RPC in the x-Kernel: Evaluating New Design Techniques*

Norman C. Hutchinson, Larry L. Peterson, Mark B. Abbott, and Sean O'Malley

Department of Computer Science
University of Arizona
Tucson, AZ 85721

Abstract

This paper reports our experiences implementing remote procedure call (RPC) protocols in the x-kernel. This exercise is interesting because the RPC protocols exploit two novel design techniques: *virtual protocols* and *layered protocols*. These techniques are made possible because the x-kernel provides an object-oriented infrastructure that supports three significant features: a uniform interface to all protocols, a late binding between protocol layers, and a small overhead for invoking any given protocol layer. For each design technique, the paper motivates the technique with a concrete example, describes how it is applied to the implementation of RPC protocols, and presents the results of experiments designed to evaluate the technique.

1 Introduction

The x-kernel is a configurable operating system kernel designed to simplify the process of implementing network protocols. The x-kernel is implemented on Sun-3 workstations and supports multiple address spaces, a protected kernel address space, light-weight processes, a library of tools used to construct protocols, and an object-oriented infrastructure that supports the composition of protocols [5, 6]. This infrastructure is similar to System V Unix streams [16]. It differs from packet filters [9] because it supports kernel-based rather than user-based protocol implementations and because it supports multiple demultiplexing points.

To date, we have implemented standard protocol configurations in the x-kernel with the intent of demonstrating that the x-kernel is general enough to accommodate a variety of protocols without imposing a significant performance penalty on any particular protocol. Our experience suggests that the x-kernel satisfies both ends of this goal. On the one hand, we have implemented many dissimilar protocols within the framework of the x-kernel; examples include the Arpanet protocol suite [8,12,13,14, 15,19], the Psync many-to-many IPC protocol [11], several application-level protocols [2,17], and numerous device drivers. On the other hand, performance studies show that protocols implemented in the x-kernel run as fast as, and sometimes faster than, their counter-part implementations in less structured environments. For example, the user-to-user round trip delay using the UDP/IP protocol suite is 2.00 msec in the x-kernel and 5.36 msec in SunOS Release 4.0 (4.3BSD Unix). These times were measured on Sun 3/75 workstations connected by a 10Mbps ethernet.

Instead of considering whether conventional protocol suites can be implemented in the x-kernel, this paper addresses a different question: It explores the extent to which the x-kernel architecture frees us to think about new protocol configurations not easily supported in other operating systems. In particular, it considers how one can take advantage of the x-kernel when implementing remote procedure call (RPC) protocols. The x-kernel has three distinguishing features that suggest novel implementations are possible:

- The x-kernel supports a uniform interface to all protocols. Thus, if two or more protocols provide the same semantics—e.g., an unreliable message delivery service—then it is easy to substitute one for another.
- The x-kernel supports late binding between protocol layers. Rather than encode into a low-level protocol's implementation information about the high-level protocols that depend on it, the high-level protocols "open" the low-level protocol at run time. It is therefore possible to delay this binding so as to select exactly the right protocol for a particular situation.
- Layers in the x-kernel are light-weight. Instead of requiring a context switch, it costs only one procedure call to pass a message from a high-level protocol to a low-level protocol, and vice versa. This makes it economical to explicitly divide a large protocol into multiple layers.

This paper shows how two new design techniques that

*This work supported in part by National Science Foundation Grants CCR-8811423 and CCR-8701516.

take advantage of these features can be used to implement RPC protocols. The first, called a *virtual protocol*, postpones the binding of the RPC protocol to a low-level message delivery protocol. The second, called *layered protocols*, involves decomposing a monolithic RPC protocol into a collection of primitive building blocks and then composing these building block protocols in different ways. While most communication systems are layered—e.g., the 7-layer ISO model—this technique refers to further subdividing an individual layer. The paper motivates each technique with a concrete problem that arises in existing systems, describes how we applied the technique to the design of RPC protocols, and reports on experiments designed to evaluate the technique.

2 Uniform Protocol Interface

This section gives a brief overview of the *x*-kernel's uniform protocol interface; a more complete description is given in [6].

When a message arrives at a network device, a kernel process is dispatched to shepherd it upward through the kernel. Should the message eventually reach the user/kernel boundary, the shepherd process crosses the boundary and continues executing in the user's address space. When a user process generates a message, the process is temporarily given kernel privileges and allowed to shepherd the message downward through the kernel. Thus, when a message does not encounter contention for resources, it is possible to send or receive a message with no process switches. When resource contention does occur, the shepherd process blocks on a semaphore.

As a shepherd process flows through the kernel, it visits a sequence of *protocol* and *session* objects that encapsulate the various network protocols provided by the kernel. Loosely speaking, each protocol object corresponds to a conventional network protocol—e.g., IP, UDP, TCP—where the relationships between protocols are defined at the time a kernel is configured. A session object is an instance of a protocol object that contains a protocol interpreter and the data structures that represent the local state of some network connection. Figure 1(a) illustrates a suite of protocols that might be configured into a given instance of the *x*-kernel. Figure 1(b) gives a schematic overview of the *x*-kernel objects corresponding to the suite of protocols in (a); protocol objects are depicted as rectangles, the session objects associated with each protocol are depicted as circles, and a shepherd process follows a path through the protocol and session objects.

Protocol objects serve two major functions: they create session objects and they demultiplex messages received from the network to one of their session objects. A protocol object supports three operations for creating session objects:

```
session = open(participant_set)
open_enable(invoking_protocol, participant_set)
session = open_done(participant_set)
```

Intuitively, a high-level protocol invokes a low-level protocol's **open** operation to create a session. Each protocol object is given a capability at configuration time for the low-level protocols upon which it depends. In the case of **open_enable**, the high-level protocol passes a capability for itself to a low-level protocol. When a message arrives from the network, the latter protocol invokes the former protocol's **open_done** operation to complete the creation of the session. The **participant_set** argument to all three operations identifies the set of participants that are to communicate via the created session. Participants identify themselves and their peers with host addresses, port numbers, protocol numbers, and so on. By convention, the first element of that set identifies the local participant. In the case of **open** and **open_done**, all members of the participant set must be given. In contrast, not all the participants need be specified when **open_enable** is invoked, although an identifier for the local participant must be present. Thus, the first operation is used by a client process to actively create a session, while the second and third operations, taken together, are used by a server to passively create a session.

In addition to creating sessions, each protocol *switches* messages received from the network to one of its sessions with a

```
demux(message)
```

operation. **demux** takes a message as an argument, and either passes the message to one of its sessions, or creates a new session—using the **open_done** operation—and then passes the message to it. Each protocol object's **demux** operation makes the decision as to which session should receive the message based on the appropriate fields in the message's header.

A session is an instance of a protocol created at runtime as a result of an **open** or an **open_done** operation. Intuitively, a session corresponds to the end-point of a network connection; i.e., it interprets messages and maintains state information associated with a connection. For example, TCP session objects implement the sliding window algorithm and associated message buffers. Sessions support two primary operations:

```
push(message)
pop(message)
```

The first is invoked by a high-level session to pass a message down to some low-level session. The second is invoked by the **demux** operation of a protocol to pass a message up to one of its sessions. Intuitively, we think of the message as a stack, where the two operations push headers onto and pop headers off of the stack.

Finally, both protocol and session objects support a

```
control(opcode,buffer,length)
```

operation. This operation is used to read and set certain object-dependent parameters. For example, one might invoke a protocol object's **control** operation to learn the maximum transmission unit (MTU) supported by the protocol. As another example, one might invoke a session's **control** operation to learn a peer's address.

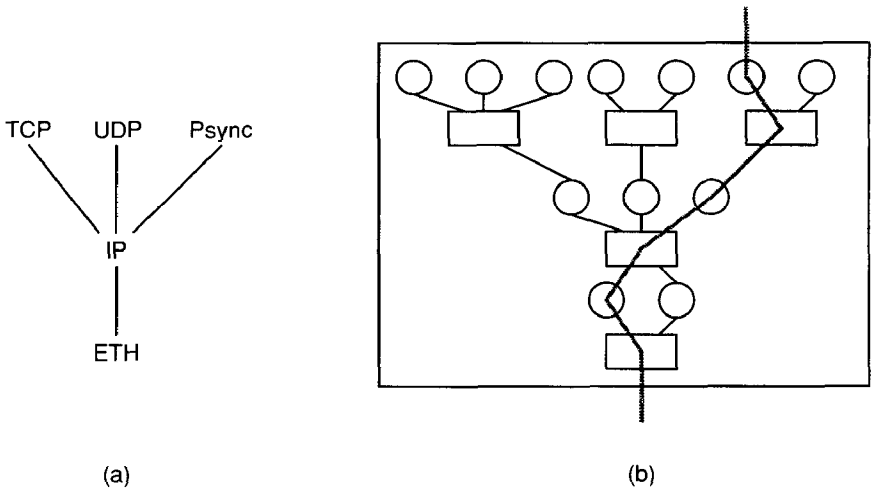

Figure 1: Example *x*-Kernel Configuration

3 Design Techniques

This section describes virtual protocols and layered protocols, and shows how they can be used to implement RPC protocols. To make the discussion more concrete, we apply the two techniques to a specific RPC protocol—Sprite RPC [20]. We chose to experiment with Sprite RPC for four reasons: (i) it supports *at most once* semantics; (ii) there are published performance results on Sun 3/75 workstations against which we can compare our results; (iii) the problem solved by virtual protocols is a real problem faced by the Sprite operating system; and (iv) it is easy to break Sprite RPC into multiple sub-protocols in a way that facilitates a fair evaluation of layered protocols.

3.1 Virtual Protocols

Consider the following problem. A remote procedure call mechanism that serves as the heart of a distributed operating system is implemented directly on top of an ethernet. While this design is efficient, it limits the size of the distributed system to a single ethernet. To accommodate a larger system, one must insert an internet protocol (e.g., IP) between the RPC protocol and the ethernet, but this has the undesirable consequence of adding a fixed overhead to every RPC, even in the common case where the client and server are on the same ethernet. To quantify the impact of inserting IP between the RPC protocol and the ethernet, consider the Sprite network operating system. The latency of the *x*-kernel implementation of Sprite RPC directly on the ethernet is 1.73 msec and the round trip cost of IP in the *x*-kernel is .37 msec. Thus, inserting IP between Sprite RPC and the ethernet automatically implies a 21% performance penalty. Note that this problem is not specific to Sprite; it will become increasingly common as more local area network applications and systems are extended into the Internet.

An alternative is to implement the RPC protocol on top of a *virtual protocol* rather than either IP or the ethernet. A virtual protocol is a header-less protocol that accepts messages from one or more high-level protocols and dynamically multiplexes them onto a collection of low-level protocols that provide approximately the same semantics. Virtual protocols differ from other multiplexing protocols like IP in that they do not add any functionality, and as a consequence, they do not attach a header to messages.

For example, VIP (Virtual IP) is a virtual protocol that provides the same semantics as IP—i.e., unreliable delivery of messages to a set of hosts identified with IP addresses—but it multiplexes messages to IP and the ethernet. Figure 2 gives an example protocol configuration, where ETH is able to deliver 1500-byte packets to hosts on the same ethernet and IP is able to deliver 64k-byte packets to any host in the Internet. Notice that IP is in turn implemented on top of ETH. Also note that multiple high-level protocols can use the virtual protocol. That is, not only does the virtual protocol make it easy to dynamically *insert* IP under RPC, it also makes it possible to dynamically *delete* IP from below protocols like Psync and UDP that normally use IP.

The *x*-kernel makes it easy to implement VIP because it delays the binding between protocols until run time and it provides a uniform interface to all protocols. The implementation of VIP involves work in the **open** and **push** operations. Consider each operation in turn.

When a high-level protocol invokes VIP's **open** operation, it identifies itself with an 8-bit IP protocol number and its peer with a 32-bit IP host address. VIP's **open** creates a new session and in turn opens either an IP or an ETH ses-

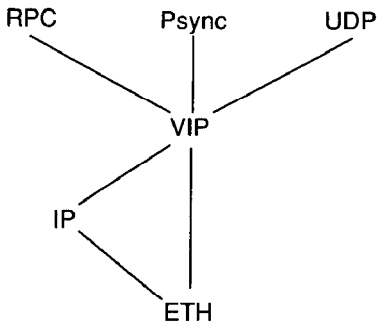

Figure 2: Example Protocol Suite

sion, or both, based on the destination address and the size of messages to be sent by the invoking protocol. VIP asks the invoking protocol about the size of messages it expects the underlying protocol to support using a **control** operation. For example, Sprite RPC reports that it never sends a message greater than 1500-bytes (it has its own fragmentation mechanism for handling larger messages), while UDP sends arbitrarily large messages (i.e., it depends on IP to fragment large messages). We assume all protocols also send messages smaller than their advertized maximum size. VIP next decides if the destination host is reachable via the ethernet by trying to resolve the IP address using ARP. If ARP can resolve the address, then the destination host must be on the local ethernet; otherwise, the destination is not on the local network. If the destination is on the local network and the high-level protocol reports a maximum message size smaller than or equal to the ethernet's MTU, then VIP invokes ETH's **open** to establish an ethernet session. If the destination host is not on the local network then VIP invokes IP's **open** to create an IP session. Finally, if the destination host is local but the maximum high-level message size is greater than the ethernet's MTU, VIP opens both an ETH session and an IP session.

Notice that this approach assumes that all hosts on the local ethernet also run VIP. A more general solution would be to maintain a table of hosts on the local network that support VIP. This table could be dynamically maintained by running a broadcast-based protocol that advertizes the protocols that a given host supports; this approach is currently used in 4.3BSD Unix to determine if trailers may be used. Also note that when VIP needs to open an ethernet session, it must map IP protocol numbers and host addresses into ethernet types and host addresses. This is easy in the case of the IP host address: it is done using ARP. In the case of the IP protocol field, the mapping is possible because IP supports only 256 high-level protocols (i.e., its protocol field is 8-bits long), while the ethernet supports 65,536 high-level protocols (i.e., its type field is 16-bits long). VIP maps IP protocol numbers onto an unused range of 256 ethernet types.

When the high-level protocol has a message to send, it applies the **push** operation to the message and the session returned by VIP's **open** operation. VIP's **push** operation, in turn, inspects the length of the message. If the length is less than the ethernet's MTU and an ethernet session had been created at open time, VIP pushes the message through the ethernet session; otherwise it pushes the message through the IP session. The *x*-kernel provides an inexpensive operation for determining the length of a given message. The important point is that once a VIP session has been opened, the only overhead it adds to message delivery is the cost of the single test in VIP **push**.

3.2 Layered Protocols

Consider the following problem. Many protocols need to be able to send large messages (i.e., bigger than one ethernet message) between a pair of hosts. For example, Sprite RPC accommodates input arguments and return values of up to 16k, Sun RPC [18] accommodates arguments and return values of up to 8k, and Psync accommodates messages of up to 16k. The most common approach to this problem is to separately embed a bulk transfer function in each protocol that needs it. While the different protocols may be able to borrow the same strategy for implementing this functionality, they are seldom able to actually *reuse* code.

An alternative approach is to start with a carefully tuned algorithm for transferring large messages, such as the one found in an existing RPC-like protocol [4], and package it as an independent protocol. Such a protocol could then be composed with other protocols to form a fully functional transport service. The principle reason for supporting this methodology is that is makes it easier to design and implement new protocols: existing protocol pieces can be reused and each individual piece is easier to implement, debug, and optimize. While this *parts programming* technique has proven useful in other settings—for example, composing Unix software tools and inserting filters in System V Unix character streams—it has not been exploited in the implementation of network protocols. Instead, network protocols are typically implemented as large, monolithic programs.

Note that carving a separate bulk transfer protocol out of an RPC protocol is significantly different than using RPC as a whole to transfer large messages. This is because RPC encapsulates too much functionality, and it particular, it provides functionality that is not compatible with all the high-level protocols that might want to take advantage of its bulk transfer capability. For example, while Psync could use a protocol that sends large messages, it does not want *at most once* RPC semantics. From the software tool perspective, existing RPC protocols encapsulate several distinguishable functions.

While many large protocols are *described* in terms of functional layers, such descriptions only serve as a method for organizing the presentation; they seldom translate into self contained software modules. Moreover, even if one implements a large protocol in separate modules according to such a layered description, these modules cannot stand alone as independent protocols. One reason is that the header fields used by each layer are *scattered* throughout the protocol's header. Another reason is that in order for a

module to be an independent protocol that can be used by multiple high-level protocols, it must have its own protocol number (type) field.

To better understand the implications of composing protocol layers, we partitioned Sprite RPC into three independent protocols, including a bulk transfer protocol that can be reused by Psync. That is, each protocol layer was designed with an eye towards being composed with other protocols in addition to being composed with each other. It is important to understand that the monolithic version of Sprite RPC and the layered version of Sprite RPC are not the same protocol in the sense that one could exchange messages with the other. They are in effect two different protocols that provide the same level of service.

The Sprite RPC protocol conveys request and reply messages between client and server processes. The protocol uses an implicit acknowledgement technique in which the receipt of a reply message by a client process acknowledges the receipt of the corresponding request message it sent to the server, and the receipt of a request message by a server process acknowledges the receipt of the previous reply message it sent to the client [1]. If no messages are lost, servers are prompt, and requests are frequent, then no extra acknowledgement messages are needed. Otherwise, timeouts trigger retransmissions which sometime elicit explicit acknowledgements. Requests or replies which are larger than the maximum packet size of the underlying network are fragmented, but the fragments are treated as parts of a single RPC; e.g., a reply implicitly acknowledges receipt of all the fragments of the request message.

We decomposed Sprite RPC into three independent layers:

- SELECT: The selection layer maps Sprite commands (procedure ids) onto procedure addresses (server processes).
- CHANNEL: The channel layer pairs request messages with reply messages while preserving *at most once* semantics.
- FRAGMENT: The fragmentation layer provides unreliable (delivery not guaranteed), but persistent (recovers from dropped fragments) transmission of large messages.

We now consider the three layers from the bottom-up.

First, FRAGMENT supports unreliable delivery of large messages between a pair of hosts. It is unreliable in the sense that messages may be delivered out of order, duplicate copies of the same high-level message may be delivered, and a given message may not be delivered at all. Specifically, each message pushed through FRAGMENT is assigned a unique sequence number. The message is then fragmented and transmitted, with a copy of the fragments saved in the local state. The protocol is persistent in that if a receiving host detects that it is missing one or more fragments, it sends a request for the missing fragments to the sending host.

FRAGMENT differs from how fragmentation is done in Sprite RPC because the receiving host never sends a positive acknowledgement (implicit or explicit) when all the message fragments have been successfully received. Instead, the sending host associates a timer with each message it sends and discards the message when the timer expires.[1] Note that it is possible that a higher-level protocol that depends on FRAGMENT expects an acknowledgement or reply message, in which case the higher-level protocol must maintain its own timer and potentially resend a message through FRAGMENT. If this happens, FRAGMENT treats the second incarnation of the message as an independent message; i.e., it is assigned a new FRAGMENT-level sequence number.

Second, CHANNEL supports request/reply transactions with *at most once* semantics. Each channel is opened as a separate *x*-kernel session and the algorithm supported by each session is the same as in Sprite: A high-level protocol pushes a message into the session (channel) and a reply message is returned. The only difficulty in implementing CHANNEL as a separate protocol is to tune its timeout mechanism to take into account that FRAGMENT exists as a separate protocol; i.e., FRAGMENT and CHANNEL each support a timeout mechanism. Specifically, CHANNEL's timer is a step function: for single fragment messages CHANNEL's timeout is small, while for multi-fragment messages CHANNEL must wait long enough to be sure that the fragmentation layer is not in the middle of transmitting the message.

Finally, SELECT maps an RPC address (procedure id) onto a procedure. The important aspect of the selection level is that it implements the caching required for good RPC performance. Since there are a fixed and predefined number of channels in Sprite, the SELECT layer simply choses one of the existing channels when an RPC is invoked; it blocks if there are none available. Since there can only be a fixed number of CHANNEL sessions, SELECT is optimized to have a fixed number of SELECT sessions. Notice that the reason for separating SELECT into a separate protocol, rather than embedding it in CHANNEL, is that we want to be able to support multiple schemes for addressing procedures. For example, we have built an alternative selection layer that does forwarding. As another example, it is trivial to build a reliable datagram protocol on top of CHANNEL.

Note that in partitioning Sprite RPC into multiple layers, we not only kept the monolithic version and the layered version semantically equivalent, we also tried to keep them as syntactically equivalent as possible. For example, FRAGMENT uses the same collection of header fields as does the fragmentation portion of Sprite RPC, CHANNEL uses a subset of the header fields that Sprite RPC uses, and so on. In other words, the union of the FRAGMENT, CHANNEL, and SELECT headers is nearly identical to the Sprite RPC header. The only significant difference is that the layered version duplicates certain fields; e.g., both FRAGMENT and CHANNEL have their own sequence number field. Also, because both FRAGMENT and CHANNEL are meant

[1] The *x*-kernel allows multiple protocol layers to maintain references to pieces of the same message. Thus, for one protocol to discard its handle on the message does not mean that the actual message is deleted.

to be used by multiple high-level protocols, their headers each include a *protocol number* field. The header for each protocol is given in the appendix.

4 Experiments

This section reports on a set of experiments designed to evaluate virtual protocols and layered protocols. The experiments involve measuring various protocol configurations for latency and throughput. The latency tests measure the round trip delay for invoking a null procedure with null request and reply messages. The throughput tests measure the round trip delay for invoking a null procedure with a series of large request messages (ranging in size from 1k-bytes to 16k-bytes) and a null reply message. The large messages are fragmented into 1500-byte packets. All the experiments are kernel-to-kernel; i.e., messages were not passed between user and kernel address spaces.

The experiments were conducted on a pair of Sun 3/75s connected by an isolated 10Mbps ethernet. The *x*-kernel and all the protocols it supports were compiled using the standard SunOS Release 4.0 C compiler. The numbers presented in this section were derived through two levels of aggregation. Each experiment involved measuring the total elapsed time needed to execute the test configuration 10,000 times, and dividing this time by 10,000. A given experiment was then repeated several times, with the average of those runs reported. Although we do not report the standard deviation of the various experiments, they were observed to be small.

Because the experiments involve comparing various combinations of protocols, we carefully identify each protocol and configuration as follows:

- N_RPC: Native implementation of Sprite RPC in the Sprite kernel [10].
- M_RPC: Monolithic version of Sprite RPC implemented in the *x*-kernel.
- L_RPC: Layered version of Sprite RPC implemented in the *x*-kernel; consists of the composition of SELECT-CHANNEL-FRAGMENT.

Also, we clearly denote the underlying delivery protocol; e.g., M_RPC-IP, M_RPC-VIP.

4.1 Virtual Protocols

To evaluate the impact of VIP on RPC performance, we tested the latency and throughput of M_RPC on top of three different message delivery protocols: ETH, IP, and VIP. Table I summarizes the results, where the *Throughput* column corresponds to the throughput measured using 16k-byte messages and the *Incremental Cost* column reports the incremental cost for each additional 1k-bytes sent in the throughput tests. Note that the throughput reported for the native implementation of Sprite RPC is an approximation.

First, observe that the *x*-kernel implementation of Sprite RPC is faster than the native implementation in the Sprite kernel: 1.73 msec versus 2.6 msec latency.[2] Also, the *x*-kernel implementation results in a better throughput rate: 860k-bytes per second versus just over 700k-bytes per second. Note that we have not performed any fine-grained optimization of M_RPC; i.e., we have not made optimizations that exploit knowledge of the compiler or the hardware. The only optimizations we have made are at the *protocol level*, for example, making sure each protocol touches the header as little as possible.

That Sprite RPC runs faster in the *x*-kernel than it does in the Sprite kernel is consistent with our experience implementing other protocols in the *x*-kernel: the structure imposed on protocols by the *x*-kernel's underlying architecture leads to efficient implementations. On the other hand, there are too many uncontrolled factors—e.g., kernel coding techniques, process switch times, and so on—to put too much weight in the *x*-kernel versus native Sprite kernel comparison. The only reason we compare the two implementations is to support the claim that the *x*-kernel implementation of Sprite RPC is reasonable; i.e., it is not so inefficient as to make the layered version look artificially fast. All the interesting performance comparisons made in this section are between various versions of Sprite RPC implemented in the *x*-kernel.

Second, VIP imposes a negligible overhead in the local case. That is, M_RPC-VIP has a round trip time of 1.79 msec, implying VIP imposes a 0.06 msec overhead on the underlying protocols. Moreover, VIP offers a modest advantage over using IP: the M_RPC-IP round trip delay is 2.10 msec. Thus, as predicted in Section 3, IP imposes a 21% latency penalty on RPC. Using VIP instead of IP eliminates most of this penalty. Using VIP instead of IP has a negligible effect on RPC throughput. This is because both protocol stacks drive the ethernet controller at its maximum rate. The M_RPC-VIP stack does use less CPU time, however.

4.2 Layered Protocols

We now evaluate the cost of decomposing RPC into multiple layers by comparing L_RPC with M_RPC. Table II summaries the results. For this experiment, both the monolithic and layered versions of RPC were composed with VIP.

The experiment shows that layering imposes a 0.14 msec penalty on Sprite RPC: layered RPC has a 1.93 msec latency and monolithic RPC has a 1.79 msec latency. Layered and monolithic RPC support approximately the same throughput, although the layered version uses slightly less CPU time. This is because only FRAGMENT—the bottom-most layer—handles the individual packets that make up each large message. That is, for each 16k-byte message, FRAGMENT handles 16 messages, but CHANNEL and SELECT

[2] The actual latency reported by Sprite is 2.8 msec, but this includes a 0.2 msec penalty for a crash/reboot detection mechanism not included in the *x*-kernel implementation. Also, Sprite recently reports a round trip time of 2.4 msec (2.2 msec without the crash protocol) when **gcc** is used. This is consistent with our experience: we have observed 10% to 20% speedups of the *x*-kernel when using **gcc**.

Configuration	Latency (msec)	Throughput (kbytes/sec)	Incremental Cost (msec/1k-bytes)
N_RPC	2.6	700+	1.2
M_RPC-ETH	1.73	863	1.04
M_RPC-IP	2.10	836	1.05
M_RPC-VIP	1.79	860	1.04

Table I: Evaluating VIP

Configuration	Latency (msec)	Throughput (kbytes/sec)	Incremental Cost (msec/1k-bytes)
M_RPC-VIP	1.79	860	1.04
L_RPC-VIP	1.93	839	1.03

Table II: Monolithic RPC versus Layered RPC

Configuration	Latency (msec)	Incremental Cost (msec/layer)
VIP	1.12	NA
FRAGMENT-VIP	1.33	0.21
CHANNEL-FRAGMENT-VIP	1.82	0.49
SELECT-CHANNEL-FRAGMENT-VIP	1.93	0.11

Table III: Cost of Individual RPC Layers

handle only one message. FRAGMENT by itself (i.e., without CHANNEL and SELECT) achieves a throughput rate of 865k-bytes/second.

In the case of latency, the overhead for layering imposed by the *x*-kernel itself is negligible: It costs only a single procedure call to go from one layer to another. Thus, the cost of layering is something inherent in the protocol, not the operating system. To better isolate the layering overhead, we measured the latency for the individual protocols that make up L_RPC. The results are reported in Table III.

The key observation is that SELECT—the least expensive layer—costs 0.11 msec. In fact, this 0.11 msec cost is consistent with our experience with other trivial protocols such as UDP. Thus, in a situation where a monolithic protocol is broken into arbitrarily many layers, we can expect each layer to cost at least 0.11 msec on a Sun 3/75, making it reasonable to think about protocol stacks with on the order of ten layers. The extra time spent in CHANNEL—the most expensive protocol—can be attributed to the cost of synchronization and process switching that is intrinsic to the request/reply paradigm.

4.3 Dynamically Removing Layers

Because we earlier claimed that the 0.37 msec penalty for using IP instead of ETH was significant, one could also conclude that the penalty for using layered RPC instead of monolithic RPC is too high. However, it is important to keep in mind the following two points. First, the cost of using IP instead of directly using ETH buys you nothing in the common case of two hosts communicating over the same ethernet. In contrast, the cost of layering buys you something significant. For example, in the case of FRAGMENT, you get a bulk transfer protocol that can be reused by other protocols. Second, even though layered RPC has higher latency, it achieves nearly the same throughput as monolithic RPC because both versions saturate the ethernet controller.

In addition, after having divided the RPC protocol into three layers, we are able to improve performance in a way not possible in monolithic RPC. Specifically, because FRAGMENT is a self-contained protocol, it can be factored out of the protocol stack. That is, we can use virtual protocols to dynamically determine whether or not to include it for each message sent by CHANNEL. Figure 4 (a) gives the original configuration of protocols that we evaluated in Section 4.2. Figure 4 (b) shows a new configuration that places FRAGMENT beneath VIP instead of above it. Specifically, the new configuration uses two different virtual protocols: VIP_{size} and VIP_{addr}. VIP_{size} selects between FRAGMENT and VIP_{addr} based on message size. Like VIP, VIP_{size} touches every message sent through the protocol stack. VIP_{addr} selects between ETH and IP. Unlike VIP, VIP_{addr} is only involved at open time; it opens a lower-level IP or ETH session and returns it rather than returning a session of its own. Thus, the overhead involved in the new configuration is the same as in the old configuration because only one virtual protocol—VIP_{size}—handles every message that is sent and received.

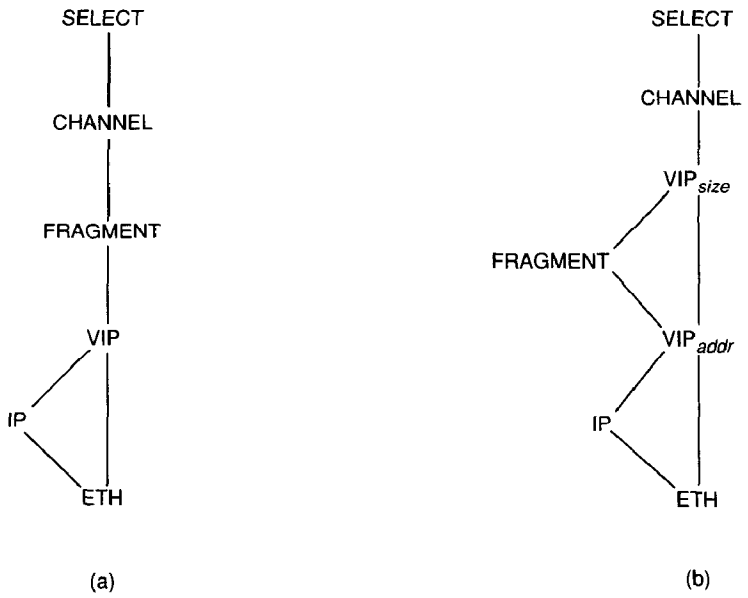

Figure 3: Alternative Configurations Using RPC Layers

Because VIP_{size} bypasses FRAGMENT when sending a small message, one would expect to save 0.15 msec in the round trip delay: subtracting 0.21 msec for bypassing FRAGMENT and adding 0.06 msec for the overhead of VIP_{size}. In fact, an experiment verifies this number: SELECT-CHANNEL-VIP_{size} has a 1.78 msec latency. In other words, we were able to eliminate the unnecessary overhead of FRAGMENT only because we were able to isolate FRAGMENT as a separate protocol, and in doing so, we are able to achieve the same latency as the monolithic version.

5 Discussion

This section discusses our experiences using virtual protocols and layered protocols to implement RPC. It also makes several comments about protocol design and implementation in general.

Generality of Virtual Protocols: While VIP was easy to implement—it consists of a little over 300 lines of C code and took a couple of hours to write and debug—virtual protocols are in general hard to design. For example, adding another internet protocol (e.g., PUP [3]) would force us to invent a new virtual address space; we could no longer use IP addresses as virtual addresses. As another example, moving a simple protocol like UDP under VIP proved to be very difficult, even though UDP supports the same unreliable semantics as ETH or IP. This is because UDP addresses include two 16-bit port numbers which cannot be completely mapped onto a single 8-bit IP protocol number. There seems to be little hope of avoiding such difficulties without standardizing the size and format of protocol type fields. As a final example, one would expect TCP to be able to use VIP since VIP provides the same semantics as IP. This doesn't work in practice, however, because TCP depends on the length field in the IP header (the TCP header does not have a length field of its own) and TCP computes a checksum that covers the IP header. The conclusion we draw from our experience trying to implement TCP on VIP is that when designing protocols, one should eliminate unnecessary dependencies on other protocols; i.e., protocols should be designed so they can be composed with any protocol that offers the same level of service. In this particular case, having TCP depend on information in the IP header contributes little to the efficiency of TCP, it is mostly an artifact of TCP and IP being designed in conjunction with each other. Despite the difficulties in implementing virtual protocols for conventional protocol stacks, we believe virtual protocols will play an important role in network environments dominated by layered protocols. This is because the taller protocol stacks resulting from subdividing protocols will provide more opportunity for dynamically bypassing individual layers.

Protocol Decomposition: Our experience strongly suggests that decomposing protocols into their primitive functional units is a good thing, independent of the exact performance costs. As one would expect, such a methodology allows one to concentrate on implementing a single function at a time. More importantly, however, dividing large protocols into smaller protocols means that (i) it is easier to test and debug each piece, and (ii) one can do a better job of optimizing each piece. This second point is critical: layering helps you isolate and correct poor performance. The central difficulty in decomposing protocols is understanding the right way to break a large protocol into independent pieces. When designing the FRAGMENT protocol, for example, we had to choose between reliable and unreliable delivery seman-

tics; RPC could use it either way. We chose to make it unreliable—i.e., not send positive acknowledgements—so that it could also be used by Psync. Identifying protocol pieces that are of general use is a topic of current research.

Potential Pitfalls of Layering: Layering provides a great opportunity for having disastrous performance. A single mistake repeated at multiple layers can quickly lead to unacceptable performance. Although we are confident that the results presented in Section 4 accurately reflect the intrinsic costs of layering rather any replicated errors, we did discover two problems that can significantly influenced the performance of layered protocols. First, unnecessarily establishing and freeing state information at each level degrades performance. The implementation of layered RPC avoids this problem by caching open sessions at all three levels. Second, the cost of manipulating message headers can be significant. In an earlier version of the *x*-kernel, we used a buffer management scheme that allocated a buffer for each new header added to a message. In contrast, the current version pre-allocates a single buffer that is large enough to hold all the headers and simply adjusts a pointer for each new header. The original approach resulted in a 0.50 msec minimum cost for each layer, whereas the current approach has a minimum cost of 0.11 msec per layer.

Information Loss: Layering, in general, has the potential to hide information that is available in a monolithic piece of code. Such information is commonly used to make decisions that affect protocol performance. Whereas a monolithic protocol learns information by looking in global data structures, a layered protocol is able to learn the same information by invoking **control** operations. While one might guess that an unwieldy number of different **control** operations would be necessary to access all the information protocols need, our experience is that a relatively small number of control operations is sufficient; i.e., on the order of two dozen. By using these control operations, layered protocols are able to gain the same advantage as monolithic protocols at the negligible cost of calling a procedure rather than reading a shared variable.

Mix and Match RPCs: In addition to experimenting with Sprite RPC, we also experimented with decomposing Sun RPC [18]. Specifically, we have divided Sun RPC into SUN_SELECT and REQUEST_REPLY layers, and we treat the various authentication mechanisms as a library of optional protocol layers. We have two main observations regarding this exercise. The first is that layering provides a natural methodology for inserting or removing optional sub-pieces such as authentication. Much of the complexity in the Sun RPC code concerns the optional authentication component. The second is that layering makes it possible to define new RPC protocols from existing pieces. For example, one can compose SUN_SELECT and REQUEST_REPLY with FRAGMENT rather than having to depend on IP to fragment large messages. FRAGMENT is superior to IP as a bulk transfer protocol because it is persistent. As another example, one can replace the REQUEST_REPLY protocol (which has *zero or more* semantics) with the CHANNEL protocol (which has *at most once* semantics). The downside of this flexibility is that applications must agree to use a particular protocol stack. We believe it is reasonable to customize a protocol stack on an application by application basis, but probably not on a per-connection basis.

Optimizing Protocol Performance: As pointed out in Section 4.1, the performance numbers for the *x*-kernel correspond to protocol implementations that have not been heavily optimized. The important point is that these protocols need not be heavily optimized in order to be efficient. This is because the *x*-kernel is highly tuned for implementing protocols. As a consequence, in order to implement a given protocol efficiently, one has only to apply a small collection of high-level "efficiency rules", such as always to cache open sessions, not touch the header any more than necessary, and so on. Our experience is that these rules apply uniformly across all protocols.

Performance Predictability: The predictability of protocol performance in the *x*-kernel is comforting, and in fact, it is necessary when one is designing new protocols. For example, by knowing the cost of individual protocol layers, one is able to predict the cost of composing those protocols. While it sounds reasonable that all operating systems have such a "predictability" characteristic, our experience is that this property is not universal. For example, to send a message using the protocol stack UDP-IP-ETH in SunOS Release 4.0 costs less than sending it using the IP-ETH protocol stack. The predictability of the *x*-kernel is a direct consequence of its uniform protocol interface, and conversely, the unpredictability in Unix is due to the lack of uniformity in the socket interface.

6 Conclusions

This paper describes how to take advantage of virtual protocols and layered protocols to implement RPC in the *x*-kernel. A set of experiments designed to evaluate the use of virtual protocols and layered protocols show that both are cost effective techniques for implementing RPC.

First, we show that VIP, a specific example of a virtual protocol, offers modest latency and throughput improvement over using IP to exchange messages over the same local area network. Moreover, VIP imposes very little overhead on end-to-end protocols that directly use the ethernet. Being able to dynamically remove IP from the protocol stack will be increasingly important as localized distributed systems migrate onto the Internet. In general, we believe virtual protocols will play a significant role in a network environment dominated by layered protocols. In such an environment, being able to dynamically insert and delete individual protocols into and out of the protocol stack will make it possible to improve performance.

Second, we demonstrate that a layered version of RPC can achieve latency and throughput performance that is comparable to a monolithic version of RPC. Because layering facilitates reuse, makes it possible to dynamically remove unnecessary layers, and makes protocols easier to debug and optimize, we conclude that decomposing monolithic

protocols into their primitive functional units is both desirable and economical. The remaining challenge is to identify and isolate important functions that can be used to build more complex composed protocols. As one example of our effort in this area, we are experimenting with using Psync as a building block protocol for implementing various protocol stacks for fault-tolerant distributed programs [7]. As another example, we have applied the technique to stream-oriented protocols with modest success. Finally, we are experimenting with a *meta-protocol* that establishes a set of "rules" for protocol design. For example, the meta-protocol defines a standard protocol type field. The idea is that protocols that adhere to the meta-protocol will be more easily composed.

References

[1] A. Birrell and B. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, Feb. 1984.

[2] A. Black, N. C. Hutchinson, E. Jul, H. M. Levy, and L. Carter. Distribution and abstract types in Emerald. *IEEE Transactions on Software Engineering*, SE-13(1):65–76, Jan. 1987.

[3] D. P. Boggs, J. F. Shoch, E. A. Taft, and R. M. Metcalfe. Pup: an internetwork architecture. *IEEE Transactions on Communications*, COM-28(4):612–623, Apr. 1989.

[4] D. R. Cheriton. VMTP: a transport protocol for the next generation of communications systems. In *Proceedings of the SIGCOMM '86 Symposium*, pages 406–415, Aug. 1987.

[5] N. C. Hutchinson, S. Mishra, L. L. Peterson, and V. T. Thomas. Tools for implementing network protocols. *Software—Practice and Experience*, 1989. To appear.

[6] N. C. Hutchinson and L. L. Peterson. Design of the *x*-Kernel. In *Proceedings of the SIGCOMM '88 Symposium*, pages 65–75, Stanford, Calif., Aug. 1988.

[7] S. Mishra, L. L. Peterson, and R. D. Schlichting. Implementing fault-tolerant replicated objects using Psync. In *Eighth Symposium on Reliable Distributed Systems*, Oct. 1989. To appear.

[8] P. Mockapetris. *Domain Names—Implementation and Specification*. Request For Comments 1035, USC Information Sciences Institute, Marina del Ray, Calif., Nov. 1987.

[9] J. C. Mogul, R. F. Rashid, and M. J. Accetta. The packet filter: an efficient mechanism for user-level network code. In *Proceedings of the Eleventh ACM Symposium on Operating System Principles*, pages 39–51, Nov. 1987.

[10] J. K. Ousterhout, A. R. Cherenson, F. Douglis, M. N. Nelson, and B. B. Welch. The sprite network operating system. *IEEE Computer*, 23–36, Feb. 1988.

[11] L. L. Peterson, N. Buchholz, and R. D. Schlichting. Preserving and using context information in interprocess communication. *ACM Transactions on Computer Systems*, 7(3):217–246, Aug. 1989.

[12] D. Plummer. *An Ethernet Address Resolution Protocol*. Request For Comments 826, USC Information Sciences Institute, Marina del Ray, Calif., Nov. 1982.

[13] J. Postel. *Internet Message Control Protocol*. Request For Comments 792, USC Information Sciences Institute, Marina del Ray, Calif., Sep. 1981.

[14] J. Postel. *Internet Protocol*. Request For Comments 791, USC Information Sciences Institute, Marina del Ray, Calif., Sep. 1981.

[15] J. Postel. *User Datagram Protocol*. Request For Comments 768, USC Information Sciences Institute, Marina del Ray, Calif., Aug. 1980.

[16] D. M. Ritchie. A stream input-output system. *AT&T Bell Laboratories Technical Journal*, 63(8):311–324, Oct. 1984.

[17] *Network File System*. Sun Microsystems, Inc., Mountain view, Calif., Feb. 1986.

[18] *Remote Procedure Call Programming Guide*. Sun Microsystems, Inc., Mountain view, Calif., Feb. 1986.

[19] USC. *Transmission Control Protocol*. Request For Comments 793, USC Information Sciences Institute, Marina del Ray, Calif., Sep. 1981.

[20] B. B. Welch. *The Sprite Remote Procedure Call System*. Technical Report UCB/CSD 86/302, University of California Berkeley, Berkeley, Calif., June 1988.

Appendix

The C structures that define the headers for monolithic RPC, SELECT, CHANNEL, and FRAGMENT, respectively. Note that while Sprite provides for 32-bit host numbers, our implementation uses IP addresses (also 32-bits) to identify hosts.

```
typedef struct sprite_hdr {
    unsigned short flags;
    IPaddr clnt_host;
    IPaddr srvr_host;
    unsigned short channel;
    unsigned short srvr_process;
    unsigned sequence_num;
    unsigned short num_frags;
    unsigned short frag_mask;
    unsigned short command;
    unsigned boot_id;
    unsigned short data1_sz;
    unsigned short data2_sz;
    unsigned short data1_offset;
    unsigned short data2_offset;
} SPRITE_HDR;

typedef struct select_hdr {
    unsigned char type;
```

```
        unsigned short command;
        unsigned char status;
} SELECT_HDR;

typedef struct channel_hdr {
        unsigned short flags;
        unsigned short channel;
        unsigned int protocol_num;
        unsigned int sequence_num;
        unsigned short error;
        unsigned boot_id;
} CHANNEL_HDR;

typedef struct fragment_hdr {
        unsigned char type;
        IPaddr clnt_host;
        IPaddr srvr_host;
        unsigned int protocol_num;
        unsigned int sequence_num;
        unsigned short num_frags;
        unsigned short frag_mask;
        unsigned short len;
} FRAGMENT_HDR;
```

Note that layered RPC does not make use of the dual data size and offset fields. Having two separate fields serves no purpose in the *x*-kernel because of the support it provides for building and tearing apart messages.