Masking the Overhead of Protocol Layering

Robbert van Renesse *
rvr@cs.cornell.edu
Dept. of Computer Science
Cornell University
Ithaca, NY 14853

Abstract

Protocol layering has been advocated as a way of dealing with the complexity of computer communication. It has also been criticized for its performance overhead. In this paper, we present some insights in the design of protocols, and how these insights can be used to mask the overhead of layering, in a way similar to client caching in a file system. With our techniques, we achieve an order of magnitude improvement in end-to-end message latency in the Horus communication framework. Over an ATM network, we are able to do a round-trip message exchange, of varying levels of semantics, in about 170 μseconds, using a protocol stack of four layers that were written in ML, a high-level functional language.

1 Introduction

Modern network technology supports very low latency communication. For example, the U-Net [1] interface to ATM allows for 75 μsecond round-trip communication as long as the message is 40 bytes or smaller. For larger messages, the latency is at least twice as long. It is therefore important that protocols that are run over U-Net use small headers, and do not introduce much processing overhead.

Distributed systems employ communication protocols for reliable file transfer, window clients and servers, RPC, atomic transactions, multi-media communication, etc. Many of these protocols are complex. The traditional approach taken to deal with this complexity is protocol layering. A problem with layering is that it introduces overhead. One source of overhead is interfacing: crossing a layer costs some CPU cycles. The other is header overhead. Each layer uses its own header, which is prepended to every message and usually padded so that each header is aligned on a 4 or 8 byte boundary. When combined with a trend to very large addresses (of which at least two per message are needed), it becomes impossible to fit the total header in 40 bytes.

To deal with both these overheads of layering, we developed the Protocol Accelerator (PA). The PA eliminates both overheads almost entirely, and has resulted in one to three orders of magnitude of latency improvement over existing protocol implementations. For example, we are using it on an Objective Caml (O'Caml) [8] implementation of Horus [13, 12]—a software framework that supports the layering of general group communication protocols[1]. O'Caml is an object-oriented dialect of ML [9], a high-level, concise, garbage-collected language, that allows to develop complex protocols quickly and relatively error-free. Furthermore, the O'Caml code is suitable for automatic verification. Unfortunately, the code is slow compared to corresponding C code. Nevertheless, between two SunOS user processes on two Sparc 20s connected by a 140 Mbit/sec ATM network, we achieve a roundtrip latency of 170 μseconds using the PA, down from about 1.5 milliseconds in the original C version of Horus [12].

Within our group of C aficionados, this result came as a surprise. The reason we set out to use the O'Caml version of Horus was as a reference implementation for the use of verification and documentation. Previous work over ML, such as the CMU FOX project [2] that uses New Jersey Standard ML, reports a round-trip time of 36 milliseconds over an Ethernet, a cost of a factor of 9.4 compared to the same protocol (TCP/IP) implemented in C. Over ATM, this would be a even larger factor, since CPU overhead will matter even more compared to network latency.

The PA achieves its results using three techniques. First, the header fields that never change are only sent once. Second, the rest of the header information is carefully packed, ignoring layer boundaries, typically leading to headers that are much less than 40 bytes. Third, a semi-automatic transformation is done on the send and delivery operations, splitting them into two parts: one that updates or checks the header but not the protocol state, and the other vice versa. The first part is then executed by a special packet filter (both in the send and the delivery path) to circumvent the actual protocol layers whenever possible. The second part is executed, as much as possible, when the application is idle or blocked.

Although these techniques are applicable to the C implementation as well, the PA is relatively hard to retrofit into the existing C code. O'Caml lends itself much better to this kind of high-level manipulation. The optimizations that result are more significant than the low-level optimizations of C code.

This paper is organized as followed. In the next section we describe how we eliminated much of the header overhead

* This work was supported by ARPA/ONR grant N00014-92-J-1866

[1] In this paper we will only deal with point-to-point communication for clarity, but the techniques extend to multicast protocols.

Horus: http://www.cs.cornell.edu/Info/Projects/HORUS U-Net: http://www.cs.cornell.edu/Info/Projects/U-Net Objective Caml: http://pauillac.inria.fr/ocaml

Table 1: URLs for various information of interest.

of layering. Section 3 shows how we eliminated much of the CPU overhead in the critical path. The implementation is described in Section 4. In Section 5, we report the resulting performance. Section 6 discusses some of the current problems of the PA.

2 Reducing Header Overhead

In traditional layered protocol systems, each protocol layer designs its own header layout. The headers are concatenated and prepended to each user message. For convenience, each header is aligned to a 4 or 8 byte boundary to allow easy access. In systems like the x-kernel or Horus, where many simple protocols may be stacked on top of each other, this may lead to extensive padding overhead.

Also, some fields in the headers, such as the source and destination addresses, never change from message to message. Instead of agreeing on these values, they are included in every message, and used as the identifier of the connection to the peer. Since addresses tend to be large, and are getting significantly larger to deal with the rapid growth of today's internet, this is no longer a good idea.

In this section, we describe how the PA reduces the header overhead significantly by eliminating padding, and agreeing on immutable fields in headers. For clarity, note that we employ a PA per connection rather than per host or process.

2.1 Header Information Classes

To enable the optimizations, the PA divides the fields in the protocol headers into four classes:

1. *Connection Identification* — fields that never change during the period of a connection, and that are typically used for identifying connections. Examples of such fields are the source and destination addresses and ports of the communicating peers, as well as byte-ordering information of their architectures.

2. *Protocol-specific Information* — fields that are important for the correct delivery of the particular message frame. We require that protocol-specific information depend only on the protocol state, and not on the contents or length of the message, or the time at which the message was sent. Examples are the sequence number of a message, or the message type (*e.g.*, data, ack, or nak).

3. *Message-specific Information* — other fields that need to accompany the message, such as the message length and checksum, or a timestamp. Typically, such information does not depend on the protocol state, but just on the message itself. However, we do not require this.

4. *Gossip* — fields that technically do not need to accompany the message, but are included for message efficiency. Piggybacked acknowledgements fall into this category. Like protocol-specific information, these fields cannot depend on the message contents itself. Often these fields do depend on the protocol state, but can be out-of-date without affecting the protocol correctness.

Each protocol layer requests a set of fields to be included in the header using calls to

handle = add_field(class, name, size, offset);

Here *class* specifies the header information class, *name* is the name of the field (which does not need to be unique), and *size* specifies the size of the field in bits. *Offset* specifies the bit offset of the field in the header if this is important, but is usually set to -1 to indicate *don't care*. The function returns a *handle* for later access.

After the initialization function of all layers have been called, the PA collects all the fields, and compiles them into four compact headers, one for each class. It does so as efficiently as possible, observing size, and if so requested, offset, but not layering. Therefore, fields requested by different layers may be mixed arbitrarily, minimizing padding while optimizing alignment. In the original Horus system, for example, each layer's header was aligned to 4 bytes, resulting in a total padding of at least 12 bytes—for a fairly small protocol stack—and going up quickly for each additional layer.

The PA provides a set of functions to read or write a field. The functions take byte-ordering into account, so that layers do not have to worry about communicating between heterogeneous machines. Also, this makes the entire system directly portable to 64-bit architectures.

2.2 Connection Cookies

The PA includes the Protocol-specific and Message-specific information in every message. Currently, although not technically necessary, Gossip information is also always included, since it is usually small. However, since the Connection Identification fields never change, they are only included occasionally because they tend to be large. The idea is similar to Van Jacobson's TCP/IP header compression technique for point-to-point links [6], but generalized for arbitrary protocols. A PA message starts out with an 8-byte header, called the *Preamble* (see Figure 1). The Preamble contains three fields:

1. Connection Identification Present Bit — a single bit that is set if and only if the Connection Identification is included. If set, the Preamble is immediately followed by the Connection Identification header, the remaining headers, and the user message itself, in that order. If cleared, the connection identification is left out.

2. Byte Ordering Bit — a single bit that is set if the ordering of the bytes in this message is little endian, and clear if the ordering is big endian. Other orderings are not supported.

3. Connection Cookie — a 62-bit magic number. It is chosen at random and identifies the connection.

The PA includes the Connection Identification on the first message. In addition, it is included on retransmissions and other unusual messages. The receiver remembers for each connection what the current (incoming) cookie

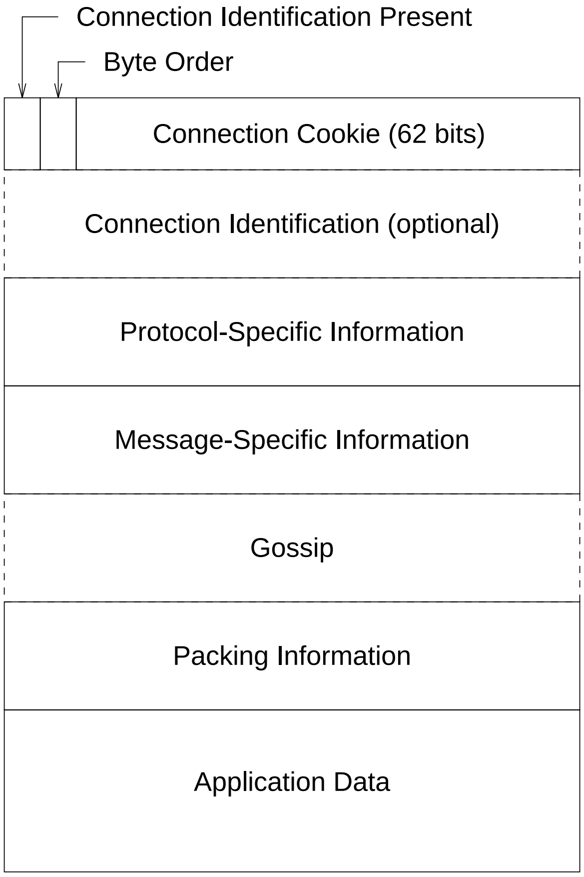

Figure 1: The format of a PA message. Note that the headers collect fields of the same category, rather than fields of the same protocol layer. Except for the first 64 bits, all headers are variable length. The Packing Information is described in Section 3.4.

is. When a message is received with an unknown cookie, and the Connection Identification Present Bit cleared, it is dropped. If the bit is set, the Connection Identification is used to find the connection.

Since the Connection Identification tends to include very large identifiers, this mechanism reduces the amount of header space in the normal case significantly. For example, in Horus, the connection identification typically occupies about 76 bytes. Cookies also reduce connection lookup time for delivery of messages. In [7], a similar idea was exploited for TCP/IP over an FDDI network of DEC Alphas, and resulted in a 31% latency improvement.

There is a problem with this approach. For example, if the first message is lost, the next message will be dropped as well because the cookie is unknown and the connection identification is not included. Currently, the PA relies on retransmission by one of the protocol layers to deal with this problem. Perhaps a better solution would be to agree on a cookie before starting to use it.

Finally, we note that the gossip information does not necessarily need to be included in each message either. For older networks, when there was a considerable overhead for each message sent, this made sense. But now that we have a low latency for small messages, it makes sense to send the gossip information in separate messages. The PA could automate this, since it knows which information is gossip. However, since we have only seen a small amount of gossip information until now, we have not dealt with this issue yet.

3 Eliminating Layered Protocol Processing Overhead

In most protocol implementations, layered or not, a lot of processing is done between the application's send operation, and the time that the message is actually sent out onto the network. The same applies between the arrival of a message and the delivery to the application. We call these paths between the application and the network the *send critical path* and the *delivery critical path*. All other processing done by the protocol can be done in parallel with the application, and is mostly of little concern (but see Section 6).

The processing that is done on the critical path is complicated by layering. The message passes through the layers, and only the state of the current layer can be inspected and updated. Without layers, the protocol implementor would be free to reorder the protocol processing arbitrarily to reduce the length of the critical path [3]. Furthermore, the processor's instruction cache is likely to be less effective if the critical path is divided over many software modules.

The PA uses three approaches to deal with this. First, it minimizes the critical path by delaying all updating of the protocol state until after the actual message sending or delivery. Secondly, it *predicts* the Protocol-specific header of the next message, so in most cases the creation or checking of the protocol-specific header can be eliminated. Finally, it uses packet filters, both in the send and delivery critical paths to avoid passing through the layers altogether.

This section discusses these techniques.

3.1 Canonical Protocol Processing

In this subsection, we observe that the send and delivery processing of a protocol layer can be done in two phases:

1. *Pre-processing Phase* — In this phase, the message header is built (in case of sending), or checked (in case of delivery), but the protocol state is left untouched. For example, a sequence number can be added to a message, or the sequence number in the message can be checked against the current one, without changing the state of the protocol.

2. *Post-processing Phase* — in this phase the protocol state is updated. For example, when sending or delivering, the sequence number maintained in the protocol state has to be incremented.

We call this *canonical protocol processing*. We use the term *pre-sending* for pre-processing of sending, and *pre-delivery* for the pre-processing of delivery. Similarly, we use the terms *post-sending* and *post-delivery*. Note that it is always possible to convert a protocol to its canonical form. If necessary, the protocol state can be checkpointed and restored to implement the pre-processing part, but typically a much less drastic and more efficient approach can be taken. In a layered system, the pre-processing at every layer may be done before the post-processing at any layer. The significance of this is that a message can be sent onto the network, or delivered to the application (in case the check succeeds), without changing the state of any protocol layer immediately. The update of the protocol state (*e.g.*, incrementing a sequence number or saving a message for retransmission) can be done afterwards, out of the critical path—in a lazy fashion—but before the next send or delivery operation.

3.2 Header Prediction

In this subsection, we observe that the protocol-specific information of a message can be predicted, that is, calculated before the message is sent or delivered. This is because this information does not depend on the message contents or the time on which it is sent.

After the post-processing has finished for a previous message, the protocol-specific header of the next message could be generated by pre-processing a dummy message. However, we found it more convenient to have the post-processing phase of the previous message predict the next protocol header immediately. This is an artifact of layers generating their own messages (acknowledgements, retransmissions, etc.). Such messages only update those fields in the predicted header that are maintained by that layer or the layers below.

Each connection maintains a predicted protocol-specific header for the next send operation, and another for the next delivery (much like a read-ahead strategy in a file system). For sending, the gossip information can be predicted as well, since this does not depend on the message contents either.

At the time of sending, only the message-specific header has to be generated. We will show in the next subsection how we do this by means of a packet filter. This packet filter may also be used to check the contents of the message to decide whether to use the predicted header or not.

When delivering, the contents of the protocol-specific header can be quickly compared against the predicted header. For delivery, the gossip information is not important. The message-specific information is checked using a packet filter in the delivery path.

Each layer can disable the predicted send or delivery header (*e.g.*,, when the send window of a sliding window protocol is full). For this, each header has a counter associated with it. When zero, the header is enabled. By incrementing the counter, a layer disables the header. The layer eventually has to decrement the counter. When all layers have done so, the header is automatically re-enabled.

3.3 Packet Filters

Not all header information, whether in the send or delivery path, can be predicted. A good example is the checksum of a message. Another example is when a message arrives out of order. To deal with these situations, the PA employs packet filters not only in the delivery path, such as done in other systems (*e.g.*, [10]), but also in the send path. The send filter is unusual in that it can update headers. Even in the delivery path, the filter is used in an atypical way: rather than doing a pattern match on the header to decide to which module or connection the message should be forwarded, it checks the message-specific information for correctness (*e.g.*,, that the checksum matches the contents of the message).

As in [10], the packet filter is a stack machine. The operations are listed in Table 2. A packet filter program is a series of such operations that operate on a message header. Particularly interesting is the POP_FIELD operation, which takes a value of the stack and stores it in the given field of a header. There are no loop or function constructs, so a packet filter program can be checked in advance, and the necessary size for the stack can be calculated (typically just a few entries). In case fields are conveniently aligned, the packet filter is optimized automatically using some customized instructions. Packet filter programs are currently interpreted. We note that in the Exokernel project, a significant performance improvement was obtained by compiling packet filter programs into machine code [4]. We intend to adopt this approach eventually.

The packet filters are constructed by the layers themselves, at run-time. Each layer adds instructions to both packet filters for their particular message-specific fields. Typically, the packet filters only need be programmed once (at protocol stack initialization time). However, if the message-specific information depends on the protocol state, part of the packet filter program may be rewritten when the protocol state is updated in the post-processing phase, at run-time.

Using the packet filter for message-specific information, along with the header prediction technique, sending and delivery pre-processing can usually be avoided altogether. This means that the protocol stack is not invoked until after an actual send or delivery, masking the overhead of the layered architecture, *and* the overhead of using a relatively slow implementation using O'Caml. Only if the predicted header is disabled, or if the packet filter returns a non-zero value, or, in case of delivery, the cookie is unknown or the protocol-specific header does not match the predicted one, the pre-delivery phase of the protocol is executed.

3.4 Message Packing

The PA as described so far will reduce the latency of individual messages significantly *if* they are spaced out far enough to allow time for post-processing. If not, messages will have to wait until the post-processing of every previous message completes. To reduce this overhead, the PA uses *message packing* [5] to deal with backlogs.

After the post-processing of a send operation completes, the PA checks to see if there are messages waiting. If there are more than one, the PA will pack these messages together into a single message. The single message is now processed in the usual way, which takes only one pre-processing and post-processing phase. When the packed message is ready for delivery, it is unpacked and the messages are individually delivered to the application.

The PA therefore uses a sixth header to describe how the messages are packed, which we call the Packing Header (see Figure 1). Currently, the PA only packs together messages of the same size. The Packing header contains the size of each of the messages. We may use a more sophisticated header, such as used in the original Horus system, so that any list of messages may be packed, if applications so demand. Special care is required in case messages are prioritized.

The PA also buffers messages in case the predicted send header is disabled, for example, in case the send window of a sliding window protocol is full. Again, this leads to a backlog which is dealt with effectively using the packing mechanism.

4 Implementation of the PA

The architecture of the PA is shown in Figure 2. Horus has a PA for each connection. Each PA maintains the layout descriptors for the headers, as well as a table of information both for sending and delivery, as shown in Table 3.

A condensed implementation of the PA appears in Figure 3. The real implementation of the PA, including the code for the packet filter and the message management code, is

Operation	Argument	Action
PUSH_CONSTANT	integer	push an integer onto the stack
PUSH_FIELD	field handle	push a field onto the stack
PUSH_SIZE		push the size of the message
DIGEST	function ptr	push a message digest
POP_FIELD	field handle	pop top of stack into a field
ADD, SUB, ...		do operation on top two entries
EQ, NE, LT, ...		comparison of top two entries
RETURN	integer	return the given value
ABORT	integer	return value if top entry non-zero

Table 2: The operations of a packet filter.

Field	Type	Purpose
mode	IDLE, PRE, or POST	state of operation
predict_msg	header[6]	all six headers
disable	integer	predicted header disabled
pre_msg	message	message to post-process
packet_filter	packet filter	packet filter to apply
backlog	list of messages	waiting for processing

Table 3: Each Protocol Accelerator maintains two tables of this information, one for sending and one for delivery.

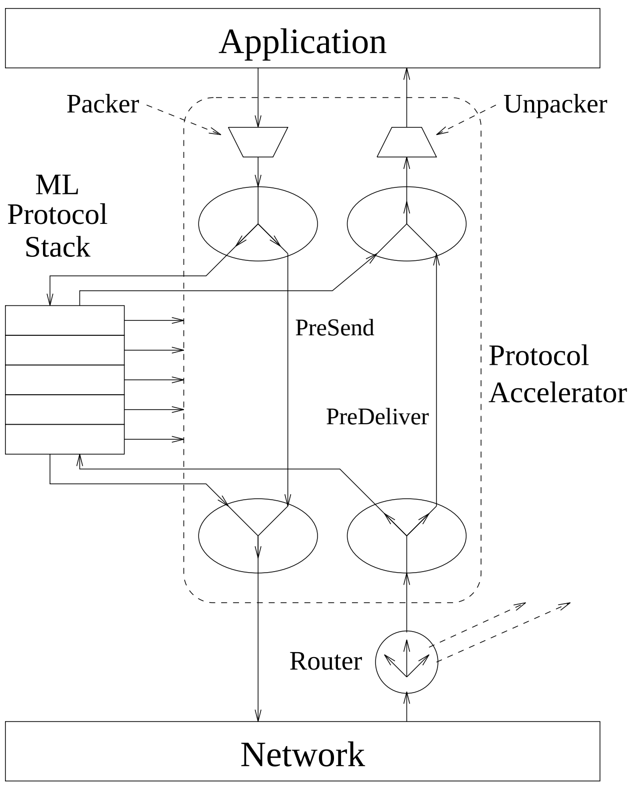

Figure 2: The architecture of the Protocol Accelerator. The application, network driver, router (which delivers messages to the correct PA), and PA itself are all written in C. The protocol stack may be written in any language, but is currently written in O'Caml—a dialect of ML.

about 1500 lines of C. When the application invokes `send()`, the PA first checks to see if the disable counter is zero. If not, the message is added to the backlog. If so, the packing information and the predicted header are added to the message, and the send packet filter is run. This packet filter fills out the message-specific and gossip information. It can be programmed to fail, for example, if the message is too large to be sent unfragmented. If successful, the connection cookie is pushed onto the message and it is sent. Then the message is passed to the protocol stack for post-processing. When this completes, the backlog will be inspected.

If this path could not be followed, the message will be passed to the protocol stack for pre-processing. This usually results in actual sending, but any layer may buffer the message until later instead. When the pre-processing completes, the message is sent out onto the network and is passed to the stack again for post-processing. This will update the predicted header for the next message.

The `conn_ident_present` bit is inspected when a message arrives from the network, If set, the connection identification is used to find the connection. If unsuccessful, the message is dropped. If successful, the connection cookie is stored. If the connection identification is not present in the message (the usual case), the cookie in the message is used to locate the connection. Next, the packet filter is run to see if the message-specific information is acceptable. If so, and if the disable counter is zero *and* the protocol-specific information is the same as the predicted information, the `deliver`() routine is invoked.

The `deliver`() routine pops the packing information of the message. If the message is not packed, it is handed to the application immediately. If the message is packed, it is unpacked and the individual messages are handed to the application separately. After completion, the protocol stack is invoked for post-processing.

If the disable counter was non-zero, or if the protocol-specific information did not match, the message is handed to the protocol stack for pre-processing first. This usually

```
/* called by application */
send(con, msg){
   if (con->send.disable > 0) {
      add_to_backlog(con->backlog, msg);
      return;
   }
   push_packing_info(msg);
   push_predict_header(con->predict, msg);
   if (run_packet_filter(con->send.filter, msg)) {
      push_cookie(con->cookie, msg);
      to_network(con, msg);
      to_protocol_stack(con, POSTSEND, msg);
   }
   else to_protocol_stack(con, PRESEND, msg);
}

/* called by network driver */
from_network(msg){
   pop_preamble(msg, &preamble);
   if (conn_ident_present(preamble)) {
      pop_conn_ident(msg, &conn_ident);
      if (!get_conn_by_ident(conn_ident, &con))
         return;
      con->cookie = preamble.cookie;
      con->conn_ident = conn_ident;
   } else
   if (!get_conn_by_cookie(preamble.cookie, &con))
      return;
   if (!run_packet_filter(con->recv.filter, msg))
      return;
   if (con->recv.disable == 0 &&
       msg->predict.prot_spec ==
                         con->predict.prot_spec) {
      deliver(con, msg);
      to_protocol_stack(con, POSTDELIVER, msg);
   }
   else to_protocol_stack(con, PREDELIVER, msg);
}

/* called by from_network() and PREDELIVER */
deliver(con, msg){
   pop_packing_info(msg, &pack_info);
   if (is_packed(pack_info)) {
      msg_list = unpack(pack_info, msg);
      for_each msg in msg_list
         to_application(msg);
   }
   else to_application(msg);
}
```

Figure 3: The code of the Protocol Accelerator.

results in an invocation of deliver() as well. However, the message may be buffered or dropped instead. When the pre-processing completes, the message is handed to the stack again for post-processing.

The post-processing of sending and delivery, as well as garbage collection, is carefully scheduled by the PA so as to minimize the round-trip latency. Since, as we will see in the next section, post-processing and garbage collection actually take longer than the U-Net round-trip time, post-processing and garbage collection are scheduled to occur after message deliveries. On slower networks, such as Ethernet, post-processing and garbage collection could be done between round-trips as well.

What	Performance
one-way latency	85 μsecs
message throughput	80,000 msgs/sec
#roundtrips/sec	6000 rt/sec
bandwidth (1 Kbyte msgs)	15 Mbytes/sec

Table 4: The basic performance of the O'Caml protocol stack using the Protocol Accelerator. Except for measuring the bandwidth, messages with 8 bytes of user data have been used.

5 Performance

In this section, we report on the user-level process to process performance of this system. We used two Sun SparcStation 20s, each running SunOS 4.1.3. The workstations were connected by a Fore 140 Mbit/sec ATM network. The firmware and driver used are not by Fore; instead we used the U-Net software [1]. The messages in our experiments contained 8 bytes of user data, unless noted otherwise. The raw U-Net one-way latency in this configuration is about 35 μsecs. U-Net provides unreliable communication, but in our experiments no message loss was detected. In spite of this, we used a protocol stack that implements a basic sliding window protocol, with a window size of 16 entries, written in O'Caml [8]. For predictable results without hiccups, we triggered garbage collection after every message reception unless noted otherwise. The basic performance results are listed in Table 4.

Figure 4 shows the breakdown of computation and communication costs. The sender (on the right) first spends about 25 μsecs before the message is handed to U-Net. The message is received 35 μsecs later. It is delivered in another 25 μsecs. The receiver immediately sends a reply message, which takes the same amount of time to delivery. The total round-trip time is therefore about 170 μsecs. After delivery of a message, the PA does the post-processing of both sending and delivery, and garbage collection. The time when this occurs depends on which layers are stacked together.

In this case, four layers have been stacked together to implement a basic sliding window protocol. The post-processing of sending takes about 80 μsecs, while the post-processing of delivery takes 50 μsecs. Garbage collection, in this case, takes between 150 and 450 μsecs, with an average of about 300 μsecs (which we used in the figure). The figure shows, using dashed lines, when the earliest next possible round-trip is possible. Because of the post-processing and garbage collection overheads, the maximum number of round-trips per second over this stack is about 1900, with an average round-trip latency of about 400 μseconds (in the worst case, about 550 μseconds). Only if fewer than 1650 roundtrips per second are done, a round-trip latency of 170 μsecs can be maintained (see Figure 5).

To see how each layer adds to the overhead, we also measured the performance for a stack, where the layer that actually implemented the sliding window was stacked twice. This layer is about 200 lines of O'Caml code. We found that the post-processing of the send and delivery operations take about 15 μsecs each. We did not find additional overhead for garbage collection.

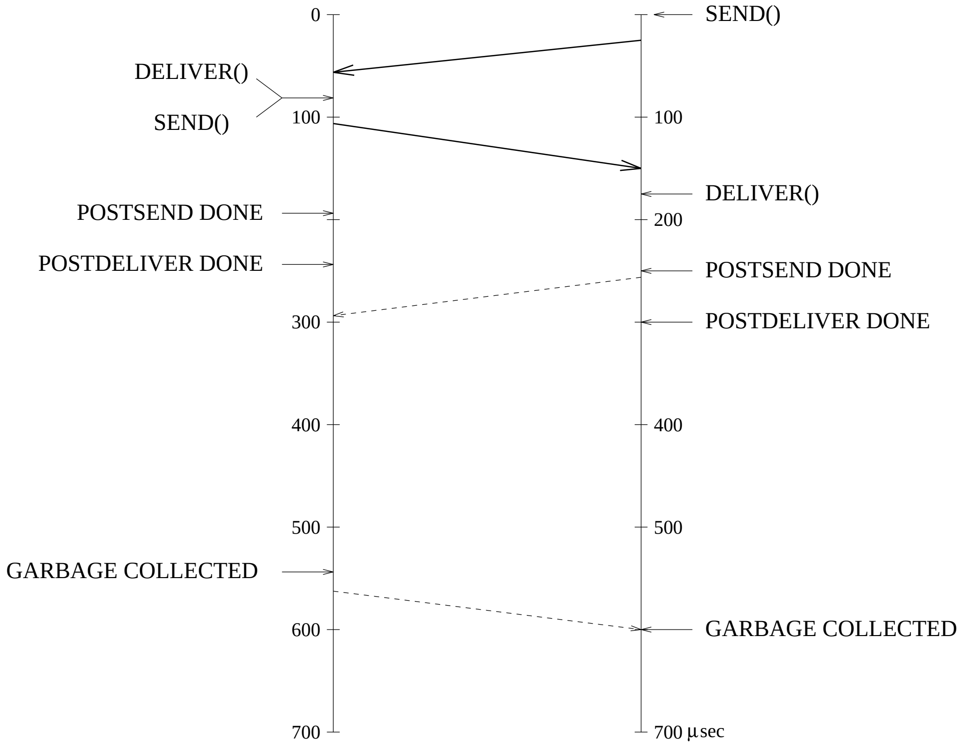

Figure 4: A breakdown of the round-trip execution. The first round-trip is the typical case. The dashed one depicts the round-trip performance if the system is pushed to its limits, and results in a higher latency.

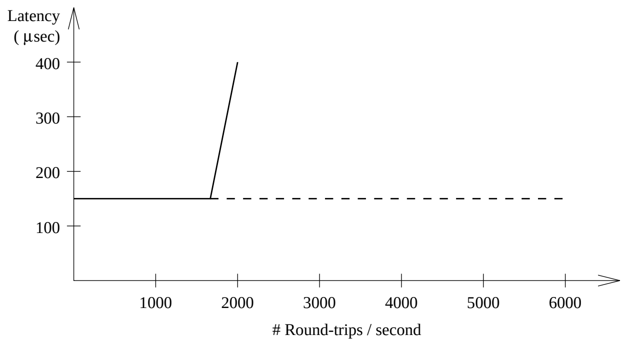

Figure 5: The round-trip latency as a function of the number of round-trips per second. The solid line is when each round-trip is followed by a garbage collection phase. The dashed line is for the case when garbage collection is only done occasionally. This results in improved performance, but leads to occasional hiccups.

It is not necessary to garbage collect after every round-trip. By not garbage collecting every time, we can increase the number of round-trips per second to about 6000. With the current protocol accelerator, this is in fact the maximum that can be achieved, because all of the post-processing is done between the actual sending and delivery of the messages. That is, even by rewriting the protocol stack in assembly this performance could not be improved. However, the garbage collection does lead to occasional hiccups which last about a millisecond.

We have seen that the PA improved the round-trip communication dramatically. The packing technique used by the PA also improves one-way streaming performance. For example, we are able to sustain about 80,000 8 byte messages per second streaming from a sender to a receiver (comparable to the C version of Horus). In addition, we achieve the full bandwidth of the underlying communication network (in this case about 15 Mbytes/sec).

These are best case behaviors. We have no reason to believe that the worst case behavior of Horus can be made worse by using the PA. Therefore the PA leads to an overall improvement of performance, no matter what the pattern of communication is.

6 Discussion

In this section, we will discuss some of the problems of the PA, and the use of a high-level language for protocol implementation. In particular, we will look at fragmentation/reassembly, the application of the PA to standard protocols like TCP/IP, the reduced maximum load that can be handled by layering, and the negative effects of garbage collection. We will present some possible solutions to each of these problems.

Fragmentation/Reassembly

Most networks only accept messages up to a certain maximum size, so that fair access to the medium can be guaranteed. Also, with any bit error rate, the chances of a large message getting through undamaged are smaller than those for a small message. For these reasons, large application messages have to be split into fragments before being sent. The fragments are reassembled at the peer side before delivery.

The PA does not fragment messages. Therefore, the pre-processing of large messages needs to be handled by the protocol stack. The fragmentation/reassembly layer adds code to the send packet filter to reject messages over a certain size to accomplish this. Also, by using a protocol-specific bit that is non-zero if and only if the message is a fragment of a larger message, it makes sure that the receiving PA does not "predict" the header, so that it is passed to the protocol stack for reassembly.

Usually, if low latency is a consideration only for small messages, this is not a problem. If ever this becomes a problem, the packing/unpacking mechanism of the PA can be extended with fragmentation/reassembly functionality.

Application to Standard Protocols

The PA may be applied to standard protocol implementations, such as TCP/IP, to improve latency. However, it cannot be used to communicate with a peer that uses a conventional implementation without the PA. The problem is that the peer would not understand the connection cookie, nor the different layout of the headers.

However, even if the header compression techniques cannot be used, the pre-processing and post-processing techniques *are* applicable. Messages would be larger, and comparing the protocol-specific information with the predicted information will be more expensive. Nevertheless, a significant latency improvement may be expected.

Maximum Load

The PA results in improvements of latency and message throughput, but not of all aspects of performance. Consider a server that uses a PA for each client. As seen in the performance section, the maximum number of Remote Procedure Calls that an individual client may do is limited to 6000 per second. Even with multiple clients, a server cannot process more than 6000 requests per second total, because the post-processing will consume all the server's available CPU cycles. For more complex stacks, this maximum is reduced even further.

This limit may be improved in at least three ways. With the current PA, the maximum number of round-trips per client that can be achieved is about 6000 roundtrips per second per connection (1/.000170), but we may be able to use less CPU time to accommodate more clients. First, an even faster implementation of the ML language may be chosen, or a different language altogether (*e.g.*,, C or Modula-3). We are currently working on annotating the critical path of the post-processing code, and generating a more efficient in-line version of it. Next, we plan to compile highly optimized code for the in-line "by-pass" function on the fly.

Secondly, modern servers are likely to be multi-processors. The protocol stacks for different connections may be divided among the processors. Since the protocol stacks are independent, there will be no synchronization necessary. This way the maximum number of RPCs per second is multiplied by the number of processors.

Finally, the server may be replicated. In this case, synchronization of the server's processing and data may be required, leading to additional, complex protocols. However, this is exactly the intention of this work—to encourage distribution. Distribution introduces complexity, but allows for higher client loads. The complexity can be managed by layering and high-level languages, while the overhead of this is masked by the PA.

Use of a High-Level Language

It is well-known that most of the execution time in a system is spent in a fraction of the system. By off-loading this "critical path" execution to a specialized piece of code, the PA, we have gained flexibility in how we implement the bulk of the system. To implement the bulk of the system, it is important to use a flexible, layered architecture, and a high-level language to minimize implementation cost, and maximize experimentation and optimization. The use of a high-level language typically implies garbage collection, which may lead to hiccups in the execution. Moreover, the lazy reclamation of resources may lead to bad caching or paging performance.

In our case, we have chosen Objective Caml, a high-level object-oriented functional language, to implement all of our system except for the PA. O'Caml uses a "stop, collect, and resume" garbage collector that does produce occasional hiccups. Obviously, exactly how often depends a lot on what code gets executed, which, in turn, depends on the application, the protocol layers that are stacked, and the particular network driver that is used. Such a garbage collector gives good performance for interactive applications, but the hiccups may be unacceptable for applications that require predictable high-performance networking.

We have been experimenting with allocating and deallocating "high-bandwidth" objects explicitly (in particular, messages)—a practice that O'Caml allows. Doing this, the number of garbage collections reduce dramatically (exact measurements are not available at this time).

The hiccups could be eliminated entirely by developing a "real-time" garbage collector that executes in parallel with the rest of the system (see, for example, [11]).

7 Conclusion

The Protocol Accelerator (PA) eliminates most of the overhead associated with layering of protocols, allowing for clean, efficient, and flexible protocol implementations. The PA achieves this using a variety of techniques, including header compression, header prediction, moving code of the critical path, and message packing. The PA has been used in an ML version of the Horus group communication system, and resulted in 170 μseconds round-trip delays (down from about 1.5 milliseconds in the original C version of Horus that does not use the PA), and a throughput of over 80,000 messages per second over an ATM network.

Acknowledgements

The idea for this work came from discussions with Mark Hayden. Mark is also responsible for several of the solutions to problems presented in Section 6, including code by-passing and explicit allocation and deallocation of high-bandwidth objects. We thank the Caml group at INRIA and the U-Net group at Cornell for their generous help, and Ken Birman, Roy Friedman, Greg Morrisett, and Werner Vogels, and the SIGCOMM reviewers for helpful comments.

References

[1] Anindya Basu, Vineet Buch, Werner Vogels, and Thorsten von Eicken. U-Net: A user-level network interface for parallel and distributed computing. In *Proc. of the Fifteenth ACM Symp. on Operating Systems Principles*, pages 40–53, Copper Mountain Resort, CO, December 1995.

[2] Edoardo Biagioni. A structured TCP in Standard ML. In *Proc. of the '94 Symp. on Communications Architectures & Protocols*, pages 36–45, University College London, UK, August 1994. ACM SIGCOMM.

[3] David D. Clark and David L. Tennenhouse. Architectural considerations for a new generation of protocols. In *Proc. of the '90 Symp. on Communications Architectures & Protocols*, pages 200–208, Philadelphia, PA, September 1990. ACM SIGCOMM.

[4] Dawson R. Engler, M. Frans Kaashoek, and James O'Toole. Exokernel: An operating system architecture for application-level resource management. In *Proc. of the Fifteenth ACM Symp. on Operating Systems Principles*, pages 251–266, Copper Mountain Resort, CO, December 1995.

[5] Roy Friedman and Robbert van Renesse. Packing Messages as a Tool for Boosting the Performance of Total Ordering Protocols. Technical Report 94-1527, Cornell University, Dept. of Computer Science, July 1995. Submitted to IEEE Transactions on Networking.

[6] Van Jacobson. Compressing TCP/IP headers for low-speed serial links. RFC 1144, Network Working Group, February 1990.

[7] Jonathan S. Kay. *PathIDs: A Mechanism for Reducing Network Software Latency*. PhD thesis, Univ. of California, San Diego, May 1994.

[8] Xavier Leroy. *The Caml Special Light system release 1.10*. INRIA, France, November 1995.

[9] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. The MIT Press, 1990.

[10] Jeffrey C. Mogul, Richard F. Rashid, and Michael J. Accetta. The Packet Filter: An efficient mechanism for user-level network code. In *Proc. of the Eleventh ACM Symp. on Operating Systems Principles*, pages 39–51, Austin, TX, November 1987.

[11] Scott Nettles and James O'Toole. Real-time replication garbage collection. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, pages 217–226, Albuquerque, New Mexico, June 23–25 1993. *SIGPLAN Notices,* 28(6).

[12] Robbert Van Renesse, Kenneth P. Birman, Roy Friedman, Mark Hayden, and David A. Karr. A Framework for Protocol Composition in Horus. In *Proc. of the Fourteenth ACM Symp. on Principles of Distributed Computing*, pages 80–89, Ottawa, Ontario, August 1995. ACM SIGOPS-SIGACT.

[13] Robbert van Renesse, Kenneth P. Birman, and Silvano Maffeis. Horus: A flexible group communication system. *Communications of the ACM*, 39(4):76–83, April 1996.