Client-Server Computing on the SHRIMP Multicomputer

Stefanos N. Damianakis, Angelos Bilas, Cezary Dubnicki, and Edward W. Felten
Department of Computer Science, Princeton University, Princeton NJ 08544 USA
{snd,bilas,dubnicki,felten}@cs.princeton.edu

Abstract

The client-server is the dominant programming model in distributed computing and has been used extensively as a structuring method for large software systems. Remote procedure call (RPC) and stream sockets form the basis of the communication system for a wide variety of distributed applications. Unfortunately advances in processor and network technology do not translate directly in performance improvements in the client-server model, mainly because of the tremendous software overhead.

However, the emergence of new network interface technology is enabling new approaches to the development of communications software.

This paper presents two implementations of RPC and one of stream sockets for the SHRIMP multicomputer. SHRIMP supports protected, user-level data transfer, allows user-level code to perform its own buffer management, and separates data transfers from control transfers so that data transfers can be done without interrupting the receiving node’s CPU.

1 Introduction

The client-server paradigm has been very useful for large classes of applications. Most client-server applications rely on RPC or the Berkeley stream sockets model for interprocess communication. RPC provides a connection-oriented bidirectional abstraction, with well-defined semantics that resemble the local procedure call semantics. Transfering data as well as control is done with a single procedure call that gets executed on a (potentially) remote system. The stream sockets interface provides a connection-oriented, bidirectional byte-stream abstraction, with well-defined mechanisms for creating and destroying connections and for detecting errors.

Technological advances in network and processor speeds do not seem to lead to equally large improvements in the performance of client-server systems. The main reason for this is that software overhead dominates communication. Thus, improvements in hardware performance do not project to user applications.

The SHRIMP project [?, ?, ?] at Princeton University supports user level communication between processes by mapping memory pages between virtual address spaces. This virtual memory-mapped network interface seems to have many advantages including flexible user-level communication, and very low overhead to initiate data transfers.

In this work we examine two implementations of RPC and sockets for the SHRIMP multicomputer that deliver to user applications almost raw hardware performance.

The first RPC library (vRPC) meets the Sun-RPC interface and achieves a round trip latency of 33 microseconds for a null call with no arguments, without compromising compatibility with the SunRPC standard.

Our experiments with vRPC show that even without changing the stub generator or the kernel, RPC can be made several times faster on the new network interface than on conventional networks. vRPC outperforms the best reported implementation (to our knowledge) for fast networks [?] by more than a factor of four. A null RPC round trip takes about 33 microseconds. Even greater gains could be achieved by applying well-known techniques that rely on changes to the stub generator.

The second RPC library, ShrimpRPC, is a full-functionality RPC system but is not compatible with any standard. ShrimpRPC achieves a round trip latency of 9.5 microseconds, or about one microsecond above the hardware minimum for round-trip communication.

The stream sockets library is compatible with the stream sockets interface; it properly detects broken connections and correctly implements the select call. The sockets library performs much better than all previous sockets implementations for small transfers, with an end-to-end latency of 11 microseconds for an 8-byte transfer. Small transfers are important because they are very common in many applications [?]. For large transfers, we obtained a bandwidth of 13.5 MBytes/sec, which is close to the hardware limit when the receiver must perform a copy.

2 SHRIMP Hardware

The SHRIMP multicomputer is a network of commodity systems. Each node is a Pentium PC running the Linux operating system. The network is a multicomputer routing network [?] connected to the PC nodes via custom-designed network interfaces. The
SHRIMP network interface closely cooperates with a thin layer of software to form a communication mechanism called virtual memory-mapped communication (VMMC) [2]. This mechanism supports various message passing packages, and applications effectively, and delivers excellent performance [2].

The network connecting the nodes is an Intel routing backplane consisting of a two-dimensional mesh of Intel Mesh Routing Chips (IMRCs) [2], and is the same network used in the Paragon multiprocessor [2].

The custom network interface [2, 7] is the system’s key component. It connects each PCI node to the routing backplane and implements the hardware support for VMMC.

3 Virtual Memory-Mapped Communication

Virtual memory-mapped communication (VMMC) [2] was developed in response to the need for a basic multiprocessor communication mechanism with extremely low latency and high bandwidth. These performance goals are achieved by allowing applications to transfer data directly between two virtual memory address spaces over the network. The basic mechanism is designed to efficiently support applications and common communication models such as message passing, shared memory, and client-server.

The VMMC mechanism consists of several calls to support user-level buffer management, various data transfer strategies, and transfer of control.

3.1 Import-Export Mappings

In the VMMC model, an import-export mapping must be established before communication begins. A receiving process can export a region of its address space as a receive buffer to either with a set of permissions to define access rights for the buffer. In order to send data to the exported receive buffer, a user process must import the buffer with the right permissions.

After successful imports, a sender can transfer data from its virtual memory into imported receive buffers at user-level without further protection checking or protection domain crossing. Communication under this import-export mapping mechanism is protected in two ways. First, a trusted daemon process implements import and export operations. Second, the hardware virtual memory management unit (MMU) on an importing node makes sure that transferred data cannot overwrite memory outside a receive buffer.

3.2 Transfer Strategies

The VMMC model defines two user-level transfer strategies: deliberate update and automatic update. Deliberate update is an explicit transfer of data from a sender’s memory to a receiver’s memory.

In order to use automatic update, a sender binds a portion of its address space to an imported receive buffer, creating an automatic update binding between the local and remote memory. All writes performed to the local memory are automatically propagated to the remote memory as well, eliminating the need for an explicit send operation.

An important distinction between these two transfer strategies is that under automatic update, local memory is "bound" to a single receive buffer at the time a binding is created. While under deliberate update there is no fixed binding between a region of the sender’s memory and a particular receive buffer. Automatic update is optimized for low latency, and deliberate update is designed for flexible import-export mappings and for reducing network traffic.

The VMMC model does not include any buffer management since data is transferred directly between user-level address spaces. This gives applications the freedom to utilize as little buffering and copying as needed. The model directly supports zero-copy protocols when both the send and receive buffers are known at the time of a transfer initiation.

The VMMC model assumes that receive buffer addresses are specified by the sender, and received data is transferred directly to memory. Hence, there is no explicit receive operation. CPU involvement in receiving data can be as little as checking a flag, although a hardware notification mechanism is also supported.

Figure 1 shows the latency and bandwidth delivered by the SHRIMP VMMC layer [2].

3.3 Notifications

The notification mechanism is used to transfer control to a receiving process, or to notify the receiving process about external events. It consists of a message transfer followed by an invocation of a user-specified, user-level handler function. The receiving process can associate a separate handler function with each exported buffer, and notifications only take effect when a handler has been specified.

4 The RPC Paradigm

RPC provides the user with a well understood communication model: remote procedure calls with well-defined semantics that are close to the semantics of local procedure calls. The basic mechanism for communication. A software system is usually divided into servers and clients. The servers export well-defined service interfaces to clients, that can access the services with simple procedure calls. SunRPC is a RPC specification available in many systems. In this section we first present some background information about SunRPC and then describe the implementation and performance of our RPC libraries.
4.1 SunRPC

SunRPC [?] is a widely used remote procedure call interface and specification. SunRPC consists of a set of library functions and a stub generator that follows the XDR [?] standard for data representation. SunRPC is a single-thread implementation.

The general structure of SunRPC is shown in Figure 2. It is implemented as a series of layers, each providing services to the layers above.

- The network layer implements the read and write system calls that transfer data across the network.
- The stream layer does buffer management. It provides a set of functions to write data (byte strings, integers, etc.) to, or read data from, a stream. The stream layer hides the details of buffer management and network packet size from the higher layers. Its existence is very important to the performance of standard SunRPC implementations, since it reduces accesses to the expensive lower layers.
- The XDR layer implements the XDR data representation specification, which insulates the higher layers from issues of machine-specific data representation. Data transferred between nodes in a network are translated to XDR format before sending, and translated back from XDR when received. The XDR layer provides a set of functions to send and receive data of a certain set of types: byte streams, longs, strings, etc.
- The RPC library implements most of the SunRPC protocol, including the management of client-server connections.
- The stub generator RPCGEN produces RPC stubs based on an interface definition supplied by the user.

As Figure 2 shows, the interface below the XDR layer separates the machine-dependent code from the machine-independent layers.

Figure 2 also shows where copying takes place. Data is copied from the user buffers into the stream buffers and then passed to the operating system functions. In SunRPC there are two copies per side per RPC call at user level, plus the copies necessary at protection boundary crossing (one or two), and potentially copies in the kernel between kernel-driver buffers. These amount to a total of at least six copies per RPC call.

Figure 3 shows the common execution path in an RPC. The client sends the header and then the data.
When the server replies the client receives first the header of the reply message and then the results. Because of the multiple layers, each call to XDR to send a datum results in a chain of several procedure calls.

4.2 vRPC

Our strategy in implementing vRPC on SHRIMP was to change as little as possible, and to remain fully compatible with existing SunRPC implementations. In order to meet these goals, we changed only the runtime library; we made minor changes to the stub generator and we did not modify the kernel.

We used two main techniques to speed up the library. First, we re-implemented the network layer directly on top of the SHRIMP network interface. Because the SHRIMP interface is simple and allows direct user to user communication, our implementation of the network layer is much faster than the standard one. Our second optimization was to collapse the stream and SBL layers into a new single thin layer that provides the same functionality, thus reducing the number of layers by one.

Communication setup For vRPC a pair of mappings is established between every client and the server. Although creating mappings is expensive (relative to the very inexpensive communication) because exporting and importing buffers requires system calls, these operations are performed only once, during the initialization phase. In that sense RPC is a typical case of communications software where one can separate the expensive setup phase from the common case. The VMMC interface takes advantage of this.

The first version of vRPC, Direct vRPC, replaces the stream layer with a simple stream communication abstraction implemented directly on top of the VMMC interface. There are still six copies per RPC. Elimination of kernel involvement in the communication path is the only difference from SunRPC. Figure 2 shows the overall structure of vRPC.

Collapsing Layers As noted above, in standard implementations of SunRPC, the stream layer is needed to decouple the XDR layer from the network layer, because calls to the network layer are expensive system calls. Now that the communication is done at user level there is no need for the stream layer. In fact, as we described above, the VMMC stream layer of Direct vRPC implements a stream. Eliminating the stream layer of SunRPC leads to the second version of vRPC, Reduced vRPC, as shown in Figure 2.

Apart from considerably reducing the layering overhead, this change also eliminates two copies per node per RPC. The only copies left are the transfers of data between library and user buffers on the receiving side. This copy is essential to maintain SunRPC semantics.

These two improvements, eliminating the overhead of a whole layer and avoiding copying, are due to

1. the low cost of the user level communication mechanisms which makes the extra stream layer unnecessary, and
2. the direct placement of data in virtual memory.

Moreover, the fact that the client and the server run on nodes of the same multiprocessor leads to two additional improvements. There is no need for translation of the data to and from an agreed transmission
format. Also, authentication can be done once, when the SHRIMP connection is set up, rather than on every RPC.

Tuning vRPC Reduced vRPC is much faster than SunRPC, and in fact is faster than any previous RPC implementation that we are aware of. We achieved this performance by merely porting and adapting existing code to the SHRIMP network interface. We can improve the performance further by tuning the RPC library for the SHRIMP and taking advantage of the SunRPC semantics and implementation.

We tuned Reduced vRPC to take advantage of these opportunities. This resulted in the final version of vRPC, Optimized vRPC, which uses single-write automatic update to transfer control information. For transferring data both synchronous deliberate update and block-write automatic update can be used. We compare these configurations in section 4.2.1.

4.2.1 Measurements

Our experiments compare SunRPC and the three versions of vRPC. We divide the code into sections and present timing measurements for each section of the code, for null calls with six argument/result sizes: zero bytes, 8 bytes, 64 bytes, 256 bytes, 1440 bytes and 3K bytes. We measured 1000 calls in each case, eliminated outlying points (due to experiments being interrupted by system daemon activity, etc.) and then averaged over the remaining runs. Full results appear in Appendix A of [2].

Figures 4 and 5 summarize these results. They present the time spent in the client and the server per section of a call, for the above argument/result sizes, as a percentage of the total cost of the call. The send and exec sections for the client and the server respectively, are almost negligible, as expected. send doesn’t do much since the transfer is already initiated (automatic update) and the procedure executed in the server is a null procedure; it just sends back a reply of the same size as the arguments to the call.

4.3 ShrimpRPC: A Specialized Implementation

While our vRPC library has very good performance, its implementation was limited by the need to remain compatible with the existing SunRPC standard. To explore the further performance gains that are possible, we implemented a non-compatible version of remote procedure call.

ShrimpRPC is not compatible with any existing RPC system, but it implements the full RPC functionality, with a stub generator that reads an interface definition file and generates code to marshal and unmarshal complex data types. The stub generator and runtime library were designed with SHRIMP in mind, so we believe they come close to the best possible RPC performance on the SHRIMP hardware.

Buffer Management The design of ShrimpRPC is similar to Bershad’s URPC [7]; the main difference is that URPC runs on shared-memory machines while ShrimpRPC runs on the distributed-memory SHRIMP system. Each RPC binding consists of one receive buffer on each side (client and server) with bidirectional import/export mapping between them. When a call occurs, the client-side stub marshals the arguments into its buffer, and then transmits them into the server’s buffer. At the end of the arguments is a flag which tells the server that the arguments are in place. The buffers are laid out so that the flag is in the same place for all calls that use the same binding, so that the flag is immediately after the data. This allows both data and flag to be sent in a single data transfer.

When the server sees the flag, it calls the procedure that the client requested. At this point the arguments are still in the server’s buffer. When the call is done, the server sends return values and a flag back to the sender.

Exploiting Automatic Update The structure of our ShrimpRPC works particularly well with automatic update. In this case, the client’s buffer and the server’s buffer are connected by a bidirectional automatic update binding; whenever one process writes its buffer, the written data is propagated automatically to the other process’s buffer. The data layout and the structure of the client stub cause the client to fill memory locations consecutively while marshaling the arguments, so that all of the arguments and the flag can be combined into a single packet by the client-side hardware.

On the server side, return values (OUT and INOUT parameters) need no explicit marshaling. These variables are passed to the serverside procedure by reference: that is, by passing a pointer into the server’s communication buffer. The result is that when the procedure writes any of its OUT or INOUT
header to be sent for every RPC, while the non-compatible ShrimpRPC system sends just the data plus a one-word flag, all of which can be combined by the hardware into a single packet. For large transfers, the difference is roughly a factor of two. This occurs because ShrimpRPC does not need to explicitly send the INOUT and OUT arguments from the server back to the client; these arguments are implicitly sent in the background, via automatic update as the server writes them.

Since the round-trip latency of a ShrimpRPC call is within one microsecond of the hardware minimum, we do not provide a detailed breakdown of how ShrimpRPC spends its time.

5 Shrimp Sockets

The SHRIMP socket API is implemented as a user library, using the VMMC interface. It is compatible and seamlessly integrated with the Unix stream sockets facility [7]. A new address family, AF_SHRIMP, was added to support the new type of socket. The stream protocol was implemented for this new family.

We implemented three variations of the socket library, two using deliberate update and one using automatic update. The first protocol performs two copies, one on the receiver to move the data into the user memory and the other on the sender to eliminate the need to deal with data alignment. We can improve the performance by eliminating the send-side copy, leading to a one-copy protocol, although we must still use the two-copy protocol when dictated by alignment. The automatic-update protocol always does two copies, since the send-side copy acts as the send operation.

It is not possible to build a zero-copy deliberate-update protocol or a one-copy automatic-update protocol without violating the protection requirements of the sockets model. Such a protocol would require a page of the receiver's user memory to be exported; the sender could then clobber this page at will. This is
not acceptable in the sockets model, since the receiver does not necessarily trust the sender.

**Design** The user-level library’s socket descriptor table contains one entry for each open socket, regardless of the socket type. When non-SHRIMP sockets are used, our library passes calls through to the regular libc calls, while still keeping a descriptor table entry. Also, our library uses calls to the regular libc functions in order to bootstrap the SHRIMP connection. Figure 7 illustrates the software layers in our implementation.

**Implementation** This section describes the data structures used to implement SHRIMP sockets. We use a straightforward implementation of circular buffers in order to manage each socket’s incoming and outgoing data. First, we will describe how to implement a simple unidirectional byte stream on SHRIMP. Then we will show how this simple structure was used as the building block for our library.

The byte stream is managed by a circular buffer that consists of a data buffer and the head and tail indices. In order to implement a simple unidirectional byte stream the sender and receiver each allocate a copy of the circular queue and cross-map the three components. The buffer and tail indices are mapped from the sender to the receiver and the head index is mapped from the receiver to the sender (see Figure 8).

Figure 9 illustrates how this simple queue structure can be used to transfer bytes. The sender writes the buffer and the tail index and only reads the head index, while the receiver only reads the buffer and tail index and writes the head index. Since writes are always exclusive, race conditions are eliminated.

```c
simple_send(msg)
{
    wait until buffer space is available
    copy msg into sendbuf
    transfer sendbuf data to receiver
    update tail pointer
    transfer tail pointer to receiver
}

simple_recv(buf)
{
    wait until data is available
    copy data from recvbuf to buf
    update head pointer
    transfer head pointer to sender
}
```

In order to construct a bidirectional byte stream we use two unidirectional byte streams described above: one for sending bytes and one for receiving bytes. There is additional complexity introduced when implementing some of the other socket functions, as well as dealing with connection establishment and byte alignment. The basic underlying implementation is simple, straightforward, and works very well with SHRIMP’s VMMC communication mechanism.

For each socket descriptor that specifies a SHRIMP socket, two data structures are maintained with data grouped based on who writes the data: incoming (from the remote process) and outgoing (to the remote process). These two structures are then used to construct the two unidirectional byte streams described above. The size of each circular buffer is determined at compile time. All of the results in this paper use 32 kbyte buffers.

**Connection Setup and Shutdown** During connection establishment, the implementation uses a regular Internet socket, via Ethernet, to exchange the data required to establish two VMMC mappings (one in each direction). The Internet socket is held open, and is used to detect if the connection breaks.

When a connection is terminated (using close) the specified socket’s descriptor table entry is freed and the associated VMMC mapping is removed.

recv must copy data that is in the incoming buffer to the user-specified buffer. Once it consumes the data, it updates the head pointer on the remote node. Each call costs one copy (or two, depending on the state of the circular buffer) and a one-word message to update the pointer. The pseudo code for recv is as follows:
• check that socket is valid/alive
• verify that data is available to receive, if not, wait for data to arrive
• copy data from socket library’s circular buffer into user buffer
• update remote node’s circular buffer start pointer (using a notification)

send must copy the user data into the circular buffer and initiate the transfer to the remote node. For deliberate update this requires an explicit SHRIMP call to initiate data transfer. Two calls are required if the data wraps around the circular buffer. When automatic update is used, copying the data into the circular buffer automatically triggers the send. Once the data is sent, another send is required to notify the remote node that new data has arrived (by updating the tail pointer). The pseudo code for send is as follows:

• check that socket is valid/alive
• verify that space is available for the remote node to receive data, if not spin
• send data from user space to remote node using SHRIMP. Either automatic or deliberate update can be used.
• update remote node’s circular buffer tail pointer (using a notification)

select deals with both regular and SHRIMP sockets and thus is the most complex function in the library. It uses notifications in order to trap the arrival of new SHRIMP data. A null user-level notification handler is used because notifications are only needed to implement select.

When a buffer is exported, notifications are option-ally activated, and a user-level handler is specified. Each exported buffer’s notifications can be in one of three states:

• ignore: notifications are dropped
• queue: notifications blocked are queued for later delivery
• deliver: all queued, and any incoming, notifications are delivered

Since notifications are implemented using interrupts we do not want the common case to pay the notification processing cost. To achieve this initially all of the socket library’s buffers ignore notifications. select must first queue notifications for the sockets in question.

The algorithm for our select was developed around the functionality of the standard select. select returns when I/O occurs or any signal is trapped. Since signals are used to implement notifications, they are also trapped by the standard select. When select completes it returns no indication of which, if any, notifications have occurred. This leads to the following implementation of select:

loop until data arrives, or timeout:

• queue notifications: we do not want to miss any that might occur
• deliver notifications: turn on notifications for the SHRIMP socket data structures
• check for new data: check the SHRIMP socket data structures to see if any data has arrived
• select: wait for a notification, other I/O, with a timeout. Since a notification can ar-rive after checking for data and before calling select we cannot block forever. Instead, we use a mini-timeout in tandem with the loop.
• ignore notifications: return notifications to the fast common case
• check for new data: do another check in case any data has arrived in the intermi

Integration with libc. Integrating the functions in the user level library with those already present in libc was straightforward. For each function imple-mented in our library, we renamed the corresponding function in libc by changing the name to up- percase. In order to keep things simple, we did this directly in the binary object modules of libc. Where necessary, our library is able to call the original socket functions.

5.1 Measurements

This section describes the performance measure-ments we obtained using our implementation of sock-ets. All tests were performed on a prototype four-node SHRIMP system. Each node is a 60 MHz Pentium PC running Linux 1.2.8. We used gcc 2.6.3 with all optimizations enabled (“O3”.

All measurements were taken for three different implementations of the socket library. The three implementations differed only in the data transfer mechanism that was used:

• DU+1copy: deliberate update with only one copy. The receiver moves the incoming data from the library’s internal buffer to the user’s buffer.

• DU+2copy: deliberate update with two copies. The sender copies the data from the source to the library’s internal buffer and the receiver moves the incoming data from the library’s internal buffer to the user’s buffer.

• AU+1copy: automatic update with one copy. The receiver moves the incoming data from the library’s internal buffer to the user’s buffer. Note that the sender also does a copy from the source to the library’s internal buffer but this copy is not counted because it acts as the actual data transfer.

Finally, we use the following three metrics from [?] to characterize performance:
Figure 10: Micro-benchmark socket bandwidth

Figure 11: Micro-benchmark socket latency

Figure 12: Ttcp socket bandwidth

Figure 13: Netperf socket bandwidth

Micro-Benchmarks To determine the bandwidth performance of our sockets library, we measured the time required for a large number of consecutive transfers in the same direction. Figure 10 shows the results. All three implementations have similar performance. For large messages, performance is very close to the raw hardware one-copy limit of about $r_\infty \approx 13.5$ MBytes/sec. Further, all three of the performance curves have a steep rise indicating that peak performance is obtained quickly: $n_T = 256$ bytes.

Figure 11 shows the latency for small transfers, measured using a ping-pong test. For small messages, we incur a one-way latency of $l = 11$ µsec, or about 7 µsec above the hardware limit. This extra time is spent equally between the sender and receiver performing procedure calls, checking for errors, and accessing the socket data structure.

Ttcp is a public-domain benchmark originally written at the Army Ballistics Research Lab. Ttcp measures network performance using a one-way communication test in which the sender continuously pumps data to the receiver. The performance of our library as reported by Ttcp is shown in Figure 12. Ttcp obtained a peak bandwidth of $r_\infty = 12.4$ MBytes/sec using 7 KByte messages. Once again, the bandwidth rises quickly as the message size increases: $n_T = 512$ bytes.

Netperf (Revision 1.7) is another public domain benchmark developed at Hewlett-Packard. Figure 13 shows the performance of our library as reported by the TCP stream test. The peak bandwidth is about $r_\infty = 13$ MBytes/sec and $n_T = 256$ bytes.

For more detailed performance results see [?].
6 Future Work

The sockets implementation we described does not allow open sockets to be preserved across fork and exec calls. With fork, the problem is arbitrating socket access between the two resulting processes. exec is difficult because SHRIMP communicates through memory and exec allocates a new memory space for the process. Maeda and Bershad discuss how to implement fork and exec correctly in the presence of user-level networking software [2]. We intend to follow their solution.

Moreover our sockets implementation ignores all fcntl calls, passing them through to the underlying kernel socket. While this behavior is correct in some cases, in others it is not. We intend to implement a better fcntl that handles all of the fcntl directives correctly.

Finally, our current sockets implementation does not support asynchronous I/O to sockets. Doing so would require a straightforward use of SHRIMP notifications, but we have not implemented it yet.

As mentioned above, VMMC delivers very high performance to communications libraries. However, certain aspects of the client-server model are not supported. Protection is one of these. Although each mapping is a one way communication channel between two processes, other processes could maliciously destroy data by guessing correctly certain pieces of information. This is not a problem for URPC since SunRPC includes provision for authentication between the client and the server. In the case of stream sockets the application would have to build its own protection mechanisms. Currently we are investigating extensions to VMMC to better support the client-server model.

7 Related Work

RPC Our approach is similar in some ways to URPC [2], since both exploit user-level communication. URPC is built on top of a shared memory architecture while we use the distributed-memory SHRIMP architecture.

Bershad’s LRPC [2] tries to optimize the kernel path for same-machine RPC calls. Since we have eliminated the kernel entirely, LRPC does not apply to our situation.

Thelkatt and Levy [2] investigated the impact of recent improvements in network technology on communication software. They point out that both high throughput and low latency are required by modern distributed systems and that newer networks strive only for high throughput. They develop techniques to achieve low latency in communication software, using RPC as a case study. Their goal along with demonstrating the new techniques is to provide guidelines for network controllers that will facilitate writing low latency communication software in traditional architectures.

Active messagees [2] are a restricted form of RPC, in which the server-side procedure may not perform any actions that might block. Active messagees achieve performance similar to Shrimp RPC on high-performance hardware, but without allowing general handlers to be invoked. The Optimistic Active Messagees [2] approach allows an arbitrary handler to be invoked, using a fast-path implementation but switching to a slower path if the handler blocks. Neither of these systems provides full RPC services, such as automatic stub generation or binding between untrusting parties.

Several papers (e.g., [2, 8]) describe optimizations that dramatically improve the performance of RPC in traditional systems. This is generally done by avoiding copying, and reducing context switching overhead and network and RPC protocol overhead.

Stream Sockets Several groups have studied how to support the socket interface on experimental high-performance network interfaces.

Boden et al. [2] describe an implementation of TCP/IP on using the Myrinet API. This implementation had a minimum user-to-user latency of over 40 µsec, which is considerably larger than ours. Two factors contribute to this: first, they implemented full TCP/IP while we support the stream socket interface directly; second, their underlying hardware has a higher communication latency than ours.

To illustrate this, consider the following measurements of two custom APIs that are not compatible with sockets. Myricom’s custom API using their interconnect results in a latency of 40 µsec and a peak bandwidth of 27 Mbytes/sec. Illinois Fast Message (FM) [2] also implement a custom API using the Myrinet hardware. FM sacrifices some features available in the Myricom API and some peak bandwidth in order to reduce the small-message latency by a factor of almost two. Their one-way latency is 24 µsec while they obtain a peak bandwidth of 19.6 Mbytes/sec.

U-Net [2] describes an architecture for user-level communications which is independent of the network interface hardware. Using Sun SparcStations and Fore Systems ATM interfaces, they measured a TCP one-way latency of 78 µsec and a bandwidth of 14.4 Mbytes/sec for 4 Kbyte packets.

The xkernel framework allows user-level implementation of a wide variety of network protocols. Experimental results for sockets on high-performance network interfaces are not yet available.

8 Conclusion

Network interfaces can have a great impact on communication performance and ease of programming. User-level communication and virtual memory mapped network interfaces embody many of the optimizations done in other systems at extra work. Simply modifying an existing communication library can give results close to or better than the most highly optimized version in traditional interfaces. Copying can be
limited to a minimum, and there are no interrupts on packet receipt and no kernel intervention.

The VMMC interface reduces the user to user latency. Using low latency mechanisms and providing support for user level communication leads to high performance communication software.

Our libraries are able to achieve very low latency by exploiting the features of SHRIMP. The freedom to use customized buffer management strategies allows us to design very efficient implementations of communication libraries. The separation of control transfer from data transfer allowed us to avoid receive-side interrupts, where this can be beneficial.

This effort shows that new architectures can open new horizons to distributed programming by providing high performance at low implementation cost.

Acknowledgments

We would like to thank Kai Li, Matt Blumrich, Liviu Itode and the rest of the SHRIMP Group at Princeton for their many useful suggestions that contributed to this work.

This project is sponsored in part by ARPA under contract N00014-95-1-1144, by NSF under grant MIP-9420653, by Digital Equipment Corporation and by Intel Corporation. Felen is supported by an NSF National Young Investigator Award.