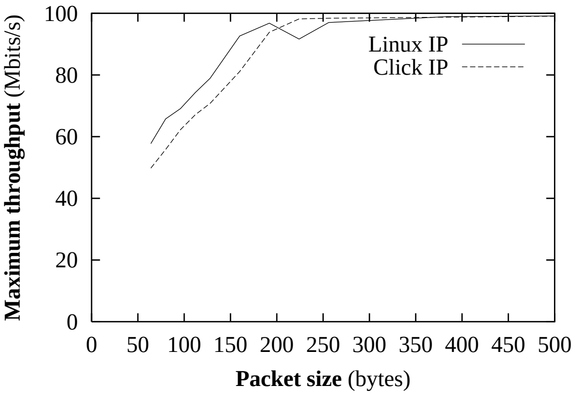

Figure 16: Effect of packet size on maximum forwarding rate.

Phase	**Linux** (µs)	**Click** (µs)
Interrupt	11.1	10.7
IP processing	1.4	2.4
Device send	1.0	1.0
Total	**13.5**	**14.1**

Table 1: Average CPU time cost for basic IP forwarding in microseconds per packet.

time with the router replaced with a wire, and dividing by two. 5.8 µs of the 33 are due to the time required to transmit a 64-byte Ethernet packet at 100 megabits per second. The latency of a router running standard Linux IP code is 28 µs.

The simple test configuration used here shows both Click and Linux in a better light than might be seen in a real network. The tests did not involve fragmentation, IP options, ICMP errors, or multiple destinations, though Figure 8 has all the code needed to handle these. Increasing the number of hosts might slow Click down by increasing the number of ARP table entries. Increasing the number of network interfaces might decrease performance by decreasing the number of packets processed per interrupt. Increasing the routing table size would also decrease performance, a problem existing work on fast lookup in large tables could address [10, 29]. Despite these issues, a simple benchmark is enough to show the performance differences between Linux and Click that are fundamentally due to the Click architecture.

6.2.1 *Detailed forwarding cost*

Table 1 breaks down the cost of forwarding an IP packet into five categories. The costs are the amount of CPU time spent in the relevant code divided by the number of packets processed. The CPU times were obtained with the Pentium cycle counter. The input load was 73,000 64-byte packets per second. Interrupts were turned off for the duration of the Click and Linux IP processing code so that the cycle counts would not include interrupt times.

The 10.7 µs per-packet interrupt cost is a function of the cost of an interrupt and the number of packets processed per interrupt. In this experiment the input Ethernet device delivered an average of 1.5 packets per interrupt. The average interrupt cost 1 µs for the CPU to save and restore its state, 6.7 µs for Linux to coordinate with the interrupt controller chip and to dispatch the interrupt, and 8.3 µs to execute the Ethernet device driver's interrupt handler. The handler moves the 1.5 packets from the receive DMA list to Linux's incoming packet queue, and frees any outgoing packets whose transmission has completed. A polling input architecture [17] might eliminate the CPU and Linux parts of the interrupt cost under high load, reducing the per-packet cost from 10.7 to $8.3/1.5 = 5.53$ µs. The difference in interrupt costs between Linux and Click in Table 1 is an artifact of interrupts being turned off while executing IP forwarding code: Click leaves interrupts off for longer, allowing more packets to accumulate for the next interrupt.

The Linux IP processing line in Table 1 includes performing IP forwarding tasks such as checksum computation and routing table lookup. The Click IP processing line includes the cost of executing the elements in Figure 8, which perform the same tasks. The Device Send line indicates the cost of placing a packet on the device's hardware DMA list.

Table 2 details the cost of each element on the forwarding path in Figure 8, obtained by repeated invocations of that element alone. Every cost but that for *Queue* includes the overhead of moving a packet from one element to the next. This overhead appears to be at least 30 nanoseconds, which indicates that at least 20% of the Click IP processing cost of 2.4 µs is due to architectural overhead rather than IP processing.

The microbenchmark times in Table 2 sum to 1.4 µs, whereas the overall measured time to execute all the Click code is 2.4 µs per packet. Part of the difference is that Table 2 is missing the *FromDevice* and *ToDevice* elements; these are hard to measure in isolation. Another source of difference is that the microbenchmarks never experience instruction cache misses, while the Pentium performance counters reveal that the complete Click router (including device driver code as well as Click elements) spends roughly 2 µs per packet waiting for instruction fetches.

To help separate the costs of IP processing from el-

Element	**Time** (ns)
Paint	38
Classifier	95
Strip	54
CheckIPHeader	299
GetIPAddress	72
LookupIPRoute	66
DropBroadcasts	48
CheckPaint	50
IPGWOptions	59
FixIPSrc	49
DecIPTTL	101
IPFragmenter	62
ARPQuerier	257
Queue	145
Total	**1400**

Table 2: Microbenchmarks of individual elements involved in IP forwarding, measured in nanoseconds per packet.

ement overhead, we wrote single elements that do the work of common groups of IP routing elements, then used the optimizer mentioned in Section 2.5 to replace those groups in Figure 8 with the single combination elements. This new configuration is equivalent to Figure 8, but has only eight elements on the forwarding path instead of 16: it merges *Paint*, *Strip*, *CheckIPHeader*, and *GetIPAddress* into a single input element, and *DropBroadcasts*, *CheckPaint*, *IPGWOptions*, *FixIPSrc*, *DecIPTTL*, and *IPFragmenter* into a single output element. The new configuration processes an IP packet in 1.9 µs instead of 2.4. When we add eight distinct no-op elements to the forwarding path of the new configuration, the packet processing time rises to 2.3 µs. This suggests that most of the reduction from 2.4 to 1.9 is due to fewer inter-element calls and fewer instruction cache misses, and not due to better compiler optimization of the larger elements.

6.3 Cost of incremental complexity

Click makes it easy to create complex and potentially slow configurations. Figure 17 shows the performance of some of the example Click configurations described in this paper, and demonstrates that small increases in complexity incur small performance costs. The line marked IP shows the performance of the basic IP configuration in Figure 8. The line marked IP+RED corresponds to a configuration in which a *RED* element is inserted before each *Queue* in Figure 8. No packets were dropped by RED in the performance test, since the router's output link is as fast as its input. The IP+SFQ line shows the performance of Figure 8 with each *Queue* replaced with a copy of the fair queuing arrangement in Figure 10. The Switch line corresponds to the Ethernet switch configuration of Figure 14, which does much less work than the IP router.

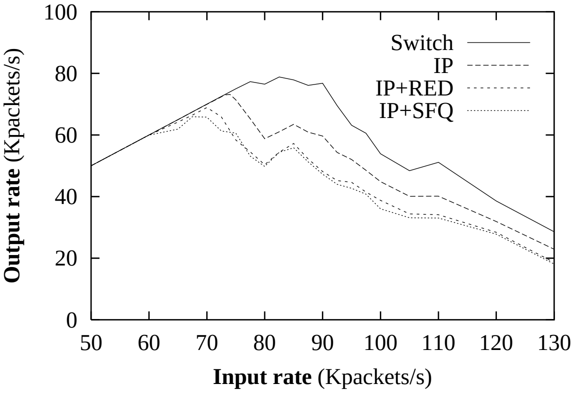

Figure 17: Forwarding rate as a function of input rate for some sample Click configurations.

6.4 Differentiated Services evaluation

We tested the diffserv configuration in Figure 13 by adding it to the IP router (Figure 8) in place of the *Queue*s. The source host generated four streams of data simultaneously, each with a different DSCP corresponding to one path through Figure 13. Figure 18 shows the results. This graph clearly shows the different policing behaviors of the four streams, and also demonstrates the livelock behavior discussed in Section 6.2. As the input rate grows large, Linux takes more and more interrupts to service the receiving interface. Eventually, there is not enough CPU time to handle the incoming packets, and new packets are discarded at the interface itself. Since packets are discarded early—before entering the Click configuration—the *Meter*s see a packet rate much smaller than the true input rate. Thus, at the right edge of the graph, the *Meter*s switch back to their non-overload behavior. Again, this livelock problem could be alleviated with a polling architecture.

6.5 Performance summary

Click performs well despite its modularity. Its 73,000 packet per second IP forwarding rate is 90% as fast as Linux on the same hardware, and faster than that of some low-end commercial routers. For example, Cisco advertises the 2621, a router with about the same cost as our hardware ($2000), as forwarding packets between its two 100 Mbit ports at 25,000 packets per second [6]. Click uses only 16% of the total CPU cycles required to forward a packet, the rest being consumed by device

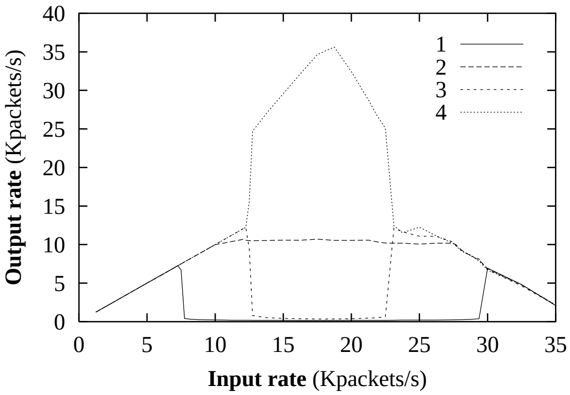

Figure 18: Performance of the diffserv configuration. Each numbered line corresponds to one DSCP; see Figure 13. The x axis corresponds to the input rate for one DSCP, so the aggregate input rate is four times this value. The performance peak is at roughly 72,000 aggregate packets per second. The line for DSCP 4 jumps up at 12,500 packets per second because, at that rate, packets with DSCP 3 are relabeled as DSCP 4.

drivers. Finally, adding a new element to the forwarding path is cheap enough that it should not deter users from taking advantage of Click's flexibility.

7 Related work

Several previous projects have investigated composable network software. These projects concentrated on end nodes, where packet motion is vertical (between the network and user level) rather than horizontal (between interfaces), so they aren't as well suited as Click for routing. None of them have pull processing, explicit queues, or flow-based router context.

The *x*-kernel [12] is a framework for implementing and composing network protocols. Like a Click router, an *x*-kernel configuration is a graph of processing nodes, and packets are passed between nodes through virtual function calls. Unlike Click, an *x*-kernel configuration graph is always acyclic and layered, as *x*-kernel nodes were intended to represent protocols in a protocol stack. This prevents cyclic configurations like the IP router (Figure 8). Connections between nodes are bidirectional—packets travel up the graph to user level and down the graph to the network. Packets pass alternately through "protocol" nodes and "session" nodes, where the session nodes correspond to end-to-end network connections like TCP sessions; session nodes are irrelevant to most routers. The inter-node communication protocols are more complex than Click's. Lastly, many protocol graph changes require recompilation.

Scout [18, 22] is better suited for routing than the *x*-kernel; for example, there are no session objects and cyclic configurations are partially supported. Execution in Scout is centered on *paths*, sequences of nodes that are run from beginning to end. Packets are classified into the correct path as early as possible, so that, for example, Ethernet packets containing MPEG data can be treated differently as soon as they arrive. Each path is executed by a thread. It is interesting to note that Click automatically supports paths without enforcing them: an early *Classifier* element can separate out MPEG-in-TCP-in-IP-in-Ethernet traffic, creating a de facto path. Each Scout path has implicit queues on its inputs and outputs. It is not clear, therefore, how many queues would be involved in a complex configuration like the IP router, which is not amenable to linearization. Scout does have some features Click currently lacks, namely a more interesting scheduler and explicit support for different kinds of inter-node communication (not just packet flow).

The UNIX System V STREAMS system [25] also provides composable packet processing modules. Every STREAMS module includes implicit queuing by default. Each module must be prepared for the next module's queue to fill up, and to respond by queuing or discarding or deferring the processing of incoming packets. Modules with multiple inputs or outputs must also make packet scheduling decisions. STREAMS' tendency to spread scheduling and queuing logic throughout the configuration conflicts with a router's need for precise control over these functions.

The router plugins system [8, 9] is designed for packet forwarding, but is only partially configurable. A router plugin is a software module executed when a classifier matches a particular flow. These classifiers can be installed at any of several *gates*, which are fixed points in the IP forwarding path. Plugins do not allow control over the path itself.

To the best of our knowledge, commercial routers are difficult to extend, either because they use specialized hardware [19, 21] or because their software is proprietary. Even open software is not enough, however. A network administrator could, in principle, implement new routing functions in Linux, but in practice, we expect few administrators have the time or capability to modify an operating system kernel. Kernel programming is harder than extending a Click configuration.

The active networking research program allows anyone to write code that will affect a router [26, 27]. However, this code is intended to teach the router new protocols, not to change core router properties like scheduling or dropping policies. Click allows a trusted user to change any aspect of a router; active networking allows untrusted packets to decide how they should be routed. The two approaches are complementary.

A number of research projects have built routers out of off-the-shelf PC hardware and public-domain software [4, 30]. In many ways this trend towards commodity hardware and software is a return to how routers were constructed 15 years ago [16]. The parts of this work that focused on making commodity routers fast use techniques that could be applied to Click.

8 Conclusion

Click is an open, extensible, and configurable router framework. Our IP router demonstrates that real routers can be built by connecting small, modular elements, and our performance analysis shows that this need not come at unacceptable cost—the Click IP router is just 10% slower than Linux 2.2.10, our base system. Interesting scheduling and dropping policies, complex queueing, and Differential Services can be added to the IP router simply by adding a couple of elements, and Click is flexible enough to support other applications as well. We have made the Click system free software; it is available for download at `http://www.pdos.lcs.mit.edu/click/`.

Acknowledgements

We thank Alex Snoeren for his work on the IPsec elements, Chuck Blake for help with hardware, and Hari Balakrishnan, Daniel Jackson, Greg Minshall, John Wroclawski, Chandu Thekkath (our shepherd), and the anonymous reviewers for their helpful comments.

References

[1] F. Baker, editor. Requirements for IP Version 4 routers. RFC 1812, Internet Engineering Task Force, June 1995. `ftp://ftp.ietf.org/rfc/rfc1812.txt`.

[2] Y. Bernet, A. Smith, and S. Blake. A conceptual model for diffserv routers. Internet draft (work in progress), Internet Engineering Task Force, June 1999. `ftp://ftp.ietf.org/drafts/draft-ietf-diffserv-model-00.txt`.

[3] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, and W. Weiss. An architecture for differentiated services. RFC 2475, Internet Engineering Task Force, December 1998. `ftp://ftp.ietf.org/rfc/rfc2475.txt`.

[4] Kenjiro Cho. A framework for alternate queueing: towards traffic management by PC-UNIX based routers. In *Proc. USENIX 1998 Annual Technical Conference*, pages 247–258, June 1998.

[5] Cisco Corporation. Distributed WRED. Technical report. `http://www.cisco.com/univercd/cc/td/doc/product/software/ios111/cc111/wred.htm`, as of October 1999.

[6] Cisco Corporation. Cisco 2600 series modular access router. Technical report, April 1999. `http://www.cisco.com/warp/public/cc/cisco/mkt/access/2600/prodlit/2600_ds.htm`, as of October 1999.

[7] David Clark. The structuring of systems using upcalls. In *Proc. of the 10th ACM Symposium on Operating Systems Principles (SOSP)*, pages 171–180, December 1985.

[8] Dan Decasper, Zubin Dittia, Guru Parulkar, and Bernhard Plattner. Router plugins: A software architecture for next generation routers. In *Proc. ACM SIGCOMM Conference (SIGCOMM '98)*, pages 229–240, October 1998.

[9] Daniel S. Decasper. *A software architecture for next generation routers*. PhD thesis, Swiss Federal Institute of Technology, Zurich, 1999.

[10] Mikael Degermark, Andrej Brodnik, Svante Carlsson, and Stephen Pink. Small forwarding tables for fast routing lookups. In *Proc. ACM SIGCOMM Conference (SIGCOMM '97)*, pages 3–14, October 1997.

[11] Sally Floyd and Van Jacobson. Random early detection gateways for congestion avoidance. *IEEE/ACM Trans. Networking*, 1(4):397–413, August 1993.

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

[13] T. V. Lakshman, Arnold Neidhardt, and Teunis J. Ott. The drop from front strategy in TCP and in TCP over ATM. In *Proc. IEEE Infocom*, volume 3, pages 1242–1250, March 1996.

[14] Steven McCanne and Van Jacobson. The BSD packet filter: A new architecture for user-level packet capture. In *Proc. Winter 1993 USENIX Conference*, pages 259–269, January 1993.

[15] P. E. McKenney. Stochastic fairness queueing. In *Proc. IEEE Infocom*, volume 2, pages 733–740, June 1990.

[16] David L. Mills. The Fuzzball. In *Proc. ACM SIGCOMM Conference (SIGCOMM '88)*, pages 115–122, August 1988.

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

[18] 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.

[19] Peter Newman, Greg Minshall, and Thomas L. Lyon. IP switching—ATM under IP. *IEEE/ACM Trans. Networking*, 6(2):117–129, April 1998.

[20] K. Nichols, S. Blake, F. Baker, and D. Black. Definition of the Differentiated Services field (DS field) in the IPv4 and IPv6 headers. RFC 2474, Internet Engineering Task Force, December 1998. `ftp://ftp.ietf.org/rfc/rfc2474.txt`.

[21] Craig Partridge et al. A 50-Gb/s IP router. *IEEE/ACM Trans. Networking*, 6(3):237–248, June 1998.

[22] Larry L. Peterson, Scott C. Karlin, and Kai Li. OS support for general-purpose routers. In *Proc. 7th Workshop on Hot Topics in Operating Systems (HotOS-VII)*, pages

38–43. IEEE Computer Society Technical Committee on Operating Systems, March 1999.

[23] J. Postel, editor. Internet Protocol. RFC 791, Internet Engineering Task Force, September 1981. `ftp://ftp.ietf.org/rfc/rfc0791.txt`.

[24] J. Postel. Internet Control Message Protocol. RFC 792, Internet Engineering Task Force, September 1981. `ftp://ftp.ietf.org/rfc/rfc0792.txt`.

[25] D. M. Ritchie. A stream input-output system. *AT&T Bell Laboratories Technical Journal*, 63(8):1897–1910, October 1984.

[26] Jonathan M. Smith, Kenneth L. Calvert, Sandra L. Murphy, Hilarie K. Orman, and Larry L. Peterson. Activating networks: a progress report. *IEEE Computer*, 32(4):32–41, April 1999.

[27] David L. Tennenhouse, Jonathan M. Smith, W. David Sincoskie, David J. Wetherall, and Gary J. Minden. A survey of active network research. *IEEE Communications Magazine*, 35(1):80–86, January 1997.

[28] Kevin Thompson, Gregory J. Miller, and Rick Wilder. Wide-area Internet traffic patterns and characteristics. *IEEE Network*, 11(6):10–23, November/December 1997.

[29] Marcel Waldvogel, George Varghese, Jon Turner, and Bernhard Plattner. Scalable high speed IP routing lookups. In *Proc. ACM SIGCOMM Conference (SIGCOMM '97)*, pages 25–38, October 1997.

[30] John Wroclawski. Fast PC routers. Technical report, MIT LCS Advanced Network Architecture Group, January 1997. `http://mercury.lcs.mit.edu/PC-Routers/pcrouter.html`, as of October 1999.