The Click Modular Router

EDDIE KOHLER, ROBERT MORRIS, BENJIE CHEN, JOHN JANNOTTI,
and M. FRANS KAASHOEK
Laboratory for Computer Science, MIT

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. A router configuration is a directed graph with elements at the vertices; packets flow along the edges of the graph. Several features make individual elements more powerful and complex configurations easier to write, including *pull connections*, which model packet flow driven by transmitting hardware devices, and *flow-based router context*, which helps an element locate other interesting elements.

Click configurations are modular and easy to extend. A standards-compliant Click IP router has sixteen elements on its forwarding path; some of its elements are also useful in Ethernet switches and IP tunneling configurations. Extending the IP router to support dropping policies, fairness among flows, or Differentiated Services simply requires adding a couple elements at the right place. On conventional PC hardware, the Click IP router achieves a maximum loss-free forwarding rate of 333,000 64-byte packets per second, demonstrating that Click's modular and flexible architecture is compatible with good performance.

Categories and Subject Descriptors: C.2.1 [**Computer-Communication Networks**]: Network Architecture and Design—*Packet-switching networks*; C.2.6 [**Computer-Communication Networks**]: Internetworking—*Routers*; D.2.11 [**Software Engineering**]: Software Architectures—*Domain-specific architectures*

General Terms: Design, Management, Performance

Additional Key Words and Phrases: Routers, component systems, software router performance

An article describing a previous version of this system was published in *Operating Systems Review* **34**(5) (*Proceedings of the 17th Symposium on Operating Systems Principles*), pp 217–231, December 1999.
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.
Address: MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139. Email: eddietwo@lcs.mit.edu, rtm@lcs.mit.edu, benjie@lcs.mit.edu, jj@lcs.mit.edu, kaashoek@lcs.mit.edu.

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 and filter packets, and act as firewalls, among other things. Furthermore, fundamental properties like packet dropping policies are still under active research [Floyd and Jacobson 1993; Lakshman et al. 1996; Cisco Corporation 1999], and initiatives like Differentiated Services [Blake et al. 1998] bring the need for flexibility close 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, it is difficult for network administrators and third party software vendors to extend a router with new functions. Extensions require access to software interfaces in the 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 creating routers. Click routers are built from fine-grained components; this supports fine-grained extensions throughout the forwarding path. The components are packet processing modules called *elements*. The basic element interface is narrow, consisting mostly of functions for initialization and packet handoff, but elements can extend it to support other functions (such as reporting queue lengths). To build a router configuration, the user chooses a collection of elements and connects them into a directed graph. The graph's edges, which are called *connections*, represent possible paths for packet handoff. To extend a configuration, the user can write new elements or compose existing elements in new ways, much as UNIX allows one to build complex applications directly or by composing simpler ones using pipes.

Several aspects of the Click architecture were directly inspired by properties of routers. First, packet handoff along a connection may be initiated by either the source end (*push processing*) or the destination end (*pull processing*). This cleanly models most router packet flow patterns, and pull processing makes it possible to write composable packet schedulers. Second, the *flow-based router context* mechanism lets an element automatically locate other elements on which it depends; it is based on the observation that relevant elements are often connected by the flow of packets. These features make individual elements more powerful and configurations easier to write. For example, in Section 4.2, we present an element that can implement four variants of the Random Early Detection dropping policy [Floyd and Jacobson 1993] depending on its context in the router: RED, RED over multiple queues, weighted RED, or drop-from-front RED. This would be difficult or impossible to achieve in previous modular networking systems.

We have implemented Click on general-purpose PC hardware as an extension to the Linux kernel. A modular Click IP router configuration has a maximum loss-free forwarding rate of 333,000 64-byte packets per second on a 700 MHz Pentium III. This is roughly four times the rate for a standard Linux router on the same hardware, mostly due to device handling improvements inspired by previous work [Mogul and Ramakrishnan 1997; Druschel et al. 1994]. It is also slightly more than the rate for a modified Linux router that uses our network device extensions, demonstrating that Click's modularity and flexibility comes at an acceptable cost.

In the rest of this paper, we describe Click's architecture in detail, including the language used to describe configurations (Section 2), then present a standards-compliant Click IP router (Section 3) and some router extensions (Section 4). After summarizing Click's kernel environment (Section 5) and evaluating its performance (Section 6), we describe related work (Section 7) and summarize our conclusions (Section 8).

2. ARCHITECTURE

A Click *element* represents a unit of router processing. An element represents a conceptually simple computation, such as decrementing an IP packet's time-to-live field, rather than a large, complex computation, such as IP routing. A Click router configuration is a directed graph with elements at the vertices. An edge, or *connection*, between two elements represents a possible path for packet transfer. Every action performed by a Click router's software is encapsulated in an element, from device handling and routing table lookups to queueing and counting packets. The user determines what a Click router does by choosing the elements to be used and the connections among them. Inside a running router, each element is a C++ object that may maintain private state. Connections are represented as pointers to element objects, and passing a packet along a connection is implemented as a single virtual function call.

The most important properties of an element are:

—**Element class.** Each element belongs to one element class. This specifies the code that should be executed when the element processes a packet, as well as the element's initialization procedure and data layout.

—**Ports.** An element can have any number of input and output ports. Every connection goes from an output port on one element to an input port on another. Different ports can have different semantics; for example, second output ports are often used to emit erroneous packets.

—**Configuration string.** The optional configuration string contains additional arguments that are passed to the element at router initialization time. Many element classes use these arguments to set per-element state and fine-tune their behavior.

—**Method interfaces.** Each element supports one or more method interfaces. Every element supports the simple packet-handoff interface, but elements can create and export arbitrary additional interfaces; for example, a queue might export an interface that reports its length. Elements communicate at run time through these interfaces, which can contain both methods and data.

Figure 1 shows how we diagram these properties for a sample element, *Tee(2)*. '*Tee*' is the element class; a *Tee* copies each packet received on its single input port, sending one copy to each output port. (The packet data is not copied—Click

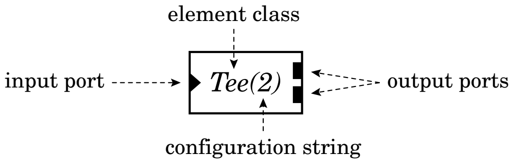

Fig. 1. A sample element. Triangular ports are inputs and rectangular ports are outputs.

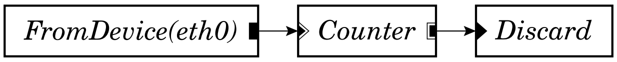

Fig. 2. A router configuration that throws away all packets.

packets are copy-on-write.) Configuration strings are enclosed in parentheses: the '*2*' in '*Tee(2)*' is interpreted by *Tee* as a request for two outputs. Method interfaces are not shown explicitly, as they are implied by the element class. Figure 2 shows several elements connected into a simple router that counts incoming packets, then throws them all away.

2.1 Push and pull connections

Click supports two kinds of connections, *push* and *pull*. On a push connection, packets start at the source element and are passed downstream to the destination element. This corresponds to the way packets move through most software routers. On a pull connection, in contrast, the destination element initiates packet handoff: it asks the source element to return a packet, or a null pointer if no packet is available. This is the dual of a push connection. (Clark called pull connections *upcalls* [Clark 1985].) Each of these forms of packet handoff is implemented by one virtual function call.

The processing type of a connection—whether it is push or pull—is determined by the ports at its endpoints. Each port in a running router is either push or pull; connections between two push ports are push, and connections between two pull ports are pull. Connections between a push port and a pull port are illegal. Elements set their ports' types as the router is initialized. They may also create *agnostic ports*, which behave as push when connected to push ports and pull when connected to pull ports. When a router is initialized, the system propagates constraints until every agnostic port has been assigned to either push or pull.[1] In our configuration diagrams, black ports are push and white ports are pull; agnostic ports are shown as push or pull ports with a double outline. Figure 3 shows how push and pull work in a simple router.

Push processing is appropriate when unsolicited packets arrive at a Click router—for example, when packets arrive from a device. The router must handle such packets as they arrive, if only to queue them for later consideration. Pull processing is appropriate when the Click router needs to control the timing of packet processing. For example, a router may transmit a packet only when

[1]The simplest way of creating an agnostic port causes each packet handoff to that port to take two virtual function calls rather than one. The first calls a general `push` or `pull` method, which is a wrapper that calls the element's "`agnostic`" method.

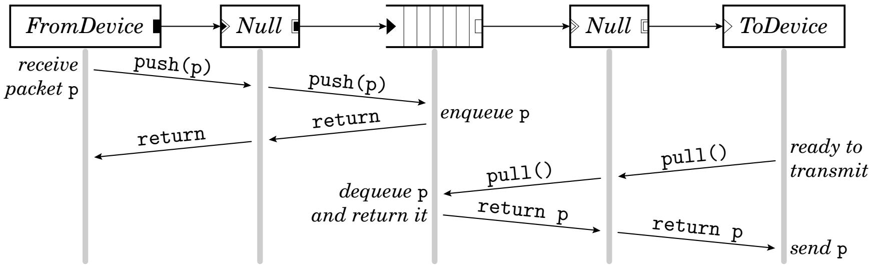

Fig. 3. Push and pull control flow. This diagram shows functions called as a packet moves through a simple router; time moves downwards. The central element is a *Queue*. During the push, control flow moves forward through the element graph starting at the receiving device; during the pull, control flow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

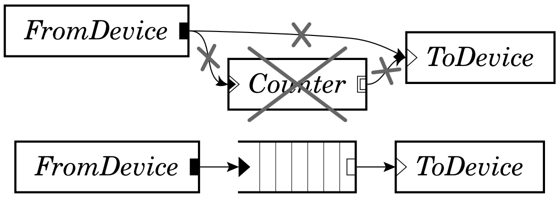

Fig. 4. Some push and pull violations. The top configuration has four errors: (1) *FromDevice*'s push output connects to *ToDevice*'s pull input; (2) more than one connection to *FromDevice*'s push output; (3) more than one connection to *ToDevice*'s pull input; and (4) an agnostic element, *Counter*,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

the transmitting device is ready. In Click, transmitting devices are elements with one pull input; they therefore initiate packet handoff, and can ask for packets only when they are ready.

Pull processing also models the scheduling decision inherent in choosing the next packet to send. A Click packet scheduler is simply an element with one pull output and multiple pull inputs. Such an element responds to a pull request by choosing one of its inputs, making a pull request to that input, and returning the packet it receives. (If it receives a null pointer, it will generally try another input.) These elements make only local decisions: different scheduling behaviors correspond to different algorithms for choosing an input. Thus, they are easily composable. Section 4.1 discusses this further.

The following properties 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 arriving on an agnostic input might be emitted immediately on one of that element's agnostic outputs, then both input and output 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 request—either pushing to an output port or pulling from an input port—is along a unique connection. These properties are automatically checked by the system during router initialization. Figure 4 demonstrates some property violations.

These properties are designed to catch intuitively invalid configurations. For example, the connection in Figure 4 from *FromDevice* to *ToDevice* is illegal because *FromDevice*'s output is push while *ToDevice*'s input is pull. But this connection is intuitively illegal, since it would mean that *ToDevice* might receive packets when it was not ready to send them. The *Queue* element, which converts from push to pull, also provides the temporary packet storage this configuration requires.

2.2 Packet storage

Click elements do not have implicit queues on their input and output ports, or the associated performance and complexity costs. Instead, queues in Click are explicit objects, implemented by a separate *Queue* element. This gives the router designer explicit control over an important router property, namely how packets are stored. It also enables valuable configurations that are difficult to arrange otherwise—for example, a single queue feeding multiple devices, or a queue feeding a traffic shaper on the way to a device. Explicit queues necessitate both push and pull connections. A *Queue* has a push input port and a pull output port; the input port responds to pushed packets by enqueueing them, and the output port responds to pull requests by dequeueing packets and returning them.

2.3 CPU scheduling

Click schedules the router's CPU with a task queue; the router's driver is a loop that processes the task queue one element at a time. (The task queue is currently scheduled with the flexible and lightweight stride scheduling algorithm [Waldspurger and Weihl 1995].) Each task is simply an element that would like special access to CPU time. Thus, the element is Click's unit of CPU scheduling as well as its unit of packet processing. A task can initiate an arbitrary sequence of push and pull requests. However, most elements are never placed on the task queue; they are implicitly scheduled when their `push` or `pull` methods are called. An element should be on the task queue if it frequently initiates push or pull requests without receiving a corresponding request. This includes device elements—for example, *FromDevice* initiates a push request when it receives a packet from a device—as well as elements that ship packets between *Queue*s inside the router configuration.

A related task structure handles timer events. Each element can have any number of active timers, where each timer calls an arbitrary method when it fires.

Click currently runs in a single thread. Thus, any packet handoff method (`push` or `pull`) must return to its caller before another task can begin. The router will continue to process each pushed packet, following it from element to element along a path in the router graph, until it is explicitly stored or dropped (and similarly for pull requests). Therefore, the placement of *Queue*s in a configuration graph determines how CPU scheduling may be performed. For example, if *Queue*s are late in the graph, the router commits to a lot of

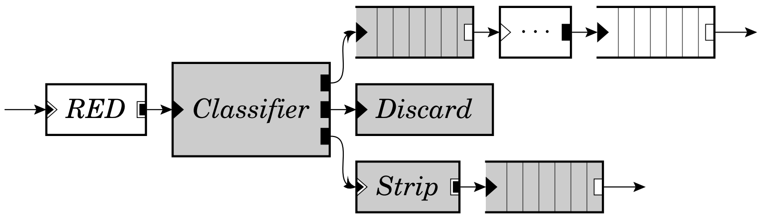

Fig. 5. Flow-based router context. A packet starting at *RED* and stopping at the first *Queue* it encountered might pass through any of the grey elements.

processing on each input packet before processing the next input packet—specifically, it commits to the processing required to push the packet to a *Queue*.

2.4 Flow-based router context

If an element A wants to use a method interface of another element B, it must first locate B. Connections solve this problem for packet handoff, but not for other method interface interactions. Instead, A can refer to B by name (for example, A's configuration string could contain the string "B"), or it can use an automatic mechanism called *flow-based router context*.

Flow-based router context describes where packets starting at a given element might end up after several handoffs, or where packets arriving at that element might have originated. This generalizes connections, which specify where a packet might travel in exactly one handoff. In practical terms, elements may ask the system questions such as "If I were to emit a packet on my second output, where might it go?"; the answer is computed using flow-based router context. The element may restrict this question by method interface; when searching for *Queue*s, it might ask "If I were to emit a packet on my second output, which *Queue*s might it encounter?" It may further restrict the answer to the closest method interfaces: "If I were to emit a packet on my second output, and it stopped at the first *Queue* it encountered, where might it stop?" This occupies a useful middle ground between purely local information (connections) and purely global information (the entire router). It can be more robust than naming elements explicitly, since it automatically adapts to changes in the router configuration. It also captures a fundamental router property: if two elements interact, then packets can usually pass from one to the other.

For example, each *Queue* exports its current length using a method interface. Elements such as *RED* (a dropping policy element described further in Section 4.2) are interested in this information, but before finding a *Queue*'s length, *RED* must find the correct *Queue*. It can do so with flow-based router context. Figure 5 shows the router context downstream of *RED*. Every element in the figure is downstream of *RED*, but only the grey elements are relevant if the search stops at the closest *Queue*s. Thus, *RED*'s flow-based router context search will return the two grey *Queue*s.

Flow-based router context is calculated by a simple data-flow algorithm that is robust in the presence of cycles in the configuration graph. Elements gen-

```
// Declare three elements . . .
src :: FromDevice(eth0);
ctr :: Counter;
sink :: Discard;
// . . . and connect them together
src -> ctr;
ctr -> sink;

// Alternate definition using syntactic sugar
FromDevice(eth0) -> Counter -> Discard;
```

Fig. 6. Two Click-language definitions for the trivial router of Figure 2.

erally ask for flow-based router context once, at router initialization time, and save its results for quick reference as the router runs. Any element that uses flow-based router context must be prepared to handle zero, one, two, or more result elements, possibly by reporting an error if there are too many or too few such results. Section 4.2 demonstrates the flexibility benefits of supporting variable numbers of results.

2.5 Language

Click configurations are written in a simple language with two important constructs: *declarations* create elements, and *connections* say how they should be connected. Its syntax is easy enough to learn from an example; Figure 6 uses it to define a trivial router.

The language contains an abstraction mechanism called *compound elements* that lets users define their own element classes. A compound element is a router configuration fragment that behaves like an element class. A user could define a compound element consisting of a *Queue* followed by a *Shaper*, and call the resulting element class *ShapedQueue*; the rest of the configuration could then use *ShapedQueue* as if it were a primitive Click element class. Compound elements can have multiple ports with arbitrary push/pull characteristics. At initialization time, each use of a compound element is compiled into the corresponding collection of simple elements.

The language is wholly declarative: it specifies what elements to create and how they should be connected, not how to process packets procedurally. This makes configurations easy to manipulate—processing a language file loses no information, and there is a canonical language file for any configuration. Router manipulation tools can take advantage of these properties to optimize router configurations off line or prove simple properties about them.

Language extensions are generally implemented through special element classes rather than new syntactic constructs. For example, the language has no special syntax for specifying CPU scheduling priorities. Instead, the user writes this information in the configuration strings of one or more *ScheduleInfo* elements. These elements have no input or output ports, and packets do not pass through them; they are used only at initialization time, for the data in their configuration strings. This design keeps the language clean and simplifies tool maintenance.

2.6 Installing configurations

There are currently two drivers that can run Click router configurations, a Linux in-kernel driver and a user-level driver that communicates with the network using Berkeley packet filters [McCanne and Jacobson 1993] or a similar packet socket mechanism. The user-level driver is most useful for profiling and debugging, while the in-kernel driver is good for production work. The rest of this paper concentrates on the in-kernel driver.

To install a Click router configuration, the user passes a Click-language file to the kernel driver. The driver then parses the file, checks it for errors, initializes every element, and puts the router on line. It breaks the initialization process into stages, allowing cyclic configurations without forcing any particular initialization order. In the early stages, elements set object variables, add and remove ports, and specify whether those ports are push or pull. In later stages, they query flow-based router context, place themselves on the task queue, and attach to Linux kernel structures.

Installing a new configuration normally destroys any old configuration; for instance, any packets stored in old queues are dropped. This starts the new configuration from a predictable empty state. However, Click supports two techniques for changing a configuration without losing information:

—**Handlers.** Each element can easily install any number of *handlers*, which are access points for user interaction. They appear to the user as files in Linux's `/proc` file system; for example, a `count` handler belonging to an element named `c` would be accessible in the file `/proc/click/c/count`. One of `c`'s methods is called when the user reads or writes this file. This lightweight mechanism is most appropriate for modifications local to an element, such as changing a maximum queue length. A Click routing table element, for example, would likely provide `add_route` and `del_route` handlers as access points for user-level routing protocol implementations. Handlers are also useful for exporting statistics and other element information.

—**Hot swapping.** Some configuration changes, such as adding new elements, are more complex than handlers can support. In these cases, the user can write a new configuration file and install it with a hot-swapping option. This will only install the new configuration if it initializes correctly—if there are any errors, the old configuration will continue routing packets without a break. Also, if the new configuration is correct, it will atomically take the old configuration's state before it is placed on line; for example, any enqueued packets are moved into the new configuration. This happens only with element classes that explicitly support it, and can be prevented by giving the new elements different names than the old ones.

Finally, element class definitions can be dynamically added to and removed from the Click kernel driver; combined with hot swapping, this makes Click an interesting platform for active networking.

2.7 Element implementation

Each Click element class corresponds to a subclass of the C++ class `Element`, which has about 20 virtual functions. `Element` provides reasonable default im-

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

Fig. 7. The complete implementation of a do-nothing element: *Null* passes packets from its single input to its single output unchanged.

plementations for many of these, so most subclasses must override just six of them or less. Only three virtual functions are used during router operation, namely `push`, `pull`, and `run_scheduled` (used by the task scheduler); the others are used for identification, push and pull specification, configuration, initialization, and statistics.

Subclasses of `Element` are easy to write, so we expect that users will easily write new element classes as needed. In fact, the complete implementation of a simple working element class takes less than 10 lines of code; see Figure 7. Most elements define functions for configuration string parsing and initialization in addition to those in Figure 7, and take about 120 lines of code. When linked with 100 common elements, the Linux kernel driver contains roughly 14,000 lines of core and library source code and 19,000 lines of element source code (not counting comments); this compiles to about 341,000 bytes of i386 instructions, most of which are used only at router initialization time. A simple element's `push` or `pull` function compiles into a few dozen i386 instructions.

2.8 Element design and architectural limitations

A Click user will generally prefer fine-grained elements, which have simple specifications, to coarse-grained elements with more complex specifications. For IP routing, for example, a collection of small elements is preferable to a single element, since the collection of small elements supports arbitrary extensions and modifications through configuration graph manipulation. However, small elements are not appropriate for all problems, since Click's main organizational principle is packet flow. Coarse-grained elements are required when control or data flow doesn't match the flow of packets. For example, complex protocol processing often requires a coarse-grained element; a routing protocol like BGP does not naturally break into parts among which packets flow.

A conventional router contains shared structures that don't participate in packet forwarding, such as routing tables, network statistics, and so forth. In Click, these structures are more naturally incorporated into the packet forwarding path. Routing tables, such as the IP routing table and the ARP cache, are encapsulated by elements that make routing decisions, and statistics are localized inside the elements responsible for collecting them. Of course, these elements can export a method interface so other elements can access the structures.

Several other modular networking systems are built around an abstraction that represents individual network flows [Hutchinson and Peterson 1991; Mosberger and Peterson 1996]. These systems automatically create and destroy modules as network flows are encountered. This is a fast, limited form of configuration installation, as each new or deleted flow changes a localized section of the configuration. Hot-swap installation is fast in Click—on a 700 MHz Pentium III, installing a 50-element configuration takes less than a tenth of a second—but not fast enough to support flow creation and deletion. Most of the benefits of a flow-based design can be realized in Click as is; many configurations only require per-flow-class state and CPU scheduling, and elements can cooperate to maintain per-flow private state. Unlike flow-based systems, however, Click cannot schedule the CPU per individual flow.

Flow-based router context is appropriate for many situations where a method interface must be located, but it is not always sufficiently specific. Consider a configuration where bad packets, which occur rarely, are sent to a set of elements for error handling. Upstream elements using flow-based router context will always include the error-handling elements in their search, which might or might not be what the user wanted. Most elements adopt a simple solution: the user can override flow-based router context with a configuration argument when more specificity is required.

The Click language contains no notion of configuration variables. For example, it would be useful to refer to "the Ethernet address of the device handled by that *FromDevice* element" in a configuration string. Currently, the user must copy such an Ethernet address everywhere it is required, which leads to duplication of information. A configuration-variable extension would be simple to add.

3. AN IP ROUTER

This section presents a real Click router configuration: an IP router that forwards unicast packets in nearly full compliance with the standards [Baker 1995; Postel 1981a; Postel 1981b]. Figure 8 shows this configuration for a bridge router with two network interfaces. (The reader may want to refer to Appendix A, a glossary of Click elements used in the IP router and elsewhere in this paper.) The rest of this section describes the IP router in more detail; Section 4 shows how to extend it by changing its scheduling and queueing behavior, and Section 6 evaluates its performance.

IP forwarding tasks that involve only local information fit naturally into the Click framework. For example, *DecIPTTL* decides if a packet's time-to-live field (TTL) has expired. If the TTL is still valid, *DecIPTTL* decrements it, incrementally updates the packet's checksum, and emits the packet on its first output; if the TTL has expired, *DecIPTTL* emits the packet on its second output (usually connected to an element that generates ICMP errors). These actions depend only on the packet's contents, and do not interact with decisions made elsewhere except as expressed in the packet's path through the element graph. Self-contained elements like *DecIPTTL* can be easily composed. For example, one could connect *DecIPTTL*'s "expired" output to a *Discard* to avoid generating ICMP errors, or insert an element that limited the rate at which errors are generated.

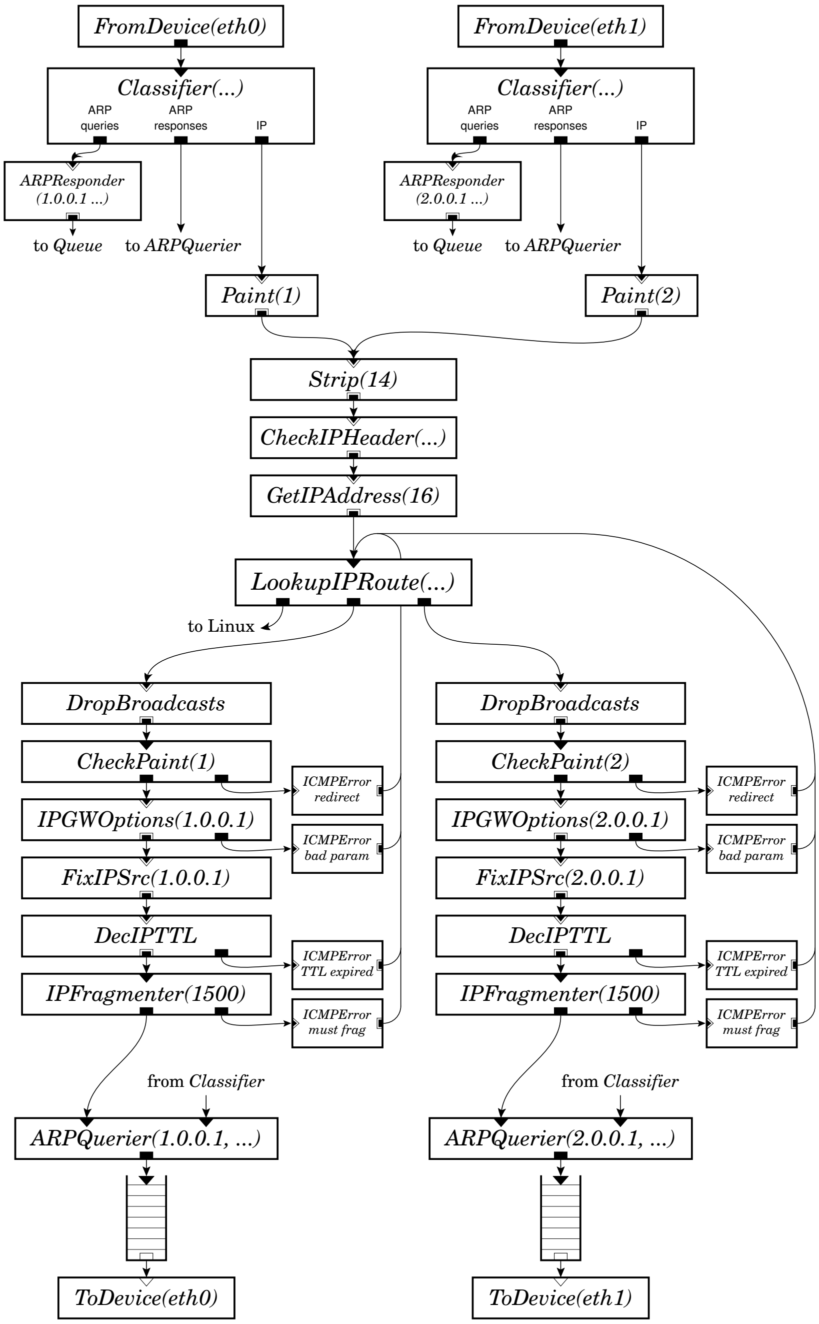

Fig. 8. An IP router configuration.

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 the packet header, but not part of the packet data; the Linux packet structure, `sk_buff`, provides 48 bytes of annotation space. The annotations used in the IP router include:

—**Destination IP address.** Elements that deal with a packet's destination IP address use this annotation rather than the IP header field. This allows elements to modify the destination address—for example, to set it to the next-hop gateway address—without modifying the packet. *GetIPAddress* copies an address from the IP header into 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 each packet with a specified 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 to another interface.

—**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 *FixIPSrc* 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 must be known at multiple points in the Click configuration: *LookupIPRoute* must decide if a packet is destined to the router itself, *CheckIPHeader* must discard a packet with any of the IP broadcast addresses as its source address, *ICMPError* must suppress responses to IP broadcasts, and *IPGWOptions* must 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 (perhaps using flow-based router context).

The IP router conforms to the routing standards both because of the behavior of individual elements and because of their arrangement. In terms of individual elements, the *CheckIPHeader* element checks exactly the properties required by the standards: the IP length fields, the IP source address, and the IP checksum.

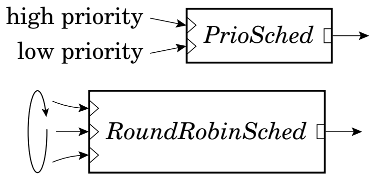

Fig. 9. Some composable packet scheduler elements.

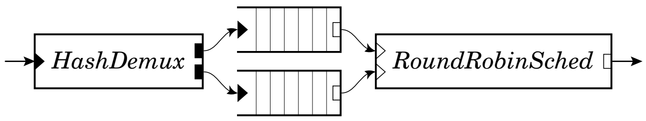

Fig. 10. A virtual queue implementing Stochastic Fairness Queueing.

Also, *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* fragments packets larger than the configured MTU, but sends large, unfragmentable packets to an error output instead. Finally, *ICMPError*, which encapsulates most input packets in an ICMP error message and outputs the result, does not respond to broadcasts, ICMP errors, fragments, and source-routed packets; ICMP errors should not be generated in response to these packets. In terms of element arrangement properties, the standards mandate the placement of *DecIPTTL* after the *LookupIPRoute* routing element; a packet's time-to-live can be decremented only after it is determined that the packet is not destined for the router itself.

4. EXTENSIONS

The Click IP router is modular—it has sixteen elements on its forwarding path—and, therefore, easy to understand and easy to extend. This section demonstrates these benefits of modularity by presenting IP router extensions and other configurations built with the same elements. We show simple extensions supporting scheduling and dropping policies, queueing requirements, and Differentiated Services; an IP tunneling configuration; and one non-IP router, an Ethernet switch. The Click software supports many other extensions not described here, including RFC 2507-compatible IP header compression and decompression, IP security, communication with wireless radios, network address translation, firewalling, and other specialized routing tasks.

4.1 Scheduling

Packet scheduling is a kind of multiplexing: a scheduler decides how a number of packet sources—usually queues—will share a single output channel. A Click packet scheduler is naturally implemented as a pull element with multiple inputs and one output. This element reacts to requests for packets by choosing one of its inputs, pulling a packet from it, and returning that packet. (If the chosen input has no packets ready, the scheduler will usually try other inputs.)

We have implemented three packet scheduler elements: *RoundRobinSched*,

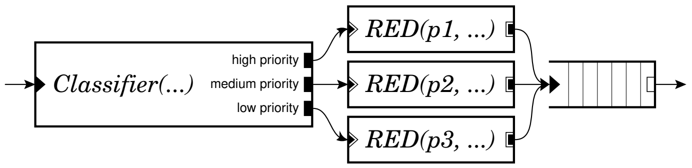

Fig. 11. Weighted RED. The three *RED* elements have different RED parameters, so packets with different priorities are dropped with different probabilities.

PrioSched, and *StrideSched*. *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 with the input cyclically following the last successful pull. *PrioSched* is a strict priority scheduler; it always tries its first input, then its second, and so forth, returning the first packet it gets. *StrideSched* implements a stride scheduling algorithm [Waldspurger and Weihl 1995] with ticket values specified by the user. Other arbitrary packet scheduling algorithms could be implemented in the same way. *RoundRobinSched* and *PrioSched* are illustrated in Figure 9.

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 act like queues but implement more complex behavior than FIFO queueing. Figure 10 shows a virtual queue that implements a version of stochastic fairness queueing [McKenney 1990]: packets are hashed into one of several queues that are scheduled round-robin, providing some isolation between competing flows. The IP router can be extended to use stochastic fairness queueing by replacing its *Queue*s with copies of Figure 10.

4.2 Dropping policies

The *Queue* element implements a simple dropping policy: a configurable maximum length beyond which all incoming packets are dropped. Other policies build on *Queue* rather than replacing it. For example, an independent *RED* element implements random early detection dropping [Floyd and Jacobson 1993]. *RED* contains only drop decision code; this separates the dropping policy from the packet storage policy, allowing either to be independently replaced. It also allows important policy variants through configuration rearrangement.

Random early detection is more likely to drop packets when there is network congestion; a link is considered congested when there are many packets in the queue servicing that link. The *RED* element therefore queries router queue lengths when deciding whether to drop a passing packet. Specifically, it queries the number of packets in the nearest downstream *Storage* elements, which it finds using flow-based router context. (*Storage* is a simple method interface implemented by *Queue* and other elements that store packets.) *RED* can handle one or more downstream *Storage* elements; if there are more than one, it adds their packet counts together to form a single virtual count. This generalization lets it implement variants like RED over multiple queues: a *RED* element placed before the *HashDemux* in Figure 10 would count both of the figure's

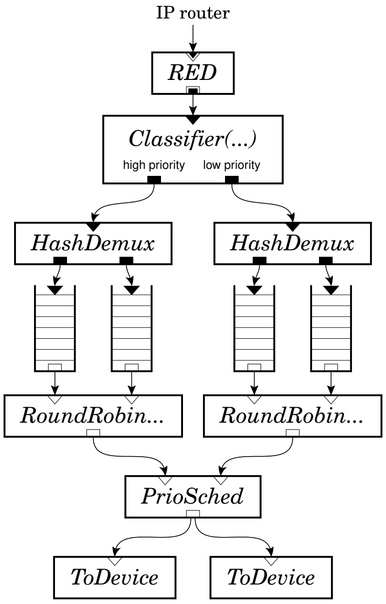

Fig. 12. A complex combination of dropping, queueing, and scheduling. The *Classifier* prioritizes input packets into two virtual queues, each of which implements stochastic fair queueing (Figure 10). *PrioSched* implements priority scheduling on the virtual queues, preferring packets from the left. 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 RED dropping over all four *Queue*s.

*Queue*s. A different arrangement implements weighted RED [Cisco Corporation 1999], where packets are dropped with different probabilities depending on their priority; see Figure 11. Finally, *RED* can be positioned after the *Queue*s instead of before them. In this case, it is a pull element and looks for upstream rather than downstream *Storage* elements, creating a strategy like drop-from-front RED [Lakshman et al. 1996].

Simple RED dropping can be added to the IP router by adding a *RED* element before each *Queue*.

4.3 Complex extensions and simpler subsets

Click is equally well suited for building simple and complex packet processors. A network administrator can easily extend the Click IP router to handle specialized routing tasks, or use a subset of its elements for some other application.

For example, imagine a complex IP 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.

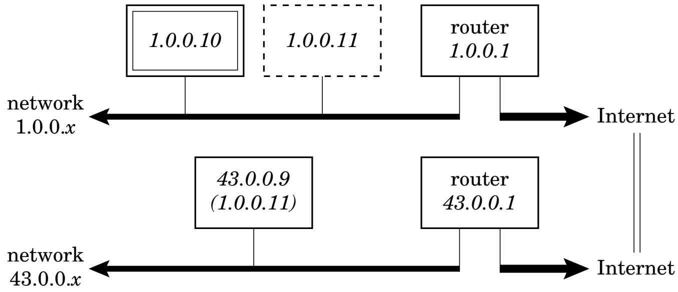

Fig. 13. Network arrangement for a configuration supporting mobility. The mobile host, 43.0.0.9, normally has the IP address 1.0.0.11. Its home node, 1.0.0.10, will encapsulate packets destined for 1.0.0.11 in packets destined for 43.0.0.9.

Click's modular scheduling, queueing and dropping policy elements make this easy; Figure 12 shows the extension.

Now consider Figure 13, where a remote mobile host (43.0.0.9) would like to appear as part of its home network (1.0.0.x) with the help of a home node on that network (1.0.0.10). A simple Click configuration for this task can reuse several of the IP router's elements. Figure 14 shows the configuration active at the home node; it proxy-ARPs for the mobile host's home IP address and performs IP-in-IP encapsulation and unencapsulation to communicate with the mobile host.

4.4 Differentiated Services

The Differentiated Services architecture [Blake et al. 1998] provides 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, queueing, and scheduling functions. We have implemented all of these components as Click elements, which gives router administrators full control over how they are arranged.

For example, Figure 15 implements a realistic diffserv traffic conditioning block. This configuration separates incoming traffic into 4 streams, based on the IP header's Differentiated Services Code Point field (DSCP) [Nichols et al. 1998]. 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 top to bottom in Figure 15, these streams are (A) limited by dropping—the stream is dropped when more than 7,500 packets per second are received on average; (B) shaped—at most 10,000 packets per second are allowed through the *Shaper*, with any excess packets left enqueued; and (C) limited by reclassification—when more than 12,500 packets per second are received, the stream is reclassified as best-effort delivery and sent into the lower priority queue.

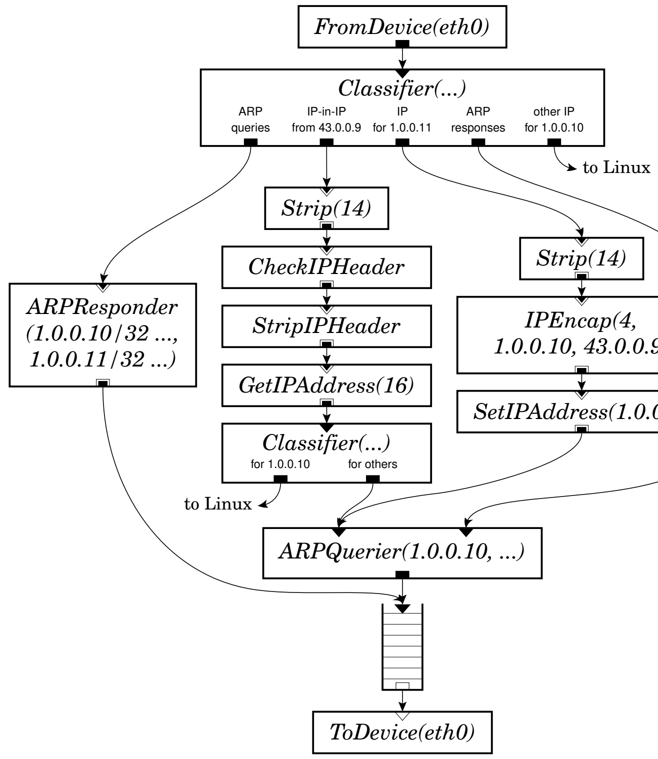

Fig. 14. A configuration for the home node, 1.0.0.10, in the arrangement of Figure 13. The home node proxy-ARPs for the mobile node (*ARPResponder*), unencapsulates packets from the remote node, sending them onto the local network (*CheckIPHeader* path), and performs IP encapsulation for packets destined for the mobile node (*IPEncap* path). Elements not shown ensure that packets generated by 1.0.0.10 itself are properly encapsulated.

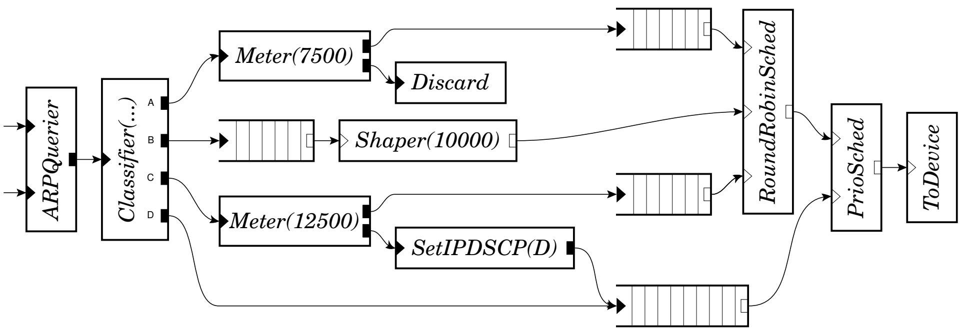

Fig. 15. A sample traffic conditioning block. *Meter*s and *Shaper*s measure traffic rates in packets per second. A, B, C, and D represent DSCP values.

4.5 Ethernet switch

Figure 16 shows a Click configuration unrelated to IP, namely a functional, IEEE 802.1d-compliant Ethernet switch. It acts as a learning bridge and par-

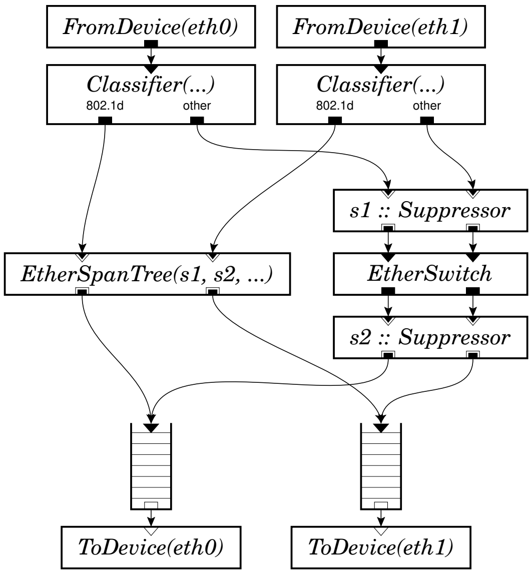

Fig. 16. An Ethernet switch configuration.

ticipates with other 802.1d-compliant bridges to determine a spanning tree for the network.

The *EtherSwitch* element can be used alone as a simple, functional learning bridge. The *EtherSpanTree* and *Suppressor* elements are necessary only to avoid cycles when multiple bridges are used in a LAN. *EtherSpanTree* implements the IEEE 802.1d protocol for constructing a network spanning tree; it works by controlling the *Suppressor* elements. *Suppressor* normally forwards packets from each input to the corresponding output, but also exports a method interface for suppressing and unsuppressing individual ports. Packets arriving on a suppressed port are dropped. *EtherSpanTree* uses this interface to prevent *EtherSwitch* from forwarding packets along edges not on the spanning tree. The *Suppressor*s cannot be found using flow-based router context, so the user must specify the *Suppressor*s by name in *EtherSpanTree*'s configuration string.

5. KERNEL ENVIRONMENT

Click runs as a kernel thread inside a Linux 2.2 kernel. The kernel thread runs the router driver, which loops over the task queue and runs each task. Only interrupts can preempt this thread, but to keep the system responsive, the router driver voluntarily gives up the CPU to Linux from time to time. (The user can configure how often with a *ScheduleLinux* element.) There is generally only one driver thread active at a time. However, during configuration installation, there may be two or more simultaneously active router configurations and driver threads—the new router plus several old ones in the process of dying.

The Click kernel module uses Linux's `/proc` filesystem to communicate with user processes. To bring a router on line, the user creates a configuration description in the Click language and writes it to `/proc/click/config` (or `/proc/click/hotconfig` for a hot-swap installation). Other files in `/proc/click` export information about the currently installed configuration and memory statistics. When installing a router configuration, Click creates a subdirectory under `/proc/click` for each element. This subdirectory contains that element's handlers (Section 2.6). The code that handles accesses to `/proc` runs outside the Click driver thread, in the context of the reading or writing process.

A running Click router contains four important object categories: elements, a router, packets, and timers.

—**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. It configures the elements, checks that connections are valid, and puts the router on line; it also manages the task queue.

—**Packets.** Click packet data is stored in a single block of memory, as opposed to a BSD `mbuf`-like structure. 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.** In the Linux kernel, Click timers are implemented with Linux timer queues, which on Intel PCs have .01-second resolution.

5.1 Polling and device handling

The original Click system [Morris et al. 1999] shared Linux's interrupt structure and device handling; our goal was to change Linux as little as possible. However, interrupt overhead and device handling dominated that system's performance, consuming about 4.8 and 6.7 μs respectively of the total of 13 μs required to forward a packet on a 700 MHz Pentium III PC. In addition, Click processed packets at a lower priority than interrupts, leading to receive livelock [Mogul and Ramakrishnan 1997]: with increasing numbers of input packets, interrupt processing eventually starved all other system tasks, leading to reduced throughput.

Click now eliminates interrupts in favor of polling. Device-handling elements —namely, *FromDevice* and *ToDevice*—place themselves on Click's task queue. When activated, *FromDevice* polls its device's receive DMA queue for newly arrived packets; if any are found, it pushes them through the configuration. *ToDevice* examines its device's transmit DMA queue for empty slots, which it fills by pulling packets from its input. These elements also refill the receive DMA list with empty buffers and remove transmitted buffers from the transmit DMA list. The device never interrupts the processor; furthermore, Linux networking code never executes, except as requested by the Click router configuration. Mogul and Ramakrishnan [1997] also used polling to eliminate receive livelock. However, their system left interrupts enabled under light load, while Click is a pure polling system—even infrequent PC interrupts are simply too expensive.

Polling required changes to Linux's structure representing network devices and the drivers for the network devices we used. The new device structure includes, for example, methods to turn interrupts on and off, to poll for received packets, and to clean the transmit DMA ring.

The introduction of polling let us eliminate all programmed I/O (PIO) interaction with the Ethernet controllers. One PIO reenabled receive interrupts after an interrupt was processed, and was not necessary in a polling system. The second PIO was used in the send routine to tell the controller to look for a new packet to transmit. To eliminate it, we configured the network card to periodically check the transmit DMA queue for new packets. As a result, all communication between Click and the controller takes place indirectly through the shared DMA queues in main memory, in the style of the Orion network controller [Druschel et al. 1994]. This saved roughly two microseconds per packet, as a PIO takes about 1 μs to execute.

These improvements eliminated receive livelock, reduced the total time required to process a packet from 13 μs to 2.8 μs, and increased Click's maximum loss-free forwarding rate of minimum-sized packets more than fourfold. Mogul and Ramakrishnan [1997] found that polling eliminated receive livelock but did not increase peak forwarding rate. We hypothesize that interrupts are relatively much more expensive on PCs than on the Alpha hardware they used.

6. EVALUATION

This section evaluates Click's performance for IP routing and for several extended configurations. We also describe and analyze sources of overhead in Click's design and implementation.

6.1 Experimental setup

The experimental setup consists of a total of nine Intel PCs running Linux—specifically, a version of Linux 2.2.14 modified to add support for Click's polling device drivers. Of the nine PCs, one is the router host, four are source hosts, and four are destination hosts. The router host has eight 100 Mbit/s Ethernet controllers connected, by point-to-point links, to the source and destination hosts. During a test, each source generates an even flow of UDP packets addressed to a corresponding destination; the router is expected to get them there. The tests use version 1.1 of the Click software.

The router host has a 700 MHz Intel Pentium III CPU and an Intel L440GX+ motherboard. Its Ethernet controllers are DEC 21140 Tulip 100 Mbit/s PCI controllers [Digital Equipment Corporation 1998] on multi-port cards, split across the motherboard's two independent PCI buses. The Pentium III has a 16 KB L1 instruction cache, a 16 KB L1 data cache, and a 256 KB L2 unified cache. The source and destination hosts are 733 MHz Pentium III CPUs and 200 MHz Pentium Pro CPUs, respectively. Each has one DEC 21140 Ethernet controller. The source-to-router and router-to-destination links are point-to-point full-duplex 100 Mbit/s Ethernet.

The source hosts generate UDP packets at specified rates, and can generate up to 147,900 64-byte packets per second. The destination hosts count and dis-

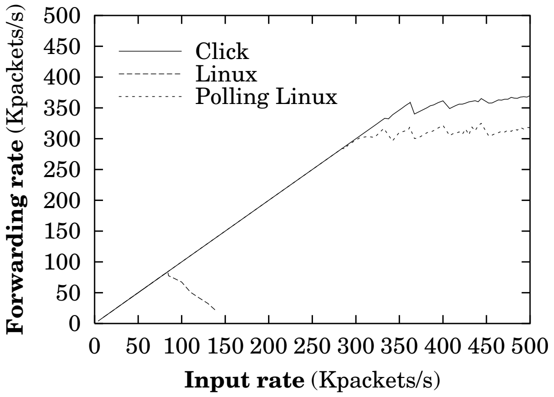

Fig. 17. Forwarding rate as a function of input rate for Click and Linux IP routers (64-byte packets).

card the forwarded UDP packets. Each 64-byte UDP packet includes Ethernet, IP, and UDP headers as well as 14 bytes of data and the 4-byte Ethernet CRC. When the 64-bit preamble and 96-bit inter-frame gap are added, a 100 Mbit/s Ethernet link can carry up to 148,800 such packets per second.

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. Minimum-size packets stress the router harder than larger packets; the CPU and several other bottleneck resources are consumed in proportion to the number of packets forwarded, not in proportion to bandwidth. Plotting forwarding rate versus input rate indicates both the maximum loss-free forwarding rate (MLFFR) and the router's behavior under overload. An ideal router would emit every input packet regardless of input rate, corresponding to the line $y = x$.

Figure 17 compares Click's and Linux's performance for IP routing. The line marked "Click" shows the performance of an eight-interface version of the Click IP router configuration in Figure 8. This configuration has a total of 161 elements (19 elements per interface plus 9 elements shared by all interfaces). The "Linux" and "Polling Linux" lines show the performance of Linux IP routing; "Linux" uses the standard interrupting device drivers, while "Polling Linux" was modified to use Click's polling drivers.

Click's maximum loss-free forwarding rate is 333,000 packets per second; its peak forwarding rate is 360,000 packets per second. When these rates are exceeded, the Ethernet controllers report that they are discarding received packets due to insufficient receive DMA descriptors, which suggests that the bottleneck is the router's CPU, not its PCI bus bandwidth. The sawtooth pattern thereafter might be caused by interactions with deterministic burst patterns in the source hosts' generated packet streams.

Standard Linux's MLFFR is 84,000 packets per second; its output rate declines as the input rate increases due to receive livelock. Polling Linux's MLFFR

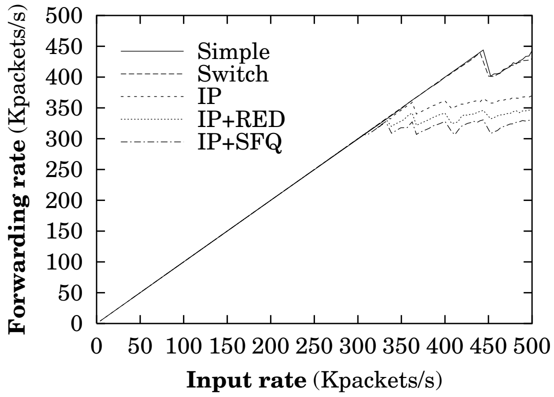

Fig. 18. Forwarding rate as a function of input rate for extended IP router configurations and some non-IP routers (64-byte packets).

of 284,000 packets per second is over three times standard Linux's MLFFR, which demonstrates the advantages of avoiding interrupts and handling device I/O efficiently. Polling Linux is slightly slower than Click, perhaps because Linux's routing table lookup algorithms are slower, but more scalable.

An otherwise idle Click IP router forwards 64-byte packets with a one-way latency of 29 μs. This number was calculated by taking the round-trip ping time through the router (measured with `tcpdump`), subtracting the round-trip ping time with the router replaced by a wire, and dividing by two. 5.8 μs of the 29 are due to the time required to transmit a 64-byte Ethernet packet at 100 megabits per second. 10 μs are due to the costs of polling eight Ethernet controllers, all but two of which are idle in this test. 2.5 μs of delay is the expected amount of time the Tulip controller will take to poll for a new packet to transmit. The latency of a router running standard Linux IP code is 33 μs.

Figure 18 shows the performance of some extended IP router configurations and some non-IP-router configurations. The "Simple" line shows the performance of the simplest possible Click router, which forwards input packets directly from input to output with no intervening processing. The MLFFR for this configuration is 444,000 packets per second. The "Switch" line corresponds to the Ethernet switch configuration of Figure 16; its MLFFR is close to that of "Simple". The "IP" line, repeated from Figure 17, shows the performance of the base IP router configuration. The "IP+RED" line corresponds to an IP router in which a *RED* element is inserted before each *Queue*. No packets were dropped by RED in the test, since the router's output links are as fast as its inputs. The "IP+SFQ" line shows the performance of an IP router with each *Queue* replaced by a stochastic fair queue—namely, a version of Figure 10 with four component *Queue*s. As router complexity increases from simple forwarding through IP routing to IP routing with stochastic fair queueing, Click's performance drops only gradually.

This test configuration shows both Click and Linux in a better light than might be seen in a real network. The tests did not involve fragmentation, IP

Task	**Time** (ns/packet)
Polling packet	528
Refill receive DMA ring	90
Push through Click forwarding path	1565
Pull from Click queue	103
Enqueue packet for transmit	161
Clean transmit DMA ring	351
Total	**2798**

Table 1. Microbenchmarks of tasks involved in the Click IP forwarding path.

options, or ICMP errors, though Figure 8 has all the code needed to handle these. Increasing the number of hosts involved might slow Click down by increasing the number of ARP table entries. Increasing the routing table size would also decrease performance, a problem addressed by previous work on fast lookup in large tables [Degermark et al. 1997; Waldvogel et al. 1997]. The configuration is nevertheless sufficient to analyze the architectural aspects of Click's performance.

6.3 Analysis

Table 1 breaks down the cost of forwarding a packet through a Click router. Costs were measured in nanoseconds by Pentium III cycle counters [Intel Corporation 1996]; each cost is the accumulated cost for all packets in a 10-second run divided by the number of packets forwarded. "Polling packet" is the time *FromDevice* spends taking a packet from the Tulip's receive DMA ring. "Refill receive DMA ring" is the time *FromDevice* spends replacing the DMA descriptor of the packet it just received with a new descriptor, so that the Tulip may receive a new packet. The Click IP forwarding path (see Figure 8) is divided into a push path and a pull path. The push path involves 15 elements; it begins when an input packet is emitted by some *FromDevice* element and ends when the packet reaches the *Queue* element before the appropriate transmitting device. The pull path involves just two elements—when a *ToDevice* element is ready to send, it pulls a packet from a *Queue*. The time consumed by Linux's equivalent of the push path is 1.65 μs, slightly more than Click's 1.57 μs. "Enqueue packet for transmit" is the time *ToDevice* spends enqueueing a packet onto the Tulip's transmit DMA ring. "Clean transmit DMA ring" is the time *ToDevice* spends removing DMA descriptors of transmitted packets. Overall, Click code takes 60% of the time required to process a packet; device code takes the other 40%.

Table 2 breaks down the 1.57 μs push cost by element. Each element's cost is the difference between the Pentium III cycle counter value before and after executing the element, decreased by the time required to read the cycle counter; it includes the virtual function call(s) that move a packet from one element to the next. (*IPFragmenter* is so inexpensive because it does nothing—no packets need to be fragmented in this test—and its packet handoff requires one virtual function call. Other elements that do nothing in this test, such as *IPGWOptions*, have agnostic ports that incur two virtual function calls per packet handoff.) The *FromDevice* and *Queue* elements are not included.

Element	**Time** (ns)
Classifier	70
Paint	77
Strip	67
CheckIPHeader	457
GetIPAddress	120
LookupIPRoute	140
DropBroadcasts	77
CheckPaint	67
IPGWOptions	63
FixIPSrc	63
DecIPTTL	119
IPFragmenter	29
ARPQuerier	106
Subtotal	**1455**

Table 2. Execution times of some of the individual elements involved in IP forwarding, in nanoseconds per packet.

The total cost of 2798 ns measured with performance counters implies a maximum forwarding rate of about 357,000 packets per second, consistent with the observed peak forwarding rate of 360,000 packets per second.

Forwarding a packet through Click incurs five L2 data cache misses (measured using Pentium III performance counters): one to read the receive DMA descriptor, two to read the packet's Ethernet and IP headers, and two to remove the packet from the transmit DMA queue and put it into a pool of reusable packets. Each of these loads takes about 112 nanoseconds. Click runs without incurring any other L2 data or instruction cache misses. 1295 instructions are retired during the forwarding of a packet, implying that significantly more complex Click configurations could be supported without exhausting the Pentium III's 16K L1 instruction cache.

6.4 Overhead of modularity

Click's modularity imposes performance costs in two ways: the overhead of passing packets between elements, and the overhead of unnecessarily general element code.

Passing a packet from one element to the next causes either one or two virtual function calls, where a virtual function call includes loading the relevant function pointer from a virtual function table as well as an indirect jump through that function pointer. Indirect jumps take about 10 nanoseconds if the Pentium III's branch target buffer successfully predicts the target address, but can take dozens of nanoseconds if it fails. Most elements in the IP router use the simplified agnostic-port interface that causes two virtual function calls per packet handoff; the CPU usually predicts one call but doesn't predict the other. As a result, the total cost of packet handoff is about 70 nanoseconds. Thus, adding an empty element to a configuration graph usually adds 70 nanoseconds of overhead, and the IP router, with 16 elements on its forwarding path, has about 1 μs of overhead. This overhead is avoidable—we have built tools that can automatically eliminate all virtual function calls from a Click config-

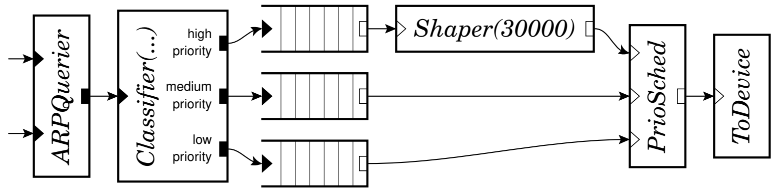

Fig. 19. Another Click diffserv traffic conditioning block. This configuration fragment implements strict priority with three priority levels; in addition, the high-priority stream is rate-limited to 30,000 packets per second.

uration [Kohler 2000].

Several elements in the IP router configuration were implemented with more generality than the IP router requires. The best example is *Classifier*, which the IP router uses to classify Ethernet packets into ARP queries, ARP responses, and IP packets. However, *Classifier* can handle many classification tasks; at run time, it interprets a small data structure (derived from its configuration string) that tells it which packet data to examine. An element specialized for the IP router's classification task could use straight-line code with compiled-in constants, avoiding interpretation overhead. We measured this overhead by writing such a specialized classifier. This element cost 24% less CPU time per packet than the generic *Classifier*, but even the generic *Classifier* takes only 4% of the per-packet CPU time expended inside the Click configuration. Since the rest of the IP router's elements offer less opportunity for specialization, we conclude that element generality has a relatively small effect on Click's performance.

To analyze the combined effects of these overheads, we wrote two single elements that combine much of the processing in Figure 8. These elements reduce the number of virtual function calls on the forwarding path, specialize some of the more general elements, and offer the compiler better opportunities for optimization. One element implements the functionality of *Paint*, *Strip*, *CheckIPHeader*, and *GetIPAddress*; the other, the functionality of *DropBroadcasts*, *CheckPaint*, *IPGWOptions*, *FixIPSrc*, *DecIPTTL*, and *IPFragmenter*. The resulting configuration is equivalent to Figure 8, but has only eight elements on the forwarding path instead of sixteen. The new configuration's push path processes an IP packet in 1.03 μs instead of 1.57 μs. When we add eight distinct do-nothing elements to the forwarding path of this new configuration, restoring the number of elements to sixteen, the packet processing time rises to 1.58 μs. This suggests that the entire reduction from 1.57 μs to 1.03 μs is due to reducing the number of virtual function calls.

6.5 Differentiated Services evaluation

Section 4.4 showed that Click can conveniently model Differentiated Services configurations; this section shows that Click can enforce diffserv policies using only packet scheduling elements like *PrioSched*. The test configuration involves three source hosts sending packets marked as high, medium, and low priority respectively. The router has a single 100 Mbit/s Ethernet output link.

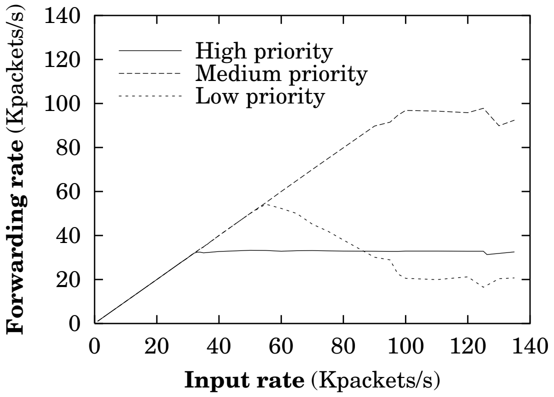

Fig. 20. Performance of the Click diffserv configuration in Figure 19. Three traffic sources send 64-byte UDP packets with three different priorities into a router with a single 100 Mbit/s output link. The sources send equal numbers of high, medium, and low-priority packets; the x axis indicates the input rate for each priority level.

It uses the configuration in Figure 8 with each *Queue* replaced with Figure 19. The configuration queues each priority level separately; it also rate-limits the highest priority traffic to no more than 30,000 packets per second. The *PrioSched* packet scheduling element implicitly schedules CPU time as well as link bandwidth.

Figure 20 shows the results. The per-source sending rate varies along the x axis; the total input load on the router is three times the x-axis value. The y axis indicates how many packets per second the router forwarded from each of the sources. Once the per-source input rate rises above 30,000 packets per second, the router starts dropping high-priority packets, since they are rate-limited. Above 57,000 packets per second, the output link is saturated, and medium priority packets start taking bandwidth away from low. Above 100,000 packets per second, the router runs out of CPU time, forcing the Ethernet controllers to drop packets. Since the controllers don't understand priority, they drop medium-priority packets as well as low. Each medium-priority packet that a controller drops leaves room on the output link for a low-priority packet, which is why the low-priority forwarding rate isn't driven down to zero.

There is probably no way to achieve perfect packet scheduling under overload; instead, a router should be engineered with sufficient processing power to carry all packets to the point where the priority decision is made. The Click user can approximate such a design by moving the classification and per-priority queues of Figure 19 earlier in the configuration—that is, nearer to *FromDevice* than *ToDevice*. This wastes less CPU time on lower priority packets that will eventually be dropped, because they travel through less of the configuration before reaching the drop point. Even with the priority decision late in the configuration, Click enforces priority reasonably well under overload and very well with ordinary loads.

7. RELATED WORK

The x-kernel [Hutchinson and Peterson 1991] is a framework for implementing and composing network protocols. It is intended for use at end nodes, where packet motion is vertical (between the network and user level) rather than horizontal (between network interfaces). Like a Click router, an x-kernel configuration is a graph of processing nodes, and packets are passed between nodes by virtual function calls. However, 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 our IP router. The x-kernel's inter-node communication protocols are more complex than Click's. 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. Bidirectional connections and session nodes are irrelevant to most routers.

Scout [Mosberger and Peterson 1996; Peterson et al. 1999] was designed for use as an operating system at end nodes, but it is better suited for routing than the x-kernel; for example, cyclic configurations are partially supported, and session nodes are not mandatory. Execution in Scout is centered on "paths", sequences of packet processing functions. Each path has implicit queues on its inputs and outputs; it is not clear, therefore, how many queues would appear in a complex configuration like the IP router, which is not amenable to linearization. Each path is run by a single thread and has a CPU priority, and packets are classified into the correct path as early as possible. For example, TCP/IP packets containing MPEG data can be routed immediately to the correct path, whose scheduling behavior might be specialized for MPEG. If desired, Click primitives can support some uses of paths: an early *Classifier* element could send MPEG-in-TCP-in-IP-in-Ethernet traffic into a special *Queue*, and the element that took packets off that *Queue* could be scheduled appropriately for MPEG data. However, Scout supports per-flow paths, where a path object is created for each individual network flow. This allows its CPU scheduler to treat flows independently; Click's CPU scheduler can only treat flow classes independently.

The UNIX System V STREAMS system [Ritchie 1984] is also built from composable packet processing modules. STREAMS modules include implicit queueing by default. Each module must be prepared for the next module's queue to fill up, and to respond by queueing 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 queueing logic throughout the configuration conflicts with a router's need for precise control over these functions.

FreeBSD contains a modular networking system called Netgraph [Elischer and Cobbs 1998] whose nodes strongly resemble Click elements. Netgraph was designed for composing coarse-grained modules such as PPP encapsulation and other framing protocols. Configurations are built up dynamically; there are commands that add and delete nodes, and that connect and disconnect "hooks" (the equivalent of ports), but no external specification for a configuration. This supports on-line configuration modification somewhat more naturally than Click, but there is no way to analyze a Netgraph configuration off line or install a new configuration atomically.

ALTQ [Cho 1998], a system for configurable traffic management, is also shipped with FreeBSD. It contains several sophisticated queueing policies, but its configurability is limited to the specification of one queueing policy per output interface.

The router plugins system [Decasper et al. 1998; Decasper 1999] was designed to make IP routers more extensible. 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". However, gates are fixed points in the IP forwarding path; there is no way to add new gates, or to control the path itself.

To the best of our knowledge, commercial routers are difficult to extend, either because they use specialized hardware [Newman et al. 1998; Partridge et al. 1998] or because their software is proprietary. However, open router software does not necessarily make a router easily extensible. A network administrator could, in principle, implement new routing functions in Linux, but we expect few administrators have the time or ability to modify the monolithic Linux kernel. Click's modularity makes it easy to take advantage of its extensibility.

The active networking research program allows anyone to write code that will affect a router [Tennenhouse et al. 1997; Smith et al. 1999]. 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, and Click may provide a good platform for active network research.

Click's polling implementation and device driver improvements were derived from previous work [Druschel et al. 1994; Mills 1988; Mogul and Ramakrishnan 1997; Wroclawski 1997].

8. CONCLUSION

Click is an open, extensible, and configurable router framework. The Click IP router demonstrates that real routers can be built from small, modular elements, and our performance analysis shows that modularity is compatible with good forwarding performance for PC hardware. IP router extensions for scheduling and dropping policies, complex queueing, and Differentiated Services simply require adding and rearranging a couple elements. Finally, Click is flexible enough to support other packet processing applications. The Click system is free software; it is available for download at `http://www.pdos.lcs.mit.edu/click/`.

ACKNOWLEDGMENTS

Massimiliano Poletto helped us enormously with tests and benchmarks. We also 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, and the anonymous reviewers for their helpful comments.

APPENDIX

A. ELEMENT GLOSSARY

This section lists 38 of the approximately 130 elements included with version 1.1 of the Click software. Each entry follows this format:

ElementName(configuration arguments) Push, pull, or agnostic (specifies port types). Port descriptions (packet types and numbers of ports). Description.

More detailed descriptions are accessible on line at `http://www.pdos.lcs.mit.edu/click/doc/`.

ARPQuerier(...) Push. First input takes IP packets, second input takes ARP responses with Ethernet headers. Output emits ARP queries and IP-in-Ethernet packets. Uses ARP to find the Ethernet address corresponding to each input IP packet's destination IP address annotation; encapsulates the packet in an Ethernet header with that destination Ethernet address.

ARPResponder(ip eth, ...) Agnostic. Input takes ARP queries, output emits ARP responses. Responds to ARP queries for IP address *ip* with the static Ethernet address *eth*.

CheckIPHeader(...) Agnostic. Input takes IP packets. Discards packets with invalid IP length, source address, or checksum fields; forwards valid packets unchanged.

CheckPaint(p) Agnostic. Input takes any packet. Forwards packets with paint annotation p to both outputs; otherwise just to first.

Classifier(...) Push. Input takes any packet. Examines packet data according to a set of classifiers, one classifier per output port. Forwards packet to output port corresponding to the first classifier that matched. Example classifier: "*12/0800*" checks that the packet's data bytes 12 and 13 contains values 8 and 0, respectively. See also *IPClassifier*.

DecIPTTL Agnostic. Input takes IP packets. Decrements input packets' IP time-to-live field. If the packet is still live, incrementally updates IP checksum and sends modified packet to first output; if it has expired, sends unmodified packet to second output.

Discard Push. Discards all input packets.

DropBroadcasts Agnostic. Input takes any packet. Discards packets that arrived as link-level broadcasts; forwards others unchanged.

EtherSpanTree Agnostic. Inputs take IEEE 802.1d messages. Implements the IEEE 802.1d spanning tree algorithm for Ethernet switches.

EtherSwitch Push. Inputs take Ethernet packets; one input and one output per interface. Learning, forwarding Ethernet switch. Learns the interfaces corresponding to Ethernet addresses by examining input packets' source addresses; forwards packets to the correct output port, or broadcasts them if the destination Ethernet address is not yet known.

FixIPSrc(ip) Agnostic. Input takes IP packets. Sets the IP header's source address field to the static IP address *ip* if the packet's Fix IP Source annotation is set; forwards other packets unchanged.

FromDevice(devicename) Push. No inputs. Sends packets to its single output as they arrive from a Linux device driver.

FromLinux(devicename, ip/netmask) Push. No inputs. Installs into Linux a fake Ethernet device *devicename*, and a routing table entry that sends packets for *ip* / *netmask* to that fake device. The result is that packets generated at the router host and destined for *ip* / *netmask* are sent to *FromLinux*'s single output as they arrive from Linux.

GetIPAddress(16) Agnostic. Input takes IP packets. Copies the IP header's destination address field (offset 16 in the IP header) into the destination IP address annotation; forwards packets unchanged.

HashDemux(offset, length) Push. Input takes any packet; arbitrary number of outputs. Forwards packet to one of its outputs, chosen by a hash of the packet's data—specifically, bytes $[\mathit{offset}, \mathit{offset} + \mathit{length})$.

ICMPError(ip, type, code) Agnostic. Input takes IP packets, output emits ICMP error packets. Encapsulates first part of input packet in ICMP error header with source address *ip*, error type *type*, and error code *code*. Sets the Fix IP Source annotation on output packets.

IPClassifier(...) Push. Input takes IP packets. Examines packet data according to a set of classifiers, one classifier per output port. Forwards packet to output port corresponding to the first classifier that matched. Example classifier: "*ip src 1.0.0.1 and dst tcp port www*" checks that the packet's source IP address is 1.0.0.1, its IP protocol is 6 (TCP), and its destination TCP port is 80. See also *Classifier*.

IPEncap(p, ipsrc, ipdst) Agnostic. Input takes any packet, output emits IP packets. Encapsulates input packets in an IP header with protocol *p*, source address *ipsrc*, and destination address *ipdst*.

IPFragmenter(mtu) Push. Input takes IP packets. Fragments IP packets larger than *mtu*; sends fragments, and packets smaller than *mtu*, to first output. Too-large packets with the don't-fragment IP header flag set are sent unmodified to the second output.

IPGWOptions Agnostic. Input takes IP packets. Processes IP Record Route and Timestamp options; packets with invalid options are sent to the second output.

LookupIPRoute(table) Push. Input takes IP packets with valid destination IP address annotations. Arbitrary number of outputs. Looks up input packets' destination annotations in a static routing table specified in the configuration string. Forwards each packet to the output port specified in the resulting routing table entry; sets its destination annotation to the specified gateway address, if any.

Meter(r) Push. Input takes any packet. Sends packets to first output if recent input rate averages $< r$, second output otherwise. Multiple rate arguments are allowed.

Paint(p) Agnostic. Input takes any packet. Sets each input packet's paint annotation to p before forwarding it.

PrioSched Pull. Inputs take any packet; one output port, arbitrary number of input ports. Responds to pull requests by trying inputs in numeric order, returning the first packet it receives. Thus, lower-numbered inputs have priority.

Queue(n) Push input, pull output. Input takes any packet. Stores packets in a FIFO queue; maximum queue capacity is n.

RED(min-thresh, max-thresh, max-p) Agnostic. Input takes any packet. Drops packets probabilistically according to the Random Early Detection algorithm [Floyd and Jacobson 1993] with the given parameters; forwards other packets unchanged. Examines nearby router queue lengths when making its drop decision.

RoundRobinSched Pull. Inputs take any packet; one output port, arbitrary number of input ports. Responds to pull requests by trying inputs in round-robin order; returns the first packet it receives. It first tries the input port just after the one that succeeded on the last pull request.

ScheduleInfo(elementname param, ...) No inputs or outputs. Specifies elements' initial task queue scheduling parameters by element name. Each scheduling parameter is a real number; an element with parameter $2p$ will be scheduled twice as often as an element with parameter p.

ScheduleLinux No inputs or outputs. Places itself on the task queue, and returns to Linux's scheduler every time it is run.

SetIPAddress(ip) Agnostic. Input takes IP packets. Sets each packet's destination IP address annotation to the static IP address *ip*. Forwards modified packets to first output.

SetIPDSCP(c) Agnostic. Input takes IP packets. Sets each packet's IP diffserv code point field [Nichols et al. 1998] to c and incrementally updates the IP header checksum. Forwards modified packets to first output.

Shaper(n) Pull. Input takes any packet. Simple pull traffic shaper: forwards at most n pull requests per second to its input. Pull requests over that rate get a null pointer in response.

Strip(n) Agnostic. Input takes any packet. Strips off each packet's first n bytes; forwards modified packets to first output.

StripIPHeader Agnostic. Input takes IP packets. Strips off each packet's IP header, including any options; forwards modified packets to first output.

Suppressor Agnostic. Inputs take any packet; arbitrary number of input ports, same number of output ports. Normally forwards packets arriving on each input port to the corresponding output port. A method interface allows other elements to ask *Suppressor* to drop packets arriving on particular input ports.

Tee(n) Push. Input takes any packet; n output ports. Forwards each input packet to all n output ports.

ToDevice(device) Pull. Input takes Ethernet packets; no outputs. Hands packets to a Linux device driver for transmission. Activates pull requests only when the device is ready.

ToLinux Push. Input takes Ethernet packets; Linux will ignore the Ethernet header except for the protocol field. No outputs. Hands input packets to Linux's default network input software.

REFERENCES

BAKER, F. 1995. Requirements for IP Version 4 routers. RFC 1812 (June), Internet Engineering Task Force. `ftp://ftp.ietf.org/rfc/rfc1812.txt`.

BLAKE, S., BLACK, D., CARLSON, M., DAVIES, E., WANG, Z., AND WEISS, W. 1998. An architecture for differentiated services. RFC 2475 (Dec.), Internet Engineering Task Force. `ftp://ftp.ietf.org/rfc/rfc2475.txt`.

CHO, K. 1998. A framework for alternate queueing: towards traffic management by PC-UNIX based routers. In *Proc. USENIX 1998 Annual Technical Conference* (June 1998), pp. 247–258.

CISCO CORPORATION. 1999. Distributed WRED. Technical report. `http://www.cisco.com/univercd/cc/td/doc/product/software/ios111/cc111/wred.htm`, as of January 2000.

CLARK, D. 1985. The structuring of systems using upcalls. In *Proc. of the 10th ACM Symposium on Operating Systems Principles (SOSP)* (Dec. 1985), pp. 171–180.

DECASPER, D., DITTIA, Z., PARULKAR, G., AND PLATTNER, B. 1998. Router plugins: A software architecture for next generation routers. In *Proc. ACM SIGCOMM Conference (SIGCOMM '98)* (Oct. 1998), pp. 229–240.

DECASPER, D. S. 1999. *A software architecture for next generation routers*. Ph. D. thesis, Swiss Federal Institute of Technology, Zurich.

DEGERMARK, M., BRODNIK, A., CARLSSON, S., AND PINK, S. 1997. Small forwarding tables for fast routing lookups. In *Proc. ACM SIGCOMM Conference (SIGCOMM '97)* (Oct. 1997), pp. 3–14.

DIGITAL EQUIPMENT CORPORATION. 1998. DIGITAL Semiconductor 21140A PCI Fast Ethernet LAN Controller Hardware Reference Manual. `http://developer.intel.com/design/network/manuals`.

DRUSCHEL, P., PETERSON, L., AND DAVIE, B. 1994. Experiences with a high-speed network adaptor: A software perspective. In *Proc. ACM SIGCOMM Conference (SIGCOMM '94)* (Aug. 1994), pp. 2–13.

ELISCHER, J. AND COBBS, A. 1998. The Netgraph networking system. Technical report (Jan.), Whistle Communications. `http://www.elischer.com/netgraph/`, as of July 2000.

FLOYD, S. AND JACOBSON, V. 1993. Random early detection gateways for congestion avoidance. *IEEE/ACM Trans. Networking 1*, 4 (Aug.), 397–413.

HUTCHINSON, N. C. AND PETERSON, L. L. 1991. The x-kernel: an architecture for implementing network protocols. *IEEE Trans. Software Engineering 17*, 1 (Jan.), 64–76.

INTEL CORPORATION. 1996. Pentium Pro Family Developer's Manual, Volume 3. `http://developer.intel.com/design/pro/manuals`.

KOHLER, E. 2000. *The Click modular router*. Ph. D. thesis, Laboratory for Computer Science, Massachusetts Institute of Technology.

LAKSHMAN, T. V., NEIDHARDT, A., AND OTT, T. J. 1996. The drop from front strategy in TCP and in TCP over ATM. In *Proc. IEEE Infocom*, Volume 3 (March 1996), pp. 1242–1250.

MCCANNE, S. AND JACOBSON, V. 1993. The BSD packet filter: A new architecture for user-level packet capture. In *Proc. Winter 1993 USENIX Conference* (Jan. 1993), pp. 259–269.

MCKENNEY, P. E. 1990. Stochastic fairness queueing. In *Proc. IEEE Infocom*, Volume 2 (June 1990), pp. 733–740.

MILLS, D. L. 1988. The Fuzzball. In *Proc. ACM SIGCOMM Conference (SIGCOMM '88)* (Aug. 1988), pp. 115–122.

MOGUL, J. C. AND RAMAKRISHNAN, K. K. 1997. Eliminating receive livelock in an interrupt-driven kernel. *ACM Trans. Computer Systems 15*, 3 (Aug.), 217–252.

MORRIS, R., KOHLER, E., JANNOTTI, J., AND KAASHOEK, M. F. 1999. The Click modular router. In *Proc. of the 17th ACM Symposium on Operating Systems Principles (SOSP)* (Dec. 1999), pp. 217–231.

MOSBERGER, D. AND PETERSON, L. L. 1996. Making paths explicit in the Scout operating system. In *Proc. 2nd Symposium on Operating Systems Design and Implementation (OSDI '96)* (Oct. 1996), pp. 153–167.

NEWMAN, P., MINSHALL, G., AND LYON, T. L. 1998. IP switching—ATM under IP. *IEEE/ACM Trans. Networking 6*, 2 (April), 117–129.

NICHOLS, K., BLAKE, S., BAKER, F., AND BLACK, D. 1998. Definition of the Differentiated Services field (DS field) in the IPv4 and IPv6 headers. RFC 2474 (Dec.), Internet Engineering Task Force. `ftp://ftp.ietf.org/rfc/rfc2474.txt`.

PARTRIDGE, C., CARVEY, P. P., BURGESS, E., CASTINEYRA, I., CLARKE, T., GRAHAM, L., HATHAWAY, M., HERMAN, P., KING, A., KOHALMI, S., MA, T., MCALLEN, J., MENDEZ, T., MILLIKEN, W. C., PETTYJOHN, R., ROKOSZ, J., SEEGER, J., SOLLINS, M., STORCH, S., TOBER, B., TROXEL, G. D., WAITZMAN, D., AND WINTERBLE, S. 1998. A 50-Gb/s IP router. *IEEE/ACM Trans. Networking 6*, 3 (June), 237–248.

PETERSON, L. L., KARLIN, S. C., AND LI, K. 1999. OS support for general-purpose routers. In *Proc. 7th Workshop on Hot Topics in Operating Systems (HotOS-VII)* (March 1999), pp. 38–43. IEEE Computer Society Technical Committee on Operating Systems.

POSTEL, J. 1981a. Internet Protocol. RFC 791 (Sept.), Internet Engineering Task Force. `ftp://ftp.ietf.org/rfc/rfc0791.txt`.

POSTEL, J. 1981b. Internet Control Message Protocol. RFC 792 (Sept.), Internet Engineering Task Force. `ftp://ftp.ietf.org/rfc/rfc0792.txt`.

RITCHIE, D. M. 1984. A stream input-output system. *AT&T Bell Laboratories Technical Journal 63*, 8 (Oct.), 1897–1910.

SMITH, J. M., CALVERT, K. L., MURPHY, S. L., ORMAN, H. K., AND PETERSON, L. L. 1999. Activating networks: a progress report. *IEEE Computer 32*, 4 (April), 32–41.

TENNENHOUSE, D. L., SMITH, J. M., SINCOSKIE, W., WETHERALL, D. J., AND MINDEN, G. J. 1997. A survey of active network research. *IEEE Communications Magazine 35*, 1 (Jan.), 80–86.

WALDSPURGER, C. A. AND WEIHL, W. E. 1995. Stride scheduling: deterministic proportional-share resource management. Technical Memo MIT/LCS/TM-528 (June), MIT Laboratory for Computer Science.

WALDVOGEL, M., VARGHESE, G., TURNER, J., AND PLATTNER, B. 1997. Scalable high speed IP routing lookups. In *Proc. ACM SIGCOMM Conference (SIGCOMM '97)* (Oct. 1997), pp. 25–38.

WROCLAWSKI, J. 1997. Fast PC routers. Technical report (Jan.), MIT LCS Advanced Network Architecture Group. `http://mercury.lcs.mit.edu/PC-Routers/pcrouter.html`, as of July 2000.