Stream Sockets on *SHRIMP*

Stefanos N. Damianakis, Cezary Dubnicki, Edward W. Felten

Department of Computer Science, Princeton University, Princeton NJ 08544, USA
{snd,dubnicki,felten}@cs.princeton.edu

Abstract. This paper describes an implementation 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 the interrupting the receiving node's CPU. Our sockets implementation exploits all of these features to provide high performance. End-to-end latency for 8 byte transfers is 11 microseconds, which is considerably lower than all previous implementations of the sockets interface. For large transfers, we obtain a bandwidth of 13.5 MBytes/sec, which is close to the hardware limit when the receiver must perform a copy. Further experiments with the public-domain benchmarks `ttcp` and `netperf` confirm the performance of our implementation.

1 Introduction

A wide variety of distributed applications rely on the Berkeley stream sockets model for interprocess communication. The stream socket interface provides a connection-oriented, bidirectional byte-stream abstraction, with well-defined mechanisms for creating and destroying connections and for detecting errors.

In this paper, we describe an implementation of the sockets interface on the *SHRIMP* multicomputer. Our implementation is compatible with the stream sockets interface; it properly detects broken connections and correctly implements the `select` call.

Our 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 [15]. For large transfers our implementation obtains a bandwidth of 13.5 MBytes/sec, which is close to the hardware limit when the receiver must perform a copy.

We achieve high performance by taking advantage of the features of the underlying *SHRIMP* architecture. *SHRIMP* provides low-latency protected user-level communication, without imposing any buffer management semantics, and without requiring an interrupt for each message. These properties allow packages like our sockets library to customize their buffer management and control transfer strategies to optimize performance.

We measured the performance of our implementation on a prototype *SHRIMP* machine, using custom micro-benchmarks in addition to two well-known benchmarks: `ttcp` and `netperf`.

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 [22] 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) [10]. This mechanism delivers excellent performance and supports various message-passing packages and applications effectively [11].

The prototype system consists of four interconnected nodes. Each node is a DEC 560ST PC containing an 60 MHz Intel Pentium processor with an external 256 Kbyte second-level cache, and 40 MBytes of main DRAM memory. Peripherals are connected to the system through the EISA expansion bus, which has main memory mastering capability and a maximum burst bandwidth of 33 MBytes/sec.

Main memory data can be cached by the CPU as write-through or write-back on a per-virtual-page basis, as specified in process page tables. The caches snoop DMA transactions and automatically invalidate corresponding cache lines, thereby keeping consistent with *all* main memory updates, including those from EISA bus masters.

The network connecting the nodes is an Intel routing backplane consisting of a two-dimensional mesh of Intel Mesh Routing Chips (iMRCs) [22], and is the same network used in the Paragon multicomputer. The backplane supports deadlock-free, oblivious wormhole routing, and preserves the order of messages from each sender to each receiver.

In addition to the fast backplane interconnect, the PC nodes are connected by a commodity Ethernet, which is used for diagnostics, booting, and exchange of low-priority messages.

The custom network interface [3, 4] is the system's key component. It connects each PC node to the routing backplane and implements the hardware support for VMMC.

3 Virtual Memory Mapped Communication

Virtual memory-mapped communication (VMMC) addresses the need for a basic multicomputer communication mechanism with extremely low latency and high bandwidth. VMMC achieves this by allowing applications to transfer data directly between two virtual memory address spaces over the network. This mechanism efficiently supports applications and common communication models such as message passing, shared memory, RPC, and client-server communication.

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 together with a set of permissions to define access rights for the buffer. In order to send data to an exported receive buffer, a user process must *import* the buffer with the appropriate permissions. After a successful import, a sender can transfer data from its virtual memory into an imported receive buffer at user-level without further protection checking and without entering the kernel. The *SHRIMP* implementation forces page granularity for all import-export mappings.

The *unexport* and *unimport* primitives can be used to destroy existing import-export mappings. Before completing, these calls wait for all currently pending messages using the mapping to be delivered.

3.2 Protection

Communication in VMMC is protected in two ways. First, communication may take place only after the receiver gives the sender permission (by calling *export*) to transfer data to a given area of the receiver's address space. Import and export requests are matched by a trusted daemon process.

Second, at send initiation, the *SHRIMP* implementation uses the hardware virtual memory management unit (MMU) to verify permission to send data to a receive buffer. For more details please see [5].

3.3 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 automatic upate there is no explicit send operation; all writes performed to the local memory are automatically propagated to the remote memory (by the *SHRIMP* hardware). 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.

An important distinction between these two transfer strategies is that under automatic update, local memory is "bound" to a receive buffer at the time a mapping is created, while under deliberate update the binding does not occur until an explicit send command is issued. Automatic update is optimized for low latency, and deliberate update is designed for flexible import-export mappings and for reducing network traffic.

Automatic update is implemented by having the *SHRIMP* network interface hardware snoop all writes on the memory bus. If a write is to an address that has an automatic update binding, the hardware builds a packet containing the destination address and the written value, and sends it to the destination node. The hardware can combine writes to consecutive locations into a single packet.

Deliberate update is implemented by having a user-level program execute a sequence of two accesses to addresses which are decoded by the *SHRIMP* network interface board on the node's expansion bus (the EISA bus). These accesses specify the source address, destination address, and size of a transfer. The ordinary virtual memory protection mechanisms (MMU and page tables) are used to maintain protection [5]. The hardware requires that both source and destination addresses are word aligned.

VMMC guarantees the in-order, reliable delivery of all data transfers. but does not include any buffer management since data is transferred directly between user-level address spaces. This gives libraries and applications the freedom to use as little buffering and copying as needed.

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 the hardware also supports a notification mechanism.

3.4 Notifications

The *notification* mechanism is used to transfer control to a receiving process, or to notify a 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.

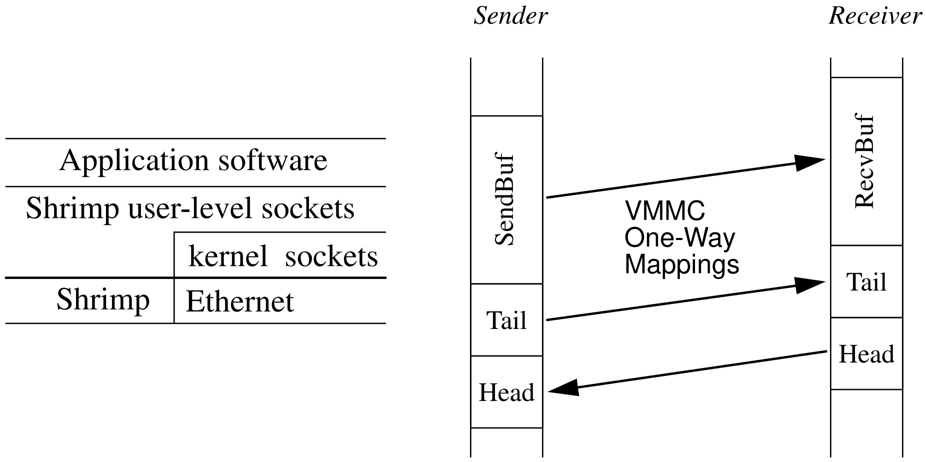
Notifications are similar to UNIX signals in that they can be blocked and unblocked, they can be accepted or discarded (on a per-buffer basis), and a process can be suspended until a particular notification arrives. Unlike signals, however, notifications are queued when blocked.

4 Shrimp Sockets

The *SHRIMP* socket API is implemented as a user-level library, using the VMMC interface. It is compatible, and seamlessly integrated with the Unix stream sockets facility [16]. We introduce a new address family, `AF_SHRIMP`, to support the new type of stream socket.

We implement three variations of the socket library, two using deliberate update and one using automatic update. The first protocol performs two copies, one on the sender to eliminate the need to deal with data alignment, and the other on the receiver to move the data into the user memory. We can improve the performance by eliminating the send-side copy, leading to a one-copy protocol. Our current protocol uses two-copies for unaligned transfers in order to minimize small message latency. The automatic-update protocol always does two copies, since the sender-side copy acts as the send operation.

On *SHRIMP*, it is not possible to eliminate the receive side copy 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



(a) Implementation Layers

(b) Unidirectional Byte Stream

Fig. 1. *Socket Diagrams*

then clobber this page at will. This is not acceptable in the sockets model, since the receiver does not necessarily trust the sender.

4.1 Implementation

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` socket calls, while still maintaining a descriptor table entry. Also, our library uses calls to the regular `libc` socket functions in order to bootstrap the *SHRIMP* connection. Figure 1a illustrates the software layers in our implementation.

We use a straightforward implementation of circular buffers in order to manage each socket's incoming and outgoing data. First, we describe how to implement a simple unidirectional byte stream on *SHRIMP*. Then we show how we use this simple structure 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 tail index and buffer are mapped from the sender to the receiver and the head index is mapped from the receiver to the sender (see Figure 1b).

Figure 2 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 to the index variables are always exclusive, race conditions are eliminated.

```
simple_send(msg)                          simple_recv(buf)
{                                         {
  wait until buffer space is available      wait until data is available
  copy msg into sendbuf                     copy data from recvbuf to buf
  transfer sendbuf data to receiver         update head pointer
  update tail pointer                       transfer head pointer to sender
  transfer tail pointer to receiver       }
}
```

Fig. 2. *Pseudo code for simple send and recv*

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. This basic underlying implementation is simple, straightforward, and works very well with *SHRIMP*'s VMMC communication mechanism.

4.2 Connection Setup and Shutdown

During connection establishment, the implementation uses a regular TCP socket connection to exchange the data required to establish two VMMC mappings (one in each direction). The TCP socket remains open, and is used to detect certain errors, such as node failure. To import a buffer a process must obtain the following information about the remote process and node:

- `IPaddr`: the node's IP address
- `squID`: similar to a process ID, obtained by a call to the *SHRIMP* library
- `bufferID`: identifies which exported buffer to import

The two VMMC mappings are established when the server process is in `accept` and the client process is in `connect`. Our implementations of these functions call an internal routine `Shrimp_Setup` that does the following:

- **pack** local data describing process and node
- **allocate buffers** for incoming and outgoing data
- **export** the receive buffer
- **client** process
  1. *TCP-send* the node's local id structure
  2. *TCP-receive* the remote node's id structure
  3. *import* the receive buffer (from the server)
- **server** process
  1. *TCP-receive* the remote node's id structure
  2. *TCP-send* the node's local id structure
  3. *import* the receive buffer (from the client)

The transfer strategy, deliberate or automatic update, is determined (using compile time defines) in `Shrimp_Setup`. When this function completes on both nodes, the two processes are ready to communicate using the *SHRIMP* hardware.

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

4.3 `send`

`send` must copy the user data into the local 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, the sender sends a one-word message to update the receiver's tail pointer. The pseudo-code for `send` is as follows:

- **check** that socket is valid and alive
- **verify** that space is available for the remote node to receive data; if not, spin until space becomes available
- **send** data to remote node with automatic or deliberate update
- **update** remote node's circular buffer tail pointer

4.4 `recv`

`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 and 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

4.5 `select`

`select` is the most complex function in the library because it deals with both TCP and *SHRIMP* sockets. It uses notifications in order to detect the arrival of new *SHRIMP* data. `select` uses a null user-level notification handler.

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

- *ignore*: notifications are dropped
- *accumulate*: notifications are queued for later delivery

– *deliver*: accumulated and 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 user-level `select` uses the kernel version of select, which we denote `SELECT`. `SELECT` returns when I/O occurs or any signal is trapped. Since signals are used to implement notifications, `SELECT` also traps them. But, when `SELECT` completes it returns no indication of which, if any, signals have occurred. This leads to the following implementation of our user-level `select`:

– *deliver notifications*: turn on notifications for Shrimp socket data
– *check for new data*: check the Shrimp socket data structures to see if any data has arrived
– *loop* until data has arrived:
  - call `SELECT`, i.e. block in the kernel for a notification or other I/O, with a timeout. The timeout is required because a notification can arrive after checking for data but before calling `SELECT`; therefore we use a timeout in tandem with the loop.
  - *check for new data*: check the Shrimp socket data structures and exit loop if any new data has arrived
– *ignore notifications*: return notifications to the fast common case

4.6 Integration with `libc`

We integrated our library with `libc` by renaming the corresponding functions in `libc` to all uppercase. 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 Performance Issues

5.1 Sending and Aligning Bytes

Socket communications operate at the byte level: all sends and receives specify byte counts, and there is no guaranteed relationship between the alignment of the send and receive buffers. The *SHRIMP* hardware is most efficient when transferring words, so byte transfers are best done by transferring a little extra data.

In addition, *SHRIMP* requires that all deliberate update transfers be word aligned. This requirement complicates the implementation. On the sending side, if the data is copied from the user buffer to the socket library's circular buffer, then the alignment problem goes away, since the extra copy into the library's buffer forces proper byte alignment. We can eliminate this extra copy by encoding byte alignment and send it along with the data; the receiver must decode this information when removing data from its buffer.

5.2 Deliberate vs. Automatic Update

We must choose when to use the deliberate update strategy and when to use automatic update. Automatic update is clearly faster for small transfers so we use automatic update to transfer control information (i.e. the circular buffers' index values).

For transferring message data, the choice is less clear-cut. Deliberate update has slightly higher asymptotic bandwidth, but automatic update is convenient to use and does not restrict buffer alignment.

When we use automatic update, the sender transmits the message data to the receiver by simply copying the data from the user memory to the send-side communication buffer. We do not count this as a data copy since this acts as the data transfer. The benefit of this transfer mechanism is that no explicit send operation is required. Thus, every automatic update protocol does only one copy, on the receive side.

5.3 Caching Effects

Our experience with *SHRIMP* shows that communication performance can be affected by the caching strategy used for send-buffer and receive-buffer memory; communication performance is better with write-through caching [2]. We do not want to change the caching strategy for the application's data, since that would probably degrade the application's performance. However, we can safely change the caching mode for pages that consist entirely of our internal data buffers.

6 Measurements

This section describes the performance measurements we obtain using our implementation of sockets. All tests are performed on a prototype four-node *SHRIMP* system. Each node is a 60 MHz Pentium PC running Linux 1.2.8. We use `gcc` 2.6.3 with all optimizations enabled (`-O3`).

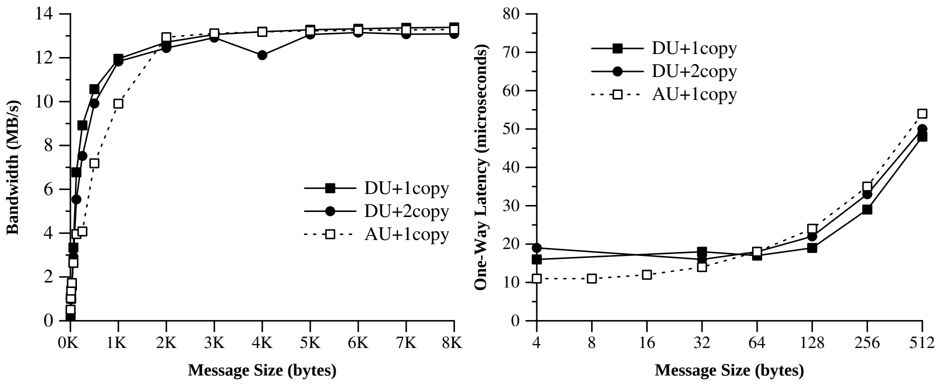
We report the performance for three different implementations of the socket library. The three implementations differ only in the data transfer mechanism:

- **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. For the current implementation, this places alignment restrictions on the sender.
- **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 [19] to characterize performance:

$$
\begin{aligned}
r_\infty &= \text{peak bandwidth for infinitely large packets} \\
n_{\frac{1}{2}} &= \text{packet size to achieve } \tfrac{r_\infty}{2} \text{ bandwidth} \\
l &= \text{one way packet latency}
\end{aligned}
$$

6.1 Micro-Benchmarks



(a) *Bandwidth* (b) *Latency*

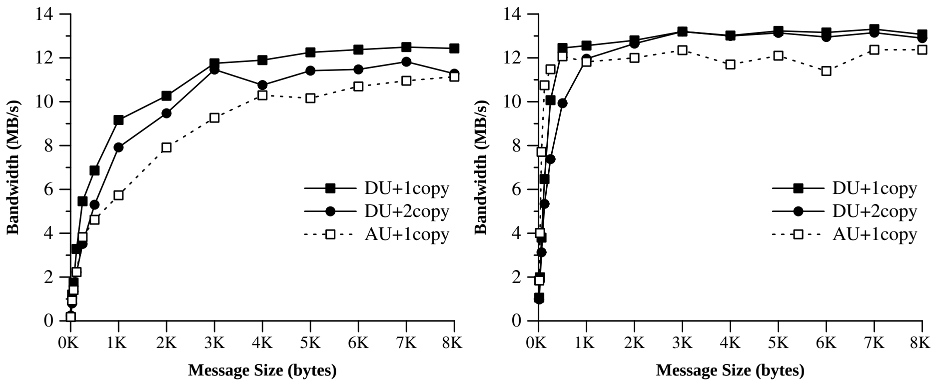
Fig. 3. *Socket Micro-benchmarks*

To determine the bandwidth performance of our sockets library, we measure the time required for a large number of consecutive transfers in the same direction. Figure 3a 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_∞=13.5 MBytes/sec. Further, all three of the performance curves have a steep rise indicating that peak performance is obtained quickly: $n_{\frac{1}{2}}$=256 bytes.

Figure 3b 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.

6.2 Ttcp

`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` appears in Figure 4a. `Ttcp` obtained a peak bandwidth of r_∞=12.4 MBytes/sec using 7 KByte messages. Once again, the bandwidth rises quickly as the message size increases: $n_{\frac{1}{2}}$=512 bytes.



(a) `Ttcp` *(b)* `Netperf`

Fig. 4. *Socket Bandwidth*

6.3 Netperf

`Netperf` (Revision 1.7) is a public domain benchmark from Hewlett-Packard. Figure 4b shows the performance of our library as reported by the TCP stream test. The peak bandwidth is about r_∞=13 MBytes/sec and $n_{\frac{1}{2}}$=256 bytes.

6.4 Discussion of Results

Our sockets implementation achieves low latency because of customized buffer management and the low software overhead of VMMC on *SHRIMP*. The available user-to-user bandwidth grows fast with the message size; $n_{\frac{1}{2}}$ is between 256 and 512 for all experiments. The bandwidth curves in Figures 3 and 4 show that automatic update closely tracks deliberate update, to within 7% - 17%. Yet for small messages (Figure 3b), automatic update's latency is 30% - 40% less than deliberate update's. Furthermore, all the bandwidth curves continue level beyond the 8K limit of the graphs; we have tested messages up to 64K. However, the maximum bandwidth is limited because we must use a memory copy.

The results for all of the experiments are consistent: the `AU+1copy` implementation performs best for small transfers, and the `DU+1copy` implementation is better for large transfers. The crossover point is at a message size of 256 bytes. This indicates that a hybrid implementation, which switches between transfer modes, based on the transfer size, would provide the best performance. We expect to implement this approach in the near future.

However, we have implemented a second hybrid approach: using deliberate update to transfer the buffer data and automatic update to transfer the index values. This results in practically no change in the latency and only a small improvement (less than 6%) in bandwidth.

7 Future Work

The 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 [18]. We intend to follow their solution.

Our current implementation ignores all `fcntl` calls, passing them through to the underlying kernel socket. While this behavior is correct is 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 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.

8 Related Work

We have previously presented this work in [9] and its results were summarized in [11]. Several groups have studied how to support the socket interface on experimental high-performance network interfaces.

The most closely related work is by Rodrigues et al. [21] who have a user-level implementation of sockets using *active messages* [24]. Their sockets can take advantage of receives that are posted before sends, which allows them a zero-copy implementation. They obtain similar bandwidth numbers, despite using different operating system, CPU, and networking hardware. Because of different network interface we obtain a significantly lower small message latency.

U-Net [23] 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.

Boden et al. [6] describe an implementation of TCP/IP 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 implement full TCP/IP while we support the stream socket interface directly; second, their underlying hardware has a higher communication latency than ours.

Even with the "restriction" of building a compatible message passing library we obtain performance comparable to "unrestricted" APIs. To illustrate this, consider the following measurements of two 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 Messages (FM) [19] 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.

Wilkes's "sender-based" communication in the Hamlyn system [25] supports user-level message passing, but requires application programs to build packet headers. They have not tried to implement message-passing libraries using the underlying communication mechanism.

Several projects have tried to lower overhead by bringing the network all the way into the processor and mapping the network interface FIFOs to special processor registers [7, 13, 8]. While this is efficient for fine-grain, low-latency communication, it requires the use of a non-standard CPU, and it does not support the protection of multiple contexts in a multiprogramming environment. The Connection Machine CM-5 implements user-level communication through memory-mapped network interface FIFOs [17, 12]. Protection is provided through the virtual memory system, which controls access to these FIFOs. However, there are a limited number of FIFOs, so they must be shared within a partition (subset of nodes), restricting the degree of multiprogramming. Protection is provided between separate partitions, but not between processes within a partition. Since the application must build packet headers, message passing overhead is still hundreds of CPU instructions.

Other high-performance networking libraries have been implemented on *SHRIMP*. Bilas [2] describes two implementations of remote procedure call, one that is compatible with SunRPC [14] and one that is not. The SunRPC-compatible library achieves latency and bandwidth comparable to our sockets implementation. One layer of this RPC implementation implements a bidirectional stream which can be viewed as a specialized implementation of our sockets library. Alpert [1] describes an implementation of the NX/2 [20] message-passing library for SHRIMP; it also achieves latency similar to ours.

9 Conclusion

Our sockets implementation was able to achieve very low latency by exploiting the features of *SHRIMP*. The freedom to use customized buffer management strategies allows us to design a very efficient implementation of bidirectional streams. The separation of control transfer from data transfer allows us to avoid receive-side interrupts.

These results correspond to the strengths of each transfer strategy. Larger messages benefit from deliberate update's DMA transfers, while automatic update snoops the writes on the memory bus and thus is better at handling smaller transfers.

The impact of a memory copy on bandwidth must not be overlooked. As I/O buses and networks get faster, the effect of a memory copy will become more pronounced. We believe the next important step in improving sockets performance is to develop a general zero-copy implementation.

Our sockets implementation, and implementations of other message-passing facilities on *SHRIMP* [1, 2] demonstrate that *SHRIMP* has met its goal of providing flexible, very high performance communication.

Acknowledgments

We would like to thank Kai Li, Matt Blumrich, Liviu Iftode, Rob Shillner, 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. Felten is supported by an NSF National Young Investigator Award.

References

1. Richard Alpert and Edward W. Felten. Design and Implementation of NX Message Passing Using SHRIMP Virtual Memory Mapped Communication. Technical Report TR-507-96, Dept. of Computer Science, Princeton University, January 1996.
2. Angelos Bilas and Edward W. Felten. Fast RPC on the SHRIMP Virtual Memory Mapped Network Interface. Technical Report TR-512-96, Dept. of Computer Science, Princeton University, February 1996.
3. M. Blumrich, K. Li, R. Alpert, C. Dubnicki, E. Felten, and J. Sandberg. A Virtual Memory Mapped Network Interface for the Shrimp Multicomputer. In *Proceedings of 21th International Symposium on Computer Architecture*, pages 142–153, April 1994.
4. Matthias A. Blumrich. *Network Interface for Protected, User-Level Communication.* PhD thesis, Dept. of Computer Science, Princeton University, June 1996. Available as technical report TR-522-96.
5. Matthias A. Blumrich, Cezary Dubnicki, Edward W. Felten, and Kai Li. Protected, User-Level DMA for the SHRIMP Network Interface. In *Proc. 2nd Intl. Symposium on High-Performance Computer Architecture*, pages 154–165, February 1996.
6. Nanette J. Boden, Danny Cohen, Robert E. Felderman, Alan E. Kulawik, Charles L. Seitz, Jakov N. Seizovic, and Wen-King Su. Myrinet: A Gigabit-per-Second Local Area Network. *IEEE Micro*, 15(1):29–36, February 1995.
7. S. Borkar, R. Cohn, G. Cox, T. Gross, H.T. Kung, M. Lam, M. Levine, B. Moore, W. Moore, C. Peterson, J. Susman, J. Sutton, J. Urbanski, and J. Webb. Supporting Systolic and Memory Communication in iWarp. In *Proceedings of 17th International Symposium on Computer Architecture*, June 1990.
8. William J. Dally. The J-Machine System. In P.H. Winston and S.A. Shellard, editors, *Artificial Intelligence at MIT: Expanding Frontiers*, pages 550–580. MIT Press, 1990.
9. Stefanos Damianakis, Cezary Dubnicki, and Edward W. Felten. Stream Sockets on SHRIMP. Technical Report TR-513-96, Dept. of Computer Science, Princeton University, February 1996.

10. Cezary Dubnicki, Liviu Iftode, Edward W. Felten, and Kai Li. Software Support for Virtual Memory-Mapped Communication. In *Proceedings of the IEEE 8th International Parallel Processing Symposium*, April 1996.
11. Edward W. Felten, Richard Alpert, Angelos Bilas, Matthias A. Blumrich, Douglas W. Clark, Stefanos Damianakis, Cezary Dubnicki, Liviu Iftode, and Kai Li. Early Experience with Message-Passing on the Shrimp Multicomputer. In *Proceedings of 23th International Symposium on Computer Architecture*, May 1996.
12. FORE Systems. *TCA-100 TURBOchannel ATM Computer Interface, User's Manual*, 1992.
13. Dana S. Henry and Christopher F. Joerg. A Tightly-Coupled Processor-Network Interface. In *Proceedings of 5th International Conference on Architectur al Support for Programming Languages and Operating Systems*, pages 111–122, October 1992.
14. Internet Request for Comments RFC 1057, Internet Network Working Group. *RPC: Remote Procedure Call Protocol Specification, Version 2*, June 1988.
15. Kimberly K. Keeton, Thomas E. Anderson, and David A. Patterson. LogP: The Case for Low-Overhead Local Area Networks. In *Hot Interconnects III*, August 1995.
16. Samuel J. Leffler, Marshall Kirk McKusick, Michael J. Karels, and John S. Quarterman. *The Design and Implementation of the 4.3BSD Unix Operating System*. Addison Wesley, 1989.
17. C.E. Leiserson, Z.S. Abuhamdeh, D.C. Douglas, C.R. Feynman, M.N. Ganmukhi, J.V. Hill, D. Hillis, B.C. Kuszmaul, M.A. St. Pierre, D.S. Wells, M.C. Wong, S. Yang, and R. Zak. The Network Architecture of the Connection Machine CM-5. In *Proceedings of 4th ACM Symposium on Parallel Algorithms and Architectures*, pages 272–285, June 1992.
18. Chris Maeda and Brian N. Bershad. Protocol Service Decomposition for High-Performance Networking. In *Proceedings of 14th ACM Symposium on Operating Systems Principles*, pages 244–255, December 1993.
19. Scott Pakin, Mario Lauria, and Andrew Chien. High Performance Messaging on Workstations: Illinois Fast Message (FM) for Myrinet. In *Supercomputing '95*, November 1995.
20. Paul Pierce. The NX/2 Operating System. In *Proceedings of Third Conference on Hypercube Concurrent Computers and Applications*, pages 384–390, January 1988.
21. Steven H. Rodrigues, Thomas E. Anderson, and David E. Culler. High-Performance Local-Area Communication With Fast Sockets. In *Proc. of Winter 1997 USENIX Symposium*, January 1997.
22. Roger Traylor and Dave Dunning. Routing Chip Set for Intel Paragon Parallel Supercomputer. In *Proceedings of Hot Chips '92 Symposium*, August 1992.
23. Thorsten von Eicken, Anindya Basu, Vineet Buch, and Werner Vogels. U-Net: A User-Level Network Interface for Parallel and Distributed Computing. In *Proceedings of 15th ACM Symposium on Operating Systems Principles*, pages 40–53, December 1995.
24. Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauser. Active Messages: A Mechanism for Integrated Communication and Computation. In *Proceedings of 19th International Symposium on Computer Architecture*, pages 256–266, May 1992.
25. John Wilkes. Hamlyn – An Interface for Sender-Based Communications. Technical Report HPL-OSR-92-13, Hewlett-Packard Laboratories, November 1993.