The Click Modular Router

by

Eddie Kohler

S.B., Mathematics with Computer Science (1995); S.B., Music (1995)
S.M., Electrical Engineering and Computer Science (1997)
Massachusetts Institute of Technology

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

Massachusetts Institute of Technology

February 2001

This version includes some corrections relative to the submitted thesis

The Click Modular Router

by

Eddie Kohler

Submitted to the Department of Electrical Engineering and
Computer Science on November 30, 2000 in partial fulfillment of the
requirements for the degree of Doctor of Philosophy

ABSTRACT

Click is a new software architecture for building flexible and configurable routers. A Click router is assembled from packet processing modules called *elements*. Individual elements implement simple router functions like packet classification, queueing, scheduling, and interfacing with network devices. A router configuration is a directed graph with elements at the vertices; packets flow along the edges of the graph. Configurations are written in a declarative language that supports user-defined abstractions. This language is both readable by humans and easily manipulated by tools. We present language tools that optimize router configurations and ensure they satisfy simple invariants.

Due to Click's architecture and language, Click router configurations are modular and easy to extend. A standards-compliant Click IP router has sixteen elements on its forwarding path. We present extensions to this router that support dropping policies, fairness among flows, quality-of-service, and transparent Web proxies. Each extension simply adds a couple elements to the base IP configuration. Other configurations, such as Ethernet switches, firewalls, and traffic generators, reuse many of the IP router's elements. Click software runs in the Linux kernel; on conventional PC hardware, its maximum loss-free forwarding rate for IP routing is 357,000 64-byte packets per second, more than commercial routers with far greater cost. Configuration optimization tools can raise this rate to 446,000 64-byte packets per second, enough to handle several T3 lines and 95% of our hardware's apparent limit.

Thesis Co-Supervisor: M. Frans Kaashoek
Title: Associate Professor

Thesis Co-Supervisor: Robert Morris
Title: Assistant Professor

Contents

Acknowledgements

This thesis describes joint work with Robert Tappan Morris, Benjie Chen, M. Frans Kaashoek, John Jannotti, and Massimiliano Poletto. Robert Morris wrote many of the IP router's elements, designed the IP router configuration, created push and pull processing, and was invaluable throughout. Benjie Chen did most of the polling and device driver work described in Section 7.1 and helped create our evaluation testbed. John Jannotti implemented the Ethernet switch. Massimiliano Poletto implemented the *FromLinux* element and helped with the *IPRewriter* elements. Douglas S. J. De Couto contributed to the Click user-level software and inspired the *ControlSocket* element. Peilei Fan created a set of elements for IPv6, Thomer Gil implemented elements for flexibly measuring rates, and Alex Snoeren wrote some IPsec elements. I am also grateful to other Click users who submitted patches and suggested improvements: Richard Mortier (University of Cambridge Computer Laboratory), Leigh Stoller (University of Utah), Saurabh Sandhir and Prem Gopalan (Purdue University), Brecht Vermeulen (Universiteit Gent), and Joe Elliott. John Wroclawski, David Black, and several anonymous reviewers made helpful comments on papers describing some of the results presented here.

Frans Kaashoek is a great advisor. I thank him inadequately and congratulate him on graduating me despite obstacles.

I am very lucky, and deeply grateful, to have met these people, and thank them, for particular things and for everything.

M. Frans Kaashoek
Robert Morris

Janet Sonenberg

Marilyn, Ted, and Clarke Kohler
Gina D'Acciaro

Ingrid V. Bassett
Mathilda van Es
Anna Rosenberg
Nicolaas and Justin Kaashoek

Paula Mickevich
Maria T. Sensale, Librarian and Notary Public
Kristen Nummerdor
Chrid Hockert

Sean Andrew Murray

Massimiliano Poletto
David Mazières
Rosalba Perna

Anne Dudfield
Celeste Winant
Chelle Gentemann
Chris Onufryk
Marcel Bruchez
Rebecca Leonardson

Elizabeth Stoehr
Federica Pasquotto
Jesse Elliott
Regina Burris

Larisa Mann
Julie Rioux
Kris Grotelueschen

Dawson R. Engler
Debby Wallach
Douglas S. J. De Couto
Jinyang Li
John Jannotti
Benjie Chen

Neena Lyall

Paul Hsiao
Sulaiman Mamdani

Cora and Olaf Stackelberg
Paul Stackelberg

David K. Gifford
Daniel Jackson
Barbara Liskov
Martin Rinard
Dorothy Curtis
Alex Hartemink
Chuck Blake
Jeanne Darling
Rachel Bredemeier
John Guttag
Andrew Myers
Mark Stephenson
Sam Larsen
Mary Ann Ladd

Ellen T. Harris
Anna Frazer
Rebecca Tyler
Elizabeth Connors

Michael Ouellette
Peter Child
Edward Cohen
Mary Cabral
Alan Brody

Jeremy Butler
Julie Park
Tara V. Perry

Jeannie Sun Anand Sarwate
Alan Pierson Erin Lavik
Charles Armesto

Darko Marinov

Bing and Beverly Hollis

Michelle Starz Nick Gaiano
David Montgomery Stacy J. Pruitt Rob Marcato
Katie Leo Monica Gomi Linda Tsang
Lin-Ann Ching Jennifer Tsuei Earle Pratt
Rachael Butcher Ken Michlitsch
Bill Fregosi Ed Darna Leslie Cocuzzo Held
Diane Brainerd Mike Katz Derik Pridmore
Andrea Zengion Adriane Stebbins Vanessa Thomas
Ryan Kershner Marivi Acuña April Griffin
Jose Luis Andrade Rony Kubat Franz Elizondo-Schmelkes
Adam Glassman Jesse Barnes Premraj Janardanan
Sean Austin Kortney Adams Melanie Pincus
Debora Lui Carolyn Chen

Kevin Simmons Aomawa Baker Sean Levin
Peter Shulman Geeta Dayal Marketa Valterova
Fernando Paiz Janet Chieh Fernando Padilla
Ricardo Ramirez Sarah Cohen

Tina Packer Joel Dawson José Luis Elizondo
Joshua Goldberg Tom Cork Max Lord
Albert Fischer Peter Cho Chad Trujillo

Forrest Larson Christie Moore
Geoff Brown Joanne Talbot Nancy Lynch

Marilyn Pierce Monica Bell

David Andersen Helen Cargill David Karger

I

Introduction

This thesis examines how software for packet processors should be designed.

A packet-processing network consists, to first order, of routers and hosts. Hosts use packets as a means to an end; they are mostly concerned with providing communication abstractions to applications. Routers, however, are pure packet processors. They are interested only in packets, which they route from place to place based on packet header information. Routing was the first packet processing application on the Internet, but many others have come to light as the network has matured. *Firewalls* limit access to a protected network, often by dropping inappropriate packets. *Network address translators* allow a large set of machines to share a single public IP address; they work by rewriting packet headers, and occasionally some data. *Load balancers* send packets to one of a set of servers, dynamically choosing a server based on load or some application characteristic. Other packet processors enforce quality-of-service policies, monitor traffic for planning purposes, detect hacker attacks or inappropriate use of resources, and support mobile hosts.

The packet processor abstraction is powerful and flexible because the Internet's network protocols generally maintain transparency. Communicating hosts can act as if they were connected by a wire; they don't care if that wire is actually interrupted by tens of hubs, switches, routers, and other packet processors. Implementing a new network-level application is as simple as dropping another packet processor onto the wire. Often, this requires no changes to other parts of the network.

The essential characteristic shared by all packet processors is the motion of packets. Packets arrive on one network interface, travel through the packet processor's *forwarding path*, and are emitted on another network interface. Contrast this with hosts, where packets lose their identity after they arrive: only data is transferred to the application. In packet processors, packets move horizontally between peers, not vertically between application layers. The

abstractions used to build a packet processor should explicitly support peer-to-peer packet transfer. Use of inappropriate abstractions, such as the inherently layered abstraction of procedure call, can result in unreadable code that is difficult to maintain.

Most software routers, to our knowledge, are built with inappropriate abstractions. All-software routers, such as the routers included with free UNIX operating systems, use procedure call exclusively, and mixed hardware/software routers use similar designs on their software paths. A router's forwarding path, however, is not easily broken into procedural units; forwarding paths behave like pipelines through which packets flow. It is therefore hard to understand or change a procedural router design. Existing research on modular networking systems [22, 30, 42] generally focuses on end node network stacks and uses layered abstractions unsuitable for routers.

As a result, most routers today 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. It is also 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 thesis presents a modular architecture and software toolkit for building routers and other packet processors. It was motivated by the extensibility and flexibility increasingly required of routers, and the wide and growing range of network applications naturally implemented by packet processors. We call this toolkit Click.

DESCRIPTION AND GOALS

Click routers are built from fine-grained software components called *elements*. To build a router configuration, the user chooses a collection of elements and connects them into a directed graph. The graph's edges represent possible paths for packet transfer. This layerless design was motivated by the peer-to-peer nature of packet processing. It also makes packet motion explicit and clear: packets move through the packet processor along the edges of the graph. Each router's forwarding path is implemented by a sequence of elements; this supports fine-grained extensions throughout, since the elements can be rearranged. To implement an extension, 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 applications with pipes.

Click's design began with these principles:

– **One rich, flexible abstraction.** Click's single component abstraction is the element. Router configurations consist of elements connected together; there are no other component-level abstractions.

  The element abstraction therefore significantly influences how Click users think about router design. Click users tend to break packet processing applications into element-sized components; that is, they create modular designs because the element abstraction encourages it. If there were many abstractions designed for specific networking problems, rather than one abstraction designed for packet processing tasks in general, the system would feel less unified, and would be less likely to improve its users' coding habits.

– **Configuration language.** Click router design divides naturally into two phases. In the first phase, users write element classes, which are configuration-independent. In the second, users design a particular router configuration by choosing a set of elements and the connections between them. Element classes are written in C++ using an extensive support library. Router configurations, however, are written in a new programming language—called, unsurprisingly, Click.

  This Click language is wholly declarative. It has features for declaring and connecting elements and for designing abstractions called *compound elements*, and that is all. This contrasts with scripting languages like Tcl, which are essentially general-purpose and, therefore, hard to analyze or manipulate. The Click language enforces hard separation between the roles of elements and configurations, leading to better element design. It makes router configurations human-readable and manipulable by automatic language tools. It has also kept the system as a whole both simple and clear.

– **Avoid restrictions.** The Click system guides users to create modular router and element designs by making modularity easy to achieve, but it does not prevent bad designs or restrict user flexibility. This means, for example, that nothing can be said for certain about an element's semantics without looking at its source code. The hundreds of elements we have created share one mechanism for transferring packets, but nothing prevents new elements from inventing another. The advantage is that possibly useful behavior is never prevented by overly aggressive restrictions.

All these principles serve a single goal: *programmability*. We wanted Click to be easy and fun to program well. We have succeeded. Creating an ele-

ment is more satisfying than writing task-specific network code because of the element's broader potential for reuse. Adding functionality to a router is exciting, especially when it takes just a single element. Click can inspire new applications as well: Section 4.6's description of the *IPRewriter* elements, and Section 5.5's illustration of a transparent Web proxy, led one reader to create a transparent DNS proxy. One year after the software's release, Click is used in at least five other universities. Click's programmability depends on the flexibility of its component architecture, its support library's ease of use, and the range of elements available by default. This thesis attempts to demonstrate programmability by example: Chapters 4 and 5 present many examples of both elements and router configurations.

We claim that, due to the Click architecture, real, standards-compliant Click routers are naturally and easily extensible. We claim that elements facilitate practical component reuse and make programming arbitrary packet processors a pleasant exercise. We claim that the Click programming language makes router configurations easy to write, and that it facilitates user-designed abstractions and debugging. We claim that tools that process Click-language router descriptions can optimize those configurations or ensure that they satisfy systemwide properties. Finally, we claim that real Click routers perform well on conventional PC hardware, and that Click's already low overhead can be mitigated by language-based optimization tools. The rest of this thesis substantiates these claims.

THESIS OVERVIEW

The first chapter of the thesis proper examines Click's architecture: the element abstraction, how packets pass through a router, how the CPU is scheduled, how configurations are installed, and so forth. It pays special attention to novel architectural features supporting programmability—for example, *push and pull processing*, two complementary kinds of packet motion, and *flow-based router context*, which uses possible paths for packet motion to aid dependency checking. Chapter 3 then describes the Click programming language, including its compound element mechanism for user-defined abstraction.

The next chapters examine specific elements and complete router configurations. Chapter 4 presents elements for classification, packet scheduling, dropping policies, and TCP/IP rewriting, among others. Chapter 5 moves from elements to routers, describing a standards-compliant IP router and a wide variety of other configurations. Several sections show how adding a couple elements to the IP router can change its queueing policy to, for example, support quality-of-service, and how its elements are useful in widely differ-

ent configurations, such as firewalls and traffic generators. These chapters exemplify Click's programmability and extensibility.

Chapter 6 describes automatic tools that understand, analyze, and modify router configurations in the Click language. Particular tools optimize router configurations, check them for errors, and ensure they satisfy simple properties. Tools can implement transformations with no analogues in other networking systems; one tool combines several component router configurations into a high-level description of an entire network, facilitating global optimizations and checks. Manipulating router configurations off line adds to Click's power and extensibility. Through tools, optimizations and checks can be added without changing the core of the system.

Chapter 7 describes our implementation of the Click architecture, which runs as software on conventional PCs. No software router on conventional hardware can reach the multigigabit speeds achieved by expensive, custom-hardware routers designed for network backbones. However, many routers, particularly boundary routers at the edges of small to medium-sized organizations, have lower performance requirements that can be achieved on conventional hardware, although with difficulty. Our performance goal was therefore to reach the limits of our hardware. Chapter 8 shows that we have achieved this goal. A Click IP router configuration, after applying language optimization tools, can route a peak of 446,000 minimum-size packets per second in our tests. This is about 95% of the apparent hardware limit, enough to handle multiple T3 lines, and more than twice the maximum forwarding rate of a common edge router, the Cisco 7200 series, that costs about an order of magnitude more than our hardware.[1]

Finally, Chapter 9 describes related work and Chapter 10 concludes.

There are two appendices: an element glossary in Appendix A describes all the elements mentioned in the text; Appendix B contains a BNF grammar for the Click language.

SUMMARY

The contributions of this thesis include:

– The Click architecture, especially flow-based router context and push and pull processing;

1. Of course, the Cisco router presumably has better reliability. It also has more features, although, outside of better routing table support, most of these features are probably not on the forwarding path.

– The Click language and its compound element abstraction;

– Particular elements, including packet schedulers, dropping policies, classifiers, and the *IPRewriter* set of elements for flexible TCP/IP rewriting;

– A standards-compliant IP router written in Click;

– Examples of other configurations and IP router extensions, demonstrating Click's flexibility;

– Tools that optimize routers and check high-level system properties by analyzing and manipulating files in the Click language;

– A thorough analysis of Click's performance on PC hardware;

– and the Click software itself.

This thesis describes release 1.1 of the Click software. This and previous releases are available online for download at `http://www.pdos.lcs.mit.edu/click/`.

2

Architecture

Click is an extensible toolkit for writing packet processors. This toolkit's architecture serves an important function: determining the design space available for its components. We therefore open our discussion of Click by examining its architecture and, in particular, the features it provides for components' use. In later chapters, we turn to examples of individual components and actual Click router configurations.

A quick overview: The Click architecture is centered on the *element*. Each element is a software component representing a unit of router processing. Elements perform conceptually simple computations, such as decrementing an IP packet's time-to-live field, rather than large, complex computations, such as IP routing. They generally examine or modify *packets* in some way; packets, naturally, are the particles of network data that routers exist to process. At run time, elements pass packets to one another over links called *connections*. Each connection represents a possible path for packet transfer. Click *router configurations* are directed graphs of elements with connections as the edges. Router configurations, in turn, run in the context of some *driver*, either at user level or in the Linux kernel.

Figure 2.1 shows some elements connected together into a simple router configuration. Elements appear as boxes; connections appear as arrows connecting the boxes together. Other features of the diagram are described in later sections. Packets pass from element to element along the arrows (connections). This router's elements read packets from the network (*FromDevice(eth0)*), count them (*Counter*), and finally throw them away (*Discard*).

The rest of this chapter examines the Click architecture's components in detail. We start with elements, packets, and connections, then move on to more specific features like push and pull processing. Later sections examine CPU scheduling and packet storage in Click, show how router configurations are installed, and present the C++ implementation of a simple element class.

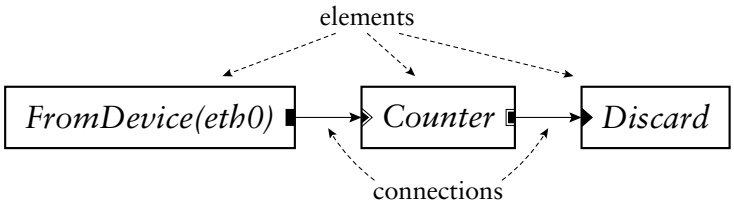

FIGURE 2.1—A simple Click router configuration.

2.1 ELEMENTS

The element is the most important user-visible abstraction in Click. Every property of a router configuration is specified either through the choice of elements or through their arrangement. Device handling, routing table lookups, queueing, counting, and so forth are all implemented by elements. Inside a running router, each element is a C++ object that may maintain private state.

Elements have five important properties: element class, ports, configuration strings, method interfaces, and handlers.

– **Element class.** An element's class specifies that element's data layout and behavior. For example, the code in an element class determines how many ports elements of that class will have, what handlers they will support, and how they will process packets. In C++, each element class corresponds to a subclass of `Element`.

– **Ports.** Each element can have any number of input and output ports. Every connection links an output port on one element to an input port on another.

  Different ports may have different roles; for example, many elements emit normal packets on their first output port and erroneous packets on their second. The number of ports provided by an element may be fixed, or it may depend on the element's configuration string or how many ports were used by the configuration. Every port that is provided must be used by at least one connection, or the configuration is in error. Ports may be push, pull, or agnostic; these terms are defined in Section 2.4.

– **Configuration string.** The optional configuration string contains additional arguments passed to the element at router initialization time. For many element classes, configuration strings define per-element state and fine-tune element behavior, much as constructor arguments do for objects.

  Lexically, a configuration string is a list of arguments separated by commas. Most configuration arguments fit into one of a small set of data types: IP addresses, for example, or integers, or lists of IP addresses.

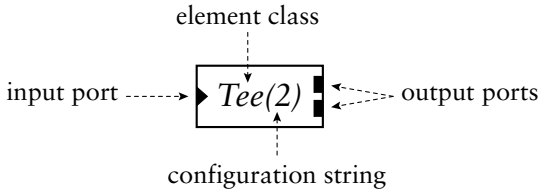

FIGURE 2.2—A sample element.

– **Method interfaces.** Each element exports methods that other elements may access. This set of methods is grouped into method interfaces.

  Every element supports at least the base method interface, which contains, for example, methods for transferring packets. Elements can define and implement arbitrary other interfaces on top of this; for example, Click's *Queue* element, which implements a FIFO packet queue, exports an interface that reports its current length.

– **Handlers.** Handlers are methods that are exported to the user, rather than to other elements in the router configuration. They support simple, text-based read/write semantics, as opposed to fully general method call semantics. For example, the *Queue* element mentioned above has a handler that reports its current length as a decimal ASCII string, and the *Counter* element in Figure 2.1 provides a handler so users can access its current count.

  In the Linux kernel driver, handlers appear as files in the dynamic `/proc` file system. At user level, handlers may be accessed via a TCP/IP-based protocol.

Figure 2.2 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. We draw input ports as triangles and output ports as rectangles. Configuration strings are enclosed in parentheses: the '2' in '*Tee(2)*' is interpreted by *Tee* as a request for two outputs. Method interfaces and handlers are not shown explicitly; they are specified implicitly by the element class.

The element glossary in Appendix A provides a quick description of *Tee* and every other element class mentioned in the text.

2.2 PACKETS

A Click packet consists of a small packet header and the actual packet data; the packet header points to the data. This structure was borrowed from the

Linux kernel's packet abstraction, `sk_buff`. In the Linux kernel driver, Click packet objects are equivalent to `sk_buff`s, which avoids translation or indirection overhead when communicating with device drivers or the kernel itself. This equivalence is hidden behind a C++ class, `Packet`. Other drivers reimplement this interface; the user-level driver, for example, has a hand-written `Packet` implementation. Thus, elements may be compiled for either driver without change. Operations on an in-kernel `Packet` have zero overhead over the corresponding Linux `sk_buff` operations.

Several packet headers may share the same packet data. When copying a packet—for example, with a *Tee* element—Click produces a new packet header that shares the original data. Shared packet data is copy-on-write. Elements that modify packet data first ensure that it is unshared; if it is shared, the element will make a unique copy of the data and change the packet header to point to that copy. Packet headers are never shared, however, so header modifications never cause a copy.

Headers contain a number of *annotations* in addition to a pointer to the packet data. Annotations may be shared with Linux or specific to Click. Some annotations contain information independent of the packet data—for example, the time when the packet arrived. Other annotations cache information about the data. For example, the *CheckIPHeader* element sets the IP header annotation on passing IP packets. This annotation marks both where the IP header begins and where the IP payload begins, freeing later elements from examining the IP header's length field. 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.

Packet data is stored in a single memory buffer. This differs from the BSD-style `mbuf` structure, where data is stored in a linked chain of buffers. Compared to `mbuf`s, Linux and Click's model allows simpler, faster access in the common case, at the cost of occasional packet copies when prepending or appending more data than will fit into a particular buffer. Such copies do not occur in our tests.

2.3 CONNECTIONS

A connection passes from an output port on one element to an input port on another. Connections are the main mechanism used for linking elements together; each connection represents a possible path for packet transfer between elements. In a running router, connections are represented as pointers to element objects, and passing a packet along a connection is implemented by a single virtual function call.

Connections are drawn as arrows; each arrow's direction represents the direction of packet flow. Each connection links a *source port* to a *destination port*. The source port is always an output port, and the destination port is always an input port. We also use *source element* and *destination element* with the obvious definitions.

Router configurations may be seen as directed graphs with elements as vertices. However, connections link ports, not elements, and each element may have many ports. A more complete model treats router configurations as directed graphs with *ports* as vertices. Port graphs such as this have two kinds of directed edges, ordinary connections and *internal edges*. Internal edges show how packets may flow from input ports to output ports within a single element; an internal edge from an element's input port *i* to its output port *o* means that a packet that arrived on input port *i* might be emitted on output port *o*. In the simplest model, every element has a complete graph of internal edges—that is, there are internal edges linking every input to every output. This is not always appropriate, however. For some elements, packets arriving on a given input might be emitted on only a subset of outputs, or perhaps on none at all. More specific internal edge information helps the system decide which elements are reachable from a given port; in the long run, it will also help check properties of configurations. Click elements can therefore specify detailed internal edge information if they choose.

If a path exists from an output port *o* to an input port *i* in the port graph representation of a router configuration, then we say that *i* is *downstream* of *o*, and, conversely, that *o* is *upstream* of *i*. This notion may also be generalized to elements.

2.4 PUSH AND PULL

Click supports two kinds of connections, *push* and *pull*, that implement complementary kinds of packet transfer. On a push connection, packets start at the source element and are passed downstream to the destination element. On a pull connection, in contrast, the destination element initiates packet transfer: it asks the source element to return a packet, or a null pointer if no packet is available. Many software routers provide only push connections; Clark called pull connections *upcalls* [8].

These forms of packet transfer are implemented by two virtual function calls, `push` and `pull`. Neither kind of call is ever intended to block. For instance, a `pull` call reaching an element with no packet ready should immediately return a null pointer rather than waiting for some packet to arrive. Click is currently single-threaded, so a blocking design would simply hang.

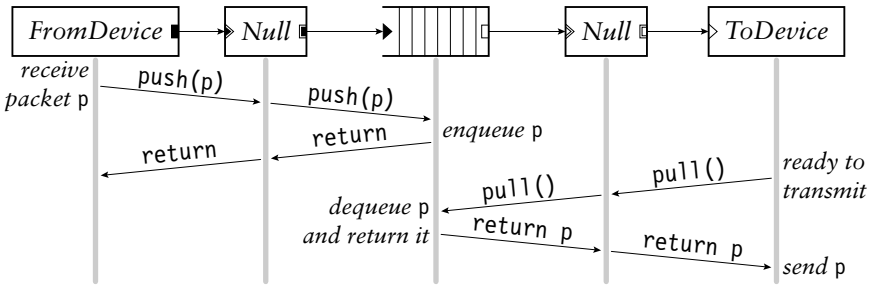

FIGURE 2.3—Push and pull control flow. This diagram shows functions called as a packet moves through a simple router; time moves downwards. During the push, control flow starts at the receiving device and moves forward through the element graph; during the pull, control flow starts at the transmitting device and moves backward. The packet p always moves forward.

The type of a connection 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. 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 2.3 shows how push and pull work in a simple router. This router forwards packets unchanged from one network interface to another. The central element in the figure is a *Queue*. This element enqueues packets on a FIFO queue as they are pushed to its input, and yields packets from the front of that queue as it receives pull requests on its output. The two *Null* elements, which pass packets through unchanged, demonstrate agnostic ports.

Push connections are 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 appear, if only to queue them for later consideration. Pull connections are appropriate when the Click router needs to control the timing of packet processing. For example, a router may transmit a packet only when the transmitting device is ready. In Click, transmitting devices are elements with one pull input. They use pull requests to initiate packet transfer only when ready to transmit. Agnostic ports model the common case that neither kind of processing is inherently required.

Pull connections also model the scheduling decision inherent in choosing the next packet to send. A Click packet scheduler is simply an element with multiple pull inputs and one pull output. It responds to a pull request by

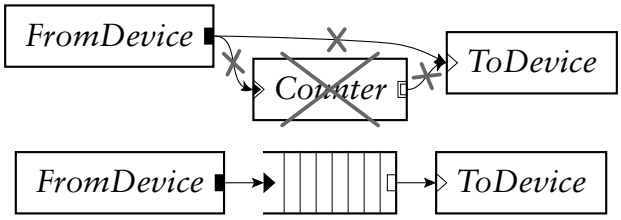

FIGURE 2.4—Push and pull violations. The top configuration has four errors: (1) *FromDevice*'s push output connects to *ToDevice*'s pull input; (2) more than one connection from *FromDevice*'s push output; (3) more than one connection to *ToDevice*'s pull input; and (4) an agnostic element, *Counter*, 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.

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 4.3 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 there is an internal edge linking an agnostic input port to an agnostic output port on the same element, then the ports must be either both push or both pull. Finally, push outputs and pull inputs must be connected exactly once; this ensures that each packet transfer request—either pushing to an output port or pulling from an input port—uses a unique connection. These properties are automatically checked by the system during router initialization. Figure 2.4 demonstrates some violations.

Configurations that violate these properties tend to be intuitively invalid. For example, the connection from *FromDevice* to *ToDevice* in Figure 2.4 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 that it was not ready to send. The *Queue* element, which converts from push to pull, also provides the temporary packet storage this configuration requires.

Some further notes on agnostic ports: Agnostic ports may be connected to other agnostic ports. At initialization time, the system will propagate constraints until every agnostic port has been assigned to either push or pull. To support agnostic ports, an element designer might write both a `push` function and a `pull` function; only the relevant function will be called. Alternatively, the

designer can write a single `simple_action` method. This causes each packet transfer to that port to take two virtual function calls rather than one—the first reaches a generic `push` or `pull`, which, in turn, calls `simple_action`.

2.5 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 implemented by a separate, explicit *Queue* element. This gives the router designer control over an important router property: 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. Each *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. *Queue* never actively passes packets through the graph—it never calls another element's `push` or `pull` function. Instead, and in accord with intuition, it exclusively reacts to requests from other elements. This design would be impossible without support for both push and pull connections: only the combination of push and pull allows *Queue* to be entirely reactive.

The user can choose a different storage policy than simple FIFO queueing simply by using a different element. For example, *Queue* drops packets from its tail when it is full. The *FrontDropQueue* element is mostly equivalent to *Queue* except that, when full, it drops packets from its head to make room for new packets at its tail.

2.6 CPU SCHEDULING

Click schedules the router's CPU with a task queue. Each router thread runs a loop that processes the task queue one element at a time. The task queue is scheduled with the flexible and lightweight stride scheduling algorithm [50]. Tasks are simply elements that would like special access to CPU time. Thus, elements are Click's unit of CPU scheduling as well as its unit of packet processing. An element should be on the task queue if it frequently initiates push or pull requests without receiving a corresponding request. For example, an element that polls a device driver should be placed on the task queue; when run, it would remove packets from the driver and push them into the configuration. However, most elements are never placed on the task queue. They are implicitly scheduled when their `push` or `pull` methods are called. Once an element is scheduled, either explicitly or implicitly, it can initiate an

arbitrary sequence of push and pull requests, thus implicitly scheduling other elements.

Click currently runs in a single thread. Thus, any `push` or `pull` packet transfer method 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. Similarly, the router will continue to process each pull request until a packet is found. The placement of *Queue*s in a configuration graph therefore determines how CPU scheduling may be performed. For example, if *Queue*s are far away from input device elements in the graph, the router must do a lot of work on each input packet before processing the next input packet: specifically, it must push the packet to a *Queue*.

This design can suffer from infinite loops caused by circular configurations. Currently, the user is expected to avoid infinite loops, although we eventually plan to write a language tool that flags potential problems. However, not every circular configuration is problematic. Consider an element that generates error packets in response to all packets that aren't errors themselves. Such an element could be placed safely in a circular configuration since it doesn't respond to packets it generates. This happens in practice with *ICMPError* elements and our IP router; see Section 5.1.

Another task structure handles timer events. Each element can have any number of active timers; each timer calls an arbitrary method when it fires.

2.7 FLOW-BASED ROUTER CONTEXT

If an element *a* wants to call a method on another element *b*, it must first locate *b*. Connections solve this problem for packet transfer, but not for other method interfaces. 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 is simply an application of depth- or breadth-first search to router configuration graphs. A search starting at an element and moving downstream describes where packets starting at a given element might end up. Similarly, a search starting at an element and moving upstream describes where packets arriving at that element might have originated. This generalizes connections, which specify where a packet might travel in exactly one transfer.

To use flow-based router context, elements ask the system questions such as "If I were to emit a packet on my second output, where might it go?" Elements may restrict this question to a subset of the configuration—namely,

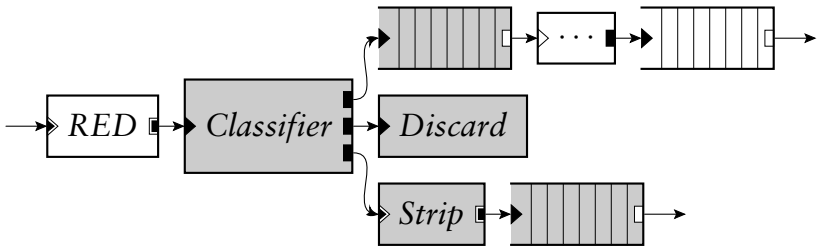

FIGURE 2.5—Flow-based router context. A packet starting at *RED* and stopping at the first *Queue* it encountered might pass through any of the grey elements.

those elements that implement a certain method interface. For example, an element 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.

Dropping policies provide one example of flow-based router context in practice. Each *Queue* exports its current length using a method interface. Elements such as *RED* (a dropping policy element described further in Section 4.4) are interested in this information; *RED* therefore locates one or more relevant *Queue*s using flow-based router context. Figure 2.5 shows the router context downstream of a *RED* element. 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 robust in the presence of cycles in the configuration graph. Elements generally 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 results. Section 4.4 demonstrates the flexibility benefits of handling any number of results.

Flow-based router context can be insufficiently precise for some applications. For example, the user might want the *RED* element in Figure 2.5 to use only one of the grey *Queue*s, or to use a different *Queue* entirely. *RED* and

other elements therefore let the user override flow-based router context with an explicit list of element names.

Click currently uses flow-based router context mostly to check assertions—that a needed element is somewhere downstream, for example. When used this way, there is no risk of finding the wrong elements, and flow-based router context simply improves the system's ability to detect configuration errors.

2.8 INSTALLING CONFIGURATIONS

Click configurations run inside a driver, which implements facilities used by all elements. There are currently two drivers: the user-level driver runs as an application at user level, and the Linux kernel driver runs as a downloadable module in the Linux kernel. The user-level driver is useful for debugging and running repeatable tests; it can even receive packets from the network, using OS mechanisms originally designed for packet sniffers. However, it cannot prevent the operating system's networking stack from handling a packet. The kernel driver can completely replace the OS networking stack, changing a conventional PC into an arbitrary router. This driver can also achieve high performance for PC hardware; see Chapters 7 and 8.

The user installs a Click configuration by writing its definition in a simple textual language and passing that definition to the appropriate driver. (Chapter 3 describes this language in detail.) The driver then parses the definition, checks it for errors, initializes every element, and puts the router on line. It breaks the initialization process into stages. 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, in the kernel driver, attach to Linux kernel structures. This staged design supports cyclic configurations better than any fixed initialization order, which might fall prey to circular dependencies.

To run a configuration in the user-level driver, the user simply runs a *click* application, passing a configuration file's pathname on the command line. Arbitrary numbers of user-level drivers can be active simultaneously.

The Linux kernel driver's installation policy is slightly more complex due to its location in the kernel. This kernel driver runs one router configuration at a time. To install a configuration, the user writes a Click-language description to the special file `/proc/click/config`. 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 several techniques for changing a configuration without losing information:

– **Handlers.** Some elements allow the user to reconfigure them with handlers as the router is running. Handler modifications are generally local to an element. For example, the *Queue* element has a handler that changes its maximum length, and a Click routing table element would likely provide `add_route` and `del_route` handlers as access points for user-level routing protocol implementations.

– **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 being 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 only when the new elements have the same names as the old ones.

– **New element classes.** Element class definitions can be added to and removed from a running Click kernel driver. A Click configuration can require a particular set of elements with a `require` statement; when that configuration is installed, the driver will automatically load an object file containing those elements. This feature could be used to support active networking, where packets can refer to specialized per-packet routing code. In Click, that code could consist of elements that are added to the driver dynamically; a new configuration using those elements could be hot-swapped in.

2.9 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 implementations for many of these, so most subclasses must override 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, handlers, and so forth.

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 2.6.

```
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(); }
};
```

FIGURE 2.6—The complete implementation of a do-nothing element: *Null* passes packets from its single input to its single output unchanged.

Most elements define functions for configuration string parsing and initialization in addition to those in Figure 2.6, 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.

2.10 DISCUSSION

Click users 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. 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 method interfaces so other elements can access the structures.

Several other modular networking systems are built around an abstraction that represents individual network flows—TCP sessions and the like [22, 30]. 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.

3
Language

The Click programming language textually describes Click router configurations. Two of its constructs are sufficient to describe any configuration graph: *declarations* create elements and *connections* connect elements together. However, a language with only these constructs would scale poorly to large configurations. Click's language contains additional structure to aid users: syntactic sugar improves readability, and *compound elements*, described in Section 3.4, provide an abstraction mechanism for configuration fragments.

Two goals guided the language's design, *readability* and *convenience for tools*. The importance of readability is clear enough; a modular networking system is only extensible as far as its configurations may be easily read and modified. The second goal, convenience for tools, means that it should be easy to design, use, and compose automatic tools that analyze and manipulate Click language files. Tools like this can optimize or transform Click configurations or ensure that application-level properties hold; see Chapter 6. We prefer a broad, open-ended set of tools to one tool, no matter how powerful that tool. For example, the Ensemble system for composing network protocol stacks [25] is built around a single tool, a theorem prover. It can perform checks and optimizations, but only with extensive human intervention. In Click, we chose to sacrifice the kind of correctness guarantees achievable with theorem proving for ease of use and ease of tool design.

These goals led us to a set of more concrete design principles.

– The language is *declarative*: it simply describes the configuration graph. Contrast this with scripting as embodied by Tcl, where directives for manipulating a system are added to a general-purpose programming language. The Berkeley *ns* network simulator [34] uses Tcl as a base; *ns* configurations are embedded in, and inseparable from, imperative Tcl scripts that may perform other arbitrary actions. Declarative languages have readability advantages, and declarative programs can be analyzed and manipulated

more easily than imperative programs.

– The language is simple. We have kept new constructs to a minimum, preferring to implement language extensions through special-purpose elements. This choice limits the mechanisms that users must learn and tools must implement while achieving great flexibility.

– Click-language programs and configuration graphs are equivalent. Any configuration corresponds to a simple program in the Click language, and it is easy to translate back and forth between programs and graphs.

– Tools should not need to fully understand Click's semantics—for example, they should not have to understand what every element does. Therefore, the language supports *blind manipulation*: tools that do not understand element semantics can transform configuration programs while preserving correctness.

The rest of this chapter describes the language we have designed based on these principles: its syntax, its abstraction mechanisms, and its limitations.

3.1 SYNTAX

Again, a configuration graph consists of elements connected together. Each element has an element class, which is specified by name, and optionally a configuration string. Elements are connected through their input and output ports. In the Click language, input and output ports are distinguished by number and elements are distinguished by name. Each element in the configuration has a unique name that the user can optionally specify. These names distinguish elements during the parsing process, and also let the user, or other programs, access particular elements once the configuration is running.

The language's fundamental constructs, declarations and connections, have the following syntax:

```
name :: class(config-string);          // declaration
name1 [port1] -> [port2] name2;        // connection
```

(All semicolons are optional.) The connection statement "*name1* [*port1*] -> [*port2*] *name2*" creates a connection from *name1*'s output port *port1* to *name2*'s input port *port2*. Elements must be declared before they are used in connections. Any configuration can be described with these two statements, but additional syntactic sugar makes configurations more pleasant to write and to read:

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

FIGURE 3.1—A Click-language description of the trivial router of Figure 2.1 (page 15).

name `::` *class*`;` // can omit empty configuration string
name1 `->` *name2*`;` // omitted port numbers are equivalent to `[0]`
name1 `[`*port1*`]` `->` `[`*port2a*`]` *name2* `[`*port2b*`]` `->` `[`*port3*`]` *name3*`;`
// can piggyback connections
name1 `->` *name2* `::` *class*`(`*config-string*`)` `->` *name3*`;`
// can declare elements inside connections

Finally, the *anonymous element declaration* syntactic sugar is described in the next section.

An additional construct, the `require` statement, lists configuration requirements. Its syntax is as follows:

`require(`*requirement* `[,` *requirement* ... `]);`

As mentioned in Section 2.8, the Click drivers treat each *requirement* as a dynamically linked object, possibly containing new element definitions, that should be loaded before the rest of the configuration is parsed.

Lexically, C++-style comments (both '/* ... */' and '// ... end of line') are treated as whitespace. Click language parsers also recognize C preprocessor-style line directives ('`#` *line-number* `"`*filename*`"`') and use them to generate reasonable line numbers for error messages. Language manipulation tools include relevant line directives in their output programs.

Figure 3.1 uses these constructs to define a trivial router. For reference, Appendix B provides a full BNF grammar for the language.

3.2 ANONYMOUS ELEMENTS

Anonymous element declarations let the user create elements without specifying their names. Such elements can be used in at most two connections, once as input and once as output. For example:

```
FromDevice(eth0) -> Counter -> Discard;
```

FIGURE 3.2—Another Click-language description of the trivial router of Figure 2.1.

```
name1 -> class(config-string) -> name2;      // one input, one output
name -> class(config-string);                // input only
class(config-string) -> name;                // output only
class(config-string);                        // no inputs or outputs
```

As usual, empty configuration strings may be omitted. The system constructs a name for each anonymous element; these names have the form '*class*`@`*i*', where *i* is an integer chosen to ensure uniqueness. Anonymous element declarations improve readability by avoiding clutter. A language without anonymous declarations could be compared, albeit with some exaggeration, to a programming language in which each subexpression had to be uniquely named. Typical Click configurations use anonymous declarations for half their elements. The Click-language description in Figure 3.2 is equivalent to Figure 3.1, but uses anonymous elements.

Besides making configurations more readable, anonymous elements provide an excellent means for implementing language extensions. For example, how might the user set the scheduling priority for an element, or define an abbreviation for a common network address? The obvious solution would involve extended syntax:

```
address eth0_addr = 00:e0:98:09:ab:af;
arpq :: ARPQuerier(18.26.4.44, eth0_addr);
arpq -> td :: ToDevice(eth0) priority 10;
```

However, this adds unnecessary complexity to the language and does not generalize well. Instead, Click implements extensions through special *information elements*—ordinary elements whose configuration strings define some extension property. For example:

```
AddressInfo(eth0_addr 00:e0:98:09:ab:af);
arpq :: ARPQuerier(18.26.4.44, eth0_addr);
arpq -> td :: ToDevice(eth0);
ScheduleInfo(td 10);
```

The `AddressInfo` and `ScheduleInfo` statements are simply anonymous element declarations. Information elements are described further in Section 4.7.

The generated names of anonymous elements are valid identifiers in the language. Thus, a tool can parse a program that uses anonymous elements and

generate an equivalent program in which all elements are named explicitly. Users are discouraged from using the distinguishing '@' character in their element names for stylistic reasons, but nothing prevents them from doing so. This property helps achieve equivalence between programs and graphs.

Correct programs containing anonymous elements can be parsed without knowing the available set of element classes. This helps support blind manipulation. The rule is simple—every identifier not previously declared is an element class. Parsing a program using this rule preserves correctness: a program generated from the parsed configuration graph will be correct if and only if the original program was correct. Thus, although tools using this rule will silently accept some kinds of errors, they will not change erroneous configurations into valid ones or vice versa.

3.3 CONFIGURATION STRINGS

Configuration strings in Click are comma-separated lists of arguments delimited by parentheses. Each individual argument is essentially a string that is passed to the element for parsing. Before parsing, however, the Click driver replaces any comments with single space characters and removes any leading and trailing space from each argument. Arguments can contain single- or double-quoted strings; these are useful if an argument contains unbalanced parentheses, leading or trailing space, or characters that might be mistaken for comments.

Strictly speaking, what elements do with the arguments is not part of the Click language. Elements are free to parse argument strings in whatever way they choose. Practically speaking, however, the configuration string "language" is important for users, who must learn it just as they learn the language for connecting elements together.

Most elements we have designed so far use a common library for parsing their configuration arguments. This library parses arguments with the following characteristics:

- A configuration argument is either a single item or a list of items. Items in a list are separated by spaces.

- An individual item can be a string ('`"this is"a_string`'), word ('`a_word`'), Boolean value ('`true`', '`false`'), integer ('`90`', '`-999`', '`0x8F`'), real number ('`0.5`'; the data will be returned in fixed-point decimal or binary representation), IP address ('`18.26.4.44`'), IP network address ('`18.26.4.0/24`'), Ethernet address ('`00:02:B3:06:36:EE`'), element name ('`elt`'), or IPv6 ad-

dress (‘1080::8:800:200C:417A’, ‘::’, ‘::18.26.4.15’). New types can be plugged into the library dynamically.

– The simplest means of accessing the library takes an entire set of arguments and produces the corresponding data or an error. This interface supports positional arguments, some of which may be optional, as well as keyword arguments like ‘APPEND true’. Elements may also use a lower-level interface that merely parses items; in this case, the elements are themselves responsible for separating arguments into items and for reporting errors.

This design emphasizes flexibility at the expense of commonality between elements. Clearly, elements can support natural representations for data types not mentioned above, since the elements themselves determine how strings are parsed into data. Several of our elements take advantage of this flexibility, such as the *Classifier* elements described in Section 4.2. However, this same property makes it difficult to write a complete typechecker for configuration strings without relying on element source code.

3.4 COMPOUND ELEMENTS

Compound element classes let users define their own element classes without writing C++ code. They are the Click language’s abstraction mechanism, providing flexible macros for router configuration graphs. Operationally, compound element classes are router configuration fragments that are treated like element classes. They are most commonly created by an `elementclass` statement, which has the following syntax:

```
elementclass Name {
  ... Click statements ...
}
```

After this statement, *Name* is an element class that corresponds to the Click configuration fragment inside the braces. Declaring an “element” of class *Name* adds a duplicate of that fragment to the configuration. We call the “element” a *compound element*; each actual element in the fragment it represents is called a *component* of the compound. For example, this compound element class has three components (an *InfiniteSource* element, a *Counter* element, and a *Discard* element):

```
elementclass Example {
  s :: InfiniteSource; c :: Counter; d :: Discard;
  s -> c -> d;
}
```

Only components remain in the final configuration graph—all compound element structure is compiled away. This process, called *flattening*, ensures that a compound element has no more overhead than the corresponding collection of components. It also lets tools manipulate flat configuration graphs, as opposed to hierarchical graphs containing compound element classes. During the flattening process, compound element components are given names that reflect their origin. A component named 'e' of a compound element named 'compound' is named 'compound/e' in the flattened configuration. For example, this configuration, with a compound element,

```
elementclass Example {
  s :: InfiniteSource; c :: Counter; d :: Discard;
  s -> c -> d;
}
e :: Example;
```

flattens into this configuration:

```
e/s :: InfiniteSource; e/c :: Counter; e/d :: Discard;
e/s -> e/c -> e/d;
```

Generated names like 'e/s' are valid identifiers in the language. As with anonymous element names, this helps achieve equivalence between programs and configuration graphs: a flattened configuration graph may be directly translated into an equivalent valid program.

The `elementclass` statement can redefine existing element classes. For example, this statement redefines the *Queue* class:

```
elementclass Queue {
  ... new definition ...
}
```

Inside the new definition, any use of '*Queue*' would refer to the original *Queue* class. It is not possible to define recursive or mutually recursive element classes.

Another `elementclass` variant defines an alias for an existing element class. For example, after this statement,

```
elementclass MyQueue Queue;
```

the identifier '*MyQueue*' is a synonym for *Queue*—specifically, for the *Queue* class that was active when the synonym statement was parsed.

The language supports nested compound element definitions. For example:

```
elementclass Example2 {
  elementclass NestedExample {
    ... Click statements ...
  }
  n :: NestedExample;
}
```

The scope of the nested compound element class is limited to the element class in which it appears.

Ports

Like any element, compound elements may have input and output ports. Each connection to or from a compound element port is transformed by flattening into a connection to or from one of its components' ports. Inside a compound element class, the special pseudoelements `input` and `output` specify how this transformation proceeds. Consider a compound element `e`. If `e/input`'s output port *i* has a connection to a component `e/c`, then every connection to a `e`'s input port *i* will flatten into a connection to its component `e/c`. The `output` pseudoelement plays a corresponding role for output ports. Compound element components are isolated from the main configuration except through `input` and `output`.

For example, in the following configuration, the *Example3* compound element has one input and zero outputs:

```
elementclass Example3 {
  input[0] -> Counter -> Discard;
}
FromDevice(eth0) -> Example3;
```

The connection between *FromDevice* and *Example3*'s input port transforms into a connection to the *Counter* component. Therefore, the above configuration flattens to something like this:

```
FromDevice(eth0) -> Counter -> Discard;
```

Figure 3.3 shows a compound element class with one input and one output as written in the language and as it might be drawn.

Particular `input` or `output` ports may be connected multiple times. This will expand each connection to that port into multiple connections to components. For example, this use of *Example4*,

```
elementclass SFQ {
  hash :: HashSwitch(...);
  rr :: RoundRobinSched;
  input -> hash;
  hash[0] -> Queue -> [0]rr;
  hash[1] -> Queue -> [1]rr;
  rr -> output;
}
```

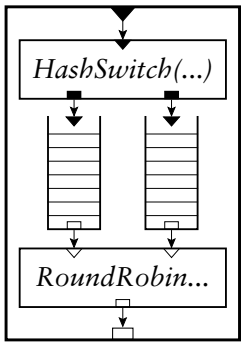

FIGURE 3.3—A simple compound element class.

```
elementclass Example4 {
  s1 :: InfiniteSource; s2 :: RatedSource;
  s1 -> [0]output; s2 -> [0]output;
}
e :: Example4 -> d :: Discard;
```

will expand into this flattened configuration:

```
e/s1 :: InfiniteSource; e/s2 :: RatedSource; d :: Discard;
e/s1 -> d; e/s2 -> d;
```

An `input` port may also be connected directly to an `output` port. For example, the following *Example5* element disappears when flattened:

```
elementclass Example5 {
  input -> output;
}
FromDevice(eth0) -> Example5 -> Discard;
```

expands into

```
FromDevice(eth0) -> Discard;
```

It is an error to use an input port of `input` or an output port of `output`, or to leave a particular `input` or `output` port unused when a higher-numbered port was used.

Anonymous compound element classes

Compound element classes may also be anonymous. To create an anonymous compound element class, the user simply writes a configuration fragment in braces where an element class was expected—for example, after '`::`' in an element declaration:

```
sfq :: {
  hash :: HashSwitch(...); rr :: RoundRobinSched;
  input -> hash;
  hash[0] -> Queue -> [0]rr; hash[1] -> Queue -> [1]rr;
  rr -> output;
}
```

or, even more anonymously,

```
FromDevice(eth0) -> { input -> Counter -> output } -> Discard;
```

Anonymous compound elements are useful for debugging as well as configuration structuring. Consider the following program:

```
q :: Queue;
// ...
Source1 -> ... -> q;
Source2 -> ... -> q;
q -> ... -> Sink;
```

Say we would like to print each packet as it enters the *Queue*. Without compound elements, one would have to change at least one of the uses of `q`; for example:

```
qp :: Print(q) -> q :: Queue;
// ...
Source1 -> ... -> qp;
Source2 -> ... -> qp;
q -> ... -> Sink;
```

With a compound element, only `q`'s declaration must be changed:

```
q :: { input -> Print(q) -> Queue -> output };
// ...
Source1 -> ... -> q;
Source2 -> ... -> q;
q -> ... -> Sink;
```

3.5 COMPOUND ELEMENT ARGUMENTS AND OVERLOADING

Compound element classes, like ordinary element classes, take configuration arguments and can have varying numbers of arguments, input ports, and output ports. *Formal parameters* define the arguments that a compound element class should take; *overloading* lets several compound element definitions with different numbers of arguments or ports share a single name.

A formal parameter is a sequence of alphanumeric characters preceded by a dollar sign, such as `$a` or `$10_4_buddy`. A compound element class may optionally begin with a list of parameters. For example, the following element takes exactly two arguments:

```
elementclass Example6 {
  $arg1, $arg2 | input -> Discard;
}
e :: Example6(a, b);                                          // OK
e2 :: Example6;                    // error—wrong number of arguments
```

Formal parameters are used inside the configuration strings of component elements. When a compound element is declared, occurrences of formal parameters in its components' configuration strings are replaced by the corresponding configuration arguments. For example:

```
elementclass GatewayDevice {
  $device | from :: FromDevice($device) -> output;
}
g :: GatewayDevice(eth0) -> Discard;
```

expands to

```
g/from :: FromDevice(eth0) -> Discard;
```

When compound elements are nested, formal parameters are lexically scoped.

Formal parameters let a compound element class support a fixed number of positional arguments. The overloading mechanism extends this to support optional arguments, as well as different behavior based on numbers of input or output ports. Every compound element class can correspond to a number of definitions, which are textually separated by '||'. These definitions are distinguished by their numbers of formal parameters, input ports, and output ports. For each declared compound element, the system checks how many arguments were supplied and how many inputs and outputs were actually used. Then it searches for a matching definition. If one is found, then it is expanded; if none is found, it is an error. For example, the following *ShapedQueue* element class supports an optional capacity argument, just like *Queue* itself:

```
elementclass ShapedQueue {
  input -> Queue -> Shaper(10000) -> output;
  ||
  $cap | input -> Queue($cap) -> Shaper(10000) -> output;
}
q1 :: ShapedQueue;                          // OK; uses first definition
q2 :: ShapedQueue(1024);                    // OK; uses second definition
q3 :: ShapedQueue(1024, 10000);             // error—no matching definition
```

The following *VerboseCheckIPHeader* element class has an optional second output port, just like *CheckIPHeader* itself:

```
elementclass VerboseCheckIPHeader {
  input -> c :: CheckIPHeader -> output;
  c[1] -> Print(CheckIPHeader) -> Discard;
  ||
  input -> c :: CheckIPHeader -> output;
  c[1] -> Print(CheckIPHeader) -> [1]output;
}
```

A variant of overloading lets the user add new definitions to an existing element class. For example, this definition adds a two-argument version of *Queue*:

```
elementclass Queue {
  ... ||                              // 'dot dot dot' is part of the syntax
  $capacity, $rate | input -> Queue($capacity)
                           -> Shaper($rate) -> output;
}
q1 :: Queue;                                        // built-in Queue
q2 :: Queue(1024);                                  // built-in Queue
q3 :: Queue(1024, 10000);           // overloaded Queue definition above
```

It is an error for a single compound element class to contain two overloaded definitions with the same signature (that is, the same numbers of arguments and input and output ports). A compound element class that overloads an existing element class ('`elementclass` *Name* { `...` `||` *definitions* }') may override a former definition with the same signature, however.

When choosing the definition that corresponds to a given compound element declaration, Click only considers the definitions that were lexically visible at the point of declaration. For example:

```
elementclass Example7 {
  Idle;
}
e1 :: Example7(test);       // error—no matching definition visible here
elementclass Example7 {
  ... || $arg | Idle;
}
e2 :: Example7(test);                                   // now it's OK
```

Thus, it is still impossible to write recursive or mutually recursive element class definitions in Click.

Compound element configuration arguments are only meaningful inside the configuration strings of components. They cannot, for example, change the element class of a given component, or cause components to be added to or subtracted from the compound. Overloading helps somewhat, but is based only on the number of configuration arguments, not on their values. However, it is possible to build a compound element that sends packets through different sets of components based on the value of its configuration string.

The keys are the *StaticSwitch* and *StaticPullSwitch* elements. *StaticSwitch* has one push input and several push outputs. Its configuration string contains an output port number; all packets arriving on its input are emitted on that output port. *StaticPullSwitch* is the pull equivalent: its configuration string contains an input port number, and all pull requests are forwarded to that input port. The following compound element uses *StaticSwitch* to implement a selective checksum check. (The *CheckIPHeader* element checks an IP header's length and checksum for sanity; *CheckIPHeader2* does the work of *CheckIPHeader* except for the checksum check.)

```
elementclass MaybeChecksum { $checksum_p |
  input -> sw :: StaticSwitch($checksum_p);
  sw[0] -> CheckIPHeader2 -> output;
  sw[1] -> CheckIPHeader -> output;
};
c1 :: MaybeChecksum(0);       // uses CheckIPHeader2, skips checksum
c2 :: MaybeChecksum(1);       // uses CheckIPHeader, checks checksum
```

The *click-undead* dead code elimination tool can remove *StaticSwitch* and any unused elements from the resulting configuration; see Section 6.4.

3.6 DISCUSSION AND LIMITATIONS

The Click programming language has achieved its goals. The language is readable and easy to work with. Compound elements let users create libraries of useful configuration fragments; they are also convenient when debugging configurations. As Chapter 6 will show, our design principles have created a convenient language for automatic processing. Also, Click's division of labor between element classes and the language works well in practice. Element classes determine how packets move locally and perform all complex calculations; the language is exclusively concerned with abstraction and how elements are combined. This division of labor has kept both language and system simple, focused, and useful.

The language is not necessarily complete, however. Possible areas for further work include adding features to compound elements and adding more information about individual element types.

Compound element classes support many of the features of native C++ element classes, but not all. For example, native classes can support keyword arguments, perform complex calculations based on their arguments, and create handlers that users may access; compound elements can do none of these things. Also, since compound elements are compiled away at router initialization time, their internal structure is visible in the created router. Compound elements are an imperfect abstraction mechanism: they do not hide themselves from the user, and they are strictly less powerful than native element classes. We plan to add some features missing from compound elements, particularly handlers, to a future release of Click. Currently, users who want these features must write their element classes in C++.

The decision to leave configuration string parsing up to individual elements has problematic consequences even beyond the difficulty of writing a type checker. For example, it is dangerous for any automatic tool to change the name of any element in a Click program: some other element might refer to that name inside its configuration string, but without knowing which configuration arguments are element names, it is impossible to patch any references that do exist. We initially chose the current design for its simplicity and because it supports blind manipulation. It was a good choice for building up the system. Now, however, as elements are more plentiful and as configurations grow bigger, checking configurations off line for errors may become more important than simplicity. In future, every element class may be accompanied by one or more specifications, such as descriptions of the types of its configuration arguments, its numbers of input and output ports and how those numbers are determined, and whether it modifies passing packets. Providing

this kind of semantic information to Click language tools would make it easier to check configurations for correctness off line, and would not burden element creators unnecessarily.

4

Elements

This chapter describes a cross-section of existing elements and discusses the design principles on which these elements were built; Chapter 5 presents a selection of router configurations that use the elements shown here. Click's available elements, and the ease of combining elements into configurations, goes to the heart of Click's utility. Section 4.1 begins with a description of our element design priciples, some of which are properties inherent in the Click architecture. The rest of the chapter describes particular elements, focusing on how these elements support composition. The packet scheduling, dropping policy, and TCP/IP rewriting elements, in particular, demonstrate how the Click element abstraction creates powerful and flexible networking components.

4.1 OVERVIEW

Every element class must address the simple set of primitives on which elements are built: configuration strings and input and output ports that can be push or pull. The characteristics of these primitives create a simple design methodology. For example, an element that divides packets into categories, such as a routing table, will have push ports, generally one output port per category. (In an element with all pull ports, each packet's output port has been determined before the packet itself is seen, so categorization based on packet contents is impossible unless the element has internal packet storage.) Click's elements follow this methodology as well as some rules we imposed ourselves:

– Separate functionality within reason. Each element should have a simple specification. For example, elements that classify packets should be separated from elements that modify them. However, separations that would result in complex inter-element communication should be avoided, particularly if the elements would not exchange packets or if they would be difficult to understand individually.

This rule argues for small elements and, therefore, large router configurations. Complex router behavior is implemented by combining simple parts.

– Whenever possible, use different ports rather than packet data to distinguish between different kinds of packets. For example, the *ARPQuerier* element takes both IP packets and ARP responses. It could examine flags in the packet header to distinguish between these packet classes, but instead, it has two input ports—one for IP, one for ARP. This makes the configuration easier to read, since different kinds of packets take distinct paths. It marginally speeds up the configuration by eliminating redundant packet header comparisons. It also tends to localize packet classification in explicit classification elements, which, in turn, facilitates optimization.

When these rules are followed consistently, elements naturally fall into categories distinguished by their input and output port characteristics.

– *Packet sources* spontaneously generate packets, either by reading them from the network (*FromDevice*), reading them from a dump file (*FromDump*), creating them from specified data (*InfiniteSource*, *RatedSource*), or creating them from random data (*RandomSource*). They have one output and no inputs.

– *Packet sinks* remove packets from the system, either by dropping them (*Discard*, *TimedSink*), sending them to the network (*ToDevice*), writing their contents to a dump file (*ToDump*), or sending them to the Linux networking stack (*ToLinux*). They have one input and no outputs.

– *Packet modifiers* change packet data. They have one input and one or two outputs. Packets arriving on the input are modified and then emitted on the first output. The second output, if present, is for erroneous packets. The input and the first output may be agnostic, but the second output is always push.[2]

2. Consider a packet modifier element whose ports are pull. After receiving a pull request on its first output port, the element will pull a packet from its input port and modify the packet it receives. Normal packets will simply be returned, satisfying the original pull request. Erroneous packets, however, must be stored or actively emitted on the second output port. This is a push operation. The element will probably return a null pointer in response to the original pull request.

– *Packet checkers* keep statistics about packets (*Counter*) or check them for validity (*CheckLength*, *CheckIPHeader*). Their ports resemble those of packet modifiers.

– *Routing elements* choose where incoming packets should go based on a packet-independent switching algorithm (*Switch*, *RoundRobinSwitch*), general characteristics of packet flow (*Meter*, *PacketMeter*, *RatedSplitter*), or an examination of packet contents (*Classifier*, *HashSwitch*, *LookupIPRoute*). They have one push input and two or more push outputs.

– *Storage elements* (*Queue*, *FrontDropQueue*) store packets in memory for later use, yielding up stored packets upon receiving pull requests. They have one push input and one pull output.

– *Scheduling elements* (*RoundRobinSched*, *PrioSched*) choose packets from one of several possible packet sources. They have one pull output and two or more pull inputs.

– *Information elements* implement language extensions (*AddressInfo*, *ScheduleInfo*) or interact with the configuration out-of-band (*ControlSocket*). They have no inputs or outputs.

Of course, not all elements fit into these categories, and some elements that do fit have slightly different port characteristics. Most elements fit into the categories remarkably well, however, which makes it possible to intuit part of an element's function without detailed knowledge of its specification.

4.2 CLASSIFICATION

Classifier is Click's generic classifier element. It compares each incoming packet's data against a set of patterns, and sends the packet to an output port corresponding to the first pattern that matched. Classification is a fundamental task, and at least one *Classifier* appears in even moderately complex configurations, so *Classifier*'s syntax and performance are both important. This section describes how we optimized its performance and made its syntax extensible.

At configuration time, *Classifier* parses its configuration string into a decision "tree" (actually, a directed acyclic graph). At run time, the tree is interpreted for every packet that is received. Leaf nodes represent output ports; internal branch nodes compare one word of packet data against a one-word constant value, perhaps with some of the packet data bits masked

off. *Classifier* cannot compare one packet header field with another or perform most greater-than or less-than tests.[3] This design makes *Classifier*'s inner loop efficient—on a Pentium Pro, that loop is just 13 instructions long—and most tests necessary in practice seem to be equality or inequality tests against constants.

Classifier optimizes its decision tree to avoid redundant tests. There are several approaches in the literature [2, 4, 17]; *Classifier*'s approach is most similar to BPF+'s [4]. Its core optimization is a form of redundant predicate elimination [52], which eliminates tests whose results can be statically determined. Optimization can be very effective, reducing the size of real *Classifier* decision trees by two-thirds or more. For example, the number of internal nodes used for the complex classifier of Figure 5.8 on page 72 drops from 181 to 50.

Comparing packet data against a data structure has significant interpretation overhead even after the data structure is optimized. One of Click's language tools, *click-fastclassifier*, addresses this cost by compiling specialized source code for each *Classifier* in a configuration. This tool expands optimized tree structure into explicit code branches and inlines all constants, for an effect somewhat like dynamic code generation [17]. Thus, the conventional optimization strategies for decision trees—tree minimization and compilation—are split in Click between the element itself and a specialized optimization tool. *Click-fastclassifier* is described further in Section 6.2.

All of *Classifier*'s tree creation and optimization methods are available for use by its subclasses. This facilitates extending its syntax. *Classifier*'s native filter language is austere, allowing only comparisons of packet data against hexadecimal constants. For example, the clause '`33/02%12`' matches packets whose 33rd data byte, after a bitwise AND with the hexadecimal mask `12`, equals the hexadecimal value `02`. *IPClassifier* is a subclass of *Classifier* and a parsing front end for it; its filter language consists of textual `tcpdump`-like expressions specialized for IP packets. For example, the clause '`tcp syn && not ack`' matches TCP packets with a SYN flag but no ACK flag; it is basically equivalent to the *Classifier* clause '`9/06 33/02%12`'. Figure 4.1 shows a sample classification task as written for *Classifier* and for *IPClassifier*. The subclass relationship between *Classifier* and *IPClassifier* is purely an implementation detail, and is not meaningful at the router configuration level. However, the

3. A limited set of greater-than and less-than tests can be compiled into equality tests. If x is a contiguous bitfield that fits within a single packet data word, then tests of the form $x < 2^k$ (or $x > 2^k - 1$, or $x \leq 2^k - 1$, or $x \geq 2^k$) are equivalent to testing some of x's upper bits for equality with zero. The *IPClassifier* element provides syntactic sugar for these tests.

Classifier

```
9/06 16/0A000014 33/02%12,

9/06 12/121A0400%FFFFFF00
    20/0050,
9/06 !22/0000%FF80
```

IPClassifier

```
tcp && dst host 10.0.0.20
    && syn && not ack,
src net 18.26.4.0/24
    && tcp src port www,
tcp dst port >= 128
```

FIGURE 4.1—A classification task for IP packets implemented by *Classifier* and *IPClassifier* configuration strings.

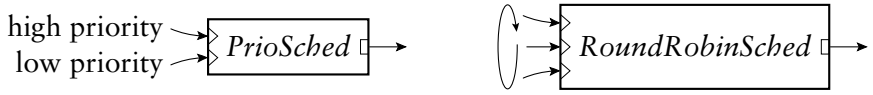

FIGURE 4.2—Some packet scheduler elements.

click-fastclassifier optimization tool takes advantage of the subtyping relationship; because they share decision tree structure, optimizations for the two elements are almost equivalent.

4.3 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 often try other inputs.

Click's packet scheduler elements include *RoundRobinSched*, *PrioSched*, and *StrideSched*. *RoundRobinSched* pulls from its inputs in round-robin order and returns 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 stride scheduling [50] with ticket values specified by the user. Arbitrary other packet scheduling algorithms could be implemented in the same way. *RoundRobinSched* and *PrioSched* are illustrated in Figure 4.2.

Packet scheduler elements make purely local decisions; any scheduling characteristics they induce in the router at large are dependent on the way the configuration is connected. This makes it easy to change a scheduling policy by plugging in a different element. It also makes it easy to implement new scheduling policies by composing packet schedulers together. For exam-

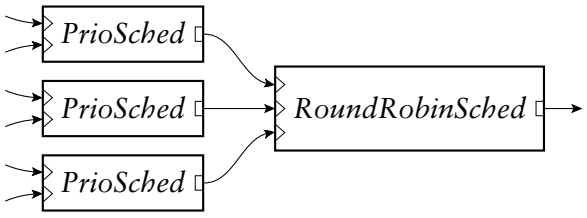

FIGURE 4.3—Composability of packet schedulers.

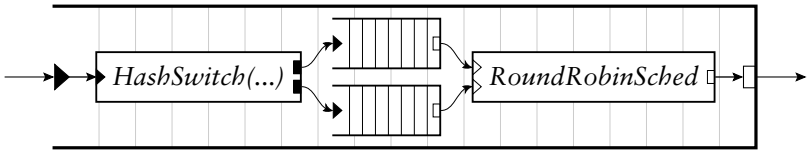

FIGURE 4.4—A stochastic fair queue as a compound element containing regular *Queue*s and a packet scheduler (*RoundRobinSched*). See Figure 3.3 on page 36 for a description in the Click language.

ple, consider a router with several traffic classes that should be scheduled round-robin, where each class has high-priority and low-priority traffic. This corresponds to a *RoundRobinSched* each of whose inputs is connected to a *PrioSched*; see Figure 4.3.

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. The compound element in Figure 4.4 is a virtual queue that implements stochastic fairness queueing [26]: packets are hashed into one of several queues that are scheduled round-robin, providing some isolation between competing flows. It has one push input and one pull output, just like a *Queue*, so it can generally be used as a drop-in replacement for *Queue*.

4.4 DROPPING POLICIES

The *Queue* element implements a simple dropping policy, a configurable maximum length beyond which all incoming packets are dropped. To implement more complex policies, the user can either create elements that replace *Queue* or build upon *Queue* with separate dropping policy elements. One separate policy element is *RED*, which implements Random Early Detection dropping [18]. This policy drops packets with higher probability when there is network congestion. *RED* contains only drop decision code. This separates the dropping policy from the packet storage policy, allowing either to be in-

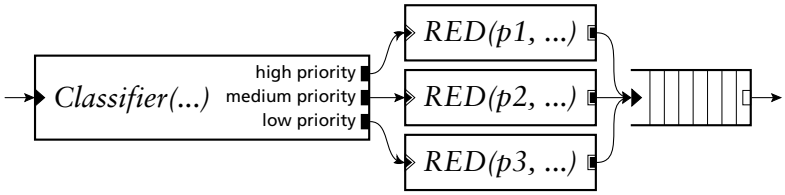

FIGURE 4.5—Weighted RED. The three *RED* elements have different RED parameters, so packets with different priorities are dropped with different probabilities.

dependently replaced. It is also more flexible, since the user can implement important RED policy variants just by rearranging the configuration.

RED considers a link congested when there are many packets in the queue servicing that link. The *RED* element therefore queries queue lengths when deciding whether to drop a passing packet. Specifically, it queries the number of packets in the nearest downstream *Storage* elements; *Storage* is a simple method interface implemented by *Queue* and other elements that store packets. It finds these elements using flow-based router context. *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 stochastic fair queue of Figure 4.4 would count both of its component *Queue*s. The arrangement of Figure 4.5 implements weighted RED [7], where packets are dropped with different probabilities depending on their priority. 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 [24]. (True drop-from-front RED would change the *Queue*s to *FrontDropQueue*s as well.)

4.5 DIFFERENTIATED SERVICES ELEMENTS

The Differentiated Services architecture [5] provides mechanisms for border and core routers to jointly manage aggregate traffic streams. 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—essentially, quality-of-service for the Internet. All of these components naturally correspond to Click elements, and implementing them as elements gives router administrators full control over how they are arranged.

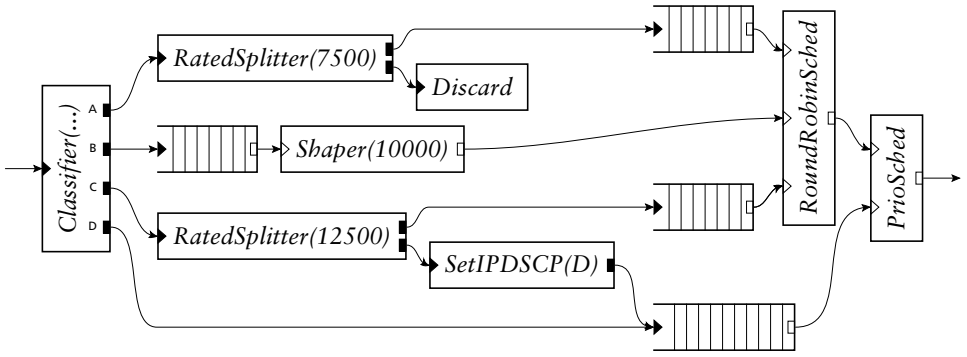

FIGURE 4.6—A diffserv traffic conditioning block. A, B, C, and D represent DSCP values.

We have already described elements for classification (*Classifier*), dropping (*Discard*), queueing (*Queue*), and scheduling (*RoundRobinSched*). *SetIPDSCP* is a simple retagger; it changes incoming IP packets' Differentiated Services Code Point field (DSCP) [33] to a fixed value and incrementally updates their IP checksums. The *RatedSplitter* and *Meter* elements classify packets based on their arrival rate. *RatedSplitter(r)* forwards at most *r* packets per second to its first output; any excess packets are redirected to the second output. *Meter(r)*, in contrast, sends the entire incoming stream to the second output once it exceeds *r* packets per second. Variants of these elements measure bandwidth (bytes per second) rather than packet rate. Finally, *Shaper* is a simple traffic shaper that regulates packets' departure times. *Shaper(r)* has one pull input port and one pull output port, and forwards pull requests to its input at a maximum rate of *r* requests per second.

Figure 4.6 uses these elements to build a realistic diffserv traffic conditioning block. This configuration separates incoming traffic into 4 streams based on DSCP. 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 4.6, these streams are (A) limited by dropping—traffic in excess of 7,500 packets per second is dropped; (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—traffic in excess of 12,500 packets per second is reclassified as best-effort delivery and sent into the lower priority queue.

4.6 IP REWRITING

This section describes the *IPRewriter* element family: *IPRewriter*, *IPRewriterPatterns*, *RoundRobinIPMapper*, *TCPRewriter*, and *FTPPortMapper*. These

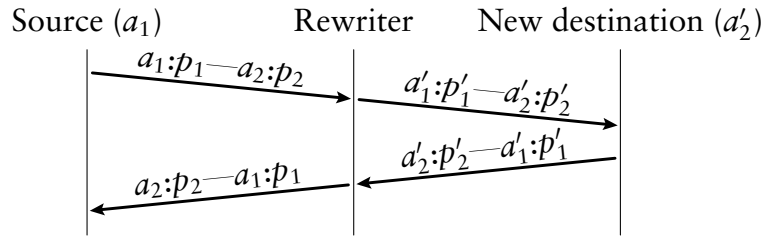

FIGURE 4.7—Action of a generic IP rewriter. Each arrow represents a packet with the given flow ID.

elements were originally designed to implement transparent proxies, but they also naturally support packet-filtering firewalls, network address/port translation, and Web server load balancing, among other applications. This demonstrates how the Click framework and its design methodology can inspire novel and flexible solutions to real routing problems.

Firewalls, load balancers, address/port translators, and transparent proxies each act as *network address translators*, or NATs [15, 45, 46]. A network address translator modifies passing packets' network addresses—and, optionally, their port numbers, or even data—to achieve some network-level goal, such as allowing many machines to share a limited number of IP addresses.

The network address translators we consider can examine and modify TCP and UDP packets' source addresses, source ports, destination addresses, and destination ports. We call this quadruple a packet's *flow ID*, and write it as "source address:source port—destination address:destination port". A network address translator will change some subset of these components, as well as any dependent header fields, such as checksums. Incoming packets that are part of the same TCP or UDP flow will share the same flow ID; their translations should share the same new flow ID. Packets replying to the translated flow will have the inverse of the new flow ID; their translations should share one flow ID, namely the inverse of the original flow ID. This design makes the translator transparent to both ends of the connection. Figure 4.7 demonstrates this general structure.

Network address translators contain a *flow mapping table* mapping old flow IDs to new flow IDs. Without such a table, it would be difficult to ensure that all packets on a session received the same new flow ID. Every translator can share a table implementation; the properties that distinguish translators are the rules used to place mappings into the table. Specifically, what happens when a translator receives a packet for which its table doesn't have a mapping? Should that packet be dropped? Should a new mapping be installed? If so, what should the new flow ID be? The rest of this section describes how

these questions are answered for four translators: a simple firewall, a network address/port translator, a Web load balancer, and a transparent proxy.

First off, consider a trivial firewall bridging an internal and an external network. This firewall allows sessions to be initiated only from the internal network. That is, the firewall forwards all packets that originated from the internal network, but forwards packets from the external network only if they correspond to an existing session. That session, by induction, must have originated internally. This corresponds to a network address translator with the following rules for *fresh packets*—that is, packets for which the table doesn't have a mapping:

1. For fresh packets from the internal network, create mappings that leave the packet's flow ID unchanged.

2. Drop fresh packets from the external network.

Packets originating externally never create new mappings, so every flow ID in the mapping table corresponds to a session initiated from the internal network. Fresh packets from the external network represent sessions initiated externally, and are properly dropped. Figure 4.8 shows a diagram of this configuration.

Network address/port translation, or NAPT [46], lets a network of machines with private IP addresses speak with the Internet by sharing one gateway with a public IP address. The NAPT gateway is a simple translator: outgoing connections are rewritten to apparently originate at the gateway; incoming replies are rewritten to apparently be destined for the relevant private IP address. This is accomplished by a network address translator with the following rules (Figure 4.9):

1. For fresh packets from the private network, create mappings that change the source address to the NAT gateway's address, and the source port to some arbitrary port. The source port must be unique among active sessions, since that field may be necessary to distinguish between them. The destination address and port remain unchanged.

2. Drop fresh packets from the Internet.

A load balancer for Web traffic takes incoming HTTP requests headed for a well-known server address and transparently redirects those requests among a set of servers that serve identical data. Its network address translation rules are (Figure 4.10):

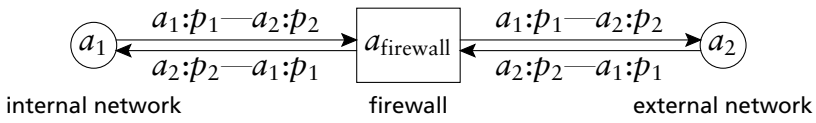

FIGURE 4.8—Rewriting behavior of a simple firewall. (In these diagrams, the flow IDs describe how packets originating at a_1 are affected by the rewriter.)

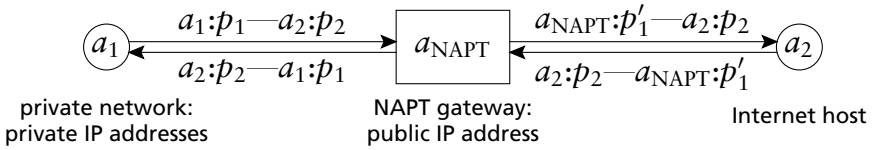

FIGURE 4.9—Rewriting behavior of a network address/port translator.

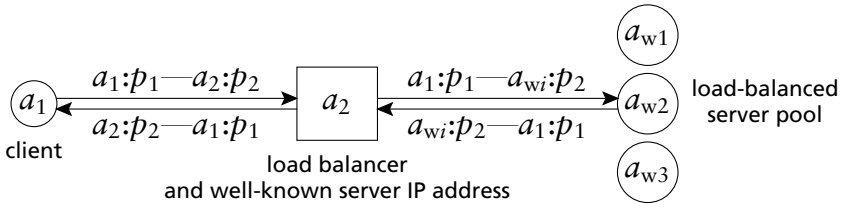

FIGURE 4.10—Rewriting behavior of a load balancer for Web traffic.

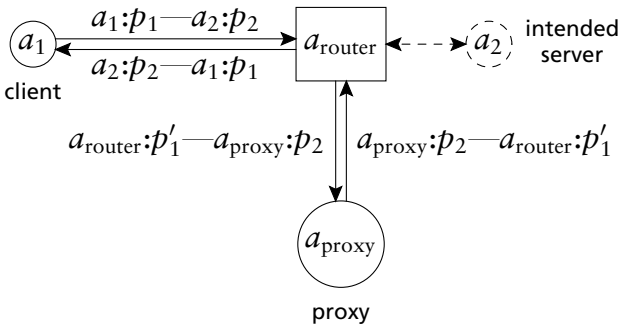

FIGURE 4.11—Rewriting behavior of a transparent Web proxy.

Application	New flow ID $a'_1{:}p'_1 — a'_2{:}p'_2$ for fresh flow ID $a_1{:}p_1 — a_2{:}p_2$	Allow fresh packets from a'_2?
Firewall	$a_1{:}p_1$—$a_2{:}p_2$	no
NAPT	$a_{\text{NAPT}}{:}p_{\text{NAPT}}$—$a_2{:}p_2$	no
Load balancer	$a_1{:}p_1$—$a_{\text{w}i}{:}p_2$, where $a_{\text{w}i} \in \{a_{\text{w}1}, \ldots, a_{\text{w}n}\}$	yes
Transparent proxy	$a_{\text{special}}{:}p_{\text{special}}$—$a_{\text{proxy}}{:}p_2$	yes

TABLE 4.1—Network address translation characteristics of four network applications.

1. For fresh packets from the Internet and destined for the well-known server address, create a mapping that changes the destination address to one of the set of server addresses. Leave the source address and port and destination port unchanged.

2. For fresh packets from the server pool, create mappings that leave their flow IDs unchanged. The server machines can transparently contact any machine they'd like.

Finally, a transparent Web proxy snoops for HTTP requests and forwards them to some program, which might, for example, cache frequently requested pages. Any replies from the proxy are rewritten to look like they originated at the intended server. This corresponds to these network address translation rules (Figure 4.11):

1. For fresh packets from a client and destined for a remote HTTP server, create a mapping that changes the destination address to the local proxy machine. The source address is also changed to a private address belonging to the router so that replies from the proxy can be rewritten.

2. Drop fresh packets from the proxy and destined for the router's private IP address. Any such packet should be in reply to some proxied connection, so it should have a corresponding mapping.

3. Pass along other fresh packets from the proxy unchanged.

Table 4.1 summarizes this discussion; it shows, in shorthand form, the translation rules for handling new flow IDs corresponding to the four applications.

IPRewriter

The *IPRewriter* element easily handles these applications, and others, by implementing a flow ID mapping table and letting the user define rules for handling fresh packets. The Click architecture lets *IPRewriter* be both specialized

and general. It is specialized for IP network address translation—other tasks, such as routing, classification, and queueing, are handled by the rest of the router configuration—but it implements translation in a fully general way.

The heart of *IPRewriter* is a flow mapping table. Each mapping in the table contains an old flow ID, a new flow ID, and the number of one of the element's output ports. Packets matching the old flow ID are rewritten to use the new flow ID and emitted onto the given output port.

Mappings are introduced into the table as fresh packets are encountered. Each of *IPRewriter*'s input ports is associated with one *rewrite rule*, which defines how mappings are added for fresh packets arriving on that input. The NAPT configuration described above, for example, requires two rules—one for traffic headed out of the private network, the other for traffic headed into it. The corresponding *IPRewriter* has two input ports, one per rule. The first input port is meant for outgoing traffic, and corresponds to the first rule; and similarly for the second input port. Rules are only applied to fresh packets, however—if a mapping exists for a packet, then that mapping is used regardless of the input port on which the packet arrived. This ensures that packets that are part of the same session are rewritten consistently, no matter which input port they arrive on.

The *IPRewriter* element thus distinguishes packet types only by the input port on which they arrived. The rest of the configuration must ensure that the right packets reach a given *IPRewriter* input port; *IPRewriter* is solely concerned with rewriting.

Mappings are removed from the table when they are stale—that is, when no packet with that mapping has been encountered for an hour. In the future, *IPRewriter* may check passing TCP packets for FIN or RST flags, which indicate that a connection is closed.

Figure 4.12 restates this discussion in pseudocode; it describes what happens when *IPRewriter* receives a packet.

Rewrite rules

Each of *IPRewriter*'s configuration arguments specifies a rewrite rule applying to a single input port. The most common, and complex, kind of rewrite rule is the '*pattern*' rule.

A pattern rule looks like '*pattern* A_1 P_1 A_2 P_2 O O_R'. A fresh packet's flow ID is rewritten according to the pattern, and the modified packet is emitted on output O.

The pattern proper has four parts: a new source address A_1, new source port P_1, new destination address A_2, and new destination port P_2. Any of these

```
For each packet p received on input port i,
    Search the flow mapping table for a mapping m
            corresponding to p's flow ID
    If no mapping was found,
        Execute the rewrite rule for input port i
        (The rule may insert new mappings into the table.)
        Search the table again for mapping m
    If no mapping was found either time,
        Drop the packet
    Otherwise, (m is a valid mapping)
        Rewrite p's flow ID according to m
        Push p onto the output port specified by m
```

FIGURE 4.12—Pseudocode describing how *IPRewriter* behaves when a packet is received.

parts can be a dash '–', which means "leave unchanged". Thus, the pattern '*1.0.0.1 – 1.0.0.2 –*' will set the packet's source and destination addresses but leave the ports as they are. The source port specification can also be a range of ports 'P_L–P_H', in which case the rewriter will choose a port between P_L and P_H. It will also ensure that any two active mappings created by this pattern have different source ports. The pattern '*1.0.0.1 1024–65535 1.0.0.2 80*', for example, sets every fresh packet's source address to 1.0.0.1, destination address to 1.0.0.2, and destination port to 80. The new source port, however, will differ for any two active sessions. Therefore, the new source port uniquely identifies an session, and every reply packet—which will contain that port number—can be mapped back to a unique original flow ID.

O is an output port number. The rewritten fresh packet, and later packets with the same flow ID, are emitted on output port O. The other port number, O_R, is used for rewritten reply packets. Different output ports are purely a convenience. *IPRewriter* assigns no semantics to its output ports—every rewriter could have exactly one output. It is often useful, however, to have *IPRewriter* demultiplex incoming packet streams based on their flow IDs. The user can decide, for example, that packets headed for networks *A* and *B* should be emitted on a rewriter's first and second outputs, respectively, and enforce this decision through rewrite rules. *IPRewriter* is already rewriting packets; demultiplexing has considerable benefit and minute additional cost.

When a fresh packet with flow ID $a_1{:}p_1{—}a_2{:}p_2$ arrives on an input port with a '*pattern*' rule, the rewriter first chooses a new flow ID $a'_1{:}p'_1{—}a'_2{:}p'_2$ according to the pattern. Next, the following pair of mappings is installed, one for the input flow ID and one for replies to the new flow ID:

$a_1{:}p_1$—$a_2{:}p_2$	maps to	$a'_1{:}p'_1$—$a'_2{:}p'_2$	with output port	O
$a'_2{:}p'_2$—$a'_1{:}p'_1$	maps to	$a_2{:}p_2$—$a_1{:}p_1$	with output port	O_R

Finally, the rewriter modifies the fresh packet to use the new flow ID and pushes this rewritten packet out on output O.

IPRewriter understands several other types of rule as well, including:

- '*drop*'. Fresh packets are dropped.
- '*keep* O O_R'. This rule is equivalent to '*pattern* – – – – O O_R': the rewriter installs mappings for the input flow and its reply flow that leave their flow IDs unchanged. Thus, future packets with either the input flow ID or its reply flow ID will be passed through the rewriter to O or O_R, even if they arrive on an input port with, for example, a '*drop*' rule.
- 'elementname'. A rewrite rule may consist of a single element name. That element should implement the *IPMapper* method interface; one of its methods is called whenever a fresh packet is encountered on this input port. The method is expected to install mappings into the *IPRewriter* element according to some algorithm. This lets the user extend the available rewrite rules by writing elements. For example, the *RoundRobinIPMapper* element, an *IPMapper*, is associated with an arbitrary number of '*pattern*' rules. It cycles through these rules in round-robin order as new flow IDs are encountered. This can implement round-robin load balancing among a pool of Web servers, for example.

Table 4.2 shows how these rewrite rules can implement the four rewriting applications of Table 4.1.

Discussion

In addition to the new flow ID and the relevant output port number, each mapping contains deltas so that the IP and TCP/UDP checksums can be updated incrementally. *TCPRewriter* is an IP rewriter specialized for TCP; its mappings contain more deltas for TCP's sequence number and acknowledgement number fields. These deltas start at zero, but other elements can easily manipulate TCP traffic by changing the deltas. The *FTPPortMapper* element uses this feature to support rewriting the File Transfer Protocol [39]. FTP requires special treatment because IP addresses are transmitted in the FTP data stream in text form. These IP addresses must be changed along with packets' source and destination IP addresses. *FTPPortMapper* rewrites embedded IP addresses

Application	Input port		Rewrite rule
Firewall	0	From internal	*keep 0 1*
	1	From external	*drop*
NAT	0	From private	*pattern* a_{NAT} p_{L}–p_{H} *– – 0 1*
	1	From Internet	*drop*
Load balancer	0	Requests	*RoundRobinIPMapper*
	1	Replies	*keep 1 0*
Transparent proxy	0	Requests	*pattern* a_{special} p_{L}–p_{H} a_{proxy} *– 0 1*
	1	Proxy's replies	*drop*
	2	Proxy other	*keep 1 0*

TABLE 4.2—*IPRewriter* configurations for the four network applications of Table 4.1. The "Input port" column informally describes the properties of packets arriving on each *IPRewriter* input port.

in FTP control packets as they pass by. This may make the FTP control packets grow or shrink, which changes the sequence numbering; *FTPPortMapper* bumps the relevant *TCPRewriter* sequence number deltas to account for this. *FTPPortMapper* demonstrates that the *IPRewriter* element family naturally extends to include application-level gateways [47].

Once a pattern rule's source port range is exhausted, that pattern will drop new packets rather than reuse active source ports. As mappings are removed, of course, the corresponding source ports become available again.

Some care is required to ensure that different fresh packets are never mapped to the same new flow ID. For example, consider these patterns:

Input port	Rewrite rule
0	*pattern 1.0.0.1 1024–4096 – – 0 1*
1	*pattern 1.0.0.1 2048–8192 – – 1 2*

If packets with flow IDs *A*:*B*—18.26.4.44:80 and *C*:*D*—18.26.4.44:80 arrive on the two input ports, the rewriter might choose 1.0.0.1:2048—18.26.4.44:80 as the new flow ID for both. This would make the reply mapping ambiguous. However, the rest of the configuration might ensure that packets arriving on input 0 had destination port 80, while packets arriving on input 1 had destination port 22. In this case, there would be no conflict between the patterns, because two new flow IDs created by the two patterns would never share the same destination port.

The user can prevent ambiguous mappings by using different source addresses, disjoint source port ranges, or *shared patterns*. A shared pattern is

not tied to a particular rule; instead, multiple *IPRewriter* rules can refer to the same shared pattern. Any two mappings created by using shared pattern will use different source ports, even if the mappings are stored in different *IPRewriter* elements. Shared patterns are specified with an information element, *IPRewriterPatterns*.

The network can generate ICMP errors, such as Destination Unreachable, Network Unreachable, and so forth, in response to any IP packet. If the ICMP error packet was generated in response to a rewritten packet, then it will not reach the original sender unaided. *IPRewriter* itself cannot help directly because the error packet is not TCP or UDP. The natural solution is to build an application-level gateway for ICMP. The *ICMPRewriter* element rewrites ICMP error packets based on the mappings stored in an existing *IPRewriter*.

This section has presented the *IPRewriter* element family in isolation. These elements shine most, however, when combined with other elements to form router configurations. Click element characteristics and our element design principles have led to uniquely flexible and modular components for IP network address translation. Users can create arbitrarily complex rewriting specifications; because of Click's input and output ports, their intentions are still clear. *IPRewriter* and friends are more flexible than other network address translators [21, 40]. For example, *IPRewriter* seems unique in its support for multiple translation tables and flexible placement of translation relative to other routing tasks. Section 5.5 presents an *IPRewriter* element in the context of a transparent Web proxy configuration.

4.7 INFORMATION ELEMENTS

The elements described so far, like most of Click's elements, participate actively in packet forwarding. However, some elements do not naturally forward packets. For example, shared *IPRewriter* patterns do not concern packets at all. Other shared structures, such as certain routing or location tables, also might belong outside the flow of packets. Furthermore, most potential language extensions do not involve forwarding packets, as Click already has good support for forwarding.

Shared state and language extensions are implemented in Click by *information elements*, which have no inputs or outputs. Information elements are a general mechanism for representing router structure that is independent of packet flow. Click's extensibility depends on decent support for non-forwarding tasks, so this section describes several information elements in depth, demonstrating the range of extensions they can implement.

Some information elements specify properties of other elements in the

configuration. Elements in this group are not active at runtime; they collect information from their configuration strings into a simple database that other elements can easily access. The *ScheduleInfo* element, for example, stores information about scheduling parameters. Section 2.6 described the task queue that schedules Click elements. Each scheduled element has a number of *tickets* that determines how often it will be scheduled in relation to other tasks; an element with two tickets will be scheduled twice as often as an element with one. Users specify these ticket amounts with *ScheduleInfo*.

Each of *ScheduleInfo*'s configuration arguments contains an element name and that element's relative priority. The default priority is 1; a priority of 2 says that an element should be scheduled twice as often as the default, 0.5 says it should be scheduled half as often, and so forth. For example:

```
fd :: FromDevice(eth0); td :: ToDevice(eth0);
ScheduleInfo(fd 0.1, td 1);
```

Elements like *ScheduleInfo* share a common problem: how should compound elements and information elements interact? This question is tricky to answer well, but finding the right solution can enhance the compound element abstraction. Consider the following configuration, where an element that must be scheduled (*Unqueue*) is buried inside a compound element (*Delayer*):

```
elementclass Delayer {
  input -> Queue -> u :: Unqueue -> output;
}
... -> d :: Delayer -> ...
```

Who can provide scheduling parameters for the *Unqueue*—the compound element, the main configuration, or both? And if both can provide parameters, how should those parameters interact?

In Click, both the compound element and the main configuration can provide scheduling parameters. For example:

```
elementclass Delayer {
  input -> Queue -> u :: Unqueue -> output;
  ScheduleInfo(u 2);
}
... -> d :: Delayer -> ...
ScheduleInfo(d 0.8);
```

This configuration will be flattened to something like the following.

```
d/Queue@1 :: Queue;
d/u :: Unqueue;
d/ScheduleInfo@3 :: ScheduleInfo(u 2);
ScheduleInfo@10 :: ScheduleInfo(d 0.8);
... -> d/Queue@1 -> d/u -> ...
```

Every element inside the compound, including the *ScheduleInfo*, has been assigned a name prefixed with `d/`. The *ScheduleInfo* named `d/ScheduleInfo@3` takes note of this and adds its `d/` prefix to each of its arguments; thus, its argument is effectively `d/u 2`, not `u 2`. The user can also specify a scheduling parameter for a prefix, such as `d 0.8` in the example. This parameter is multiplied with parameters specified inside the prefix. Therefore, in this configuration, the `d/u` element will have scheduling priority $0.8 \times 2 = 1.6$. Compound element designers and users collaborate to set scheduling parameters in this design. A compound element can suggest that it be scheduled more or less often than the default; if the user does not specify a scheduling parameter, Click will follow the element's suggestion.

In an alternative design, compound elements might specify how their parts should be scheduled relative to one another, but only the main configuration could say how the compound should be scheduled relative to other elements. The chosen design might be viewed as a breach of encapsulation when compared to this alternative. However, the chosen design is more flexible, that flexibility is important in practice, and the alternative design breaches encapsulation even more. With Click's design, a compound element is truly equivalent to the corresponding collection of simple elements; if two schedulable elements with priority 1 are included in a compound, their scheduling parameters are the same as before. In the alternative design, their effective priorities would drop to 0.5, since the compound as a whole would have priority 1. To make the compound usable, then, every user would have to give every use of the compound twice the scheduling parameter they might expect.

Other information elements extend the language's syntax. For example, Click configurations often refer to common IP addresses, Ethernet addresses, and IP network prefixes repeatedly. The *AddressInfo* element can provide shorthand names for these addresses, localizing the addresses and making the configuration easy to adapt as addresses change. Each of *AddressInfo*'s configuration arguments has a name and several addresses; for example,

```
AddressInfo(me 18.26.4.15 0:90:27:97:5a:76);
```

says that '`me`' is shorthand for an IP address (18.26.4.15) and an Ethernet address (0:90:27:97:5A:76). The address type chosen for a particular use of '`me`' depends on the kind of address required there.

Some information elements monitor the router configuration at run time or respond to user requests. For instance, *PeekHandlers* and *PokeHandlers* call read and write handlers on specified elements at specified times; this is useful for running repeatable tests. The *ControlSocket* element, available only in the user-level driver, opens a TCP/IP socket and listens for commands. Other programs may connect to this socket to query read handlers, call write handlers, or find out other information about the current router. Still other information elements stop the router when packet sources have completed, poll network interface status, or set up performance counter measurements.

Language extensions in general are well suited for implementation by information elements. When implemented as elements, extensions fit into the Click language without change, interact naturally with compound element classes, and, given anonymous element declarations, still provide convenient, readable syntax. Some extensions do not fit well into this framework—compound element classes, for example, obviously could not be implemented by information elements alone. However, most extensions do not alter the element abstraction, they merely augment it. Information elements are good for extensions of this kind. They make the Click language and system extensible without pain.

5
Routers

Now that elements have been described, we explore how they combine to form routers. This chapter presents several full Click router configurations, starting with a standards-compliant IP router. The Click IP router is modular—it has sixteen elements on its forwarding path—and, therefore, easy to understand and easy to extend. The next sections present IP router extensions for different queueing and packet scheduling policies and for transparent proxying, as well as simpler configurations that use subsets of the IP router's elements. Of course, Click is suitable for many packet processing tasks other than IP routing, and the chapter closes with examples of non-IP configurations and IP non-routers—specifically, an Ethernet switch, a simple firewall, and the traffic generator used in our tests.

5.1 IP ROUTER

Figure 5.1 shows a Click IP router that forwards unicast packets in nearly full compliance with the standards [3, 37, 38]. The version in the figure has two network interfaces, but easily generalizes to three or more. The rest of this section describes this configuration in more detail; Chapter 8 evaluates its performance with eight network interfaces, and Appendix A contains quick descriptions of every element class it uses.

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, which is 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 com-

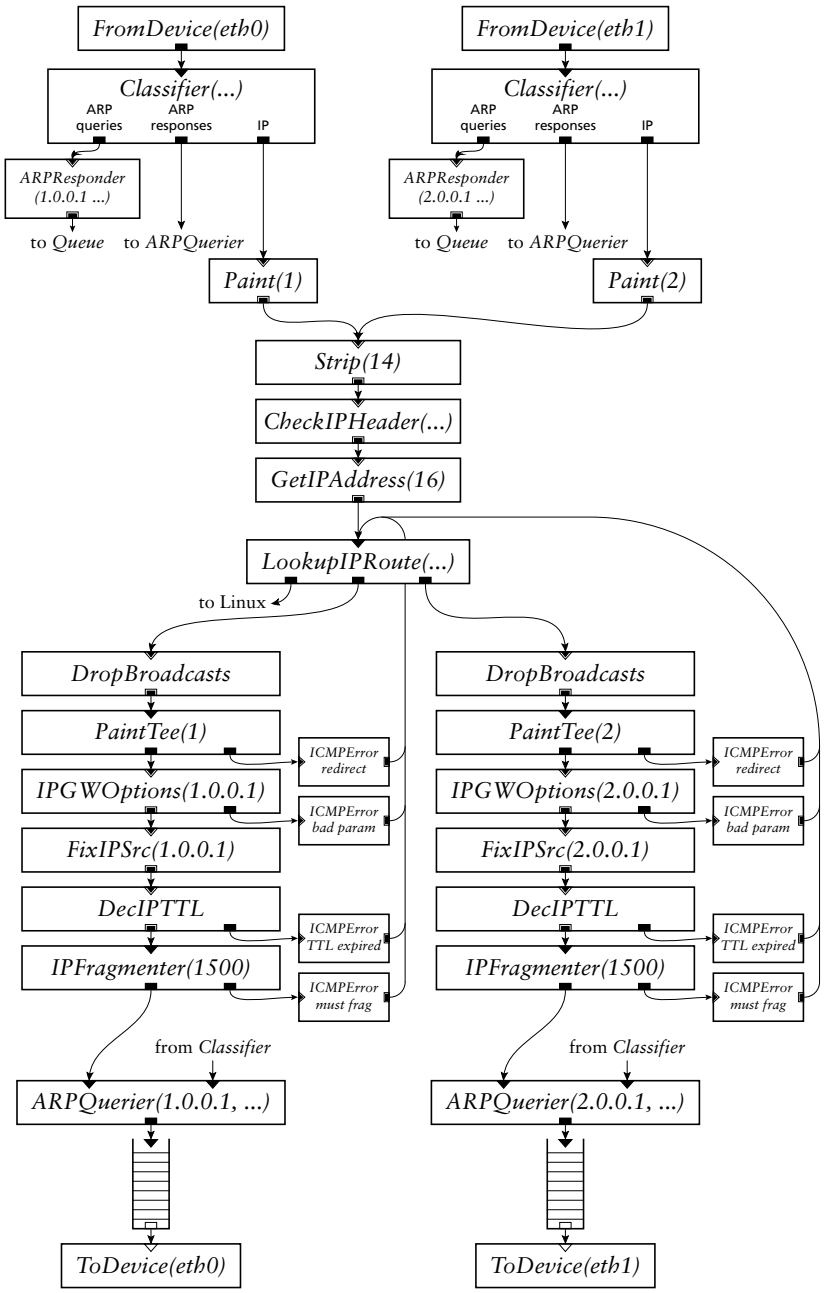

FIGURE 5.1—An IP router configuration.

posed. 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.

Some forwarding tasks require that information about a packet be calculated in one place and used in another. The IP router uses packet annotations (Section 2.2) to carry such information along, including the following:

- **Destination IP address.** Elements that deal with a packet's destination IP address use this annotation rather than the value of the IP header's destination 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". *PaintTee* 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 color 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 *DropBroadcasts* 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 individual elements' behavior and because of their arrangement. For example, the *CheckIPHeader* element checks exactly the properties required by the standards: the IP length fields, the IP source address, and the IP checksum. *IPGWOptions* processes just the Record Route and Timestamp options, since source route options should be processed only on packets addressed to the router. *IPFragmenter* fragments packets larger than the configured MTU, but sends large packets marked as unfragmentable 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. Likewise, the Record Route IP option must be processed only on packets not destined for the router, so *IPGWOptions* must follow *LookupIPRoute*; and the router should not process IP options on link-level broadcast packets, so *IPGWOptions* follows *DropBroadcasts*.

5.2 EXTENSIONS AND SUBSETS

Click's design makes it easy to extend the IP router by adding elements, or to implement simpler IP networking tasks by reusing a subset of its elements. Figure 5.2 shows IP router extensions that implement different dropping and packet scheduling policies, including Differentiated Services; the elements supporting these policies were described in Sections 4.3, 4.4, and 4.5. Many of the extensions consist of just a couple added elements. The extensions only affect how the router stores packets, not how routing decisions are made, so the changes required are conveniently localized to the area surrounding the *Queue*s.

Click's modular scheduling, queueing and dropping policy elements make even complex functionality easy to implement. For example, consider the

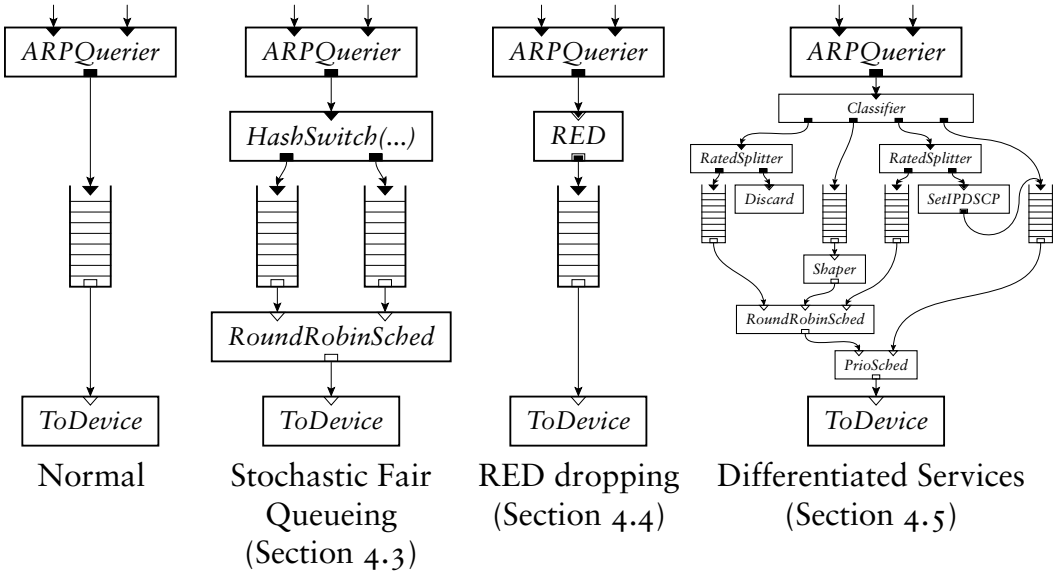

FIGURE 5.2—Three queueing extensions for the IP router of Figure 5.1.

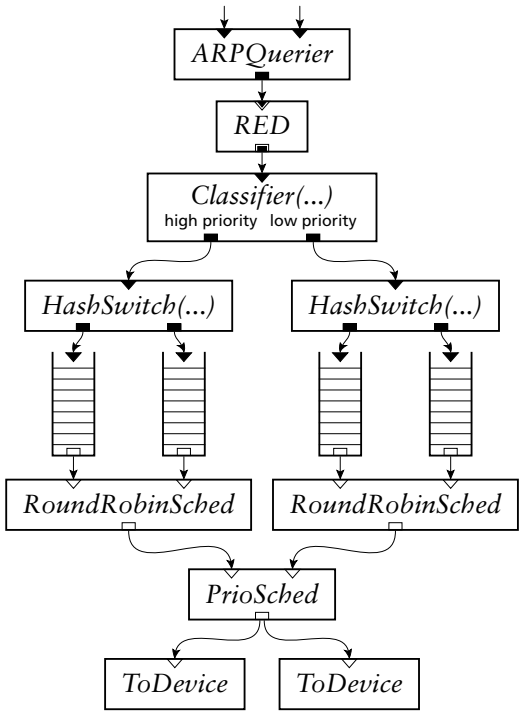

FIGURE 5.3—A complex combination of dropping, queueing, and scheduling.

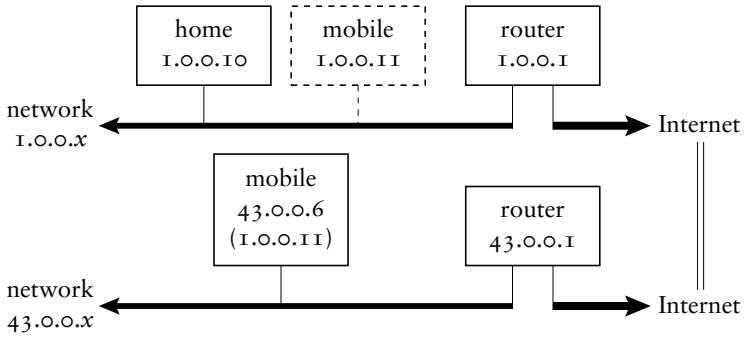

FIGURE 5.4—Network arrangement for a configuration supporting mobility. The mobile host 1.0.0.11 is currently located on network 43, where it has the temporary address 43.0.0.6. Its home node, 1.0.0.10, should encapsulate packets destined for 1.0.0.11 and forward them to 43.0.0.6.

following requirements:

– two parallel 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;

– and RED dropping driven by the total number of packets queued.

Figure 5.3 shows the corresponding extension.

We now turn to a different task, namely supporting host mobility. This configuration can reuse several of the IP router's elements in a simpler context. Figure 5.4 shows a network with a mobile host, 1.0.0.11, that has currently roamed to network 43 and accepted a local IP address there (43.0.0.6). We would like packets sent to the mobile host's home address to be transparently forwarded to its current location. Assume, however, that we cannot change the home router. We do have control over 1.0.0.10, a normal machine on the home network. Figure 5.5 shows a Click configuration that might be active at this home node. The configuration proxy-ARPs for 1.0.0.11, the mobile host's home IP address, essentially pretending to be the mobile host (the *ARPResponder* element). Packets it receives that are meant for 1.0.0.11 are then encapsulated in a new IP header and sent to 43.0.0.6, the mobile host's remote location (the *IPEncap* path). Reply packets from 43.0.0.6 are unencapsulated and sent onto the local network (the *CheckIPHeader* path). Not shown are elements that deal with fragments or ensure that packets generated by 1.0.0.10 itself are properly encapsulated.

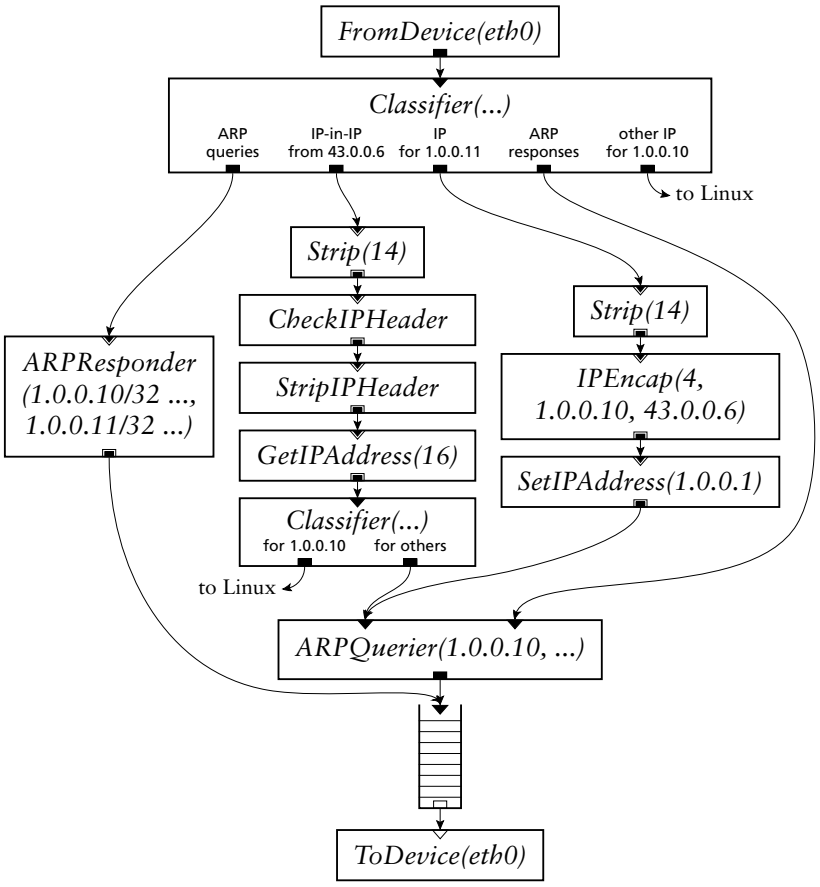

FIGURE 5.5—A configuration for the home node, 1.0.0.10, in the arrangement of Figure 5.4.

5.3 ETHERNET SWITCH

Figure 5.6 shows a Click configuration unrelated to IP, namely an IEEE 802.1d-compliant Ethernet switch. It acts as a learning bridge and participates 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. *EtherSwitch* can switch packets among an arbitrary number of links, which are represented by pairs of one input port and one output port. Intuitively, each port pair corresponds to an Ethernet interface, and this is how the figure is arranged; as a modular element, however, *EtherSwitch* can easily support other configurations. When a packet arrives on an *EtherSwitch* input port, it is normally copied and sent to every other output port. This corresponds to the action of a hub. *EtherSwitch* additionally keeps a table

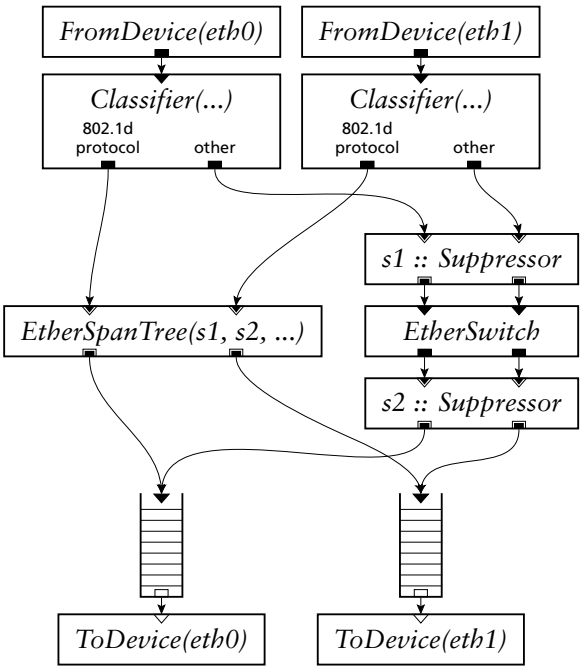

FIGURE 5.6—An Ethernet switch configuration.

mapping Ethernet addresses to port numbers. When a packet with Ethernet source address *a* arrives on input port *p*, this indicates that the machine with address *a* must be on, or directly reachable via, link *p*, so *EtherSwitch* stores a mapping $a \rightarrow p$ in the table. Until that mapping expires, all packets destined for *a* will be forwarded only to port *p*.

The *EtherSpanTree* and *Suppressor* elements are necessary only to avoid cycles when multiple bridges are used in a LAN. *Suppressor*, a generic element, 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* implements the IEEE 802.1d protocol for constructing a network spanning tree. It suppresses certain ports on the *Suppressor* elements, thus preventing *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 give their names in *EtherSpanTree*'s configuration string.

5.4 FIREWALL

Firewalls attempt to prevent unauthorized traffic from entering or leaving an internal network. Packet filtering firewalls, in particular, work by deciding, for

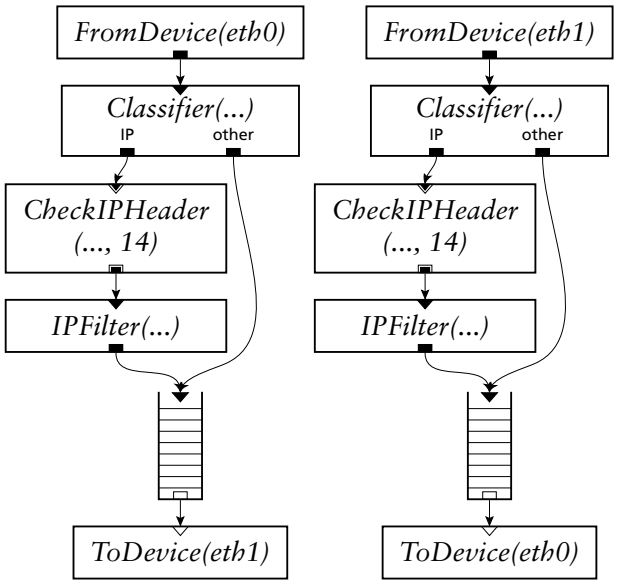

FIGURE 5.7—A simple IP firewall.

each packet, whether or not to forward it. Figure 5.7 shows a simple packet filtering firewall built in Click. This firewall acts as a transparent bridge; it has two interfaces, and all packets arriving on one interface are either passed to the other interface or dropped. Its IP addresses, and even its Ethernet addresses, are not visible to other machines on the network. It behaves like an Ethernet cable—for example, it forwards ARP queries and responses unchanged. The firewall itself has no holes that could be exploited by a network attacker,[4] since it does not process packets except by forwarding them. This also means that it cannot be contacted over the network.

The firewall's packet filtering behavior depends on its *IPFilter* elements. *IPFilter* is another element built upon *Classifier*; its syntax resembles *IPClassifier*'s, but the user writes a set of filtering rules for each output rather than one classifier per output. Figure 5.8 shows one set of filtering rules suitable for *IPFilter*. The rules were directly translated from an example in *Building Internet Firewalls* [54, pp 691–2]; they are the incoming packet filtering rules for the network configuration of Figure 5.9. For example, the following rules:

```
allow src host BASTION && dst host INTERNAL_SMTP                // SMTP-2
   && tcp && src port 25 && dst port > 1023 && ack,
```

4. Except, of course, for any bugs in the firewall's elements or its network drivers. One of Click's advantages is that elements are relatively simple and can be checked easily for correctness.

```
IPFilter(deny src net INTERNAL,                                    // Spoof-1
         allow src host BASTION && dst net INTERNAL                // HTTP-2
            && tcp && src port www && dst port > 1023 && ack,
         allow dst net INTERNAL                                    // Telnet-2
            && tcp && src port 23 && dst port > 1023 && ack,
         allow dst net INTERNAL                                     // SSH-2
            && tcp && src port 22 && ack,
         allow dst net INTERNAL                                     // SSH-3
            && tcp && dst port 22,
         allow dst net INTERNAL                                     // FTP-2
            && tcp && src port 21 && dst port > 1023 && ack,
         allow dst net INTERNAL                                     // FTP-4
            && tcp && src port > 1023 && dst port > 1023 && ack,
         allow src host BASTION && dst net INTERNAL                 // FTP-6
            && tcp && src port 21 && dst port > 1023 && ack,
                                                          // FTP-7 omitted
         allow src host BASTION && dst net INTERNAL                 // FTP-8
            && tcp && src port > 1023 && dst port > 1023,
         allow src host BASTION && dst host INTERNAL_SMTP          // SMTP-2
            && tcp && src port 25 && dst port > 1023 && ack,
         allow src host BASTION && dst host INTERNAL_SMTP          // SMTP-3
            && tcp && src port > 1023 && dst port 25,
         allow src host NNTP_FEED && dst host INTERNAL_NNTP        // NNTP-2
            && tcp && src port 119 && dst port > 1023 && ack,
         allow src host NNTP_FEED && dst host INTERNAL_NNTP        // NNTP-3
            && tcp && src port > 1023 && dst port 119,
         allow src host BASTION && dst host INTERNAL_DNS            // DNS-2
            && udp && src port 53 && dst port 53,
         allow src host BASTION && dst host INTERNAL_DNS            // DNS-4
            && tcp && src port 53 && dst port > 1023 && ack,
         allow src host BASTION && dst host INTERNAL_DNS            // DNS-5
            && tcp && src port > 1023 && dst port 53,
         deny all                                                 // Default-2
         );
```

FIGURE 5.8—Packet filtering rules for incoming packets on the *Building Internet Firewalls* "interior router" example [54, pp 691–2], implemented by an *IPFilter* element. The rule FTP-7 contains greater-than and less-than tests that cannot be implemented with *IPFilter*. Capitalized words like `BASTION` are addresses specified elsewhere with *AddressInfo*. See also Figure 5.9.

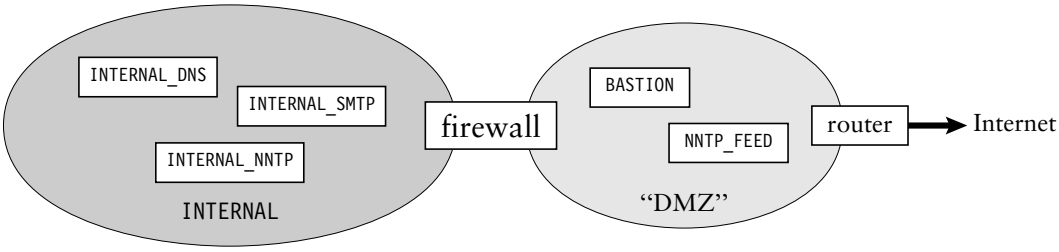

FIGURE 5.9—The network configuration for the firewall of Figure 5.8. The firewall protects machines on the INTERNAL network. Machines in the "DMZ" are partially trusted by the internal network although they are outside the firewall.

```
allow src host BASTION && dst host INTERNAL_SMTP                    // SMTP-3
   && tcp && src port > 1023 && dst port 25
```

allow the exchange of SMTP traffic between the BASTION host in the "DMZ" and the INTERNAL_SMTP host in the internal network. Either host can initiate a connection to the other host's SMTP port, port 25. The BASTION host presumably receives mail from the Internet, and perhaps checks it for viruses and other problems; once it is satisfied, it may forward traffic to the INTERNAL_SMTP host, from which it will reach the internal recipient.

The firewall's *CheckIPHeader* elements mark each packet's IP header, which is required by *IPFilter*. The *14* in their configuration strings says that the IP header starts 14 bytes into each packet—that is, just after the Ethernet header.

5.5 TRANSPARENT PROXY

Figure 5.10 shows a transparent Web proxy extension for the IP router configuration. It expects to run on a router that receives all of its clients' Web requests—for example, on a bridge router that is the clients' single link to the Internet. Outgoing Web requests are diverted to a user-level proxy running on the router machine. The proxy might, for example, answer the queries from a cache.

Figure 5.10 catches outgoing traffic just before it reaches the outgoing interface's *ARPQuerier* and separates Web traffic from other traffic using an *IPClassifier*. Web traffic passes through an *IPRewriter* element, which rewrites packets' flow IDs as described in Section 4.6. The rewrite rule for outgoing packets is "*pattern FAKE 1024–65535 LOCAL 8000 0 1*", so packets are rewritten and sent to port 8000 on the local machine. The proxy should be listening on that port. The packets' source addresses are also changed, to *FAKE*.

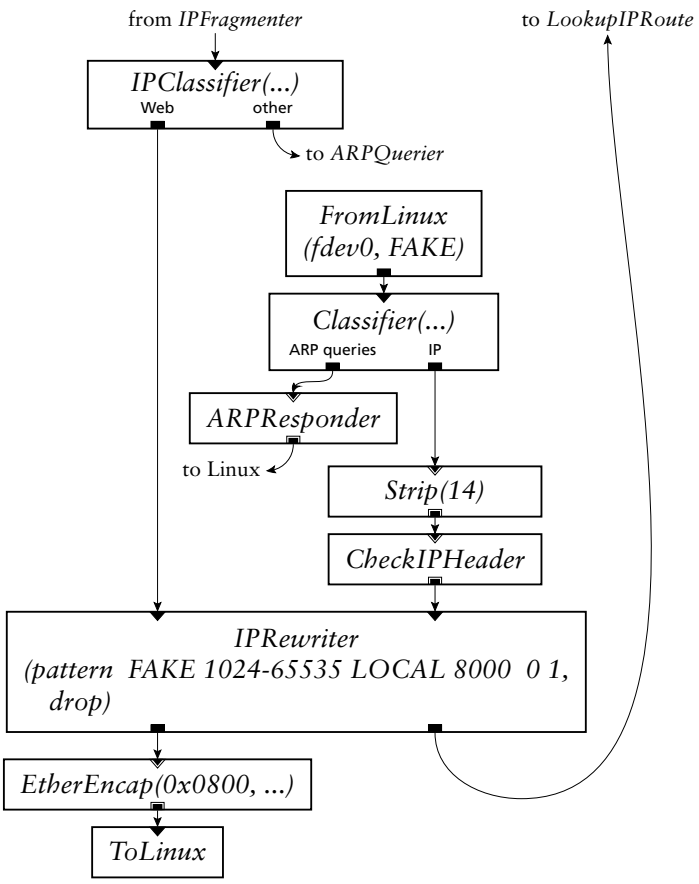

FIGURE 5.10—A transparent Web proxy extension for the IP router configuration.

The *FromLinux* element captures replies from the proxy. *FromLinux* installs a fake device into Linux's device table; here, that device is named *fdev0* and its native IP address is *FAKE*. Linux's routing table is manipulated so that packets destined for the address *FAKE* are sent to the fake device. Since packets directed to the proxy have their source addresses changed to *FAKE*, replies from the proxy will have destination address *FAKE*, and will be sent to the fake device. As packets arrive on that device, they are emitted into the Click router configuration by *FromLinux*. The configuration must respond to ARP requests sent to the fake device, which explains the *Classifier* and *ARPResponder* elements. IP packets from the fake device—namely, the proxy's replies—are sent through the *IPRewriter*, where they are rewritten to look like replies from the intended server and properly forwarded via the configuration's routing table.

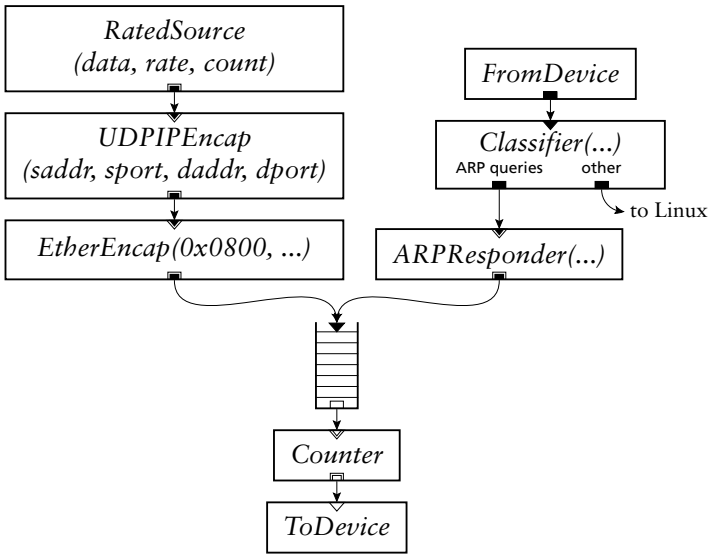

FIGURE 5.11—A configuration that generates an even flow of UDP traffic at a given rate.

5.6 TRAFFIC GENERATOR

Figure 5.11 shows a Click traffic generator. We used this configuration in our experiments to generate flows of minimum-size UDP/IP packets at specified rates. Click's efficiency and location within the Linux kernel allows a simple configuration to generate more traffic more evenly than any user-level program. The configuration also demonstrates that Click is well-suited for packet processing tasks that are nothing like routing.

The *RatedSource* element actually generates the traffic. It creates and emits packets containing the specified data at the specified rate without burstiness. Later *UDPIPEncap* and *EtherEncap* elements encapsulate this traffic in correct UDP/IP and Ethernet headers. Most packets received during the test are passed off to Linux for normal processing, but the Click configuration responds to ARP queries itself. ARP queries must be answered in a timely fashion to support our evaluation setup; placing ARP responses under Click's control ensures that responses will be generated relatively quickly, and allows us to schedule Linux less often. The *Counter* element counts the number and rate of packets actually sent.

6

Language tools

This chapter explores the router manipulation tools made possible by Click's programming language. Tools in general can optimize router configurations, transform them, and check whether they satisfy simple properties. We discuss, in particular, a tool for transforming a configuration based on user-specified patterns; two optimization tools that generate specialized C++ source code; a tool that eliminates dead code from a configuration; a tool that ensures that routers satisfy a global property, namely that packet data is always correctly aligned; and a tool that combines two or more router definitions into one configuration, enabling global optimizations and analyses on a collection of routers. We built these tools to make routers faster—for many of them, we report how much they speed up the Click IP router in Figure 5.1—and to allow Click to run on non-x86 machines. More fundamentally, though, they demonstrate how powerful programming language techniques can be when applied at the level of system components.

Again, the tools depend on several important properties of the language. First, every router configuration can be expressed in the language, so a language file is effectively equivalent to a router configuration. Furthermore, because the language is wholly declarative, a configuration can be considered apart from its eventual execution environment. Without this property, it would be difficult to manipulate router descriptions outside of a running router.

Each tool reads a configuration, analyzes and manipulates that configuration, reports errors, and writes out the (possibly modified) configuration. Being a language file, the output of one tool is suitable as input for another, so tools can be composed in any order. We categorize the tools according to three properties:

1. **Language-only or language plus source code?** Some tools work entirely at the configuration language level, while others additionally generate C++

source code defining new elements. This source code is attached to the configuration; a tool for installing configurations can compile the source code and dynamically load it into a Click driver.

2. **Element semantics.** Some tools work without understanding the semantics of individual elements. Others need to partially understand some elements' semantics, ranging from whether an element's ports are push or pull to how an element affects packet data alignment.

3. **Graph analyses and manipulations.** Some tools require only simple graph analyses and manipulations, others run relaxation-based data flow algorithms on graphs, and others perform expensive graph operations like subgraph isomorphism tests.

6.1 PATTERN REPLACEMENT

The pattern replacement tool, *click-xform*, is a generic search-and-replace engine for configuration graphs. First, it reads a router configuration and a collection of patterns and replacements. It then searches the configuration for occurrences of each pattern, replacing each match it finds with the corresponding replacement. When there are no more occurrences of any pattern, it outputs the transformed configuration.

Click-xform patterns and replacements are written as compound elements. A pattern matches a subset of a router configuration iff replacing that subset with an occurrence of the pattern's compound element would result in an identical configuration, modulo element names. Element configuration strings in each pattern may also contain variables, such as $*a*, which are set to the corresponding text in the matched subset. This definition is quite flexible—for example, the user can specify that the pattern must be isolated from the rest of the router, or that the pattern is connected to the router in a constrained way. Searching a configuration graph for an occurrence of a pattern is a variant of subgraph isomorphism, a well-known NP-complete problem. Luckily, the patterns and router configurations seen in practice are well-served by Ullmann's algorithm for subgraph isomorphism [49], and *click-xform*'s observed performance is good, taking about one minute to make hundreds of replacements on router graphs with thousands of elements (and much less on normal-sized routers).

One optimization enabled by *click-xform* is replacing a slow element, or a collection of slow elements, with a single, more specialized, faster element. For example, we have designed elements that combine the work of common IP router elements. Figure 6.1 shows a portion of the IP router configuration

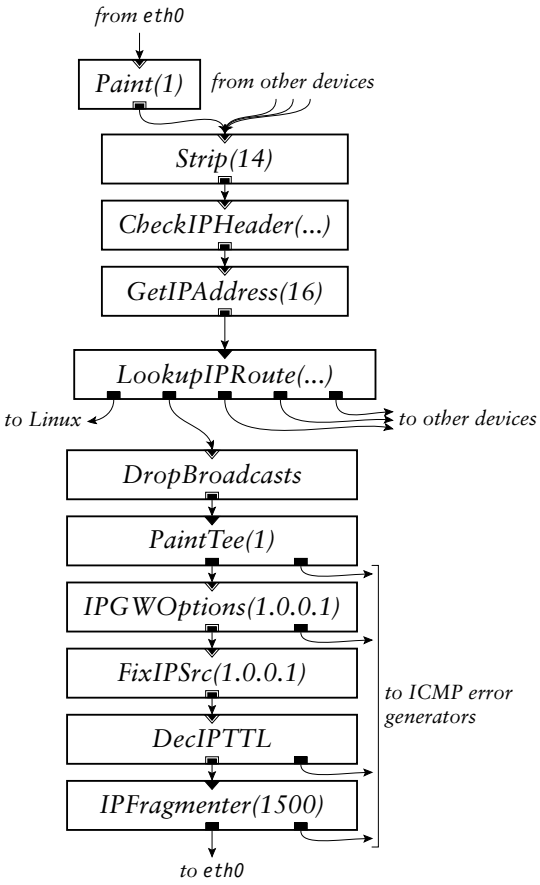

FIGURE 6.1—A portion of the IP router configuration (Figure 5.1 on page 64).

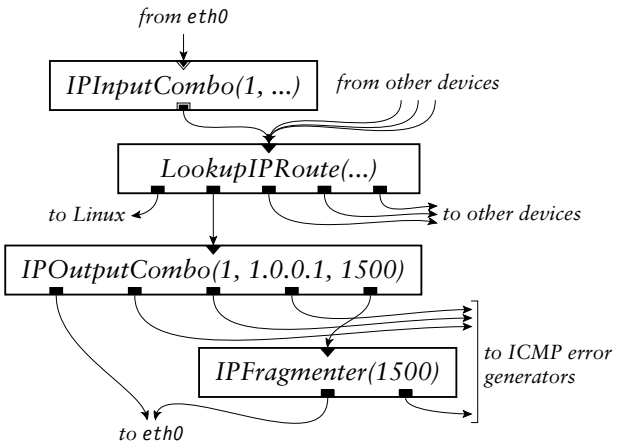

FIGURE 6.2—A router fragment equivalent to Figure 6.1 using faster "combo" elements.

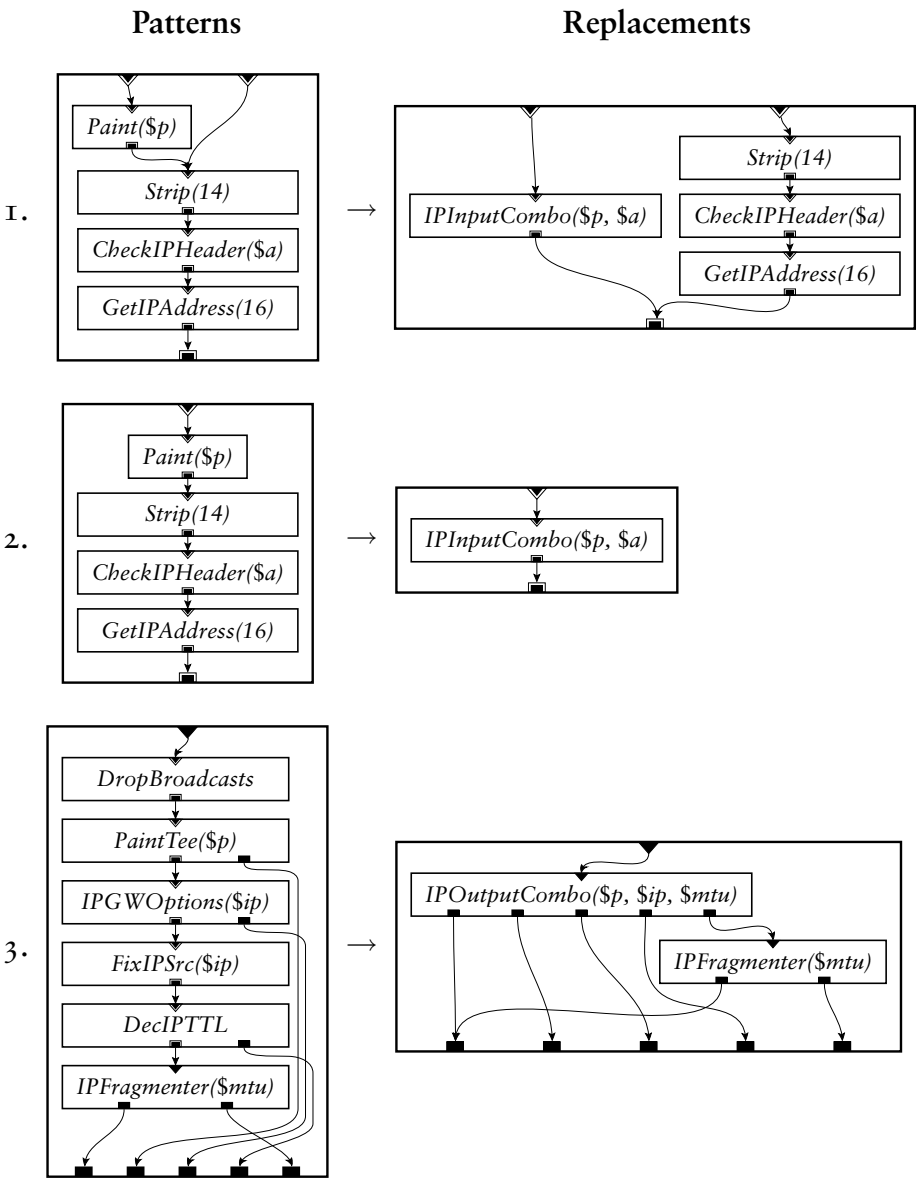

FIGURE 6.3—Three pattern-replacement pairs suitable for the *click-xform* tool. Together, they transform Figure 6.1 into Figure 6.2.

(Figure 5.1 on page 64) corresponding to one network interface. The fragment in Figure 6.2 behaves equivalently, but uses the combination elements. This combination-element configuration takes 29% less CPU time to process a single packet than the original. With the simple pattern set shown in Figure 6.3, *click-xform* can automatically transform the original fragment into the combination-element version.

Every element class mentioned in a replacement must already exist. Click users, for example, wrote the combination element classes in Figure 6.2 by hand. *Click-xform* simply changed configurations to use the combination elements wherever possible. A single set of patterns can apply to an infinite number of configurations, freeing the user from having to update all routers by hand. Furthermore, *click-xform* lets users work with the original, more modular configurations, which tend to be more flexible and easier to understand, without sacrificing the performance of the specialized versions. These are the conventional arguments for any optimizer: a high-level language plus optimizations is often preferable to a low-level language. In Click, however, programs in the high-level and low-level languages are distinguished not by their syntax, but by the element classes they use, and the user can mix high- and low-level languages at will.

Click-xform is also suitable for general-purpose transformations. It could, for example, insert extensions into existing configurations. Well-written patterns could upgrade configurations to use new elements or fix bugs.

Click-xform works entirely at the language level and without any knowledge of element semantics. Semantic knowledge is encoded in the patterns and replacements, but these are written by users. It performs extensive, and expensive, graph analysis and manipulation.

Pattern replacement has been implemented in several systems—for example, Scout [36] has a rule-based global optimizer, and instruction selectors like BURG [19] attack similar problems for tree structures. *Click-xform*'s pattern language is more powerful than any of these.

6.2 FAST CLASSIFIERS

The fast classifier tool, *click-fastclassifier*, optimizes configurations that contain *Classifier*, *IPClassifier*, and *IPFilter* elements. As described in Section 4.2, these elements traverse a decision tree whose intermediate nodes check packet data against fixed values, and whose leaves represent particular output ports. *Click-fastclassifier* makes each such element faster by compiling its decision tree into C++ code, inlining constants and making branches explicit. Different *Classifier*s are compiled into different C++ classes.

Click-fastclassifier also combines adjacent *Classifier* elements when possible, creating a single fast classifier that does the work of both. This reduces the number of elements a packet must travel through, and is particularly useful in the presence of compound elements, where a user may create adjacent *Classifier*s without realizing it by connecting two compound elements together.

For example, the generated code for *Classifier(12/0800, –)*, a particularly simple classifier, contains the following function:

```
inline void
FastClassifier_a_ac::length_unchecked_push(Packet *p)
{
  const unsigned *data = (const unsigned *)(p->data() - 0);
 step_0:
  if ((data[3] & 65535U) == 8U) {
    output(0).push(p);
    return;
  }
  output(1).push(p);
  return;
}
```

The corresponding function in an unmodified *Classifier* contains a loop and many operations involving the decision tree structure:

```
void
Classifier::push(int, Packet *p)
{
  const unsigned char *packet_data = p->data() - _align_offset;
  Expr *ex = &_exprs[0]; // avoid bounds checking
  int pos = 0;
  // ...
  do {
    if ((*((unsigned *)(packet_data + ex[pos].offset)) & ex[pos].mask.u)
        == ex[pos].value.u)
      pos = ex[pos].yes;
    else
      pos = ex[pos].no;
  } while (pos > 0);
  checked_push_output(-pos, p);
}
```

The *click-fastclassifier* tool relies on Click itself to produce the decision trees. It starts a user-level Click driver, passes it a stripped-down version of

the router configuration, and calls *Classifier* read handlers that output the optimized decision trees in text form. This avoids a possibly buggy reimplementation of *Classifier*'s configuration string parsing and decision tree optimizations.

Click-fastclassifier both changes the configuration program and compiles new element classes. It depends intimately on the semantics of *Classifier*, but requires little graph analysis or manipulation.

Similar optimization strategies have been developed for BPF [4], DPF [17], and other packet filters. The Click language provides a generic context for this filter optimization work, showing that it exemplifies a class of optimizations—where a faster element is swapped in for a slower one. For example, a tool that generated specialized code for some other element class could share much of *click-fastclassifier*'s graph manipulation code.

An IP router configuration with fast classifiers takes 3% less CPU time to process a single packet than the original. The IP router's *Classifier* is very simple; *click-fastclassifier* is particularly effective on more complex classifiers. For example, consider the complex *IPFilter* of Figure 5.8 (page 72). This element takes 388 nanoseconds to process a packet matching rule DNS-5, more than five times the cost of many of the IP router's elements (see Table 8.2 on page 103). *Click-fastclassifier* speeds this up to 188 nanoseconds per packet. This 200 nanosecond-per-packet improvement is large enough to have impact on end-to-end performance metrics.

6.3 DEVIRTUALIZATION

The *click-devirtualize* tool optimizes configurations by removing virtual function calls from their elements' source code. Virtual function calls, or dynamic dispatches, are made through a table of function pointers; on a Pentium III, a mispredicted virtual function call takes dozens of cycles. Packet transfer between elements is implemented in Click with virtual function calls, which makes C++ element classes independent of how their elements are used. However, once the router configuration is known, every packet transfer virtual function call could conceivably be replaced by the relevant direct function call, since each element's position in the configuration is known. *Click-devirtualize* implements this optimization.

The tool reads a configuration program, then reads and partially parses the C++ source code for each element class used in that configuration. Finally, it generates one new C++ class per element in the configuration. This code is based on that element's original code, but direct function calls replace each virtual function call for packet transfer. For example, consider an element

whose first output port is connected to the first input port of a *Counter* element. Then code like this in the element's normal class definition:

```
Element *next = output(0).element();
// call goes through virtual function table
next->push(output(0).port(), packet_ptr);
```

is transformed by *click-devirtualize* into code like this:

```
Counter *next = (Counter *)output(0).element();
// call is resolved at compile time
next->Counter::push(0, packet_ptr);
```

Besides removing the virtual function call, *click-devirtualize* has inlined the next input port number. That is, the reference to '`output(0).port()`' has been changed to '`0`'. *Click-devirtualize* inlines other method calls as well, such as those that return how many inputs or outputs an element has.

While *click-devirtualize* can generate a new element class for every element, it usually does not, because even specialized elements can often share code. For example, all *Discard* elements can share code, since *Discard* throws away every packet it receives: there are no virtual function calls to remove. Because of this, two elements can share code if they have the same class—say, *Counter*—and they are each connected to a single *Discard* element. (The two *Discard* elements share a C++ implementation, so the *Counter*s' `push` virtual function calls both resolve to the same function, `Discard::push`.) Similarly, two elements with the same class, each connected to one of the two *Counter*s, can also share code, and so forth. Two elements *cannot* share code if any of the following properties is true:

1. The elements have different classes.
2. The elements have different numbers of input or output ports.
3. There exists an input or output port number where that port is push on one element, but pull on the other.
4. There exists an input port number where the corresponding ports are both pull, but the two elements connected to those ports cannot share code. (For example, the port is connected to a *Counter* on one element, but a *Strip* on the other.)
5. There exists an output port number where the corresponding ports are both push, but the two elements connected to those ports cannot share code.

A simple data flow algorithm based on the above rules calculates the elements for which code sharing is allowed. In our IP router configurations, such as Figure 5.1, analogous elements in different interface paths can always share code.

Virtual function calls do have some advantages. For example, infrequently executed paths may not be important enough to devirtualize—the i-cache cost of expanded source code may outweigh the performance savings of removing virtual functions. To address this, the user can tell *click-devirtualize* not to devirtualize certain elements.

Click-devirtualize can inline function calls as well as devirtualizing them. This can improve performance significantly, but it currently requires some hand intervention.

The devirtualization tool both changes the configuration program and compiles new element classes. It depends only on the push/pull properties of each element, not on more complex semantic properties; although it requires that every element's source code be available, it does not analyze or understand that source code in depth. It requires configuration graph analysis, both to discover the push/pull properties of the graph and to determine whether elements can share code. Both analyses use relatively simple data flow algorithms. It does not require substantial graph manipulation.

Devirtualization is a well-known technique in object-oriented programming languages such as Java. Mosberger et al. [31] demonstrate that *path inlining*, essentially devirtualization with inlining, is useful for decreasing protocol latency in a modular networking system (the *x*-kernel [22]), but they implement it by hand. To our knowledge, neither the *x*-kernel nor Scout [36] can implement devirtualization automatically.[5]

A devirtualized IP router configuration takes 23% less CPU time to process a single packet than the original.

6.4 DEAD CODE ELIMINATION

The *click-undead* tool removes dead code from a router configuration. Dead code, such as unused or useless elements, rarely occurs in configurations written entirely by hand. Configurations containing compound elements will sometimes contain dead code, however, as the user will tend to connect unneeded compound element inputs and outputs to *Discard* or *Idle* elements. This might, for example, connect one of a *Tee*'s branches to an *Idle*. This

5. Scout paths are automatically "specialized", but this optimization largely consists of removing unneeded queues. Click avoids unneeded queues a priori.

branch obviously serves no purpose except to slow down the configuration: it is dead code.

Click-undead first removes all *Null*, *StaticSwitch*, and *StaticPullSwitch* elements from the configuration. *Null* never occurs in real configurations, but *StaticSwitch* and *StaticPullSwitch* are useful for creating conditional branches in compound elements, as described in Section 3.4. *Click-undead* redirects connections around *StaticSwitch*, sending packets to the specified output port directly. This eliminates one packet transfer on the corresponding path and may create opportunities for further optimization.

Click-undead next decides which elements are live. An element is live if it is reachable from both a packet source element and a packet sink element. Sources and sinks are explicitly marked as such by element designers; sources include *InfiniteSource*, *FromDevice*, and *FromDump*, while sinks include *TimedSink*, *ToDevice*, and *Discard* (when used as a pull element). Discovering liveness requires a simple data flow analysis to spread reachability downstream from sources and upstream from sinks. When this is complete, any dead elements are removed. Elements can be explicitly marked as live. Information elements, for example, are always live, so they are not removed even though they are never reachable.

Next, the tool removes unused ports on scheduler elements and tees. This saves the time that would be spent copying a packet only to throw it away, or checking upstream for a packet that could never exist. A port is considered unused if everything connected to it was dead. *Click-undead* also removes redundant scheduler elements and tees—that is, schedulers or tees with only one input and one output.

Dead code elimination works entirely at the language level. It does require some knowledge of element semantics—namely, source, sink, and liveness information for some elements, and more detailed knowledge about the switches, schedulers, and tees specifically mentioned above. It relies on relatively simple data flow analysis and graph manipulations.

None of the configurations described in this thesis contain dead code, so we do not measure *click-undead*'s effect on performance. *Click-undead* becomes useful only for configurations with large numbers of compound elements.

6.5 PACKET DATA ALIGNMENT

The data contained in a Click packet is stored in a flat array of bytes. However, many elements require that data to be aligned on a word boundary. For example, elements such as *CheckIPHeader* expect the IP header to be word-aligned. Other elements, such as *Classifier*, can adapt to any single alignment,

Before

```
RatedSource(...)
  -> Strip(14)
  -> c :: Classifier(...)
  -> CheckIPHeader
  -> ...;
```

After

```
RatedSource(...)
  -> Align(4, 2)
  -> Strip(14)
  -> c :: Classifier(...)
  -> CheckIPHeader
  -> ...;
AlignmentInfo(Strip@2 4 2,
  c 4 0,
  CheckIPHeader@4 4 0,
  ...);
```

FIGURE 6.4—The *click-align* tool at work on a simple configuration.

but require that every packet have the same alignment. On many architectures, incorrect packet data alignment will cause the entire router to crash.

The alignment of a packet's data depends on both its initial alignment and on any modifications that happened on its path through the router. For example, the *Strip* element strips a specified number of bytes from the packet's header by bumping a pointer; a packet's alignment after passing through *Strip(1)* will be different than its alignment before. The *Align* element fixes misaligned packets by copying them. Its configuration string contains an alignment specification, such as '*4, 0*' to request word-aligned packets. It copies every passing packet whose alignment does not match that specification, fixes its alignment, and emits the fixed packet.

It is clearly too expensive, in packet copies and in sheer number of elements, to place *Align*s everywhere. The packet data alignment tool, *click-align*, improves on this by globally calculating both the required and expected alignments at every point in the configuration. It can then insert a small number of *Align*s exactly where they are needed.

Click-align calculates alignments using a data flow analysis resembling availability analysis [1]. The analysis takes the alignment effects of elements like *Strip* into account. The tool then inserts *Align* elements at every point where the existing alignment is incorrect. After rerunning the analysis, it removes redundant *Align* elements and adds an *AlignmentInfo* element, which informs every other element of the alignments they can expect. Our IP router configuration requires no *Align* elements, but the tool is necessary anyway, as it provides the *Classifier* element with alignment information.

Figure 6.4 shows *click-align*'s effects on a simple configuration. The *RatedSource* element produces packets with the wrong alignment, so the tool adds

a single *Align* element. The *AlignmentInfo* element informs the *Classifier* `c` that its received packets will be word-aligned (‘`4 0`’).

On x86-based machines, the *click-align* tool is optional—unaligned accesses are legal, and not any slower than aligned accesses. (An unaligned access can cause two d-cache misses if it straddles two cache lines, but this is not a problem in practice, as the packet data is always in the cache.) On other architectures, however, configurations that have not been checked with *click-align* cause Click to crash.

The alignment tool works entirely at the language level and requires knowledge of the alignment-related semantics of each element. Its data flow analyses are relatively complex—for example, two different analyses are required to discover what alignments elements need and what alignments they can expect. However, its graph manipulations are simple. It has no effect on performance unless you count that on some architectures, unaligned configurations do not work at all.

6.6 MULTIPLE-ROUTER CONFIGURATIONS

The *click-combine* tool can combine several router configurations into one larger configuration that gives a detailed picture of the overall network. For example, if the output interface of one router is connected to the input interface of another by a point-to-point link, then the combined configuration will have explicit connections between the relevant interfaces. This unified Click graph can be analyzed and manipulated using language techniques like the ones described above, ensuring properties of the network—for instance, that the frame formats at either end of each link are compatible—or removing redundant computation performed by more than one router. The *click-uncombine* tool can then separate the combined configuration into its component router parts. For example, Figure 6.5 shows part of a combined router generated by *click-combine*. Router A’s *ToDevice(eth0)* was connected by a point-to-point link to router B’s *PollDevice(eth1)*.

One optimization enabled by router combination is ARP elimination. If two routers are connected by a point-to-point Ethernet link, there is no need to have a full ARP mechanism in place on that link; each router can be preprogrammed with the Ethernet address of the other. The ARP elimination optimization removes *ARPQuerier* and *ARPResponder* elements from a configuration, replacing them with the corresponding *EtherEncap* element. Another optimization might remove redundant *IPFragmenter* elements. Each link has a hard-wired maximum transmission length, or MTU; *IPFragmenter* breaks packets larger than this length into fragments before sending them.

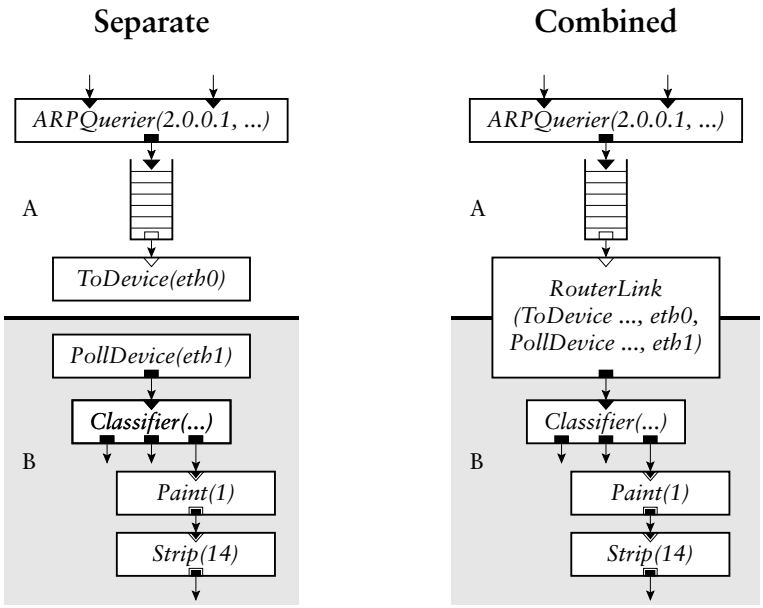

FIGURE 6.5—Portions of two routers, A and B, and the corresponding combined configuration generated by *click-combine*.

However, if the links upstream of a router have the same or smaller MTU as its downstream links, then *IPFragmenter* is unnecessary, as any arriving packets will already be small enough to fit.

Of course, these optimizations are inherently dangerous. They produce correct results only if the user correctly and completely represents every router attached to a particular link as a Click configuration. Furthermore, the user must re-run the optimizations every time the devices on a link are changed: if a cable is unplugged, the optimizations should be run again. These requirements are only realistic when all routers on a link are controlled by the same administrative entity, or when the routers automatically exchange information about their configurations. A safer use for *click-combine* and *click-uncombine* is to check for properties like loop freedom.

For completeness, we measured the performance impact of ARP elimination. The most general form of ARP elimination would require complex data flow analyses to ensure that the required properties hold in the configuration graph; we did not implement these analyses, but instead used simple *click-xform* transformation patterns to capture the common case. The impact was small—an IP router with ARP eliminated via *click-combine* and *click-xform* takes just 3% less CPU time to process a single packet than the original.

Click-combine and *click-uncombine* work entirely at the language level

and do not need to understand element semantics. Their graph manipulations and analyses are extensive, but simple. Individual multiple-router optimizations like ARP elimination resemble some kinds of programming language optimizations, such as interprocedural optimizations.

6.7 DISCUSSION

The partial information that some language tools need about element classes is often quite simple. For example, several tools only need to know whether ports are push or pull. Extracting this information from the C++ implementation would be a difficult, if not impossible, task. Instead, we supply several parallel definitions for each element class. For example, each element has a C++ definition and a push/pull definition; some elements have additional definitions as required by the tools. In this approach, the user creates simple, high-level specifications for each element and each property required. These specifications are focused and easy to write, and they facilitate incremental specification—the behavior of an element is specified as tools are built. However, they force the user to write many specifications for each element, and as new tools become available, old element classes may need to be updated with new specification information. Furthermore, the additional specifications are not required by the drivers, so element designers may tend to put off writing them.

One could instead write element implementations in a language that facilitated analysis, then extract all necessary information from those implementations. For example, in the Ensemble system for composing network protocol stacks [25], components are written in Objective Caml. However, despite its relative convenience for analysis, Objective Caml is still so general that a complex theorem prover and extensive human intervention are required to analyze Ensemble components. Click's design avoids these costs.

Tools are relatively simple to write, but certainly not as simple as elements. A tool library has functions for reading configurations, flattening them to remove compound elements, searching configuration graphs for specific elements or connection patterns, and so forth. Writing a tool is nontrivial despite this. After all, the Click language is still a programming language, however simple, and tools have some similarities to compiler passes. Manipulating any program is a difficult task because of the exhaustive analysis required to handle all cases, and the compiler knowledge required to design a tool is significantly greater than the system knowledge required to design an element. A higher-level library that made it easier to develop new tools would be useful.

7
Implementation

This chapter describes how the Click kernel driver was implemented, including the changes to Linux required to achieve good performance.

The Click kernel driver runs as a kernel thread under Linux 2.2. The kernel thread loops over the current router's task queue and runs each task. Only interrupts can preempt this thread, but to keep the system responsive, it voluntarily gives up the CPU to Linux from time to time; the user can specify how often with a *ScheduleLinux* information element. There is generally only one Click 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 driver 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 the `/proc/click` directory export information about the current configuration, a list of element classes known to the driver, and so on. When installing a router configuration, Click creates a subdirectory under `/proc/click` for each element. This subdirectory contains that element's handlers (Section 2.8). The code that handles accesses to `/proc` runs outside the Click thread in the context of the reading or writing process.

Besides packets, important objects in a running Click router include:

– **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. Element prototypes can be added to and removed from the running driver.

– **Router.** The single router object collects general information relevant to a given router configuration. It configures the elements, checks that con-

nections are valid, and puts the router on line; it also manages the task queue.

– **Timers.** In the kernel driver, Click timers are implemented with Linux timer queues. On Intel PCs, these have .01-second resolution.

7.1 POLLING AND DEVICE HANDLING

The original Click system [29] 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 [28]: as the number of input packets increased, interrupt processing eventually starved all other system tasks, leading to reduced throughput.

Click now eliminates interrupts in favor of polling. A new device-handling element, *PollDevice*, is the polling equivalent of *FromDevice*; it puts the relevant device into polling mode and puts itself on Click's task queue to wait for packets. When activated, *PollDevice* polls its device's receive DMA queue for newly arrived packets; if any are found, it pushes them through the configuration. *ToDevice* also places itself on the task queue. When activated, it 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.[6] Mogul and Ramakrishnan [28] 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 to the drivers for our network devices. 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.

6. The *FromDevice* element still uses interrupts, however. With *FromDevice*, Click can use devices whose drivers don't support polling. On an interrupt, *FromDevice* stores packets in an internal ring buffer. Like *PollDevice*, it places itself on the router's task queue. It then outputs packets from that ring buffer as it is activated. This avoids race conditions in configurations containing both *FromDevice* and *PollDevice*, but does make *FromDevice*-only configurations slower than they were before. Our test configurations all use *PollDevice*.

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 Osiris network controller [14]. 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 [28] 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.

8

Evaluation

This chapter 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, and demonstrate the practical effectiveness of the language tool optimizations.

Our goal is to understand the performance of a particular Click router, the performance limitations of the Click architecture, and the performance characteristics of our PC hardware. We will show that the Click architecture's only significant performance limitation is the cost of virtual function calls, and that the language tool optimizations successfully address this cost. We also argue that our PC hardware is limited, by either the PCI bus or the memory system, to a sustained forwarding rate of about 400,000 64-byte packets per second, and show that our optimized IP router configurations easily achieve this rate. Finally, we show that Click can enforce quality-of-service constraints very well under ordinary loads, and reasonably well even under overload.

An unoptimized Click IP router achieves a maximum loss-free forwarding rate of 357,000 64-byte packets per second. With language tool optimizations, this rate rises to 446,000 packets per second. For comparison, a common edge router, the Cisco 7200 series, can route 200,000 packets per second. The cost of this router is about a factor of ten more than the cost of the hardware we used. (Of course, the two routers are not really comparable. Cisco router hardware is presumably more reliable than PC hardware. Furthermore, the Cisco router has functionality not currently available in Click—such as routing protocol support—although most of this functionality is probably not on the forwarding path.)

8.1 EXPERIMENTAL SETUP

The experimental setup consists of a total of nine Intel PCs running a modified version of Linux 2.2.16. (We patched Linux 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 router host has a 700 MHz Intel Pentium III CPU and an Intel L440GX+ motherboard. Its eight DEC 21140 Tulip 100 Mbit/s PCI Ethernet controllers [13] are on multi-port cards split across the motherboard's two independent PCI buses. The Pentium III has a 16 KB level-1 instruction cache, a 16 KB level-1 data cache, and a 256 KB level-2 unified cache. The source and destination hosts have 733 MHz Pentium III CPUs and 200 MHz Pentium Pro CPUs, respectively. Each host 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 discard 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.

This setup is easier on routers than a real network might be. The tests did not involve fragmentation, IP options, or ICMP errors, though Figure 5.1 has all the code needed to handle these. More destination hosts might slow Click down by increasing the number of ARP table entries. A larger routing table might also slow down Click's simple routing table implementation; previous work on fast lookup in large tables has addressed this problem [12, 51]. The setup is nevertheless sufficient to analyze the architectural aspects of Click's performance.

Click's modularity is as convenient for measuring performance as it is for building routers. Our source hosts generate packets using the Click configuration described in Section 5.6; the destination hosts use another Click configuration to receive them. Particular measurements made use of Click elements that check Pentium III cycle counters and performance counters, and that poll Tulip network interface status. All of these elements are distributed with the Click software.

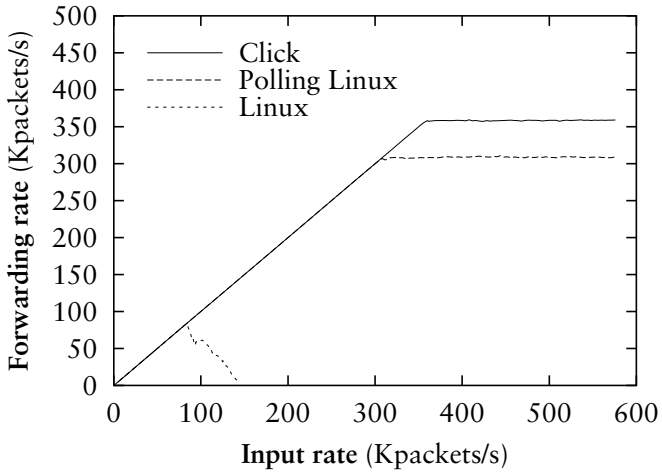

FIGURE 8.1—Forwarding rate as a function of input rate for Click and Linux IP routers (64-byte packets). The "Linux" line used one sender-receiver pair (theoretically 148,800 packets/s maximum), while the other lines used four sender-receiver pairs (theoretically 595,200 packets/s maximum).

8.2 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 $y = x$. This section focuses on maximum loss-free forwarding rates for a number of Click router configurations, including optimized IP routers. The next section turns to router behavior under overload.

Figure 8.1 shows the performance of the Click IP router—namely, the IP router configuration of Figure 5.1 extended to a total of eight network interfaces. This configuration has a total of 161 elements, 19 elements per interface plus 9 elements shared by all interfaces. The "Click" line shows this router's performance with eight interfaces. To provide context, 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 357,000 packets per second.

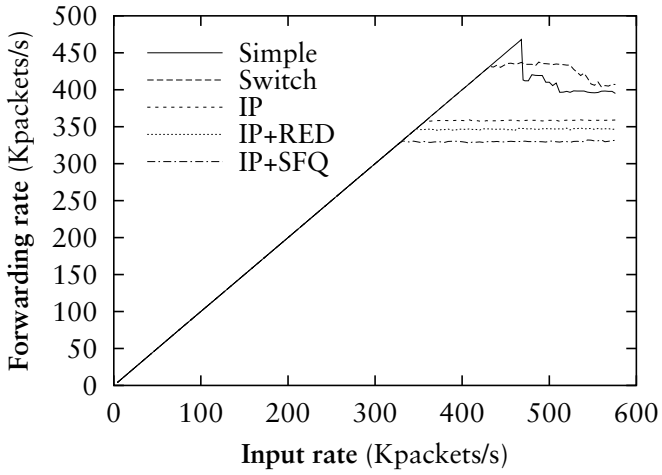

FIGURE 8.2—Forwarding rate as a function of input rate for extended IP router configurations and some non-IP routers (64-byte packets).

Click was always fair during this test: the router forwarded an equal number of packets for each sender-receiver pair.

Polling Linux's MLFFR is 308,000 packets per second. This is clearly comparable to Click's MLFFR, and shows that Click's modularity has acceptable cost when compared to conventional router software. Polling Linux's MLFFR may be less than Click's because of the Linux routing table implementation, which is slower than *LookupIPRoute*, but more scalable. Standard Linux's MLFFR is significantly less than either Click's or Polling Linux's because of interrupt-driven receive livelock.

Figure 8.2 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 via a *Queue* with no intervening processing. Its MLFFR is 468,000 packets per second; its overload behavior is analyzed in the next section, but is basically due to the configuration being PCI or memory limited rather than CPU limited. The "Switch" line corresponds to the Ethernet switch configuration of Figure 5.6. The "IP" line, repeated from Figure 8.1, shows the performance of the base IP router configuration. The "IP+RED" configuration is 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" configuration is an IP router with each *Queue* replaced by a stochastic fair queue—namely, a version of Figure 4.4 with four component *Queue*s. As router complex-

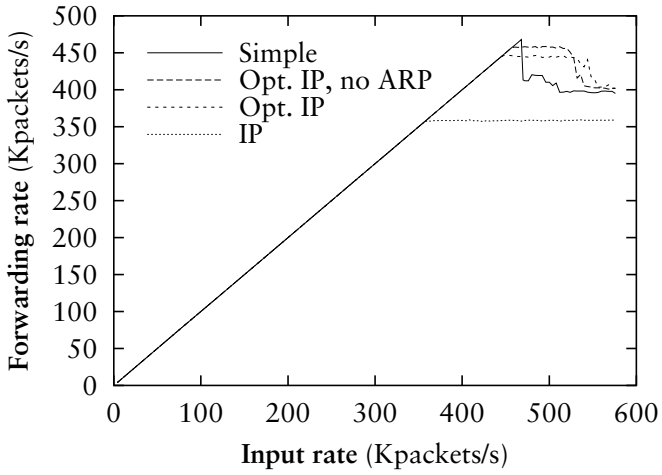

FIGURE 8.3—Forwarding rate as a function of input rate for optimized IP router configurations (64-byte packets). "Opt. IP" is the IP router transformed by *click-xform*, *click-fastclassifier*, and *click-devirtualize*. "Opt. IP, no ARP" adds ARP elimination as well.

ity increases from simple forwarding through IP routing to IP routing with stochastic fair queueing, Click's performance drops only gradually.

Figure 8.3 illustrates the effectiveness of the language-based optimizations described in Chapter 6. The "IP" and "Simple" lines are repeated from Figure 8.2. The "Opt. IP" configuration is an IP router transformed by three optimization tools: *click-xform* with the patterns of Figure 6.3, *click-fastclassifier*, and *click-devirtualize*. The "Opt. IP, no ARP" configuration is "Opt. IP" additionally transformed to eliminate ARP queries, as described in Section 6.6. "Opt. IP" has a maximum loss-free forwarding rate of 446,000 packets per second, which is 25% more than the base IP router. Thus, our language-based optimizations have a significant impact on the performance of IP routing. Eliminating ARP adds 11,000 more packets per second to this MLFFR, which is perhaps not enough of an improvement to justify this risky optimization.

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.

8.3 OVERLOAD BEHAVIOR

Before analyzing these routers' performance under overload, we must closely examine how packets move through a Click router's network interfaces and CPU. Figure 8.4 describes, at a high level, the paths our test packets take while traveling through our router. Each packet arrives on a receiving Tulip card and is written to memory. Then the CPU reads the packet from memory and pushes it to a *Queue* in the configuration. After spending some time in the *Queue*, the packet is pulled out and placed on a DMA queue for the transmitting Tulip card. The transmitting Tulip card reads the packet from memory and sends it. In other tests, packets might take paths not illustrated in the figure; the Click configuration's *CheckIPHeader* will drop packets with erroneous IP headers, and several Tulip error conditions, such as transmission aborts, never arise in our experiments. We derived these flowcharts from Tulip documentation [13], code inspection, and experiments.

Each packet's path ends in one of four different outcomes, which are shown in the figure as boxes with heavy borders. Packets may be dropped on the receiving Tulip card because the Tulip is out of memory ("FIFO overflow") or if there are no receive DMA descriptors available after two tries ("missed frame"). The packet may be dropped at the Click *Queue* when packets are arriving faster than they can be sent ("*Queue* drop"). Every packet that survives these obstacles is sent ("packet sent"). If a packet is to be dropped, it is best to drop it at the receiving card's FIFO, as this early drop point avoids wasting resources. In particular, it avoids all PCI bus and memory operations, which are shown in the figure as shaded boxes.

All of these events can be measured. The Tulip cards provide counters that report missed frames and FIFO overflows, if polled frequently; Click's *Queue* has a handler that reports how many packets it has dropped; and the test's receiver machines report how many packets they received, which is equivalent to the number of packets the router sent. Therefore, we can account for every packet received by the router. We measured these packet outcomes for several router configurations over a range of input rates. Figure 8.5 on page 101 shows the results. The solid line in each graph corresponds to the forwarding rate. The other lines cumulatively add other drop outcomes to the forwarding rate. The sum of all outcomes is the input rate—that is, the straight line $y = x$.

The baseline IP router configuration is clearly CPU-limited. All of its input packets are either forwarded or dropped as missed frames. Missed frame drops occur when a Tulip card unsuccessfully searches for an open DMA descriptor twice. This indicates that the CPU is emptying and refilling the relevant receive DMA ring slower than the Tulip card is filling it—that is, the

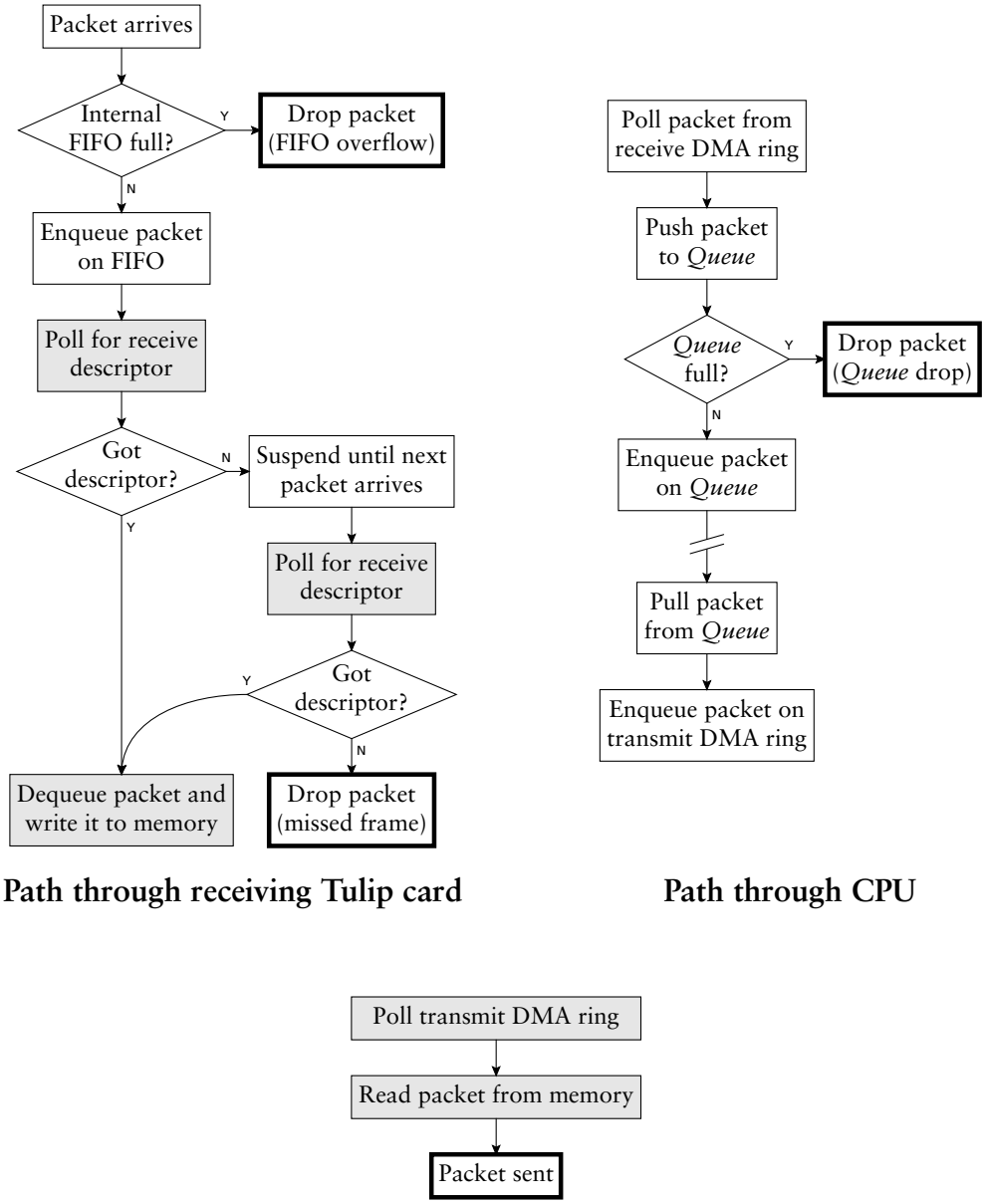

FIGURE 8.4—Flowcharts showing the paths our test packets take while passing through a Click router with Tulip network interface cards. Packet outcomes are shown as boxes with thick borders; shaded boxes are steps that use the PCI bus.

CPU is overloaded, so the relevant *PollDevice* is running infrequently.

The "Simple" configuration is not CPU-limited. None of the packets dropped by "Simple" are missed frames; they are either FIFO overflows or *Queue* drops. Both of these outcomes indicate that the PCI bus or memory system is overloaded. FIFO overflows occur when a receiving Tulip card gets packets faster than it can request DMA descriptors and write packets to memory. *Queue* drops occur when packets are added to a *Queue* faster than *ToDevice* removes them. Instrumenting the *ToDevice*s revealed that they were often idle—that is, they chose not to pull packets—because their devices' transmit DMA queues were full. Thus, the CPU wanted to send packets faster than the transmitting Tulip cards could process them. These possible bottleneck tasks—reading packets from memory, writing packets to memory, and requesting DMA descriptors—are limited not by the CPU, but by the PCI bus or the memory system. Our analysis is not detailed enough to determine the bottleneck resource precisely. Other researchers have suggested in personal communication that the memory system is a likely candidate—in particular, that validating data received from the PCI bus against the CPU's caches is an expensive operation.

The "Simple" forwarding rate drops when *Queue*s begin dropping packets. This happens soon after the *ToDevice* elements begin noticing full transmit DMA queues. The forwarding rate may drop, rather than holding steady, because each packet dropped at a *Queue* has already consumed some critical resources—namely, the PCI or memory bandwidth required to fetch a receive descriptor, write the packet into memory, and read it from memory into the CPU's cache. Thus, dropping the packet at a *Queue* is worse than never receiving the packet at all. This waste causes livelock. The forwarding rate flattens out again when packets begin dropping due to FIFO overflows. These packets have consumed no critical resources, so they don't cause livelock.

The "Optimized IP, no ARP" configuration is CPU-limited over a range of input rates, but then becomes PCI- or memory-limited. At input rates up to 500,000 packets per second, all dropped packets are due to missed frames, indicating that the CPU is the bottleneck. The forwarding rate is flat for a time, but then begins to drop. Again, this may be due to livelock—while missed frames are not as wasteful as *Queue* drops, they do exhaust some PCI bandwidth by requesting useless receive descriptors. At 500,000 packets per second, however, FIFO overflows start to occur. This indicates that the CPU is no longer the bottleneck. The Tulip card's FIFO is filling up, so it must be receiving a decent number of receive descriptors; otherwise, the FIFO would empty due to missed frames. Once the Tulip's FIFO is full, the forwarding rate flattens out, again to around 400,000 packets per second. All four fast router

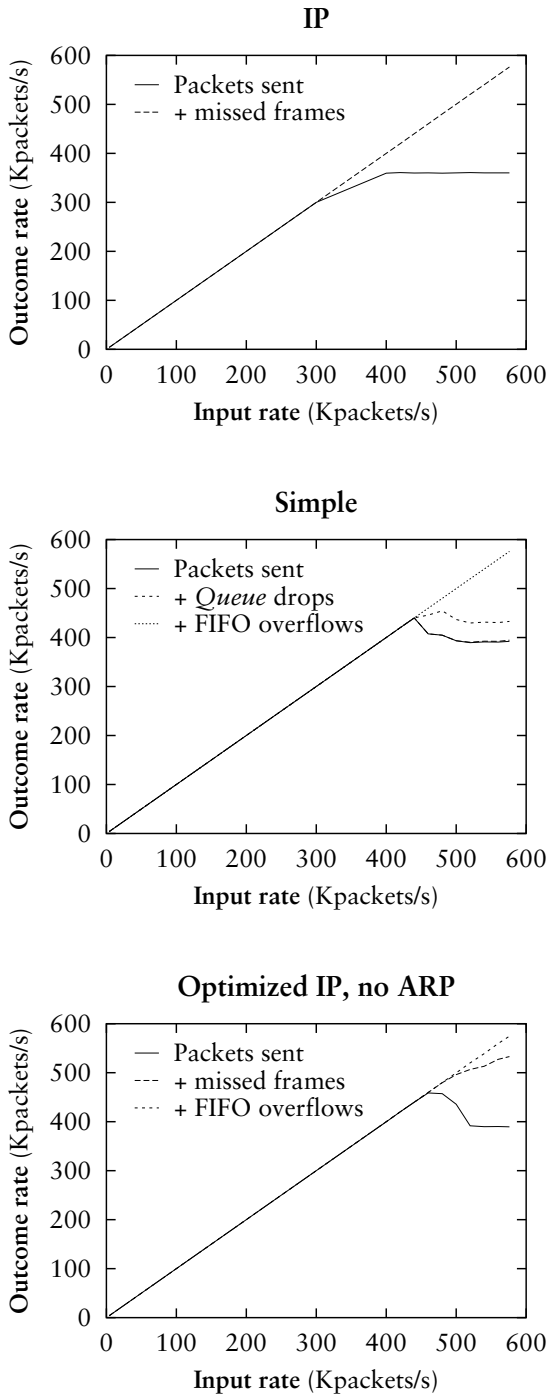

FIGURE 8.5—Cumulative outcome rates as a function of input rate for three Click router configurations.

Task	**Time** (ns/packet)
Polling packet	562
Refill receive DMA ring	139
Click forwarding path: push	1572
Click forwarding path: pull	85
Enqueue packet for transmit	135
Clean transmit DMA ring	412
Total	**2905**

TABLE 8.1—CPU time cost breakdown for the Click IP router.

configurations—"Simple", "Switch", and the optimized IP routers—flatten out to this rate after dropping. Because FIFO overflows, which do not use any critical resources, eventually occur whenever PCI or memory bandwidth is overloaded, it seems possible that PCs using Tulip cards may be able to achieve forwarding rates of about 400,000 64-byte packets per second even under loads much higher than those we tested.

8.4 CPU TIME BREAKDOWN

Table 8.1 breaks down the CPU time cost of forwarding a packet through the baseline Click IP router of Figure 5.1. Costs were measured in nanoseconds by Pentium III cycle counters [23]. Each measurement is the accumulated cost for all packets in a 10-second run divided by the number of packets forwarded. These measurements are larger than the true values as using Pentium III cycle counters has significant cost. Most of the tasks performed by a Click router's CPU are included in the table, except for the overhead of the Click task scheduler.

"Polling packet" measures the time it takes *PollDevice* to read packets from the Tulip's receive DMA ring. *PollDevice* then adds new descriptors to the receive DMA ring so that the Tulip may receive more packets; this cost is measured by "Refill receive DMA ring". The two "Click forwarding path" costs measure how much time it takes a packet to traverse the IP router's configuration graph. The "push" cost involves 15 elements, starting after *PollDevice* and ending at a *Queue*. The "pull" cost 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 placing a packet on the Tulip's transmit DMA ring.

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

TABLE 8.2—Execution times of some of the individual elements involved in IP forwarding.

ToDevice must also remove transmitted packets from the DMA ring; this cost is measured by "Clean transmit DMA ring". The Click push forwarding path is by far the most expensive task. Overall, Click code takes 60% of the time required to process a packet; device code takes the other 40%.

Table 8.2 partially breaks down the 1.57 μs Click 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 transfer 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 transfer.

The total cost of 2905 ns measured with performance counters implies a maximum forwarding rate of about 344,000 packets per second, which is within 4% of the observed MLFFR of 357,000 packets per second.

Pentium III performance counters report that forwarding a packet through Click incurs five level-2 data cache misses: 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

Configuration	Time (ns/packet) Push	Pull	Click	% of IP	Total Time	% of IP
Simple	97	91	187	11%	1715	59%
Switch	958	83	1040	63%	2324	80%
Optimized IP, no ARP	991	69	1061	64%	2256	78%
Optimized IP	1033	68	1101	66%	2414	83%
IP + *click-xform*	1093	81	1174	71%	2480	85%
IP + *click-devirtualize*	1207	67	1274	77%	2532	87%
IP + ARP elimination	1516	84	1600	97%	2809	97%
IP + *click-fastclassifier*	1536	75	1611	97%	2819	97%
IP	1572	85	1657	—	2905	—
IP + RED	1711	80	1792	108%	3091	106%
IP + SFQ	1665	230	1895	114%	3102	107%

TABLE 8.3—CPU time costs per packet for several router configurations. "Optimized IP" is the IP router transformed by *click-xform*, *click-fastclassifier*, and *click-devirtualize*. "Optimized IP, no ARP" adds ARP elimination as well.

reusable packets. Each of these loads takes about 112 nanoseconds. Click runs without incurring any other level-2 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 level-1 instruction cache.

Table 8.3 shows Click push and pull costs and total CPU time costs of the other router configurations mentioned in Section 8.2. The "Click" column is the total Click cost, push and pull; the "Total Time" column is the total CPU time cost, including both Click and the four device-related tasks.

When applied to the IP router configuration, the language tool optimizations mostly reduce the cost of pushing a packet through the Click forwarding path. Applying all router-local optimizations—*click-fastclassifier*, *click-devirtualize*, and *click-xform*—reduces this cost from 1657 ns to 1101 ns. The optimizations have significant effects even when device drivers are included, as shown by the table's last columns.

Pattern replacement with combination elements (*click-xform*) is the most effective individual optimization, reducing the Click cost by 30%. Devirtualization is also quite effective on its own, reducing that cost by 23%. These optimizations address the same problem, the number of packet transfers on the path, so applying them together is not much more useful than applying either one alone.

Although the fast classifier optimization reduces the cost of the push path only slightly (to be precise, by 36 cycles), it improves the performance of the

IP router's *Classifier* element by an average of 24%. This optimization is more effective on configurations with complex *Classifier*s, such as the IP firewall of Section 5.4.

8.5 ARCHITECTURAL OVERHEAD

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 a simplified agnostic-port interface that causes two virtual function calls per packet transfer; the CPU usually predicts one call but doesn't predict the other. As a result, the total cost of packet transfer is about 70 nanoseconds, 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 can be avoided with the *click-devirtualize* tool.

Several elements in the IP router configuration, such as *Classifier*, are implemented with more generality than the IP router requires. We used the *click-fastclassifier* tool to analyze the cost of this generality; it produces an element specialized for the IP router's classification task, including straight-line code and compiled-in constants. The fast classifier element cost 24% less CPU time per packet than *Classifier*, but even *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 used the combination elements described in Section 6.1. 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. Again, one element implements the functionality of *Paint*, *Strip*, *CheckIPHeader*, and *GetIPAddress*; the other, the functionality of *DropBroadcasts*, *PaintTee*, *IPGWOptions*, *FixIPSrc*, *DecIPTTL*, and *IPFragmenter*. The forwarding path of the transformed configuration has only eight elements instead of sixteen. Its push path processes an IP packet in 1.03 μs instead of 1.57 μs.

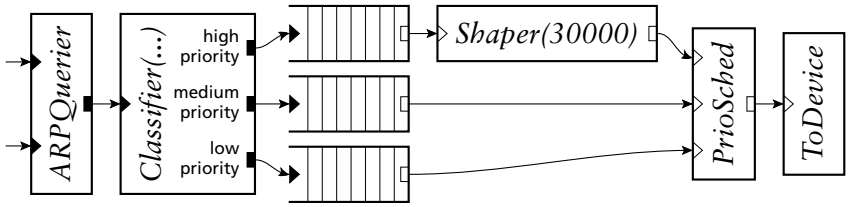

FIGURE 8.6—A Click diffserv traffic conditioning block with three priority levels.

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.

8.6 DIFFERENTIATED SERVICES

Section 4.5 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*. Our simple test policy has three priority levels, where the highest priority is additionally rate limited to no more than 30,000 packets per second. Figure 8.6 shows a configuration fragment that implements this policy. To test this configuration, we inserted it into the IP router of Figure 5.1 and measured per-priority forwarding rates at a range of input rates. Three source hosts send equal numbers of packets marked as high, medium, and low priority respectively. The router has a single 100 Mbit/s Ethernet output link which all priority levels must share.

Figure 8.7 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

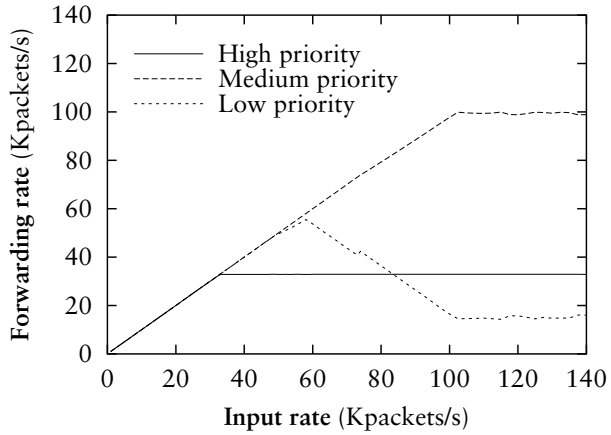

FIGURE 8.7—Performance of the Click diffserv configuration in Figure 8.6. Three traffic sources send 64-byte UDP packets with three different priorities into a router with a single 100 Mbit/s output link.

carry all packets to the point where the priority decision is made. A Click user can approximate such a design by moving the classification and per-priority queues of Figure 8.6 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.

8.7 SUMMARY

We have shown that a modular Click router can achieve performance near the limits of high-end PC hardware. Of its sources of overhead, only virtual function calls have a significant impact on performance, and language tools described in Chapter 6 can mitigate or eliminate their impact. A Click IP router optimized with these tools can achieve a peak forwarding rate of 446,000 64-byte packets per second, 95% of the apparent hardware bound. This router's maximum sustained forwarding rate is roughly 400,000 64-byte packets per second, enough to handle several T3 lines. Measurements of extended Click configurations show that incremental extensions have incremental costs. Finally, Click can enforce quality-of-service constraints—namely, priorities—reasonably well even under overload.

9

Related work

The *x*-kernel [22] 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 [30, 36] 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 [42] 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 [16] whose nodes 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 [6], 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 [10, 11] 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 [32, 35] 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 [44, 48]. 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 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 [14, 27, 28, 53].

10

Conclusion

Click routers are modular, extensible, and flexible. Elements, the building blocks of router functionality, can support wide varieties of tasks through composition. Click configurations for real routers, such as a standards-compliant IP router, are readable and extensible. We showed how adding support for scheduling and dropping policies, complex queueing, and transparent Web proxies can require simply adding a couple elements to the IP router, and how IP router elements are useful for other packet processing applications. Router configurations are written in a simple, declarative language. Careful language design means that router configurations are easy to process with tools, which can optimize configurations, check them for errors, or ensure they satisfy simple invariants. Our performance analysis shows that Click's modularity is compatible with good forwarding performance for PC hardware, and that our optimization tools are effective, reducing the Click IP router's per-packet CPU time by 34%. The optimized Click IP router's steady-state forwarding rate is 400,000 minimum-size packets per second; its peak forwarding rate is 95% the hardware's apparent capacity. Last but not least, Click is fun to program.

Several universities are already using Click for research. At the University of Utah, researchers adopted it as a prototypical component-based system against which they compared their own component architecture [41]. Students at Purdue University are modifying it to better support multicast and active networking [20]. The Mescal project at the University of California, Berkeley used the Click IP router as the definition of IP routing, upon which they are building their own modular router in a hardware-focused setting; they appear to have chosen Click because its high-level description of IP routing is so easy to understand [43]. At MIT, Click is in use as a mixed simulation/execution environment for the Grid ad-hoc routing system [9].

FUTURE WORK

Since the optimized Click IP router appears close to the performance limits of our hardware, future performance work must focus on breaking these limits.

For instance, it might be a pleasant exercise to compile Click configurations for real router hardware—FPGAs or ASICs. Since elements tend to have compact specifications, they might correspond naturally to short definitions in some hardware description language. A Click-language tool might read in configurations and write out the corresponding hardware descriptions.

Of course, some configurations are still CPU-limited. Click currently runs in a single thread on a single processor. Taking advantage of a symmetric multiprocessing platform might speed Click up dramatically. Its element structure may facilitate dividing a configuration into regions, each of which could be active on a different processor.

The tools described in Chapter 6 are relatively simple; they cannot, for example, prove that every packet reaching a certain point in a configuration is an IP packet. Tools should be written that check configurations for such high-level correctness properties. These properties are difficult to check and require more information about element semantics than the tools we have now. The current system of writing tools based on informal element semantics has many advantages, including that writing tools and elements is easy and requires no theorem-proving expertise. Nevertheless, as tools are written that establish more complex properties, a parallel language for expressing element semantics in an integrated fashion may begin to make more sense.

SUMMARY

Click's philosophy of components, a declarative language for combining components, and tools manipulating that language has been a success. The three parts of the system—components, language, and tools—are in good balance. The system works well as a whole, with each part becoming more powerful because of the others; for example, information elements are more useful because tools can add them automatically, and compound elements are more powerful because elements like *ScheduleInfo* were designed to support them. Click appears easy to learn, as witnessed by its use outside MIT. The most important decision in its creation, I believe, was the initial division of the system into components and language. Without that early division, it would have been tempting to make either the component architecture or the language more complicated than was necessary. With the early division, complexities in the component architecture were resolved with help from the language, and vice versa, resulting in a simple, natural, and usable system. I would like to build more systems this way.

A

Element glossary

This section lists 59 of the approximately 150 elements included with version 1.1 of the Click software. Each entry follows this format:

ElementName(configuration arguments) Port types (push, pull, or agnostic). Port descriptions (packet types and numbers of ports). Description. (Optional reference to section with more detailed description.)

Some elements also have a reference to a section where more information can be found. Detailed manuals for each element are accessible on line at `http://www.pdos.lcs.mit.edu/click/doc/`.

AddressInfo(name addrs, ...) No inputs or outputs. Creates shorthand names for IP and Ethernet addresses and IP network prefixes. (§4.7, p 61)

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.

CheckIPHeader2(...) Agnostic. Input takes IP packets. Discards packets with invalid IP length or source address fields; forwards valid packets unchanged. Unlike *CheckIPHeader*, does not check the IP checksum.

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*. (§4.2, p 45)

ControlSocket(type, addr) No inputs or outputs. Creates a socket and listens for connections on that socket. Speaks a simple protocol allowing remote machines to query read handlers and set write handlers. Only available at user level.

Counter Agnostic. Input takes any packet; forwards packets unchanged. Its read handlers provide information about the number of packets passed and their arrival rate.

DecIPTTL Agnostic, but second output is push. 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.

EtherEncap(p, ethersrc, etherdst) Agnostic. Input takes any packet, output emits Ethernet packets. Encapsulates input packets in an Ethernet header with protocol *p*, source address *ethersrc*, and destination address *etherdst*.

EtherSpanTree Agnostic. Inputs take IEEE 802.1d messages. Implements the IEEE 802.1d spanning tree algorithm for Ethernet switches. (§5.3, p 70)

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. (§5.3, p 69)

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. Use *FromDevice* for devices that do not support polling. See also *PollDevice*.

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 emitted from *FromLinux*'s single output as they arrive from Linux.

FrontDropQueue(n) Push input, pull output. Input takes any packet. Stores packets in a FIFO queue; maximum queue capacity is *n*. When a new packet arrives on a full queue, the packet at the head of the queue is dropped rather than the new packet. See also *Queue*. (§2.5, p 21)

FTPPortMapper(...) Agnostic. Input takes TCP/IP packets representing an FTP control connection. Forwards most packets unchanged, but looks for PORT commands embedded in passing packets. These commands are rewritten according to an *IPRewriter* pattern. (§4.6, p 57)

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.

HashSwitch(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 [*offset*, *offset* + *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.

Idle Agnostic. Any number of inputs and outputs. Discards all packets as they arrive, and always returns a null pointer in response to pull requests.

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* and *IPFilter*. (§4.2, p 46)

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*.

IPFilter(...) Push. Input takes IP packets. Examines packet data according to a set of classifiers, where each classifier contains an action such as "*allow*" (meaning forward to first output port) or "*drop*". Performs the action of the first classifier that matched. Example action and classifier: "*allow 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; if this is true, the packet is forwarded to the first output. See also *IPClassifier*. (§5.4, p 70)

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, but second output is push. Input takes IP packets. Processes IP Record Route and Timestamp options; packets with invalid options are sent to the second output.

IPRewriter(...) Push. Inputs take IP packets; any number of inputs and outputs. Rewrites TCP/IP and UDP/IP packets according to specified rewrite rules. See also *TCPRewriter*. (§4.6, p 54)

IPRewriterPatterns(...) No inputs or outputs. Stores shared *IPRewriter* patterns. (§4.6, p 58)

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 all packets to first output if recent input rate averages $< r$, second output otherwise. Multiple rate arguments are allowed. See also *RatedSplitter*.

Null Agnostic. Input takes any packet; forwards every input packet to its output unchanged.

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

PaintTee(p) Agnostic, but second output is push. Input takes any packet. Forwards packets with paint annotation *p* to both outputs; otherwise just to first.

PollDevice(devicename) Push. No inputs. Sends packets to its single output as they arrive from a Linux device driver. Uses polling. See also *FromDevice*. (§7.1, p 91)

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. (§4.3, p 47)

Queue(n) Push input, pull output. Input takes any packet. Stores packets in a FIFO queue; maximum queue capacity is *n*. See also *FrontDropQueue*. (§2.5, p 21)

RandomBitErrors(prob, action) Agnostic. Input takes any packet. Changes each bit on each input packet with probability *prob*. *Action* determines the kind of change: *set* sets bits to 1, *clear* sets bits to 0, and *flip* flips bits.

RatedSource(data, rate, count) Push. No inputs; output emits packets containing *data* as data. Sends *rate* packets per second up to a maximum of *count* packets.

RatedSplitter(rate) Push. Input takes any packet. Forwards up to *rate* input packets to its first output unchanged. Any remaining packets are forwarded to the second output. (§4.5, p 49)

RED(minthresh, maxthresh, maxp) Agnostic. Input takes any packet. Drops packets probabilistically according to the Random Early Detection algorithm [18] with the given parameters; forwards other packets unchanged. Examines nearby router queue lengths when making its drop decision. (§4.4, p 48)

RoundRobinIPMapper(...) No inputs or outputs. A round-robin *IPRewriter* mapper, useful, for example, for load balancing. (§4.6, p 57)

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. (§4.3, p 47)

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. (§4.7, p 59)

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 [33] 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.

StaticPullSwitch(p) Pull. Input takes any packet; arbitrary number of inputs. Forwards each pull request to input port p. (§3.5, p 40)

StaticSwitch(p) Push. Input takes any packet; arbitrary number of outputs. Forwards each input packet to output port p. (§3.5, p 40)

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. (§5.3, p 70)

TCPRewriter(...) Push. Inputs take TCP/IP packets; any number of inputs and outputs. Rewrites TCP/IP packets according to specified rewrite rules. This element supports sequence number rewriting as well as flow ID rewriting. See also *IPRewriter*. (§4.6, p 57)

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.

UDPIPEncap(ipsrc, srcport, ipdst, dstport) Agnostic. Input takes any packet, output emits UDP/IP packets. Encapsulates input packets in a UDP-in-IP header with source address *ipsrc*, source port *srcport*, destination address *ipdst*, and destination port *dstport*.

Unqueue Pull input, push output. Input takes any packet. Places itself on the task queue; whenever it is scheduled, pulls packets from its input and forwards them to its output.

B

Language grammar

B.1 LEXICAL ISSUES

Click identifiers are nonempty sequences of letters, numbers, underscores '_', at-signs '@', and slashes '/' that do not begin with a slash. The system uses '@' and '/' for special purposes: '@' in constructed names for anonymous elements and '/' in names for components of compound elements. Users are discouraged from using these characters in their own identifiers. Identifiers are case-sensitive.

The keywords '`connectiontunnel`', '`elementclass`' and '`require`' may not be used as identifiers. The normal identifiers '`input`' and '`output`' have special meaning inside compound element definitions.

The following characters and multi-character sequences are single Click tokens: `->` `::` `;` `,` `(` `)` `[` `]` `{` `}` `|` `||` `...`

Whitespace and comments, as defined by C++, separate Click tokens. Either form of comment—from '//' to the end of the line, or from '/*' to the next '*/'—terminates an identifier, so this Click fragment

```
an/identifier/with/slashes//too/many
```

has an identifier '`an/identifier/with/slashes`' and a comment '`//too/many`'. No identifier contains two consecutive slashes.

Parameters look like Perl variables. A parameter consists of a dollar sign '$' followed by one or more letters, numbers, and underscores.

A configuration string (config-string in the grammar) starts immediately following a left parenthesis '(' and continues up to the next unbalanced right parenthesis ')'. However, parentheses inside single or double quotes or comments do not affect balancing. Here are several examples; in each case, the configuration string consists of the underlined text.

```
A(simple string)
B(string with (balanced parens))
C(string with ")quoted" paren)
D(// end-of-line comment)
   still going!)
E(/* slash-star comment) */ and backslash \)
```

Click programs may contain C preprocessor-style line directives. These are lines that start with '#' and have the form '# *linenumber* "*filename*"' or '`#line` *linenumber* "*filename*"'; they change the filenames and line numbers used for error messages. The "*filename*" portion is optional. Other preprocessor directives are ignored with a warning. Preprocessor directives are not recognized inside configuration strings.

B.2 BNF GRAMMAR

The Click language is LALR(1), although this grammar is not. (There are ambiguities between the *declaration* and *connection* nonterminals, and between identifiers used as element names and element class names.)

stmts ::= *stmts stmt* | ϵ
stmt ::= *declaration* | *connection*
| *tunnel-stmt* | *elementclass-stmt* | *require-stmt* | **;**
declaration ::= *element-names* **::** *class opt-config*
element-names ::= *element-name* | *element-names* **,** *element-name*
element-name ::= identifier
class ::= identifier | { *compounds* } | { **...** || *compounds* }
compounds ::= *compound* | *compounds* || *compound*
compound ::= *stmts* | *opt-formals* | *stmts*
opt-formals ::= *formals* | ϵ
formals ::= parameter | *formals* **,** parameter
connection ::= *element opt-port* **->** *opt-port connection-tail*
connection-tail ::= *element* | *connection*
element ::= *element-name* | *element-name* **::** *class opt-config*
| *class opt-config*
opt-config ::= **(** config-string **)** | ϵ
opt-port ::= **[** number **]** | ϵ
tunnel-stmt ::= `connectiontunnel` identifier **->** identifier
elementclass-stmt ::= `elementclass` identifier *class*
require-stmt ::= `require` **(** config-string **)**

Bibliography

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, Massachusetts, 1986.

[2] Mary L. Bailey, Burra Gopal, Michael A. Pagels, and Larry L. Peterson. PathFinder: A pattern-based packet classifier. In *Proc. 1st USENIX Symposium on Operating Systems Design and Implementation (OSDI '94)*, pages 115–123, October 1994.

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

[4] Andrew Begel, Steven McCanne, and Susan L. Graham. BPF+: Exploiting global data-flow optimization in a generalized packet filter architecture. In *Proc. ACM SIGCOMM '99 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 123–134, August 1999.

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

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

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

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

[9] Douglas S. J. De Couto and Robert Morris. Personal communication, 2000.

[10] Dan Decasper, Zubin Dittia, Guru Parulkar, and Bernhard Plattner. Router plugins: A software architecture for next generation routers. In *Proc. ACM SIGCOMM '98 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 229–240, October 1998.

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

[12] Mikael Degermark, Andrej Brodnik, Svante Carlsson, and Stephen Pink. Small forwarding tables for fast routing lookups. In *Proc. ACM SIGCOMM '97 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 3–14, October 1997.

[13] Digital Equipment Corporation. DIGITAL Semiconductor 21140A PCI Fast Ethernet LAN Controller Hardware Reference Manual, March 1998. `http://developer.intel.com/design/network/manuals`.

[14] P. Druschel, L. Peterson, and B. Davie. Experiences with a high-speed network adaptor: A software perspective. In *Proc. ACM SIGCOMM '94 Conference on Communications Architectures, Protocols and Applications*, pages 2–13, August 1994.

[15] K. Egevang and P. Francis. The IP Network Address Translator (NAT). RFC 1631, Internet Engineering Task Force, May 1994. `ftp://ftp.ietf.org/rfc/rfc1631.txt`.

[16] Julian Elischer and Archie Cobbs. The Netgraph networking system. Technical report, Whistle Communications, January 1998. `http://www.elischer.com/netgraph/`, as of July 2000.

[17] Dawson Engler and M. Frans Kaashoek. DPF: Fast, flexible message demultiplexing using dynamic code generation. In *Proc. ACM SIGCOMM '96 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 53–59, August 1996.

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

[19] Christopher W. Fraser, Robert R. Henry, and Todd A. Proebsting. Burg—fast optimal instruction selection and tree parsing. *SIGPLAN Notices*, 27(4):68–76, April 1992.

[20] Prem Gopalan and Saurabh Sandhir. Personal communication, 2000.

[21] Michael Hasenstein. IP network address translation. Diplomarbeit, Technische Universität Chemnitz, Chemnitz, Germany, 1997. Available on line at `http://www.suse.de/~mha/linux-ip-nat/diplom/nat.html` as of November 2000.

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

[23] Intel Corporation. Pentium Pro Family Developer's Manual, Volume 3, 1996. `http://developer.intel.com/design/pro/manuals`.

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

[25] Xiaoming Liu, Christoph Kreitz, Robbert van Renesse, Jason Hickey, Mark Hayden, Kenneth Birman, and Robert Constable. Building reliable, high-performance communication systems from components. In *Proc. 17th ACM Symposium on Operating Systems Principles (SOSP)*, pages 80–92, December 1999.

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

[27] David L. Mills. The Fuzzball. In *SIGCOMM '88, Proc. ACM Symposium on Communications Architectures & Protocols*, pages 115–122, August 1988.

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

[29] Robert Morris, Eddie Kohler, John Jannotti, and M. Frans Kaashoek. The Click modular router. In *Proc. 17th ACM Symposium on Operating Systems Principles (SOSP)*, pages 217–231, December 1999.

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

[31] David Mosberger, Larry L. Peterson, Patrick G. Bridges, and Sean O'Malley. Analysis of techniques to improve protocol processing latency. In *Proc. ACM SIGCOMM '96 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 73–84, August 1996.

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

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

[34] NS. UCB/LBNL/VINT network simulator NS homepage. Available from http://www-mash.cs.berkeley.edu/ns/.

[35] Craig Partridge, Philip P. Carvey, Ed Burgess, Isidro Castineyra, Tom Clarke, Lise Graham, Michael Hathaway, Phil Herman, Allen King, Steve Kohalmi, Tracy Ma, John Mcallen, Trevor Mendez, Walter C. Milliken, Ronald Pettyjohn, John Rokosz, Joshua Seeger, Michael Sollins, Steve Storch, Benjamin Tober, Gregory D. Troxel, David Waitzman, and Scott Winterble. A 50-Gb/s IP router. *IEEE/ACM Transactions on Networking*, 6(3):237–248, June 1998.

[36] Larry L. Peterson, Scott C. Karlin, and Kai Li. OS support for general-purpose routers. In *Proc. 7th Workshop on Hot Topics in Operating Systems (HotOS-VII)*, pages 38–43. IEEE Computer Society Technical Committee on Operating Systems, March 1999.

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

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

[39] J. Postel and J. Reynolds. File Transfer Protocol (FTP). RFC 959, Internet Engineering Task Force, October 1985. ftp://ftp.ietf.org/rfc/rfc0959.txt.

[40] Darren Reed. IP Filter TCP/IP packet filtering package. Technical report, 2000. http://coombs.anu.edu.au/~avalon/, as of November 2000.

[41] Alastair Reid, Matthew Flatt, Leigh Stoller, Jay Lepreau, and Eric Eide. CpU: Component composition for systems software. In *Proc. 4th Symposium on Operating Systems Design and Implementation (OSDI 2000)*, October 2000.

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

[43] Michael Shilman. Personal communication, 2000.

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

[45] P. Srisuresh and D. Gan. Load sharing using IP Network Address Translation (LSNAT). RFC 2391, Internet Engineering Task Force, August 1998. ftp://ftp.ietf.org/rfc/rfc2391.txt.

[46] P. Srisuresh and M. Holdrege. IP Network Address Translator (NAT) terminology and considerations. RFC 2663, Internet Engineering Task Force, August 1999. ftp://ftp.ietf.org/rfc/rfc2663.txt.

[47] P. Srisuresh, G. Tsirtsis, P. Akkiraju, and A. Hefferman. DNS extensions to Network Address Translators (DNS_ALG). RFC 2694, Internet Engineering Task Force, September 1999. ftp://ftp.ietf.org/rfc/rfc2694.txt.

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

[49] J. R. Ullmann. An algorithm for subgraph isomorphism. *Journal of the ACM*, 23(1):31–42, January 1976.

[50] Carl A. Waldspurger and William E. Weihl. Stride scheduling: deterministic proportional-share resource management. Technical Memo MIT/LCS/TM-528, MIT Laboratory for Computer Science, June 1995.

[51] Marcel Waldvogel, George Varghese, Jon Turner, and Bernhard Plattner. Scalable high speed IP routing lookups. In *Proc. ACM SIGCOMM '97 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 25–38, October 1997.

[52] Mark N. Wegman and Kenneth Zadeck. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems*, 13(2):181–210, April 1991.

[53] John Wroclawski. Fast PC routers. Technical report, MIT LCS Advanced Network Architecture Group, January 1997. `http://mercury.lcs.mit.edu/PC-Routers/pcrouter.html`, as of July 2000.

[54] Elizabeth D. Zwicky, Simon Cooper, and D. Brent Chapman. *Building Internet Firewalls, Second Edition*. O'Reilly and Associates, Sebastopol, California, 2000.