Eddie Kohler

MIT Laboratory for Computer Science
545 Technology Square, Room 521a
Cambridge, MA 02139
USA
+1 617 253 5261

eddietwo@lcs.mit.edu
http://www.pdos.lcs.mit.edu/~eddietwo/

Education

Massachusetts Institute of Technology

2000 PhD in Electrical Engineering and Computer Science (expected)
Thesis title: The Click modular router as a system and a programming language
Supervisors: M. Frans Kaashoek and Robert Morris

1997 SM in Electrical Engineering and Computer Science
Thesis title: Prolac: a language for protocol compilation
Supervisor: M. Frans Kaashoek

1995 SB in Mathematics with Computer Science
SB in Music

Research interests

Computer systems, with specific interests in networking, programming languages, and operating systems. Also, user interfaces and graphic design.

Experience

1995– **Research assistant** – *MIT LCS Parallel & Distributed Operating Systems Group*

Click modular router: Click is an architecture for building routers from modular software components. Modern routers are expected to implement a large, changing feature set, from interesting dropping policies and quality-of-service to firewalling and network address translation. With Click, a network administrator can implement novel routing features simply by rearranging components. I co-designed the Click system, created the language Click uses to describe router configurations, wrote much of the core code of the Click router, and designed and implemented tools that process Click language files. More information is available at http://www.pdos.lcs.mit.edu/click/.

Prolac protocol language: Prolac is an object-oriented language designed for writing readable, modular, extensible, and efficient network protocol implementations. I designed the Prolac language, implemented its compiler, and co-developed a prototype TCP specification in Prolac. I also informally co-supervised a Prolac-related M.Eng. thesis. More information is available at http://www.pdos.lcs.mit.edu/~eddietwo/prolac/.

Binary analysis: Developed a sandboxer (which enforces safety properties by rewriting machine code) for the MIPS architecture, and a live register analyzer for Alpha binaries.

1998 **Summer intern** – *Microsoft Research, Cambridge, England*

Co-designed and implemented Java prototype for an electronic book. Created a fast algorithm for text layout supporting arbitrary page designs and both document and user preferences.

1996–9 **Consultant** – *Bitstream, Inc., Cambridge, MA; SightPath, Waltham, MA*
Contract work in software implementation and information design.

1992–4 **Undergraduate research** – *MIT*
Programming Methodology Group, LCS: Developed a language-based foreign function interface for CLU, and implemented a driver that improved the CLU compiler's performance by up to 40%.

Visible Language Workshop, Media Lab: Implemented graphical user interface objects for the VLW's proprietary window system.

Information Services: Co-implemented GUI, network protocol, and back end for an electronic-forms system.

1991– **Free software**
Author and maintainer of several widely used free software packages, including PostScript font manipulators, a GIF animation manipulator, a program that reminds you to take wrist breaks, a game, a joke, and a graphical instant messaging client used by half to two-thirds of MIT undergraduates.

Teaching experience

1997 **Recitation instructor** – *MIT course 6.821: Programming languages*
Taught weekly recitation sections to about 25 students. Answered students' questions, graded problem sets, led quiz reviews. Taught lecture when Prof. Gifford was absent. Developed course material, including problem sets, exams, and code (a reconstructor for side effect specifications).

1996 **Teaching assistant** – *MIT course 6.033: Computer system engineering*
Answered students' questions, graded reports, led quiz reviews. Assisted students' writing. Edited course lecture notes and all other course documents.

1994–5 **Laboratory assistant** – *MIT course 6.170: Laboratory in software engineering*
Answered students' online questions.

Publications

Robert Morris, Eddie Kohler, John Jannotti and M. Frans Kaashoek. "The Click modular router." *Proc. 17th ACM Symposium on Operating Systems Principles,* Kiawah Island, South Carolina, December 1999, pages 217–231.
This paper was selected as one of four best-in-conference papers at SOSP '99 and has been selected for fast-track publication in ACM Transactions on Computer Systems.

Eddie Kohler, M. Frans Kaashoek and David R. Montgomery. "A readable TCP in the Prolac protocol language." *Proc. ACM SIGCOMM '99 Conference,* Cambridge, Massachusetts, August 1999, pages 3–13.

Eddie Kohler, Massimiliano Poletto and David R. Montgomery. "Evolving software with an application-specific language." *Workshop Record of WCSSS '99: 2nd ACM SIGPLAN Workshop on Compiler Support for Systems Software,* Atlanta, Georgia, May 1999, pages 94–102.

Service

1998–2000 **Graduate student representative** – *MIT Committee on Campus Race Relations*
Helped define areas for the committee to investigate, including minority graduate student admissions.

1991– *MIT Dramashop*
Served variously as Vice President, Publicity Director, and Publicity Designer for this campus arts organization. Responsibilities included co-producing shows and extensive graphic design work.

Graphic design

1993– **Print**: Poster and other design for arts organizations in the Boston area, including MIT Dramashop, Boston Musica Viva, the John Oliver Chorale, and Open City Theater. Much of this work is archived at http://www.lcdf.org/~eddietwo/design/.

Web: Web site design and implementation for campus organizations and classes, including the Committee on Campus Race Relations (assistant designer), the LCS Parallel and Distributed Operating Systems group, Dramashop, the Gay, Lesbian and Bisexual Graduate Student Coffeehouse, and the 6.033 course (computer system engineering).

Other interests

Theater, music, writing, and visual arts. Wrote four one-act plays performed at MIT; acted in others. Wrote music to accompany a Shakespeare production.

Awards

1998 *Frederick C. Hennie III Award*
Recognizes excellence in teaching by an MIT EECS graduate student.

1997 *Kristen E. Finnegan Prize*
Recognizes contributions of a graduate student to MIT's effort to improve undergraduate writing skills.

1996 *National Science Foundation Graduate Research Fellowship*

1995 *Laya and Jerome B. Weisner Award*
Recognizes outstanding achievement in and contributions to the arts at MIT.

Member, Phi Beta Kappa academic honor society

Member, Sigma Xi engineering honor society

References

Prof. David Gifford
MIT Laboratory for Computer Science
545 Technology Square, Room 401
Cambridge, MA 02139, USA
+1 617 253 6039
gifford@lcs.mit.edu

Prof. M. Frans Kaashoek
MIT Laboratory for Computer Science
545 Technology Square, Room 522
Cambridge, MA 02139, USA
+1 617 253 7149
kaashoek@lcs.mit.edu

Prof. Barbara Liskov
MIT Laboratory for Computer Science
545 Technology Square, Room 528
Cambridge, MA 02139, USA
+1 617 253 5886
liskov@lcs.mit.edu

Prof. Robert Morris
MIT Laboratory for Computer Science
545 Technology Square, Room 509
Cambridge, MA 02139, USA
+1 617 253 5983
rtm@lcs.mit.edu

Dr. Chuck Thacker
Microsoft Research
543 Tennyson Ave
Palo Alto, CA 94301, USA
+1 650 323 9422
cthacker@microsoft.com

Statement of Research Interests — Eddie Kohler

Many systems today have poor extensibility, flexibility, correctness, and security, all of which are easier to fix when a system is easy to understand. Understanding a real, complex software system should be as easy as reading a page or two of code. This would require high-level comprehension of the system; however, we design systems with architectures that don't facilitate higher-level comprehension, and the tools we use to program, analyze, test, and debug them don't help either. I want to create new architectures and tools that will facilitate higher-level comprehension and test them on real-world systems. I want to build systems that satisfy tough performance requirements, but are still flexible and understandable. This work will combine my interests in systems, programming languages, and software design with a willingness to rethink the way systems should be built.

As a first approach, I want to treat systems as programming languages, an avenue inspired by my graduate school research. In this model, a new programming language is an integral part of each system. This language specifies system-level properties, rather than the data layout and control flow issues controlled by languages like C and C++. Its constructs correspond directly to high-level properties of the system, which may be different even for two systems in the same domain. (For example, an operating system language might specify how file systems, network connections, and other kernel-level objects interact. Different operating systems would probably have different languages.) A program in the language gives us a readable, high-level description of the system, and we can manipulate this program with language processors to optimize or statically check the system as a whole.

The first of two projects that inspired this approach is Prolac, an object-oriented language for network protocol implementation. Prolac was designed to make protocol implementations readable, modular, extensible, and efficient. It includes advanced language features, appropriate syntax, tight integration with unforgiving environments (Unix kernels), and compiler optimizations. Our Prolac TCP implementation is organized in a new, readable and modular way; for example, each TCP protocol extension is readably located in one small set of subclasses. The compiler optimizes most of this organization away, leaving code that performs comparably to commodity TCPs.

The Prolac language is low-level and detailed, however, which complicates adoption. Most people don't want or need to learn another systems programming language. Therefore, in my next project—the Click modular router—we kept the programming language higher level, specifying the interaction between components rather than the instruction-by-instruction behavior of each component.

Click is an architecture for building software routers from modular components called *elements*. Element definitions are written in C++; the Click programming language specifies how elements should be connected in a particular router. A router's high-level behavior is thus exactly determined by its definition in the Click language. Network administrators can create arbitrary feature combinations by manipulating easy-to-read Click-language files, rather than by hacking kernel source code.

Like formal specifications, the Click language can be used to detect errors statically and to discover global router properties. Unlike formal specifications, the language compiles into working routers, so people can test and modify the way their packets are routed by manipulating Click-language files. Programs analogous to conventional language processors can manipulate these files as well, to optimize them, analyze them, and so forth. We have written several such programs already: a pattern-based optimizer that replaces common element arrangements with faster equivalents; an aligner, which examines a configuration and adds elements to ensure that packet memory is correctly aligned; and a specializer, which creates new, faster elements tailored to how the elements in a configuration are actually being used.

We have implemented the Click architecture on conventional PC hardware with good results. Click routers are flexible—their configurations can be easily read, understood, and changed. Already, performance is good enough for many demanding applications. Furthermore, having the Click language made the core system better by guiding us towards better internal structures and element designs.

Now I want to expand from these lessons into other systems. While my previous work has been in networking, my interest in systems is omnivorous. Networking, operating systems, and distributed applications—particularly over networks of small devices—all seem like fertile ground for this research. Regardless of area, I want to build real systems; I believe you learn from interacting with complexity, and little is more complex than the real world. These new systems probably won't be much like Click or Prolac, because they will be tailored for different problems. Nevertheless, successful system languages may share some basic principles, which I would like to pin down.

This work is fundamentally interdisciplinary in nature. Building a successful system language requires expertise in languages, systems in general, and the particular kind of system being built. I look forward to working in many system areas through close collaboration with peers and, especially, students.

Statement of Teaching Interests   Eddie Kohler

I love learning, and believe it is a responsibility, a privilege, and a joy to teach in return. My love for learning and teaching is broader than any one field, so much of this statement is about teaching in general. So far, I have taught classes in programming languages and systems, and would like in future to teach undergraduates or graduate students in languages, operating systems, networking, and/or elementary courses. My teaching is driven by flexibility, clarity, and the desire to reach all students.

At MIT, I have taught informally, as a lab assistant, and twice as a teaching assistant: for an undergraduate course and a graduate course. My most extensive teaching experience was the graduate course on programming languages. I taught recitation (weekly sections with 25 to 30 students) and lecture once or twice when the professor was out of town; answered students' questions; led quiz reviews; and developed course material, including quizzes and code. I was honored with two awards for the teaching assistantships, one for contributions to students' writing skills and one for general excellence.

My goal in teaching is to reach every student. (Many students reported on evaluations that they "truly believed that [I] cared about their performance in the course.") To reach everyone, a teacher must be flexible and provide different ways to learn the material; then students can choose the ones that work for them. I concentrate my efforts on clear, intuitive explanations, but let students help guide the pace and path of my lectures, and use other techniques whenever they help—from design problems to metaphors to games.

Teaching can also be inspirational, exciting students about computer science and computer systems. Inspiration, I believe, comes mostly from working on well-chosen, difficult, and rewarding problems. Lectures are important for providing intuition, but working on problems is what makes intuition stick; and only solving problems provides the adrenaline rush of inspiration. I put a lot of effort and creativity into creating problems and projects that are meant to inspire.

Of course, the most rewarding problems are research problems, and research and teaching are deeply linked, sharing a concern for the clearest explanation and the simplest solution. Advising graduate students (and undergraduate researchers) requires flexibility and excitement, even more than teaching a lecture. But then the rewards are greater—you end up with a collaborator and a peer.

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

The Click modular router

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

Abstract

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

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

1 Introduction

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

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

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

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

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

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

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

SOSP-17 12/1999 Kiawah Island, SC

© 1999 ACM 1-58113-140-2/99/0012 . . . $5.00

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

2 Architecture

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

The most important properties of an element are:

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

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

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

Click provides two kinds of connections between elements, *push* and *pull*. In a push connection, the upstream element hands a packet to the downstream element; in

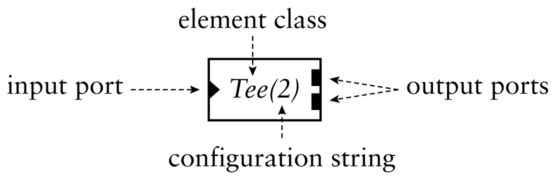

Figure 1: A sample element. Triangular ports are inputs and rectangular ports are outputs.

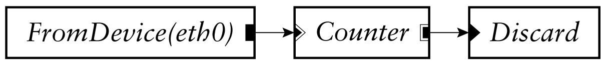

Figure 2: A router configuration that throws away all packets.

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

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

2.1 Control flow and queues

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

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

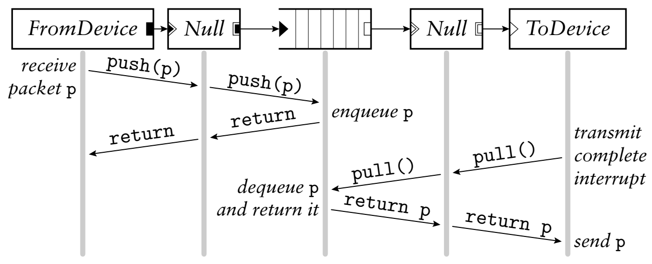

Figure 3: Push and pull control flow. This diagram shows functions called as a packet moves through a simple router. The central element is a *Queue*. During the push, control flow moves forward through the element graph starting at the receiving interface; during the pull, control flow moves *backward*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

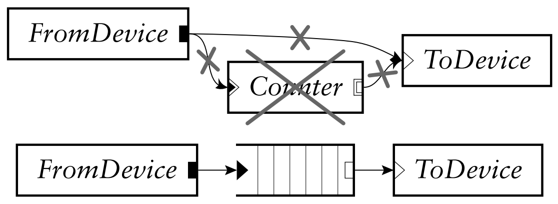

Figure 4: Some invariant violations. The top configuration has four errors: (1) *FromDevice*'s push output connects to *ToDevice*'s pull input; (2) more than one connection to *FromDevice*'s push output; (3) more than one connection to *ToDevice*'s pull input; and (4) the agnostic element *Counter* is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

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

2.2 Push and pull processing

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

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

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

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

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

2.3 Flow-based router context

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

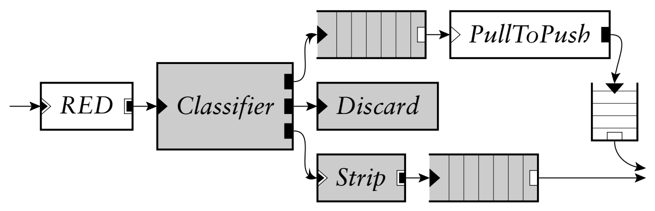

Figure 5: The elements downstream of *RED*, found by flow-based router context with a filter that stops at *Queue*s. The downstream elements are colored grey.

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

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

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

2.4 Implementation

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

Subclasses of `Element` are easy to write, so we expect users will have no problem writing new element classes as needed. In fact, the complete implementation of a simple working element class (*Null*, which passes packets

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

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

from its single input to its single output unchanged) takes less than 20 lines of code; see Figure 6. Most elements define functions for parsing configuration strings and initialization in addition to those in Figure 6, and take about 120 lines of code.

2.5 Language

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

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

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

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

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

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

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

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

3 An IP router

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

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

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

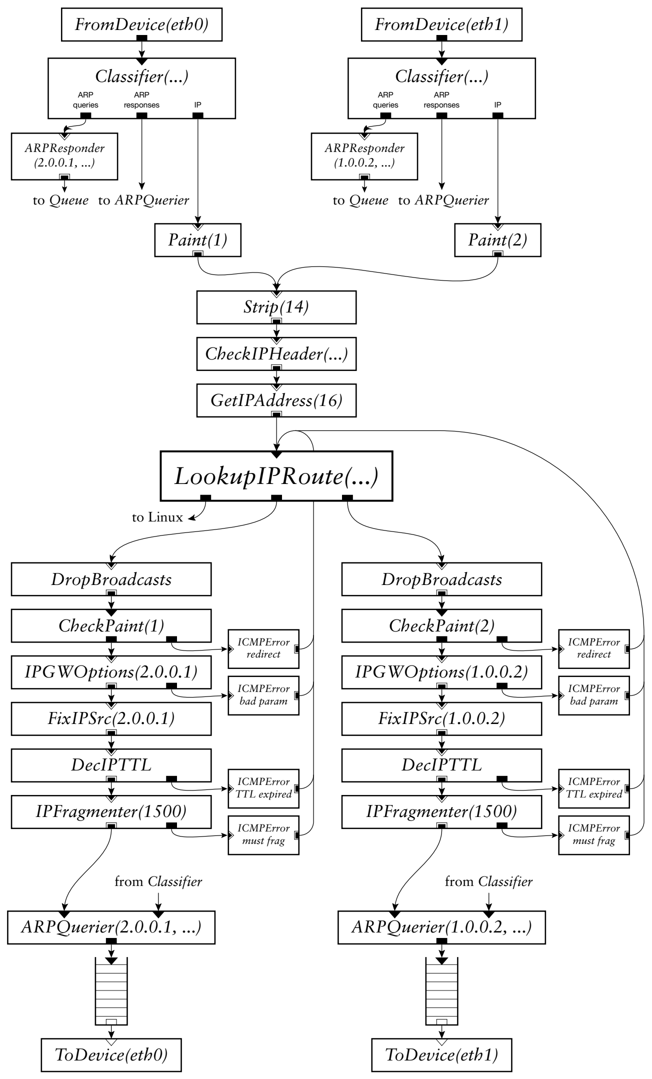

Figure 8: The IP router configuration.

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

Figure 9: Element glossary.

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

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

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

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

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

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

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

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

4 Extensions

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

4.1 Scheduling

With pull processing, a packet scheduler can be implemented in Click as a single element that maintains only local knowledge of the router configuration. Packet scheduling is a kind of multiplexing—a scheduler decides how a number of packet sources (usually queues) will share a single output channel—and a Click scheduler is a pull element with multiple inputs and one output. It reacts to requests for packets by choosing one of its inputs,

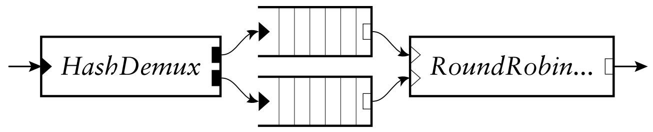

Figure 10: A virtual queue implementing Stochastic Fairness Queueing.

pulling a packet from it, and returning it. (If the chosen input has no packets ready, the scheduler will usually try other inputs.)

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

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

4.2 Dropping policies

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

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

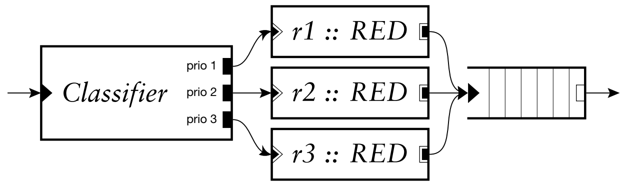

Figure 11: Weighted RED. The three *RED* elements can have different RED parameters, allowing packets with different priorities to be dropped with different probabilities when the router is under stress.

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

4.3 Complex queueing

Imagine a router with the following requirements:

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

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

4.4 Differentiated Services

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

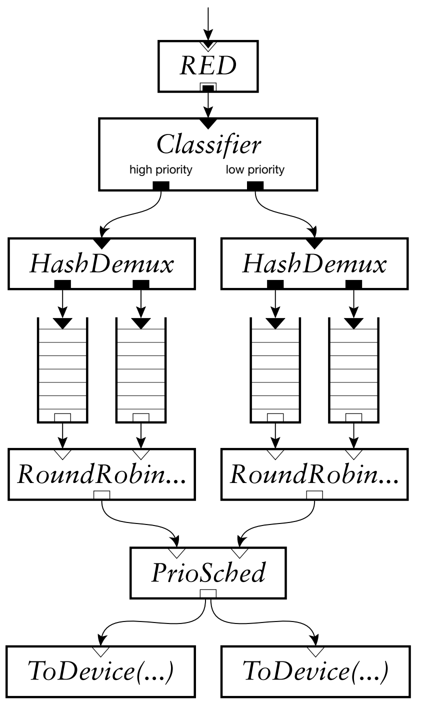

Figure 12: A complex combination of dropping, queueing, and scheduling. The *Classifier* prioritizes input packets into two virtual queues, each of which implements stochastic fair queueing (see Figure 10). *PrioSched* implements priority scheduling on the virtual queues, preferring packets from the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

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

4.5 Ethernet switch

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

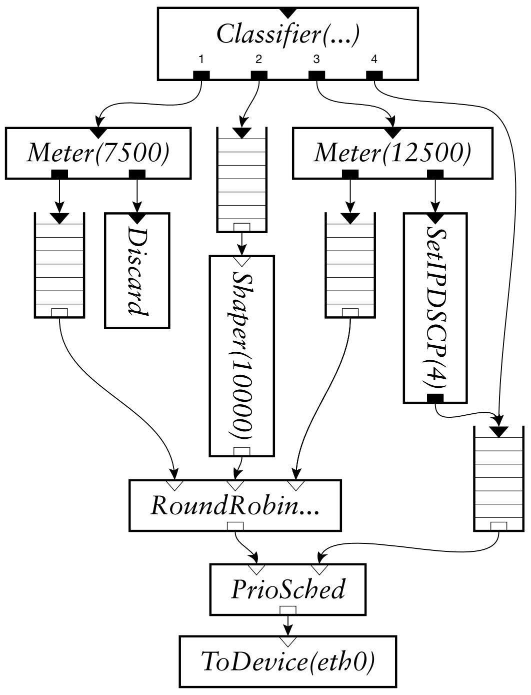

Figure 13: A sample traffic conditioning block. *Meter*s and *Shaper*s measure traffic rates; they are available in varieties that measure bytes per second or packets per second. This example uses packets per second. 1, 2, 3, and 4 represent DSCP values.

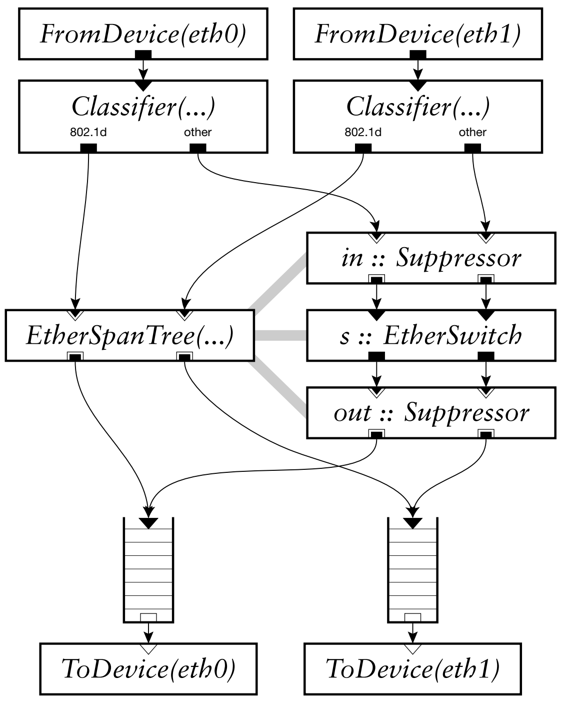

Figure 14: The Ethernet switch configuration.

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

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

4.6 Limitations

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

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

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

5 Implementation

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

5.1 System components

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

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

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

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

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

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

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

5.2 Linux kernel environment

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

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

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

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

6 Evaluation

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

6.1 Experimental setup

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

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

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

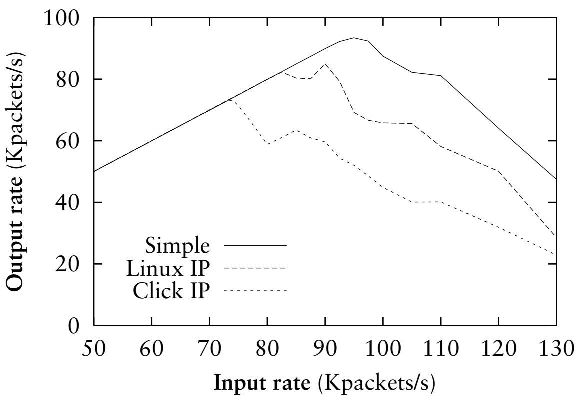

Figure 15: Forwarding rate as a function of input rate for 64-byte packets. An ideal router that forwarded every packet would appear as a straight line $y = x$. The Simple plot is the measured performance of a Click configuration that does no processing other than to emit each input packet. The Linux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

6.2 Forwarding rates

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

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

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

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

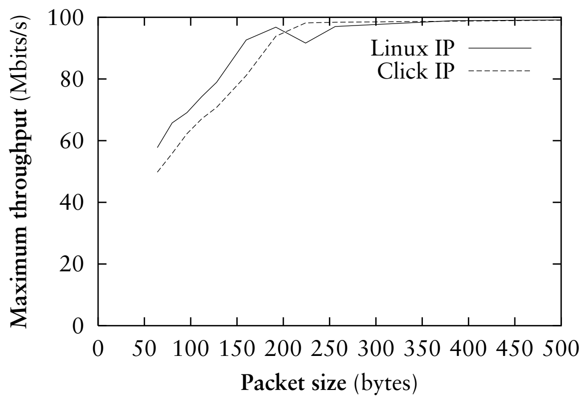

Figure 16: Effect of packet size on maximum forwarding rate.

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

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

6.2.1 *Detailed forwarding cost*

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

The 10.7 μs per-packet interrupt cost is a function of the cost of an interrupt and the number of packets processed per interrupt. In this experiment the input Ethernet device delivered an average of 1.5 packets per interrupt. The average interrupt cost 1 μs for the

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

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

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

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

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

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

To help separate the costs of IP processing from el-

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

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

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

6.3 Cost of incremental complexity

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

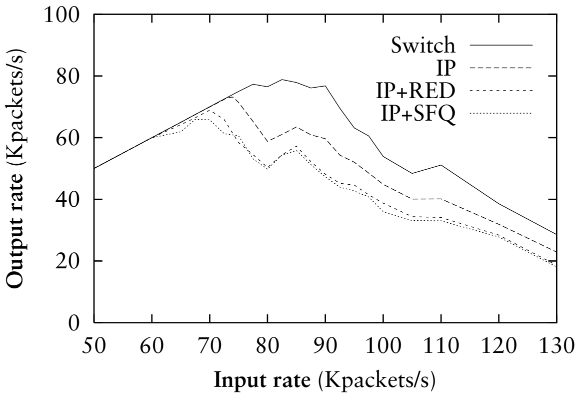

Figure 17: Forwarding rate as a function of input rate for some sample Click configurations.

a copy of the fair queuing arrangement in Figure 10. The Switch line corresponds to the Ethernet switch configuration of Figure 14, which does much less work than the IP router.

6.4 Differentiated Services evaluation

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

6.5 Performance summary

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

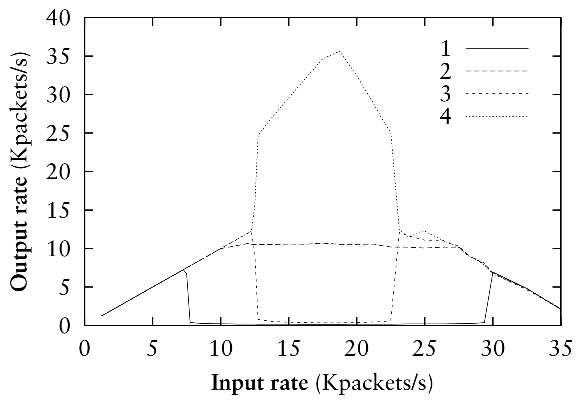

Figure 18: Performance of the diffserv configuration. Each numbered line corresponds to one DSCP; see Figure 13. The x axis corresponds to the input rate for one DSCP, so the aggregate input rate is four times this value. The performance peak is at roughly 72,000 aggregate packets per second. The line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

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

7 Related work

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

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

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

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

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

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

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

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

8 Conclusion

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

Acknowledgements

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

References

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

ACM SIGCOMM '99 Conference: Applications, Technologies, Architectures, and Protocols for Computer Communications
Published as *Computer Communication Review* **29**(4):3–13, October 1999

A Readable TCP in the Prolac Protocol Language

Eddie Kohler, M. Frans Kaashoek, and David R. Montgomery
M.I.T. Laboratory for Computer Science
Cambridge, MA 02139, USA
{eddietwo, kaashoek, dmontgom}@lcs.mit.edu
http://www.pdos.lcs.mit.edu/

ABSTRACT

Prolac is a new statically-typed, object-oriented language for network protocol implementation. It is designed for readability, extensibility, and real-world implementation; most previous protocol languages, in contrast, have been based on hard-to-implement theoretical models and have focused on verification. We present a working Prolac TCP implementation directly derived from 4.4BSD. Our implementation is modular—protocol processing is logically divided into minimally-interacting pieces; readable—Prolac encourages top-down structure and naming intermediate computations; and extensible—subclassing cleanly separates protocol extensions like delayed acknowledgements and slow start. The Prolac compiler uses simple global analysis to remove expensive language features like dynamic dispatch, resulting in end-to-end performance comparable to an unmodified Linux 2.0 TCP.

1 INTRODUCTION

Most familiar programming idioms handle network protocols badly—even modern languages are stressed by common protocol characteristics like complicated control flow, soft modularity boundaries, and stringent efficiency requirements. This makes protocol code hard to read, verify and maintain. Specialized languages are a promising area for solutions to this problem, and network protocol languages and compilers have been an active research area for decades [1, 4, 7, 10, 11, 17].

Most existing protocol languages focus on verification. Their underlying theoretical models are designed for testing and provability, often making pragmatic goals like real-world implementation difficult to achieve. Even languages designed with pragmatism in mind can have theoretical models that are difficult to program.

In this paper, we describe a language that takes a different approach. Prolac is a lightweight object-oriented language tailored for network protocol implementation. It is focused on readability rather than provability, and on the human programmer rather than a machine verifier; protocol implementation requirements inspired its design. No part of Prolac is difficult to compile into efficient low-level code, as we demonstrate with our TCP implementation. Section 3 describes the Prolac language and its compiler in more detail.

Section 4 presents the reimplementation of most of TCP in Prolac. Our TCP is modular, readable, and extensible compared to other implementations in 4.4BSD and Linux 2.0. Modules and methods are used to break complex functionality into focused parts, and the protocol's top-down design remains visible in the final implementation; all this has no significant performance overhead. Four TCP extensions (delayed acknowledgements, slow start, fast retransmit, and header prediction) are implemented through subclassing as add-ons to a clean base. These extensions are simple—each one fits in a single source file with less than 60 lines of Prolac—and can be independently turned on with no changes to the base protocol. The Prolac TCP runs inside a Linux 2.0 kernel, interfaces with the networking subsystem, and is able to exchange packets with other, unmodified TCPs with roughly the same end-to-end performance as unmodified Linux, as discussed in Section 5.

The contributions of this work are the Prolac language, including several novel language features; a new way of structuring a TCP implementation in Prolac, giving superior readability and extensibility; and a preliminary performance analysis of this Prolac TCP.

2 RELATED WORK

This section discusses both how Prolac relates to other protocol languages and compilers, and how the Prolac TCP implementation relates to other TCPs, including modular TCPs written in conventional languages like C++.

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

2.1 Protocol languages

Many previous protocol languages have been designed for verification, not readability or implementation. Prolac uses ideas from some of these languages, but we found that specific language features designed with protocols in mind—for example, parallelism to model both sides of a connection—often worked against readability, implementability, extensibility, or all three. Prolac's final design is more conventional and less domain-specific than these languages; the protocol domain generally affected the details of our versions of common concepts, not specific language features.

Two protocol languages, or "formal description techniques," were originally designed for developing the OSI protocol suite: LOTOS [4] and Estelle [10]. Estelle, the language intended for implementation, is Pascal-like; it structures a protocol as a set of finite state machines running in parallel and communicating via broadcast signals. We find Estelle specifications difficult to read because of this, although it is well suited for state analysis and test generation. Semi-automatic implementations of Estelle specifications have been built [20], but finite state machines make specifications complicated and difficult to change, even for carefully layered protocols [23].

Esterel [5] addressed some of Estelle's implementation difficulties by removing its asynchronous parallelism, leaving a completely sequential language. This worked. Impressive performance results are reported for a restricted Esterel version of TCP [7], better than a similarly restricted BSD TCP; this convinced us to leave parallelism out of Prolac. However, Esterel still shares Estelle's formal model, interlocking finite state machines, and the problems this causes: complexity, unfamiliarity, unreadability, and difficulty of modification or extension. The Esterel TCP did not include connection establishment, and appears not to include important extensions like congestion avoidance.

RTAG [2] is based on a different formal model: context-free attribute grammars. RTAG is more readable than LOTOS and Estelle, but large RTAG specifications, like large attribute grammars generally, become hard to read since the namespace is flat. An early version of Prolac resembled RTAG, but readability and other issues have pushed it in the direction of conventional programming languages. RTAG's performance is problematic, again due to parallelism in the language.

The *x*-kernel [12], which introduces an explicit architecture for constructing and composing protocols, is orthogonal to Prolac. We focus on making a single protocol implementation readable; the *x*-kernel provides a uniform interface between protocols and aims to improve the structure and performance of protocol layering.

Morpheus [1], another object-oriented language for protocol implementation, is based on *x*-kernel ideas. To force clean protocol designs and enable domain-specific optimizations, it puts many constraints on the programmer. As a result, existing protocol specifications may not be implementable in Morpheus. Its compiler has not been written.

2.2 TCP specifications

TCP specifications [19, 22] and existing C implementations of TCP—particularly the 4.4BSD implementation as presented by Stevens [21, 24]—have greatly influenced our TCP implementation, suggesting code structures to emulate and to avoid. Prolac TCP was rewritten for readability from 4.4BSD's TCP, using both general object-oriented techniques and techniques specific to Prolac.

The Fox project's structured TCP [3], which is based on *x*-kernel ideas, uses a functional language—a dialect of Standard ML—to explore the advantages and disadvantages of using a non-traditional language in the systems domain. They report readability and modularity benefits similar to Prolac's. Their TCP is not built for protocol extensibility, however, and because of advanced language features, it is unsuitable for in-kernel implementation and performance is low.

Although Prolac resembles object-oriented languages like C++ and Java, it is designed to be more useful for network protocols than these languages. We initially tried to implement a modular TCP in C++, but were foiled by C++'s programming paradigm, which pushed us toward a conventional inheritance structure and a small number of types. Additionally, C++ has inflexible access control, function definitions are syntactically expensive, and most programmers habitually avoid virtual functions. These factors suggest that a C++ TCP would combine most protocol data into one large class (avoiding access control issues at the expense of modularity), tend towards larger functions, and use dynamic dispatches only rarely (making it less extensible). Many of these properties occur in *ns*, the Berkeley network simulator [18], which contains a C++ implementation of TCP.

3 THE PROLAC LANGUAGE

This section is an introduction to the Prolac language. We do not provide a thorough description of the language (see the reference manual [14] for that); instead, we focus on general features and design goals. We kept Prolac largely conventional, hoping it would be easy to grasp, but our focus on protocol-related issues and an additional concentration on simplicity has led to some novel language contributions. Two of these, module operators and implicit methods, are described below.

3.1 Methods and computation

All computation in a Prolac program is performed by *methods*, functions that belong to a module. A method's body is simply an expression: Prolac is an expression language, like Lisp, ML or Haskell, so it has no concept of "statement". All of C's operators (including assignments), plus a few additions, are usable in Prolac expressions.

Prolac method bodies tend to be very short compared with C function bodies—most are 5 lines or less. There are several reasons for this: Prolac makes it easy and efficient to name parts of a computation, so large methods tend to be broken up into sensibly-named parts; furthermore, large expressions can become unreadable, so there is pressure to keep methods small.

The choice of an expression language was influenced by a desire to eliminate syntax, particularly routine or boilerplate syntax. We find that lightweight syntax makes small methods more readable, as the substance of the code is the only thing on display.

Much like Yacc parsers [13], Prolac is wedded to the C language through uninterpreted *actions*. A C action may be included in any Prolac expression; the Prolac compiler, which generates C, will copy the action to its output when compiling that expression. C actions can easily refer to Prolac objects and change their values, as well as perform arbitrary computation in C. They are extremely useful for interfacing with the environment a Prolac specification is embedded in.

Figure 1, which is extracted from the Prolac TCP specification, should give a flavor of what Prolac is like. It gives a concrete example of Prolac code and explains some of the language's features.

3.2 Modules and object orientation

Prolac *modules* represent groups of methods and data (data members are called *fields*). Modules may extend other modules through inheritance, and may provide new definitions for their superclass's methods; the correct definition is chosen at runtime (dynamic dispatch). Thus, Prolac is object oriented, and modules are similar to C++ or Java classes.

Like Java, the Prolac language is statically typed, all code is part of some module, a module can have at most one parent, every method is potentially subject to dynamic dispatch, and Prolac source code is completely order-independent. However, not all Java features translate to Prolac—there is no interface inheritance, for example.

In our TCP implementation, we use inheritance both for subclassing and to build complex subsystems from smaller parts. For example, the module representing the base transmission control block is built through successive inheritance from 6 submodules (basics and connection state, windows, timeouts, round-trip time measurements, retransmission, and output). The submodules serve more as grouping constructs than as types with individual identities.

In the interests of flexibility and simplicity, Prolac does not provide primitives for manipulating heap storage. Instead, the user can get memory inside a C action (using `kmalloc`, for example) and use Prolac to initialize it.

3.3 Naming

Descriptive naming makes any program more readable, but in a programming language like Prolac, which encourages the

```
module Trim–To–Window ... {

  trim–to–window :> void ::=
     (before–window ==> trim–old–data),
     (after–window ==> trim–early–data),
     (sending–data–to–closed–socket ==> reset–drop);

  before–window ::= seg–>left < receive–window–left;
  trim–old–data {
    trim–old–data ::=
       (syn ==> trim–syn),
       (whole–packet–old ==> duplicate–packet)
       || seg–>trim–front(receive–window–left – seg–>left);
    whole–packet–old ::=
       seg–>right <= receive–window–left;
    duplicate–packet ::=
       clear–fin, mark–pending–ack, ack–drop;
  }

  after–window ::= seg–>right > receive–window–right;
  trim–early–data {
    trim–early–data ::=
       (whole–packet–early ==> early–packet)
       || seg–>trim–back(seg–>right – receive–window–right);
    whole–packet–early ::=
       seg–>left >= receive–window–right;
    early–packet ::=
       ((receive–window–empty
         && seg–>left == receive–window–left)
         ==> mark–pending–ack)
       || { PDEBUG("early packet\n"); }, ack–drop;
  }

... }
```

Figure 1: Part of Prolac TCP's code for trimming incoming packets to fit the current receive window. The current packet is stored in the 'seg' field. Code is split into small, readably named methods, which are grouped into namespaces. Packets have wide interfaces: both 'seg–>seqno' and 'seg–>left' refer to the first sequence number in the packet, but read well in different situations. Methods not defined in the figure are taken from other modules, such as the transmission control block (TCB); their purposes should be clear from their names. Methods ending in '-drop' are exceptions. There is one C action, in early–packet; as in Yacc, C actions are enclosed in braces. Syntax notes: Rule definitions look like '*rule-name* ::= *expression*;', possibly with arguments in parentheses or a return type following ':>'. Hyphens are allowed in identifiers. Parentheses may be left off when calling methods that take no arguments. Most operators behave as in C. The new operator '==>' is used for simple if statements; '*x* ==> *y*' is equivalent to '*x* ? (*y*, true) : false'.

use of many small methods, sensible naming is an imperative. In the Prolac TCP implementation, we try to use method names that make their purposes immediately clear; without this property, a reader would have to jump nonlinearly from method to method to have any hope of understanding the code. In this section, we discuss a range of Prolac features that encourage and support sensible naming. Together, these features make naming in Prolac significantly more flexible than in C++ or Java.

Prolac supports namespaces both inside and outside modules, allowing methods and modules to be grouped into related units. More flexibility is provided by *module operators*, operators that affect the compiler's behavior rather than the running program's behavior. The hide and show module operators support loose, flexible access control. If M is a module with a feature named x, then 'M hide x' is the same module, except that its x feature is inaccessible. The Prolac TCP implementation uses hide to hide implementation details from module users. But hard access control is a disadvantage in the protocol domain: protocol extensions often work by changing deeply buried, almost random bits of protocol code that cannot be determined a priori. This suggests that access control should be overridable, which the show operator supports by making hidden names accessible again.

The *implicit method* mechanism was also inspired by the protocol domain. In most object-oriented languages, it is syntactically easier to call an object's methods from within another method of that object, since you can leave off the reference to 'this' or 'self'. When a piece of code uses many of a particular object's methods, the user will therefore tend to write it as a method, since it's so much easier that way. Protocol implementations differ from most programs in that data objects are small and limited in number: TCP, for example, deals with transmission control blocks (TCBs) and packets, and not much else. But all TCP processing deals intimately with TCBs; does that mean all of TCP's code should be situated in a TCB module?

The implicit method mechanism solves this problem by allowing the programmer to refer to *another* object's methods with the same syntactic convenience. The programmer can mark a field with the using module operator. When the compiler finds an undefined name, it transparently looks for methods with that name on any fields marked with using. If a unique method is found, it is used implicitly. With implicit methods, TCP processing can be broken into modules based on control flow structure, rather than the less revealing structure of the data, without giving up on readability.

3.4 Compilation and optimization

The Prolac compiler compiles Prolac into C. It generates high-level C, featuring large expressions resembling the Prolac input, reasonable indentation, and relatively few introduced temporaries. The result is reasonably readable, debuggable with C debuggers, and, with some C compilers, results in better object code than an equivalent lower-level version.

The compiler accepts an entire Prolac program at once. This is not a problem even for our relatively complex TCP implementation; with full optimization, the Prolac compiler processes it in under a second on a 266 MHz Pentium II laptop.

The Prolac language has many features that are potentially expensive to implement—universal dynamic dispatch, many small functions, exceptions, modules, and so forth. We carefully arranged the details of these features to minimize their overhead, and simple compiler optimizations can remove that overhead almost entirely. The remainder of this section describes some of these optimizations.

3.4.1 Static class hierarchy analysis

The most important optimization the compiler performs is *static class hierarchy analysis* [9], a simple global analysis that removes every dynamic dispatch in our TCP implementation. The idea is simple: if the compiler can prove that the method being called was not overridden—it is a leaf in the inheritance graph—then that method can be called directly, without the need for dynamic dispatch.

Removing dynamic dispatches is absolutely necessary for performance. A dynamic dispatch is slightly more expensive than a conventional function call, but the real problem is that Prolac will not inline a dynamically dispatched method. The language encourages the use of small methods for naming extremely simple computations; the only hope of having good performance is therefore aggressive inlining.[1]

To show the magnitude of the problem, we removed static class hierarchy analysis from the Prolac compiler. Even when allowing the compiler to inline or directly call methods that were only defined once, the number of dynamic dispatches jumps to 62, many for trivial methods that obviously should be inlined. Considering that *every* Prolac method is potentially dynamically dispatched, however, the situation is even worse: a naive compiler (equivalent to an average C++ or Java compiler) would generate 1022 dynamic dispatches in the Prolac TCP implementation.

Our implementation of static class hierarchy analysis was motivated by, and works so well because of, characteristics of network protocols. The type-related behavior of TCP, for example, is static at runtime: it deals with one kind of control block, one kind of packet, one kind of header, and so on. In other words, there is only one kind of TCP running at any time. Since we don't have to demultiplex among varieties of TCP, we can use inheritance purely for grouping related methods and including extensions that should always be used. In this style, the module we want will always be the most derived module (the TCB we want is the most derived TCB,

1. It is possible to inline dynamic dispatches with mechanisms such as speculative inlining, which inlines one version of the code in question and generates a check to see if that version is the correct one. However, these mechanisms are complex and have nonzero overhead.

and so forth). But every method in a most derived module is a most derived method, so static class hierarchy analysis will always succeed.

Of course, it would be perfectly possible to use inheritance to demultiplex packets or kinds of processing—to derive TCP and UDP modules from a superclass representing Internet transport protocols, for example. In this case, static class hierarchy analysis would appropriately fail, and the necessary dynamic dispatches would be generated. The analysis would continue to be effective within the module hierarchies for the individual protocols.

3.4.2 Inlining and outlining

Mosberger et al. [16] list a number of useful techniques for improving protocol efficiency. Prolac has direct support for three of these: inlining, path inlining, and outlining. Inlining is replacing a function call with the function's body; path inlining is recursive inlining, where functions called by an inlined body are replaced with their bodies, and so on; and outlining is moving code for uncommon cases out of common-case code, thus improving i-cache behavior.

The programmer is given fine-grained control over these optimizations through expression operators and module operators. Module operators are especially useful, as they allow the programmer to specify, without cluttering either the call site or the method's definition, that a method should inlined—and, unlike C++'s `inline` declaration, module operators can be overridden.

4 A READABLE, EXTENSIBLE TCP IMPLEMENTATION

This section describes the structure of the Prolac TCP implementation, highlighting its readability and extensibility and describing several of its subsystems in detail.

4.1 Overview and status

The core of the Prolac TCP is a near-full reimplementation of 4.4BSD's TCP as described by Stevens [24], rethought and reorganized from the ground up for greater readability and extensibility. We implement full input and output processing including retransmissions, slow start, fast retransmit and congestion avoidance, TCP options, and header prediction. We do not yet fully implement keep-alive or persist timers or urgent processing. Also, some changes were made to emulate some of Linux 2.0 TCP's behavior; for example, Linux TCP occasionally delays an ACK for at most .02 sec where BSD would send an ACK immediately.[2] Packet comparisons using `tcpdump` show that Linux 2.0–Prolac TCP exchanges are indistinguishable from Linux 2.0–Linux 2.0 TCP exchanges, except for keep-alive and persist timers and urgent processing.

2. This happens when responding to a packet whose PSH bit is set.

Prolac TCP currently runs inside the Linux kernel as a dynamically loadable kernel module. It works alongside Linux's default TCP; packets directed to specific configurable ports are routed to Prolac instead of the default TCP stack. Prolac TCP is fully integrated with lower-level Linux networking code, including IP processing, network devices, and memory management. This integration even extends to sharing data structures; we use C actions and a Prolac structure-punning feature[3] to make Prolac's Segment module an alias for Linux's internal packet representation, `struct sk_buff`. We have begun integration with higher-level networking code, particularly the `struct sock` structure representing BSD sockets, but for the results in this paper we used an alternative interface to communicate with user level: a handful of new system calls for connection, data transfer, and polling that bypass the socket interface.

Most Linux-specific code is localized in a handful of modules, which should make it easier to port Prolac TCP to other operating systems. The Linux TCP is only slightly modified from a TCP that runs at user level using Berkeley Packet Filters [15].

4.2 Organization

The Prolac TCP implementation was guided by the goal of separating TCP into small, focused modules, or *microprotocols,* handling one job each. TCP extensions are separated from the base protocol into independently selectable units. This principle was also used within the base protocol: we divided complex functionality, like connection state and input processing, into several microprotocols each. Input window management, for example, can be considered a microprotocol within TCP; it is localized in two modules, one for the transmission control block (Window–M.TCB) and one for input processing (Trim–To–Window, which was shown in Figure 1).

The current Prolac TCP implementation consists of 21 source files and about 2100 nonempty lines of code. This is about one-third the size of Linux 2.0's TCP implementation, although that TCP does have more functionality than ours (SYN cookies, for example). The Prolac files are combined by the C preprocessor and the resulting preprocessed source is passed to the Prolac compiler.

Modules in the base TCP implementation fall into six categories: *utilities,* for byte-swapping and checksumming routines; *data,* for data-centric protocol modules—IP and TCP headers, TCP packets, and the transmission control block; *input,* for processing received packets; *output,* for sending packets; *timeouts,* for slow and fast timeout events; and *interfaces,* for communicating with the rest of the system. Figure 2 lists the modules constituting the base protocol, many of which are described in detail in the following sections.

3. The programmer can control how a module is laid out in memory by giving specific byte offsets for its fields. Prolac automatically generates any required padding and warns when field offsets conflict.

Utilities	
Byte–Order	Byte-swapping
Checksum	Checksumming
Data	
Headers.IP	IP header
Headers.TCP	TCP header
Segment	Packet
TCB	Transmission control block
Base.TCB	Basics and connection state
Window–M.TCB	Send and receive windows
Timeout–M.TCB	Timeouts
RTT–M.TCB	Round-trip time measurement
Retransmit–M.TCB	Retransmission
Output–M.TCB	State for BSD-like output
Input	
Base.Input	General input processing
Base.Listen	Handle input in *listen* state
Base.Syn–Sent	Handle input in *syn-sent* state
Base.Trim–To–Window	Trim packet to fit receive window
Base.Reset	Process RST
Base.Ack	Process ACK
Base.Reassembly	Reassembly
Base.Fin	Process FIN
Output	
Base.Output	Output processing
Timeouts	
Base.Timeout	Timeouts
Interfaces	
Tcp–Interface	User-level interface (`read`, `write`, etc.)
Base.Socket	Interface to socket layer

Figure 2: Module structure of the Prolac TCP implementation: the base protocol.

4.3 The TCB

In the RFC definition of TCP [19], all persistent TCP-specific data about a connection is stored in a single structure, the transmission control block (or TCB). The 4.4BSD TCP implementation and our Prolac TCP implementation follow this organization. The TCB is large—the 4.4BSD TCB structure has 48 fields, while our Prolac TCB structure currently has 42. This is too large to be readably defined in a single module, especially if methods are included. Therefore, as mentioned above, we build the TCB by successive inheritance from six components: basics and connection state, windows, timeouts, round-trip time measurement, retransmission, and output. Each of these components is made self-contained through hide, the access control module operator; private fields and methods are hidden from other components. This defines a public interface for the module, which has the usual benefit of making it easier to safely change module internals.

The TCB is mostly *passive,* meaning that it does not usually act upon other modules—other modules act upon it. This resembles 4.4BSD's non-object-oriented implementation, where the TCB simply a flat structure. Even in Prolac, however, passive organization seems right for the TCB: TCP processing is so complex that separating control flow from data generally improves readability.

Even our passive TCB still benefits from object-oriented design. First, the TCB provides small, descriptive methods that perform simple calculations, so users need never touch the fields themselves. For example, there are two ways to determine whether a received acknowledgement, ackno, is valid for the current connection:

```
valid–ack(ackno :> seqint) ::=
        ackno >= snd_una && ackno <= snd_max;
unseen–ack(ackno :> seqint) ::=
        ackno > snd_una && ackno <= snd_max;
```

(snd_una and snd_max are fields maintained by the TCB. All variables have type seqint, so the arithmetic comparison operators are actually circular comparison mod 2^{32}.) valid–ack and unseen–ack both return true iff they are given a good acknowledgement number, but valid–ack allows duplicate acknowledgements while unseen–ack does not. The method names make this clearer than the expressions, which differ only subtly. Calling these methods makes code easier to read, since the reader doesn't need to parse expressions; it also helps prevent errors, since the programmer must actively choose between them. The choice between valid–ack and unseen–ack puts the issues more clearly at stake than the choice between > and >=, which makes the programmer more likely to choose carefully and correctly.

Second, some TCP events, such as receiving a new acknowledgement, trigger complex behavior that cuts across Prolac's module structure. To model this cleanly, the TCB uses *hooks,* methods that are called to mark the occurrence of a protocol event. Hooks exist to be extended; a base hook defined in Base.TCB often does nothing—the action takes place in overriding definitions from later TCB components. Here are a few of the TCB's hooks, including the event that triggers each and typical actions they perform.

- receive–syn–hook(*seqno* :> seqint)

  Called when a SYN is received on a connection. *seqno* is the SYN's sequence number. Effects: Sets various TCB fields (like irs, the initial received sequence number, and rcv_next, the sequence number we expect to receive next).

- new–ack–hook(*ackno* :> seqint)

  Called when a new acknowledgement is received. Effects: Removes newly acknowledged data from the retransmission queue, updates snd_una (the first unacknowledged sequence number sent), adjusts the send window, and updates the current round-trip time estimate if appropriate.

- total–ack–hook

  Called when all outstanding data has just been acknowledged. Effects: Cancels the retransmission timer.

- send–hook(*seqlen* :> uint)

  Called when a packet is sent. *seqlen* is its length in sequence numbers. Effects: Moves snd_next and snd_max forward, clears the pending-acknowledgement and delayed-acknowledgement flags, adjusts the send window, and optionally starts round-trip time measurement and the retransmission timer.

Most hooks are multiply overridden, with each overriding definition adding behavior to the previous definition. Figure 3 shows how this works in practice for send–hook, which has five definitions total (four in base modules and one in the delayed-ack extension). Each individual definition is small, focused, and clear, although the aggregate behavior is sophisticated.

The individual TCB submodules are similarly readable: each contains limited, focused processing, with complex behavior only created through the modules' combination. This style does obscure the aggregate behavior—the code in Figure 3 was taken from five source files—but when suitably natural hooks are chosen, this doesn't tend to be a problem.

```
Base.TCB.send–hook(seqlen :> uint) ::=
  // This is the base hook. It adjusts some fields and clears
  // some flags
  clear–flag(F.pending–ack | F.pending–output),
  snd_next += seqlen,
  snd_max max= snd_next;
Window–M.TCB.send–hook(seqlen :> uint) ::=
  // The window TCB additionally adjusts the send window
  // and clears another flag
  inline super.send–hook(seqlen),  // calls Base.TCB.send-hook
  clear–flag(F.need–window–update),
  snd_wnd –= seqlen;
RTT–M.TCB.send–hook(seqlen :> uint) ::=
  // Decide whether to measure this packet's round-trip time.
  // After inline super.send-hook, the sent packet's sequence
  // number is snd_next - seqlen, not snd_next
  inline super.send–hook(seqlen),
  (seqlen && !retransmitting && !timing–rtt ==>
    start–rtt–timer(snd_next – seqlen));
Retransmit–M.TCB.send–hook(seqlen :> uint) ::=
  // Start the retransmit timer if necessary
  inline super.send–hook(seqlen),
  (!is–retransmit–set && !recently–acked ==>
    start–retransmit–timer);
Delay–Ack.TCB.send–hook(seqlen :> uint) ::=
  // Clear the delayed acknowledgement flag
  inline super.send–hook(seqlen),
  clear–flag(F.delay–ack);
```

Figure 3: The five send-hook methods defined by the Prolac TCP implementation, from the initial definition (in Base.TCB) to the most derived version (in Delay-Ack.TCB). Each method except the first explicitly calls its predecessor with super.send-hook(seqlen), resulting in cumulative behavior.

4.4 Input and output

The Prolac TCP implementation divides input processing into eight independent modules based on processing steps specified in the original TCP RFC [19]. 4.4BSD TCP also follows the RFC in outline, but obscures that relationship by hand-inlining large chunks of code. Prolac, in contrast, keeps the high-level structure crystal clear: Figure 4 demonstrates this by comparing an excerpt from our input processing code with headings from the TCP RFC. This top-down organization has no associated cost in Prolac, since the methods can all be inlined.

The base input processing module, Input, declares exceptions and convenience methods and directs control flow through the other modules. (The methods defined in Figure 4 are all taken from Input.) The other seven input modules—Listen, Syn–Sent, Trim–To–Window, Reset, Ack, Reassembly, and Fin—all inherit from Input and use its exceptions and convenience methods.

The relevant TCB and the input packet being processed are stored in Input, as fields named tcb and seg. This allows them to be passed implicitly from method to method within each module, and enables implicit method search as described in Section 3.3, making the code more readable by avoiding fussiness. If the packet and TCB weren't fields, for example, the user would have to pass them as parameters to every method—which, with many small methods, would quickly become annoying. There is a performance penalty: the packet and TCB are structure members, and therefore not stored in registers by some compilers; and creating Input objects, or objects derived from Input like Ack and Reassembly, has a small but nonzero overhead.

Output processing, which is smaller and simpler than input processing, is implemented in a single module. Output processing follows the 4.4BSD model: a single routine, Output.do, is called whenever any normal kind of output is needed; the Output module then decides exactly what kind of packet to send. Several small changes were made, including consistently using sequence number length rather than data

do–segment ::=	
(closed ==> reset–drop)	If the state is CLOSED . . .
|| (listen ==> do–listen)	If the state is LISTEN . . .
|| (syn–sent ==> do–syn–sent)	If the state is SYN-SENT . . .
|| other–states;	
other–states ::=	Otherwise,
trim–to–window,	first check sequence number . . .
(rst ==> do–reset),	second check the RST bit, . . .
(syn ==> reset–drop),	fourth, check the SYN bit, . . .
(!ack ==> drop), do–ack,	fifth check the ACK field, . . .
process–data;	
process–data ::=	
(urg ==> check–urg),	sixth, check the URG bit, . . .
let is–fin = do–reassembly in	seventh, process the segment text, . . .
(is–fin ==> do–fin)	eighth, check the FIN bit, . . .
end,	
send–data–or–ack;	and return. [19]

Figure 4: The Prolac implementation, at left, directly echoes the TCP RFC specification, at right. (The RFC's third step, "check security and precedence", is missing.)

length (sequence number length is data length plus any SYN and FIN flags). These changes, which were only intended to make the code more consistent and therefore readable, ended up discovering a bug in the 4.4BSD code as reported by Stevens [24]: if a packet just fits in a maximum segment size, but doesn't quite fit when options are included, that code could leave a FIN on the packet when it should have been removed. While this small bug had already been fixed in our OpenBSD kernel, our independent discovery eloquently demonstrates the advantages of code readability.

4.5 Extensions

TCP has been extended over time, with some of these extensions becoming standard—slow start, congestion avoidance, fast retransmit, and fast recovery, for example [22]. We used subclassing to extend the Prolac TCP without cluttering its base definition. We have currently implemented four TCP extensions: delayed acknowledgements, slow start and congestion avoidance, fast retransmit and fast recovery, and header prediction. A C preprocessor mechanism called *hookup* makes these extensions both transparent and independent: almost any subset of them can be turned on without changing the rest of the system in any way.

Each extension consists of several modules that override modules from the base protocol. All modules relating to a particular extension are placed in a single source file. The extension is turned on only if that source file is `#included` into the preprocessed source. Figure 5 lists the modules that constitute some of the extensions and their corresponding source files.

This arrangement makes extending TCP simple, natural, and convenient. None of our extensions takes more than 60 lines of Prolac proper. Each extension is concentrated and readable, since extension-related code is contained in one file rather than spread throughout thousands of lines of other protocol processing. Finally, the extension code runs without additional runtime overhead, thanks to static class hierarchy analysis and inlining. All this makes Prolac a good platform for developing protocol extensions.

Delay–Ack.*	Delayed acknowledgements
Delay–Ack.TCB	(in `delayack.pc`)
Delay–Ack.Reassembly	
Delay–Ack.Timeout	
Slow–Start.*	Slow start and congestion
Slow–Start.TCB	avoidance (in `slowst.pc`)
Slow–Start.Ack	
Fast–Retransmit.*	Fast retransmit
Fast–Retransmit.TCB	(in `fastret.pc`)
Fast–Retransmit.Ack	
Header–Prediction.*	Header prediction
Header–Prediction.Input	(in `predict.pc`)

Figure 5: Module structure of the Prolac TCP implementation: some protocol extensions.

4.6 Discussion

If, while extending our TCP, we discover the need for a new hook, we simply add it to the base protocol with an empty definition. This can make the implementation easier to follow, but similar techniques would work without changing the base protocol at all. A user can add a new hook by overriding the method or methods that should call the hook, and adding a call of the hook itself. Changing a send–segment method to

	End-to-end latency (μs)	**Processing time** (cycles)
Linux TCP	184	3360
Prolac TCP	181	3067
Prolac without inlining	228	6833

Figure 6: Microbenchmark results for the echo test. The test machine sends 4 bytes of data to an unmodified Linux 2.2.7 machine's `echo` port and waits for an ack. Results are averaged over five trials, each consisting of 1000 round-trips, for a total of 10000 packets: 5000 input and 5000 output. Processing time represents the average number of cycles it took to process a packet.

include a hook might look like this:

```
Base.TCB.send–segment(s :> *Segment) ::= ...;
Extension.TCB.send–segment(s :> *Segment) ::=
   super.send–segment(s),
   send–hook(s–>seqlen);
Extension.TCB.send–hook(seqlen :> uint) ::= ...;
```

The extension framework we have described works best for extensions that do not fundamentally change the base TCP's behavior or data structures. An extension implementing extended sequence numbers, for example, would be much more complex than our delayed-ack extension.

5 PROLAC TCP NETWORK PERFORMANCE

This section describes experiments that compare Prolac TCP with an unmodified Linux 2.0 TCP implementation. These experiments show that Prolac's high-level language features come with little or no associated performance cost; when Prolac does worse, it seems to be due to implementation artifacts like packet copies.

We compared the Prolac TCP loadable kernel module, running on a Linux 2.0.36 kernel, with Linux 2.0.36's native TCP. There are important differences between the two. Linux TCP is generally more reliable, well tested, and complete than Prolac TCP, although Prolac does have some features Linux lacks, such as header prediction. In addition, Linux TCP communicates with user level through the socket API, while Prolac TCP uses its own system-call-based API and a private socket-like structure. We tested sources of overhead in both TCPs and found that only one was significant: due to implementation artifacts, Prolac TCP copies packets one more time on input and two more times on output than Linux. The sole input copy and one output copy are due to Prolac's socket-like API, and affect only end-to-end measurements like latency and throughput; the other output copy is in output processing proper and affects cycle counts as well.

The test machines were 200 MHz Pentium Pro desktop PCs with DEC Tulip-based Ethernet cards (SMC 10/100 EtherPower). One machine ran either Linux 2.0.36 or Linux 2.0.36 with Prolac TCP; the other always ran Linux 2.2.7. They communicated over an otherwise idle 100 Mbit/s Ethernet with one hub.

Figure 6 shows the results of an echo test, which measures end-to-end latency and protocol processing overhead. In this test, the Prolac machine repeatedly writes four bytes of data to the other machine's `echo` port; the other machine echoes the data. Prolac's extra data copies do not affect this test significantly as the data size is small. To measure protocol processing time in isolation, we instrumented Linux and Prolac input and output processing functions using Pentium performance counters. Although both Linux and Prolac can output packets because of input events—for example, sending an ACK or more data in response to an input packet—this does not occur in the echo test. Linux IP layer processing time is included in output processing time.

Results show that Linux and Prolac TCP have comparable end-to-end latency to within a few microseconds. Prolac did slightly better in terms of cycles per packet (3067, versus 3360, average cycles to process a packet). The difference may be due to the two TCPs' timer implementations. Linux sets multiple fine-grained millisecond timers per connection to handle various timeouts; Prolac, following the 4.4BSD model, uses one fast timer (with 200 ms resolution) and one slow timer (with 500 ms resolution) for all of TCP. In the echo test, where timers are being set and cleared on each round trip, this results in Linux having significantly more timer overhead.

We also measured the impact of the compiler's inlining optimizations on Prolac TCP. With no inlining whatsoever, Prolac TCP processing time jumps by more than 100% to 6833 cycles per packet on the echo test, and end-to-end latency increases by 25%.

Prolac does significantly worse on a test measuring write throughput. In this test, the Prolac machine writes 8000 Kbytes of data to the other machine's `discard` port. Prolac's end-to-end write bandwidth was 8 Mbyte/s compared to Linux's 11.9 Mbyte/s. This is probably due to Prolac's two extra data copies, a hypothesis cycle count measurements tend to confirm. While Prolac's cycle count is lower than Linux's by 10% in the echo test, it is roughly twice as high as Linux's in the throughput test, and the only significant difference in the packets processed is the amount of data attached to them. Also, load instruction count on the

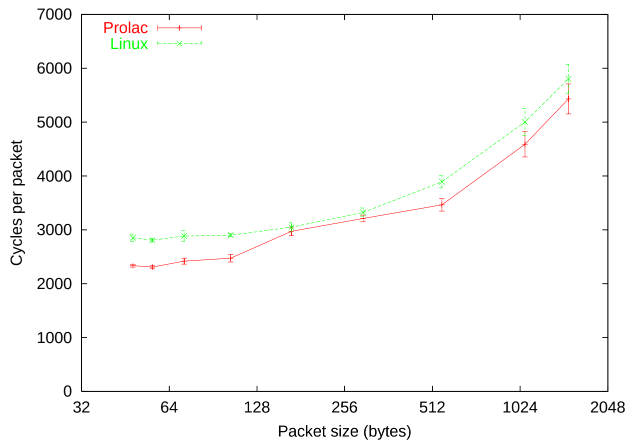

Figure 7: Input packet processing, in cycles per packet, for different packet sizes (echo test). Packet sizes include TCP and IP headers. The vertical bars indicate one standard deviation either way from the average.

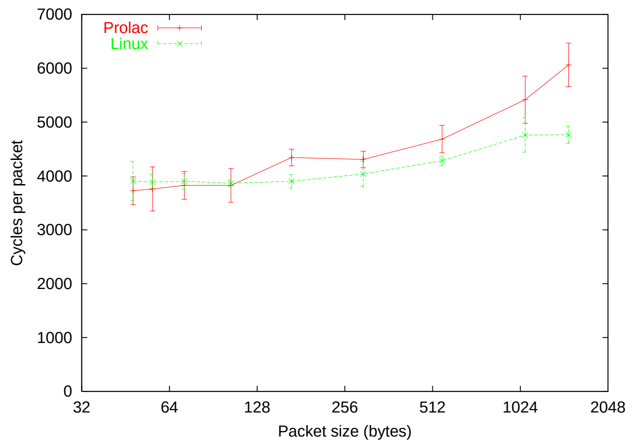

Figure 8: Output packet processing, in cycles per packet, for different packet sizes (echo test).

throughput test (using a slightly different configuration) was much higher for Prolac than for Linux.

Figures 7 and 8 justify this hypothesis further by showing, for the echo test, the effect of packet size on cycles per packet for both Linux and Prolac TCP. On the input processing path, Prolac has no extra copies and always slightly outperforms Linux; on the output processing path, however, there is one extra copy, and Prolac TCP performs worse on larger packets.

Overall, these results show that the Prolac language's performance overhead is minimal, even for a highly modularized implementation of a large, complex protocol like TCP.

Prolac may become more efficient in the future. First, we could eliminate the extra data copies in the input and output paths, which, as we have shown, are the key difference between Linux and Prolac TCP behavior. Second, we haven't yet applied all the compiler optimizations we have implemented, such as outlining. Third, there are optimizations we have not yet tried; Prolac's natural extensibility, combined with the compiler's ability to optimize modularity away, may allow us to exploit layer collapsing as discussed by Clark and Tennenhouse [8].

6 DISCUSSION

Prolac is intended to be robust, readable, and efficient enough for real-world use. Our focus on real-world application has made Prolac a better language: simpler, faster, more readable, more familiar. Nevertheless, it's fair to ask whether Prolac is truly suited for production use. This section discusses arguments for and against using Prolac in the real world.

We have discussed Prolac's advantages of readability, modularity, and extensibility throughout this paper. Due to careful structuring and Prolac's module-manipulation facilities, the Prolac TCP is substantially easier to understand piece by piece than other TCPs we have seen. Our TCP is certainly easier to extend than conventional TCPs: the protocol extensions are among the clearest sections of the Prolac TCP, and are guides for those wanting to extend the protocol further. It is easy to integrate Prolac into an existing C-based system. Finally, and in contrast to C or C++, Prolac's aggressive inlining and dynamic dispatch removal make it possible to use these ideas without sacrificing performance.

But there are compelling reasons to keep Prolac out of production code as well. The Prolac compiler, which is not small (20,000 lines of C++), would effectively become part of the system's code base. The compiler is stable but not bulletproof, and would need to be maintained. Furthermore, while we find Prolac very readable, it is also not C: it takes work to learn Prolac, particularly if you are used to large functions instead of small, interconnected ones. Some of Prolac TCP's design features may be usable in a more conventional language, but those languages' syntactic qualities, and the runtime costs of unoptimized dynamic dispatch, would make some of its best features less attractive.

We have shown that Prolac makes it much easier to extend protocols, but how common is that? If you don't need to modify a protocol, Prolac's ease of extension and modification is irrelevant. However, even production TCPs are changed all the time: extended with security measures like SYN cookies or optimizations like TCP Vegas [6], or even completely rewritten (Linux's TCP input processing functions were redone between Linux 2.0 and Linux 2.2).

7 CONCLUSION

Prolac's readability and features tailored for network protocols made writing TCP a pleasant experience, and the resulting specification is significantly more readable than any other we have seen. Its extensibility should be useful for protocol teaching, research, and development. Prolac's high-level language features were carefully designed to have minimal runtime costs, as demonstrated by experimental results.

ACKNOWLEDGEMENTS

We would like to thank Charles Blake for helping with our experimental setup, and Robert T. Morris and the anonymous reviewers for insightful comments.

REFERENCES

[1] Mark B. Abbott and Larry L. Peterson. A language-based approach to protocol implementation. *IEEE/ACM Transactions on Networking*, 1(1):4–19, February 1993.

[2] David P. Anderson. Automated protocol implementation with RTAG. *IEEE Transactions on Software Engineering*, 14(3):291–300, March 1988.

[3] Edoardo Biagioni. A structured TCP in Standard ML. In *Proceedings of the ACM SIGCOMM 1994 Conference*, pages 36–45, August 1994.

[4] Tommaso Bolognesi and Ed Brinksma. Introduction to the ISO specification language LOTOS. In Peter H. J. van Eijk, Chris A. Vissers, and Michel Diaz, editors, *The formal description technique LOTOS*, pages 23–73. North-Holland, 1989.

[5] Frédéric Boussinot and Robert de Simone. The ESTEREL language. Technical Report 1487, INRIA Sophia-Antipolis, July 1991.

[6] Lawrence S. Brakmo, Sean W. O'Malley, and Larry L. Peterson. TCP Vegas: new techniques for congestion detection and avoidance. In *Proceedings of the ACM SIGCOMM 1994 Conference*, pages 24–35, August 1994.

[7] Claude Castelluccia, Walid Dabbous, and Sean O'Malley. Generating efficient protocol code from an abstract specification. In *Proceedings of the ACM SIGCOMM 1996 Conference*, pages 60–71, August 1996.

[8] David D. Clark. Modularity and efficiency in protocol implementation. RFC 817, IETF, July 1982.

[9] Jeffrey Dean, David Grove, and Craig Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *Proceedings of the ECOOP 1995 Conference*, pages 77–101, August 1995.

[10] P. Dembinski and S. Budkowski. Specification language Estelle. In Michel Diaz, Jean-Pierre Ansart, Jean-Pierre Courtiat, Pierre Azema, and Vijaya Chari, editors, *The formal description technique Estelle*, pages 35–75. North-Holland, 1989.

[11] Diane Hernek and David P. Anderson. Efficient automated protocol implementation using RTAG. Report UCB/CSD 89/526, University of California at Berkeley, August 1989.

[12] 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.

[13] Stephen C. Johnson. Yacc—Yet Another Compiler-Compiler. Comp. Sci. Tech. Rep. #32, Bell Laboratories, July 1975. Reprinted as PS1:15 in *Unix Programmer's Manual*, Usenix Association, 1986.

[14] Eddie Kohler. Prolac language reference manual. Available from http://www.pdos.mit.edu/∼eddietwo/prolac/, January 1999.

[15] Steven McCanne and Van Jacobson. The BSD packet filter: a new architecture for user-level packet capture. In *USENIX Technical Conference Proceedings*, pages 259–269, San Diego, Winter 1993. USENIX.

[16] David Mosberger, Larry L. Peterson, Patrick G. Bridges, and Sean O'Malley. Analysis of techniques to improve protocol processing latency. In *Proceedings of the ACM SIGCOMM 1996 Conference*, pages 73–84, August 1996.

[17] Linda Ness. L.0: a parallel executable temporal logic language. In Mark Moriconi, editor, *Proceedings of the ACM SIGSOFT International Workshop on Formal Methods in Software Development*, pages 80–89, September 1990.

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

[19] Jon Postel. Transmission Control Protocol: DARPA Internet Program protocol specification. RFC 793, IETF, September 1981.

[20] Deepinder Sidhu, Anthony Chung, and Thomas P. Blumer. A formal description technique for protocol engineering. Technical Report CS-TR-2505, University of Maryland at College Park, July 1990.

[21] W. Richard Stevens. *TCP/IP Illustrated, Volume 1: The Protocols*. Addison-Wesley, 1994.

[22] W. Richard Stevens. TCP slow start, congestion avoidance, fast retransmit, and fast recovery algorithms. RFC 2001, IETF, January 1997.

[23] Gregor v. Bochmann. Methods and tools for the design and validation of protocol specifications and implementations. Publication #596, Université de Montréal, October 1986.

[24] Gary R. Wright and W. Richard Stevens. *TCP/IP Illustrated, Volume 2: The Implementation*. Addison-Wesley, 1995.