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].

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.

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++.

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.

```
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'.

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 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

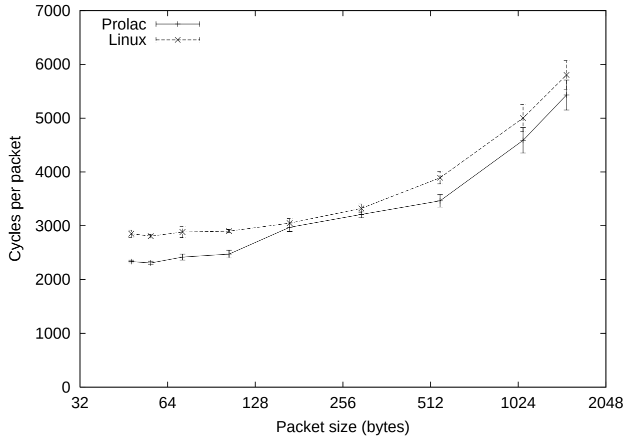

Figure 7: Input packet processing, in cycles per packet, for different packet sizes (echo test). Packet sizes include TCP and IP headers. The vertical bars indicate one standard deviation either way from the average.

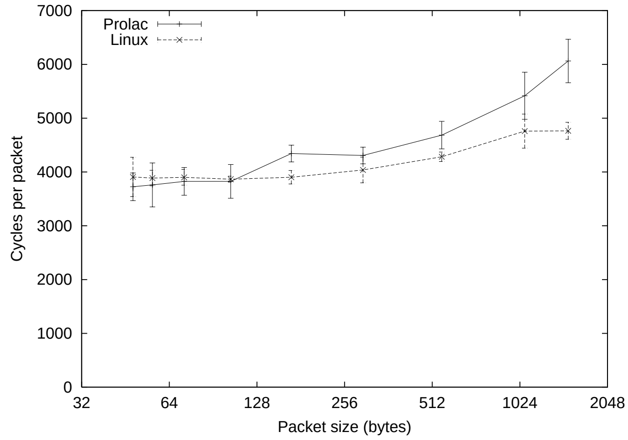

Figure 8: Output packet processing, in cycles per packet, for different packet sizes (echo test).

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.