Efficient Connection-Oriented Communication on High-Performance Networks

Stefanos Nektarios Damianakis

A Dissertation
Presented to the Faculty
of Princeton University
in Candidacy for the Degree
of Doctor of Philosophy

Recommended for Acceptance
by the Department of
Computer Science

June 1998

This thesis is dedicated to my parents, Stylianos and Eleftheria

Στους Γονεις *μ*ου, Στυλιανο και *Ε*λευθερια

Abstract

Network protocols have traditionally been implemented in the kernel in order to provide protected communication between processes. As gigabit network fabrics become available, the kernel protocol stack becomes a communication performance bottleneck, affecting both latency and bandwidth. This dissertation investigates how to use protected, user-level communication mechanisms to support efficient, connection-oriented communication.

Most user-level communication subsystems are designed for custom communication models. To support fast user-level communication, new network interfaces place incoming data directly in user memory, at an address specified by the sender. This capability enables the construction of efficient communication subsystems since the network interface can deliver messages without any software intervention. Such communication subsystems do not support connection-oriented communication well for two reasons. First, direct data transfer does not provide message detection as required by the connection-oriented model, and second, sender-based communication does not provide the anonymity of receivers required by the connection-oriented model.

This dissertation studies how to support efficient, user-level, connection-oriented communication on high-performance networks. First, the issue of message arrival detection for user-level communication models is addressed. The traditional approach uses interrupts to indicate the arrival of new messages, which defeats the benefit of user-level communication since a protection crossing is needed with every message. To minimize the interrupt overhead, a competitive mechanism that combines polling and interrupts to efficiently detect the arrival of new messages at user-level is proposed. Second, the mismatch between connection-oriented APIs and sender-based communication models is addressed. The traditional approach requires receivers to copy messages to their ultimate destinations, which are unknown by senders. To enable zero-copy, connection-oriented protocols, a

mechanism that allows the receiver to dynamically change the destination of a message in a sender-based model is proposed.

These ideas were integrated into a user-level communication model and evaluated using a stream sockets implementation on two different multiprocessor platforms. The results from experiments with benchmarks and applications show that these techniques provide significant performance improvements for user-level, connection-oriented communication.

Acknowledgments

I am fortunate to have the "dream team" of thesis advisors [7]: Kai Li, Edward Felten, and Douglas Clark. Working with them has been a wonderful learning experience. I want to thank them for their help and support. I see each of them, in their own way, as role a model. I am especially grateful to my principal advisor, Kai, for his patience and kindness.

The computer science department at Princeton University is a terrific place to study and work. I want to thank Andrea LaPaugh and Jaswinder Pal Singh for being on my thesis committee. I am grateful to the faculty for making my "stay" so memorable; in particular David Dobkin, David Hanson, Eric Ristad, Robert Sedgewick, and Andy Yao. I also want to thank the department's system and office staff who put up with my countless requests. I am grateful to Melissa Lawson, Jim Roberts, and especially Patricia "patty" Zabielski.

Princeton is a wonderful place, but it's the people that make it special. I am blessed to be surrounded with with so many good-hearted people. I am indebted to Matthias "king-of-all-hardware" Blumrich, his wife Aimee "hiney" Cohen, and their daughter Sarah "tiny-hiney" for always being there for me. I also want to thank Chris "fabio/boo-boo/skinny-little-friend" Dunworth, Zicheng Liu and his wife Xue Li for their friendship.

Working on the SHRIMP Project has been a great experience largely because of the people involved. In particular, I want to thank "los" Angelos Bilas, Matthias "king-of-all-hardware" Blumrich, Yuqun "yoochie-coochie" Chen, Cezary "chuck" Dubnicki, and Rob "sir-hell-born" Shillner for their contributions to this dissertation. They are wonderful colleagues and good friends. I also want to the thank following SHRIMPers: Richard "link/boots" Alpert, Liviu "count-livula" Iftode, Sanjeev "scooby" Kumar, and Bin "no" Wei.

I want to thank *all* my fellow graduate students who have made my stay so much "fun", in particular I'm grateful to Pripal "aladdin/ymca" Ahuja, Matthias "hansa" Blume, Dimitrios "dg" Gunopoulos, Jeff "jasmine" Korn, Jim "hotdog" Plank, Mukund "mookie"

Raghavachari, Scott Rosenberg '95 (okay, Scott's not a grad student, but we were office-mates for a summer), and Bob "hoss" Thomas.

Of course I cannot forget an important source for many nicknames: the department sports teams. Our softball team, *The Cache Hitters*, won back-to-back B-league championships in 1995 and 1996. Quite an amazing accomplishment for a team without *the bat*. Our basketball team, *The Infinite Hoops*, was never quite as successful but just as much fun.

Now that the dream team has stopped reading, I can get to their nicknames. Ed got the name "big-hurt" by doing some serious damage to the softball! And everyone knows that Kai is "the-man" so it's no coincidence that he also goes by "don-kailione." Only Doug has successfully avoided getting a nickname, for now...

I need to thank George and Emily Roussos at GEORGE'S ROASTERS AND RIBS for their friendship and wonderful "fabio-approved" food.

I am grateful to Dr. Karl and Marion Przybilla of ULTRALIGHT AG in Liechtenstein for their friendship and generous support. They made my visits to the Principality memorable with fine dining and wonderful company.

I want to thank Dr. Glen Johnson at the IBM T.J. Watson Research Center who gave me the opportunity to work at a world class research lab. I learned a great deal from his approaches to both science and life.

I am grateful to the Natural Science and Engineering Research Council (NSERC) of Canada for awarding me the *1967 Scholarship*, Canada's most prestigious undergraduate academic award. This award helped fund my stay at Princeton.

Special thanks to my friend Peter "the-ferg" Giakoumatos for always being able to join me in finding ways to avoid work (Ackerman decay!). And of course, the occasional night out in NYC.

I cannot forget my spiritual father, Rev. George Saitanis at Evangelismos Tis Theotokou Greek Orthodox Church in Montreal. His guidance inspired me and his prayers gave me strength.

I want to thank *all* of my uncles, aunts, and cousins who live a short drive from Princeton. Unfortunately, I cannot list them all, but I do want to mention my cousin John P. Kalogridis who I have always been able to count on.

Also, I am extremely grateful to my Uncle Emmanuel ("sneek") and Aunt Anna Kalogridakis and my cousins Amalia ("countessa"), Asimina, and Stelios for welcoming me into their home and making me part of their lives.

My Uncle Antonis and Aunt Anna Kalogridis and my cousins Chrisoula ("chrisoulatos"), John ("ink"), Antonis Jr. ("potoko"), who made me feel part of the their family. Uncle Antonis is an incredible person; his kindness, motivation, and determination are awe inspiring. My Aunt Anna is a warm and generous person and an *amazing* cook!

I want to acknowledge the lasting impact of my late grandfather, Stefanos Damianakis. I spent the summers of my youth with him on the beautiful Greek island of Crete. My fond memories of him will be with me always.

I am blessed to have a loving and supportive family. My sister, Maria, is a caring, selfless, and giving person who I have always looked up to. She stressed the importance of education, and set a high standard for me to follow. My niece Eleftheria ("kokolina") and nephew Nicholas ("cookie") are a source of endless joy; they warm my heart and touch my soul. Their visits gave life to my sometimes-lonely NJ summers.

I am grateful to my brother John and his wife Anna. John will always be my best friend.

I dedicate this dissertation to my parents, Stylianos and Eleftheria, for their unconditional love and unwaivering support. They unselfishly sacrificed so much for their children.

Finally, I want to acknowledge my patron Saint Nektarios for the many miracles in my life, and the Lord Jesus for always helping me in times of need.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The client-server paradigm is useful for a large class of applications. Separating service providers from clients leads to clean and modular designs. Large scale client-server systems, and fault tolerant and high performance environments are based on connection-oriented communication. The challenge is to make the implementation of the communication model perform well for new-generation networks.

The traditional communication protocol stack was designed with the assumption of a slow network. Consequently, the overhead of protection boundary crossings, copying, and interrupts was relatively small. This assumption is no longer true since the performance of new-generation networks is at least an order of magnitude better.

Using TCP/IP—the traditional communication protocol stack—on a new-generation network results in a negligible performance improvement. This is because the overhead of crossing protection boundaries, copying, and interrupts has seen little improvement [2]. In order to realize the performance improvement of new-generation networks, communication mechanisms must avoid these sources of overhead.

As an example, consider two implementations of TCP/IP, one on 100 Mbit/s fast Ethernet and the other on 1.28 Gbit/s Myrinet [12], a new-generation network. Both use the traditional TCP/IP protocol stack for communication. For 64 KByte transfers the fast Ethernet network achieves a bandwidth of 10 MBytes/s while the gigabit Myrinet network achieves 13 MBytes/s. The latency results are similar: single word transfers take 175 μs using fast Ethernet and 259 μs using Myrinet. With TCP/IP, an order of magnitude performance improvement in network hardware is not translated to applications due to software overhead imposed by the traditional protocol.

1.1 Challenges for Efficient, Connection-Oriented Communication

Network protocols are traditionally implemented in the kernel in order to provide protected communication between processes; messages are sent and received by the kernel on behalf of user processes. The kernel approach works well for local and wide area networks but it incurs an unacceptable performance penalty in high performance new-generation networks. Involving the kernel results in poor performance for two reasons:

- *protection boundary crossings:* every send and receive crosses the kernel boundary for protection checking and message reception. The kernel checks to make sure the process is allowed to send or receive the message. These protection boundary crossings are relatively more expensive with high-performance networks because the cost of system calls and interrupt handler invocations does not scale well with increased processor performance [2, 53].
- *buffer copying:* messages are received in kernel memory and are then copied to user memory. Traditionally, the copy bandwidth was assumed to be much higher than the network bandwidth. However, the bandwidth of high-performance networks has improved to match the copy bandwidth. Therefore, copying every message once reduces the effective bandwidth by half.

Due to the poor performance of the kernel approach, many new communication models try to avoid crossing protection boundaries when sending and receiving messages. In other words, communication occurs at the *user-level* and the kernel is not involved. To accomplish this, new network interfaces place incoming data directly in user memory, bypassing the kernel [11, 12, 16, 34, 70]. This capability enables the construction of very efficient communication subsystems since the network interface can deliver packets without any software intervention.

These communication subsystems are typically based on custom-designed communication models, such as Virtual Memory Mapped Communication (VMMC) [31] and Active Messages [69]. Such custom models do not support connection-oriented communication Application Programming Interfaces (APIs) very well. Our work addresses this problem and proposes several solutions.

One challenge for user-level, connection-oriented communication is message detection by receivers. In traditional LAN-based communication protocols, every message generates an interrupt and invokes a handler that copies the message to a kernel-level buffer. Chapter 2 shows how the cost of the interrupt and the handler invocation alone can substantially increase the latency of short messages.

Another challenge with designing user-level, connection-oriented communication models is copy avoidance. In connection-oriented communication, processes communicate in pairs but they do not trust each other. As a result each process' memory address space is kept separate; all communication occurs through named connections. Therefore, connection-oriented communication is traditionally implemented with kernel-level copying. A major challenge for user-level connection-oriented communication is to maintain the untrusting nature of the connection while eliminating the overhead of copying.

1.2 Thesis Contributions

This dissertation describes solutions to the challenges for efficient connection-oriented communication. The first solution is a competitive spin-blocking mechanism for message detection that avoids unnecessary protection crossings by the receiver. The second solution is a user-level transfer redirection mechanism that gives the receiver dynamic control over the ultimate destination of an incoming message in order to avoid unnecessary copying in the protocol.

In the traditional, interrupt-based implementation of message reception, the kernel is responsible for (i) receiving incoming messages and (ii) informing the user process that messages arrived. Each message arrival is required to cross a protection boundary at least once. Our approach for reducing the overhead of message detection is to do most of the work at the user level in order to avoid as many protection boundary crossings as possible.

To detect that new data has arrived, a user-level library must either poll, or block and be awakened later by an interrupt. This requires both a policy for when to poll and when to block, and a mechanism for efficient blocking. Our idea is to use a competitive spin-blocking mechanism which spin-waits for a message before deciding to block. We consider the questions of which policy and which mechanism to use. We first describe the tradeoffs between polling and blocking, drawing on previous work on spinning versus blocking tradeoffs in multiprocessor locking. We discuss which mechanisms for blocking and unblocking are the most efficient.

The main difficulty in avoiding copying in connection-oriented communication is that the sender knows only the connection name and not the location of the receiver's buffer. Our approach is a transfer redirection mechanism that allows the receiver's network interface to redirect data transfer without CPU intervention. A connection-oriented communication protocol can be implemented at the user level with a default receive buffer from which data transfers are redirected when the destination buffer is known.

The timing between connection-oriented senders and receivers also plays a critical role. Let t_r be the time the receiver decides the destination of a message (i.e., the receiver reaches the appropriate place in the instruction stream). If a message is sent *before* t_r, its destination is not known and therefore it must be buffered. At some time after t_r it will be moved from the temporary buffer to the receiver-specified destination. As a result, either the sender or the receiver is forced to copy the message, depending on the implementation.

In this dissertation we address the issues of trust and timing in the context of message copying. We describe how to design and implement the low-level data transport protocol so that high-level connection-oriented protocols can be implemented with little overhead due to copying.

1.3 Implementation and Experiments

We demonstrate the two proposed techniques on two SHRIMP systems [7, 8, 11, 30] (Figure 1.1) at Princeton University. We implemented a stream sockets [47] user-level library using VMMC in order to evaluate the performance of our connection-oriented communication mechanisms. 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. A wide variety of distributed applications rely on the stream sockets model for interprocess communication.

We designed, implemented, and evaluated the competitive spin-blocking technique for user-level message detection on a SHRIMP system called Shrimp-C [7, 11, 34]. The Shrimp-C system uses a custom-built network interface to connect sixteen 60 MHz Pentium PCs. The network is a two-dimensional mesh routing backplane from an Intel Paragon multicomputer [40]. The mechanism itself is implemented in the user-level stream sockets library (Figure 1.1).

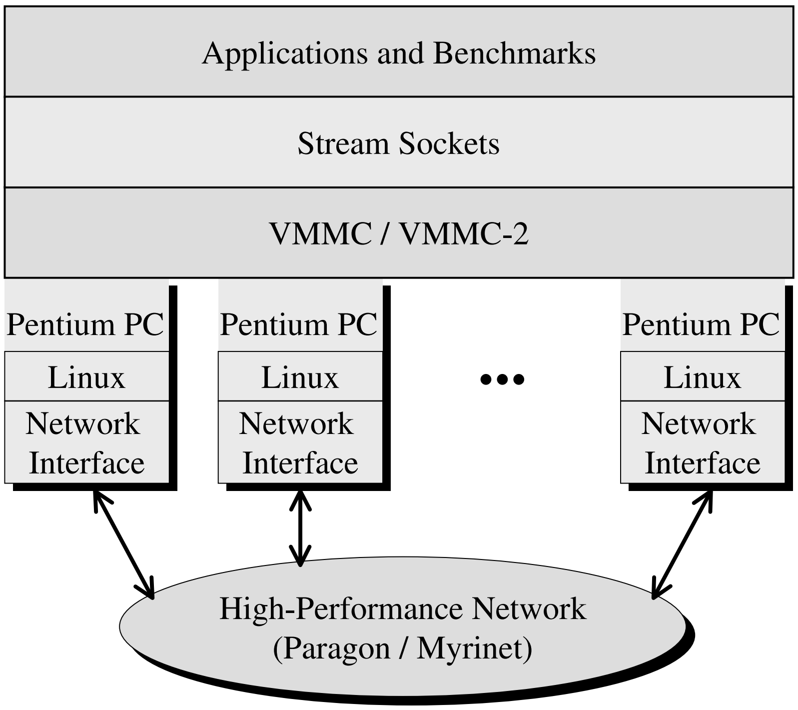


Figure 1.1: SHRIMP System overview

We evaluated the spin-blocking mechanism, along with several other strategies, for various workloads in a multi-programming environment. Our experiments show that a combination of polling and blocking gives consistently good performance in a wide variety of situations. Further, reducing the interrupt service time in addition to the number of unnecessary interrupts results in significant performance improvement.

We designed, implemented, and evaluated the copy avoidance technique based on a transfer redirection mechanism on a SHRIMP system called Shrimp-M [30] (Figure 1.1). Shrimp-M uses Myrinet programmable network interfaces and switches [12] to connect sixteen 166 MHz Pentium PCs. We extended the VMMC communication model to provide support for copy avoidance in connection-oriented communication and called it VMMC-2. We programmed the network interface firmware to implement most of the functionality of our custom network interface as well as the VMMC-2 extensions. We did not use the

Shrimp-C system for this work because modifying the custom network interface would require a significant engineering undertaking.

To evaluate the performance of transfer redirection, we experimented with user-level stream sockets, as described in Chapter 3. As an example of the results, the performance of a bidirectional ping-pong benchmark sending 128 KByte messages improved from 34 MBytes/s without redirection to 84 MBytes/s with redirection. We obtained almost identical results for the standard ttcp benchmark [67].

In summary, we were successful in designing a connection-oriented communication model that addresses two outstanding problems in user-level communication: efficient message detection and copy avoidance. Our implementation and experiments show that both mechanisms are effective.

We have published some of the material in Chapter 2 [24, 25, 26]. The transfer redirection mechanism presented in Chapter 3 was designed together with Kai Li, Angelos Bilas, Yuqun Chen, and Cezary Dubnicki, who was responsible for the implementation[29].

1.4 Related Work

This dissertation deals with mechanisms for improving the performance of connection-oriented communication. Specifically, mechanisms for preserving the security and anonymity of the connection-oriented model while taking advantage of user-level communication are developed and studied. Therefore, the related work can be drawn from many sources, both strongly related and barely related to connection-oriented communication.

User-level communication has been extensively studied, and several practical implementations of user-level data transport layers exist. The work described in this dissertation grows directly out of the SHRIMP project [11] at Princeton University. Closely related efforts with similar goals are Active Messages [69], Fast Messages [54], Hamlyn [70], and

Flash [45]. Though none of these related projects has provided adequate solutions to the problems discussed in this dissertation, our solution strategies should work for all of them.

Spector [64] proposed an early remote memory reference model to perform communication over a local area network and his implementation is programmed in a processor's microcode. This model has been revived by Thekkath et al. [65] using fast traps. They did not deal with the issues of supporting connection-oriented high-level protocols.

Efficient user-level communication is often supported by network interface hardware, as is the case for SHRIMP [7]. Similar efforts include Galactica Net [71], Hamlyn [16], Flash [45], and Telegraphos [52].

Chapter 2 identifies receive-side interrupts to be a major source of communication overhead, and we discuss several mechanisms to reduce the number of interrupts and make those that are necessary cheaper. These ideas are partially derived from the work of Karlin et al [44, 43], who studied the effects of the spinning versus blocking tradeoff on multiprocessor performance in the context of locks. A related technique is the *polling watchdog* [51], which tries to minimize the number of interrupts by delaying them so that they will accumulate. The spin-block co-scheduling side-effect is further investigated with simulations of "bulk-synchronous" SPMD applications [32]. Similar work uses *dynamic co-scheduling* [63] as part of a working system that explicitly treats all message arrivals as scheduling requests.

There are several experimental systems that attempt to improve the performance of TCP/IP and/or sockets.

Druschel et al. [27] proposed the concept of application device channels which provide protected user-level access to a network interface. U-Net [4] uses a similar abstraction to support high-level protocols such as TCP/IP. Their implementation requires applications to use special, pinned buffers. Hamlyn uses a sender-based communication model and uses a network interface to support user-level message passing [15], but it requires application programs to build headers using pinned memory. They did not address the issues of supporting connection-oriented high-level protocols.

The Nectar system [19] implements TCP on a programmable network interface, with a 16Mhz Sparc as the communication processor. S-Connect implements a retransmission protocol in hardware to provide reliable communication. They obtain 90% of the hardware bandwidth between two network interfaces, but only 30% between between user applications. The significant performance reduction is due to protocol software overhead.

Active Messages(AM) [33] is a user-level communication mechanism based on fast remote execution facility. Recently, a fast sockets facility has been built with AM handlers [59]. They used a receiver-based approach called *receive posting*. The arriving data is first placed in an anonymous staging area in memory. Posting a receive will move the data to the final destination. FastSockets achieves 30 μs one-way latency and 33 MBytes/s maximum bandwidth between two UltraSPARC 1 connected by a Myrinet network. Our transfer redirection is different: our default buffer is used only when the receive address is not posted in time.

Many methods have been proposed to avoid copying between I/O devices and application memory. IBM's OS/360 can specify a file I/O buffer area for applications to avoid copying. Page remapping has been used to avoid copying in several past and recent projects [46, 18, 42, 28, 49]. Lately, several groups have proposed letting applications directly access communication interface memory [19, 23, 62].

Chapter 2

User-Level Message Detection

2.1 Introduction

Traditional message passing systems involve the operating system (i.e., the kernel) in communications. Messages are sent and received by the kernel on behalf of user processes. Kernel message buffering has several advantages:

- The kernel guarantees that messages and processes are protected. In a multiprogramming environment, the kernel protects messages from other eavesdropping processes, and does not allow processes to send unauthorized messages.
- User memory is swappable and may be unavailable for message transfers. Kernel buffers are pinned in physical memory.
- Network interfaces use physical memory addresses. User programs are only aware of virtual memory. The kernel deals with both virtual and physical address and can act as the intermediary. For example, a user process sees virtual memory as a linear address space so a single user-buffer occupies a contiguous virtual memory region. But, the corresponding physical memory pages are not allocated consecutively.

The kernel approach works well for most networks but these benefits come at a cost. They incur an unacceptable performance penalty in high performance (gigabit) system area networks for two reasons:

- *buffer copying:* messages are sent to kernel memory, they are then copied to user memory. Copying every message reduces the effective bandwidth to the memory copy bandwidth.
- *protection boundary crossing:* every send and receive crosses the kernel boundary. The kernels on the sender and receiver perform the message transfer as shown in Figure 2.1. Further details on the causes for the poor performance are covered in [53, 2].

Figure 2.1 illustrates the steps required to send and receive a large message with kernel involvement on a Paragon multicomputer [40]. The tasks are divided between three distinct units: (i) *send system call*, (ii) *receive device driver*, and (iii) *receive system call*. This figure assumes that the send system call does not need to copy in order to align transfers; it DMAs data directly from user memory. A copy is used by the receive system call due to kernel buffering. In this example, we have a very fast network with a hardware message transfer time of 1 μs. Yet, the total transfer time, i.e. from user memory on the sender to user-memory receiver, is 50 μs. Software overhead accounts for 98% of the transfer time.

As a result of kernel limitations, many new communication models try to eliminate kernel involvement from the sending and receiving of messages. In other words, communication occurs completely at the *user-level.* To accomplish this, new network interfaces place incoming data directly in user memory, bypassing the kernel [11, 12, 16, 34, 70]. This capability enables the construction of very efficient network software since the network interface can deliver packets without any software intervention. On such architectures, communication can be handled entirely in user-level libraries. The network software is typically based on custom-designed communication models, as we have done with VMMC.

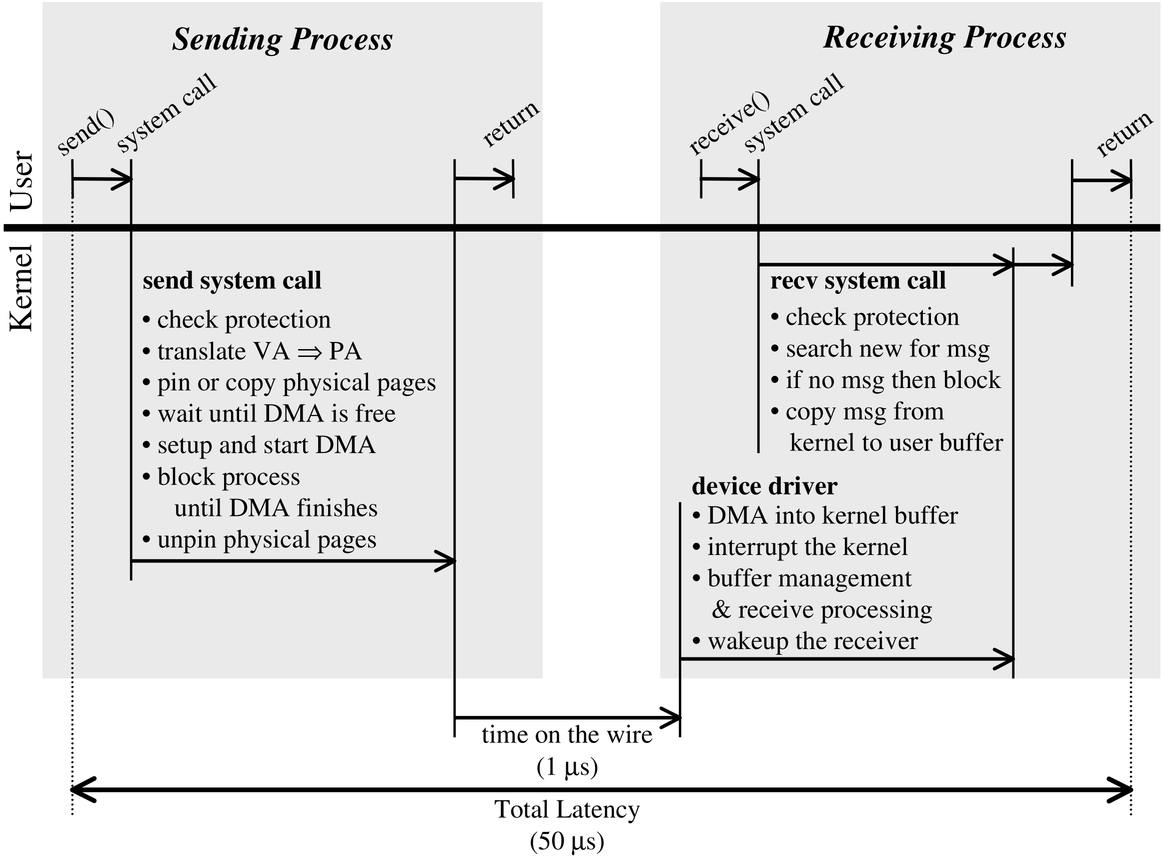


Figure 2.1: Message passing latency for the Paragon kernel-based implementation

These custom models do not support connection-oriented communication APIs very well. Our work addresses this problem and proposes several solutions.

But by removing the kernel we have introduced new issues that need be addressed. The kernel was responsible for (i) receiving incoming messages and (ii) informing the user process that messages arrived. Therefore we would like a user-level mechanism that satisfies two requirements:

- *no copying*: incoming data is placed directly in user memory with no software overhead

- *fast message detection*: a user process can quickly detect that a new message has arrived

The traditional solution is to interrupt the kernel and have it deal with each incoming message. This approach defeats the purpose of user-level communication since each message requires the kernel to service an interrupt and schedule user code to process the message. The impact of interrupts is more pronounced with new-generation networks and PCs.

2.1.1 Message Flags

For a receiver to detect the arrival of messages at the user-level, without the kernel (interrupts), the sender must *flag* messages. That is, every message is sent using either:

- *implicit flags*: the last word of the message indicates the arrival of the message
- *explicit flags*: a second, separate, message is sent to indicate the arrival of a previous message. A pair of *counters* can be used as a shared explicit flag.[1]

Without using the kernel, a process must spin-wait (i.e. poll) on either type of flag to determine the arrival of a new message.

To detect that new data has arrived, a user-level library must either poll, or block and be awakened later by an interrupt. This requires both a policy for when to poll and when to block, and a mechanism for efficient blocking [25]. In this chapter we consider the questions of which policy and which mechanism to use. We first describe the tradeoffs between polling and blocking, drawing on previous work on spinning versus blocking tradeoffs in multiprocessor locking [43, 44]. We discuss which mechanisms for blocking and unblocking are the most efficient.

[1] After sending a data message, the sender increments a counter and explicitly sends it to the receiver. The receiver compares the sender's counter with a local counter that is updated for each message received. If the two counters are not equal then there is at least one new message.

We present an implementation on the prototype Shrimp-C multicomputer [11, 34]. We also describe the implementation of our user-level stream sockets library, which is used for experiments. The experiments include micro-benchmarks, larger benchmarks, and a distributed file system. Our results show that a hybrid spin/block policy is best in a wide range of situations, and that reducing the interrupt service overhead significantly improves performance.

2.2 Architectural Assumptions

This section discusses the general architectural assumptions on which this work is based, followed by details of the SHRIMP architecture on which a specific implementation was tested.

Our approach makes the following two assumptions about the communication architecture:

- the network interface (NI) delivers incoming data directly to user memory in the order it was sent. No explicit receive is required by user or kernel code. This is referred to as a *direct data transfer*.
- the receiving node can dynamically enable and disable receive interrupts at page granularity. In other words, incoming messages may or may not interrupt the receiver, and the receiver has fine-grain control over when interrupts can occur.

These assumptions imply that user-level communication software on the receiver can be given responsibility for detecting the arrival of new data. It can achieve this by polling or, with appropriate kernel support, by blocking the process and activating interrupts for specific incoming data.

We do not make assumptions about the cost of sending data, since it is relatively independent of which policies we use. The sender's job is easier than the receiver's because the sender knows when a message will be sent, but the receiver does not know when a message will arrive.

2.3 Strategy

Our design improves performance in two ways: by reducing the cost of interrupt service, and by taking interrupts only when it is helpful to do so. This section describes our approaches to these goals.

2.3.1 Reducing Interrupt Service Time

To make interrupt service faster, we have to reduce the amount of work done as a result of the interrupt. Although many systems execute a user-level handler when messages arrive, doing this adds overhead that is not necessary for building connection-oriented, user-level communication libraries. In early implementations of our system, we found ourselves installing null handlers and using the notification merely to awaken a sleeping process.

All that is really needed is a way to reawaken a blocked process when a suitable message arrives. If the process blocks waiting for message arrival, then it will certainly look for the message immediately upon reawakening. For connection-oriented communications, there is no reason to disrupt the flow of control by executing a user-level handler.

Generally, for a network interface device driver interrupt handler to invoke a user-level handler (instead of simply setting the receive process's state to runnable), the system has to perform the following additional work:

- the device driver must send a signal (or similar lightweight notification) to the sleeping user process;
- the user process must process the notification:
  - save the process context on the stack;

- set up a stack frame for the handler;
- jump to the handler;
- restore the process state;

This amount of work is significant, and typically takes 45 - 150 microseconds [65]. Since we do not need the functionality of a user-level handler, we can eliminate it and reduce the software overhead. This allows us to improve performance while maintaining the functionality we need.

Our new mechanism adds a system call that blocks the calling process until specific data arrives. The user-level process specifies which memory locations it is watching, and the device driver interrupt handler wakes up the process when one of those locations is modified by an incoming message. An implementation of this mechanism is presented later in this chapter.

With a programmable network interface we can further reduce interrupt overhead by filtering interrupts on the network interface instead of the device driver. We can push some of the device driver functionality to the network interface and enable it to decide if an interrupt is needed.

2.3.2 Controlling the Number of Interrupts

Our second goal is to carefully control when interrupts are taken. It may seem, at first, that we should try to minimize the number of interrupts, but this is not quite correct. Certainly, we want to avoid taking an interrupt when there is no reason to do so, but there are cases in which interrupts are helpful. For example, if a process is waiting for a message and has nothing useful to do until the message arrives, and we know that the message will not arrive soon, then it is best for the process to block and wait for an interrupt to reawaken it. This gives other processes a chance to run.

Our ultimate goal is to minimize the amount of CPU time that a process uses to detect the arrival of messages. This includes time spent in the kernel for system calls, context switching, and interrupt processing. We try to achieve the optimal number of interrupts, because fewer interrupts is not necessarily better. For example, always spinning (no interrupts) is clearly not the right approach because even though time spent in the kernel is zero, the spin time becomes large. Likewise, not spinning uses no user time but requires significant operating system time for every message transfer (system call, context switching, and interrupts).

The right choice between spinning and blocking depends on the *arrival delay*: the length of time before the message will arrive. The cost of spinning is equal to the arrival delay, while the cost of blocking is a constant T_{block}, independent of arrival delay:

$$T_{block} = 2 \cdot T_{cs} + T_{intr} + T_{setup} \tag{2.1}$$

where T_{cs} is the context switch time, T_{intr} is the kernel interrupt service time, and T_{setup} is the overhead for making the system call and setting up the network interface to interrupt the kernel on the arrival on incoming data. What makes this problem difficult is that we do not know in advance how long it will be before a message arrives. So we use T_{block} to determine the amount of time we spin at user-level before blocking the process.

For the *optimal off-line* policy, the process always knows the arrival delay so it will spin if the arrival delay is less than T_{block}, and block otherwise. In reality there is no way to predict the arrival delay, so the optimal off-line policy is not realizable, though it does provide an upper bound on the performance of practical policies.

The two simplest policies are spin and block. Each is much worse than optimal in some cases. If the arrival delay is very long, spin uses a lot of CPU time spinning when blocking would have been much better. If the arrival delay is very short, block incurs a significant constant overhead when spinning would have been much better.

A simple hybrid spin-block policy works fairly well: spin for time T_{block} and then block if the message has not shown up yet. This policy does well in the two limiting cases of very small and very large arrival delay. Karlin et. al. [43, 44] show by a simple argument that this policy is 2-competitive: its cost is never more than twice that of the optimal off-line policy regardless of the sequence of arrival delays.

Briefly, the 2-competitive argument is as follows. Let T_{block} be the time the spin-block policy delays and T_a the time the message arrives relative to the start of the receive call (i.e. T_a can be positive or negative). The optimal policy "knows" the value of T_a, so it will spin if $T_a \leq T_{block}$ and block if $T_a > T_{block}$. On the other hand, the spin-block policy will always spin for T_{block} before blocking. When messages arrive at $t \leq T_{block}$ both policies perform equally well. But, when messages arrive at $t > T_{block}$ the optimal policy always blocks immediately and incurs overhead of T_{block} for context switching. The spin-block policies unnecessarily spins for T_{block} before blocking and thus incurs overhead of T_{block} for spinning and another T_{block} for context switching for a total of $2 \cdot T_{block}$. Therefore spin-block is no worse than two times the optimal algorithm.

2.3.3 Message Detection Strategies

We identified seven strategies that processes can use when waiting for messages:

- *spin*: sit in a loop polling for message arrival
- *sched*: sit in a polling loop, but yield the processor after each iteration of the loop
- *block*: block the process and use interrupts to reawaken it
- *notify*: block the process and use a user-level handler to reawaken it
- *spin-sched*: spin for a while and then switch to the sched policy
- *spin-block*: spin for a while and then switch to the block policy
- *spin-notify*: spin for a while and then switch to the notify policy

Strategy	User Spin	Interrupts	System Calls	Kernel Reschedules per System Call
Spin	✔		none	–
Block		✔	always	1
Spin-Block	✔	✔	sometimes	1
Notify		✔	always	1
Spin-Notify	✔	✔	sometimes	1
Sched			always	≥ 6
Spin-Sched	✔		sometimes	≥ 3

Table 2.1: Summary of the differences between the various message detection strategies

In addition to studying the tradeoffs between spinning and blocking, these strategies enable us to analyze the performance of three different mechanisms that detect data arrival: fast-interrupts, user-level notifications, and the kernel scheduler. Table 2.1 illustrates the main differences among the strategies.

2.4 Message Detection Implementation

This section describes how the seven strategies were implemented as well as the hardware and software platforms that were used.

There are three software components: (i) the VMMC communication layer (Section 2.4.2), (ii) our implementation of user-level stream sockets, and (iii) benchmarks and applications. Stream sockets are described later in this section while benchmarks and applications are presented in the next section.

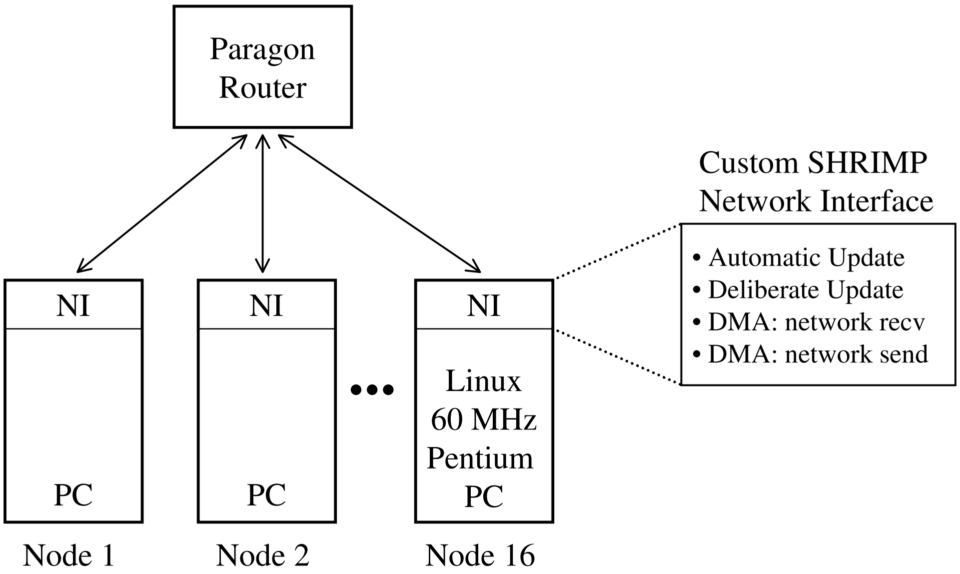


Figure 2.2: Shrimp-C Architecture

2.4.1 The Shrimp-C Platform

The strategies were implemented on the Shrimp-C multicomputer [9, 11, 34]. The Shrimp-C hardware (Figure 2.2) consists of a network of sixteen nodes running the Linux operating system. Each node is a DEC 560ST PC with a 60 MHz Pentium CPU [60]. The nodes are connected to a multicomputer routing network [66] via custom-designed network interfaces [7, 11]. The network itself is an Intel routing backplane consisting of a two-dimensional mesh of Intel Mesh Routing Chips (iMRCs) [66], and is the same network used in the Paragon multicomputer [40]. The backplane supports deadlock-free, oblivious worm-hole routing [22], and preserves the order of messages from each sender to each receiver.

The receive-side network interface transfers data from network packets directly to user memory. If conditions are satisfied, the network interface can interrupt the processor either before or after the data has been deposited. Two conditions must be satisfied to trigger a receive-side interrupt:

- per message interrupt bit is set: the message was sent with an interrupt bit set.
- per page interrupt bit is set: exported page receiving the message has an interrupt bit set. Either, or both, bits that specify when to generate an interrupt can be set: (i) before the message is delivered, (ii) after the message is delivered.[2]

Therefore both the sender and receiver must agree in order to trigger an interrupt.

The receive-side network interface transfers data from network packets directly to user memory. In addition, if the sender has requested (and the receiver agrees), the network interface interrupts the processor either before or after the data has been deposited. Further, the receiver can control the interrupts by enabling or disabling them for any page that has been mapped to receive incoming data.

2.4.2 The VMMC Model

VMMC is a model that allows an application process to directly access the memory of another process running on any node in the system, while respecting protection domains. Depending on the implementation, memory can be accessed directly or through user-level transfer operations. In either case, transfer is between virtual addresses and does not involve the operating system.

Protection

The basic VMMC model supports direct data transfer to *receive buffers*, which are variable-sized regions of contiguous virtual memory. In order to make the memory of a receive buffer available to other processes, the owning process *exports* the buffer together with a set of permissions. Any other process with the proper permission can *import* the receive buffer and access it directly. Note that any region of virtual memory can be exported as a receive buffer, including dynamically allocated memory.

[2]Setting both results in two interrupts per message

After a successful import, a local representation of the remote receive buffer—called a *proxy receive buffer*—is created. The proxy receive buffer is the same size and alignment as the actual receive buffer, but exists in the virtual address space of the importing process. Unlike the actual receive buffer, the local proxy representation is not backed by any real, physical memory.

In the VMMC model, data transfers are performed between a processes' local virtual memory and imported receive buffers. The model does not define the actual operations for moving the data, only that those operations should transfer the data directly, without copying. Currently, there are two such operations supported by various implementations of the VMMC model: deliberate update and automatic update.

Transfer Strategies

Deliberate update is an explicit operation that requests a transfer of a contiguous block of data from local memory to a remote receive buffer. To initiate the transfer, a process specifies a virtual address in its memory, a virtual address within a proxy receive buffer, and a transfer size. This causes the communication subsystem to transfer a block of the specified size starting at the specified memory address to the remote receive buffer represented by the specified proxy address. The block can be transferred to any offset within the receive buffer, but is not allowed to exceed the bounds of the buffer.

Deliberate update illustrates some fundamental advantages of the VMMC model. The export-import mechanism enforces protection and insures that there is a destination for direct data transfer. Therefore, the transfer itself can be performed entirely at user level, without operating system involvement at the sender or receiver. The basic idea is to "factor out" the overhead of protection checking and buffer management from the critical path of communication.

Automatic update is an implicit data transfer operation that occurs as a side-effect of a local memory write. To use automatic update, a block of virtual memory is *bound* to an imported receive buffer (or portion thereof), forming a direct connection between the local and remote memory. Any write to the local, bound memory causes an update of the remote memory with the same data, at the same offset. No explicit data transfer operation is required.

Automatic update has the advantage of extremely low latency because the remote update is generated at the same time as the local data is written. The implementation disadvantage of message volume can be mitigated by combining updates into large, aggregate transfers.

Notifications

Although the VMMC model specifies direct data transfer, control transfer is often also desired for notification or synchronization. To support this, the model allows a receive buffer to be associated with a user-level handler such that any data transfer to the buffer causes the handler to be invoked. This control transfer is called a *notification*, and is similar in semantics to a UNIX signal. The model provides no guarantee as to when the notification is delivered to the handler, and no guarantee that the data is not over-written by the time the handler is invoked. However, the notifications are guaranteed to all be delivered, and in the order they were generated.

The VMMC model supports low-overhead communication because data transfers are performed from user-level, do not involve copying, and do not require a control transfer. For example, automatic update has no software overhead whatsoever, and deliberate update has been implemented with only two user-level memory references as software overhead [10].

The basic VMMC model is intended to be sufficiently general to support all kinds of communication mechanisms, while encouraging efficient implementations. It can be used to implement message passing as well as shared memory.

Because communication is performed at the application level, various models—existing as well as novel—can be implemented in user-level libraries. This gives developers the freedom to choose the kind of support provided by the communication subsystem.

2.4.3 VMMC Stream Sockets

A wide variety of distributed applications rely on the Berkeley stream sockets model [47] for interprocess communication. The stream sockets interface provides a connection-oriented, bi-directional, byte-stream abstraction with well-defined mechanisms for creating and destroying connections and for detecting errors.

In this section we describe an implementation of the sockets interface using the VMMC communication model. Our implementation is compatible with the stream sockets interface; it properly detects broken connections and correctly implements the select call.

The implementation we describe 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 [50].

The user-level sockets library allocates a socket descriptor table that contains one entry for each open socket, regardless of the socket type. When non-VMMC 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 VMMC connection. Figure 2.3 illustrates the software layers in our user-level implementation.

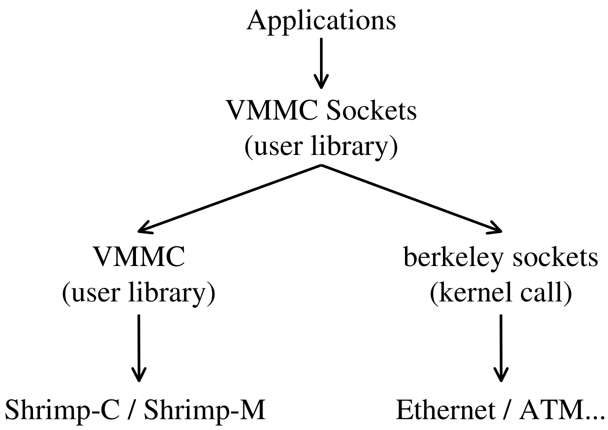


Figure 2.3: Socket layers

Circular Byte Streams

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 VMMC. Then we show how to use this simple structure as the building block for the stream sockets library.

The byte stream is managed by a circular buffer that consists of a default buffer and the head and tail indices. In order to implement a simple unidirectional byte stream the receiver allocates and exports memory for a *default buffer* and a *tail index*. The sender imports them in proxy space (not backed by memory). Similarly, the sender allocates memory for the *head index* which the receiver imports (Figure 2.4).

Figures 2.5 and 2.6 illustrate how the simple queue structure can be used for one-way byte transfers. The sender writes to the default buffer and the tail index and only reads the head index, while the receiver reads only from the default buffer and tail index and writes the head index. Since writes to the index variables are always exclusive there are no race conditions.

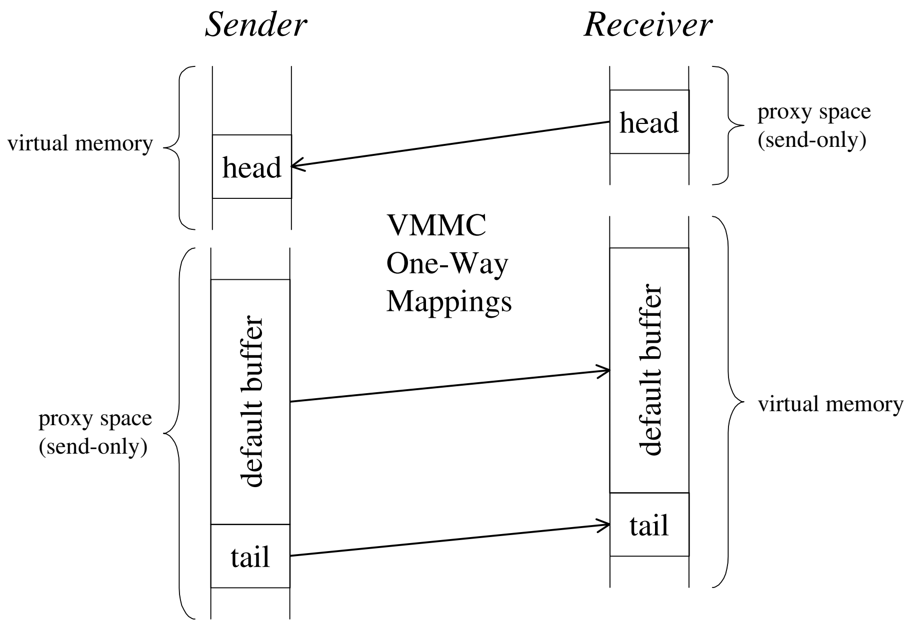


Figure 2.4: VMMC mappings for a unidirectional byte stream

In order to construct a bi-directional byte stream we use two unidirectional byte streams: one for sending bytes and one for receiving bytes. The basic underlying implementation is simple, straightforward, and works very well with the VMMC communication mechanism.

Sending and Aligning Bytes

Socket communications operate at the byte level; all sends and receives specify byte counts. There is no guaranteed relationship between the byte alignment of the send and receive buffers.

Neither the Shrimp-C nor the Shrimp-M network interfaces support byte transfers; their minimum transfer unit is one word. Further, all transfers must be word aligned and a integer number of words in length. These network interface characteristics result in a mismatch with the byte-oriented nature of stream sockets. Figure 2.7 illustrates the set of problems

```
simple_send (message)
{
    wait until buffer space is available
    transfer message to receiver's default buffer
    update tail pointer
    transfer tail pointer to receiver
}
```

Figure 2.5: Pseudo code for simple send

```
simple_recv (final buffer)
{
    wait until data is available
    copy data from default buffer to final buffer
    update head pointer
    transfer head pointer to sender
}
```

Figure 2.6: Pseudo code for simple receive

that arise. In the worst case, both the source and destination buffers are not word aligned (by different offsets). In addition, the message can be an odd length and not end on a word boundary.

To address this mismatch we add an *alignment queue* to the unidirectional byte stream. The alignment queue, like the unidirectional byte stream, is a circular queue that is used to send unaligned bytes. Each entry in the queue contains at most three bytes of message data, a byte count, and a placement index. The placement index positions the alignment bytes relative to the data in the unidirectional byte stream (Figure 2.8).

When the sender detects that the user data is not word aligned, it sends an alignment queue entry containing up to three bytes (to get to the next word boundary). It then uses the unidirectional byte stream to send an integral number of words. Finally, if the user data

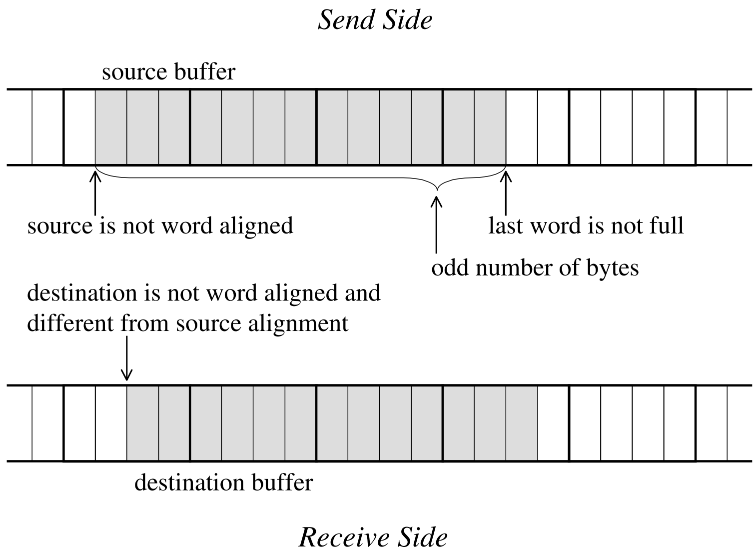


Figure 2.7: The byte alignment problem

does not end on a word boundary, it will need to send another alignment queue entry to complete the message transfer (Figure 2.9).

The receiver must first check the alignment queue to determine if it should reconstruct an unaligned message. In the common case, when aligned data is sent, the overhead imposed by the alignment queue is one user-level comparison.

Send

send does not copy except for unaligned data (at most six bytes). Transfers are initiated directly from the user-specified buffer. Deliberate update requires an explicit user-level VMMC call to initiate data transfers. Two data transfers are required if the circular buffer wraps around. When using Shrimp-C's automatic update transfer mode, the user buffer must be copied into the mapped circular buffer that automatically triggers the transfer. Once the buffer is sent, the sender transfers a one-word message to update the receiver's tail pointer. The pseudo-code for send is shown in Figure 2.10.

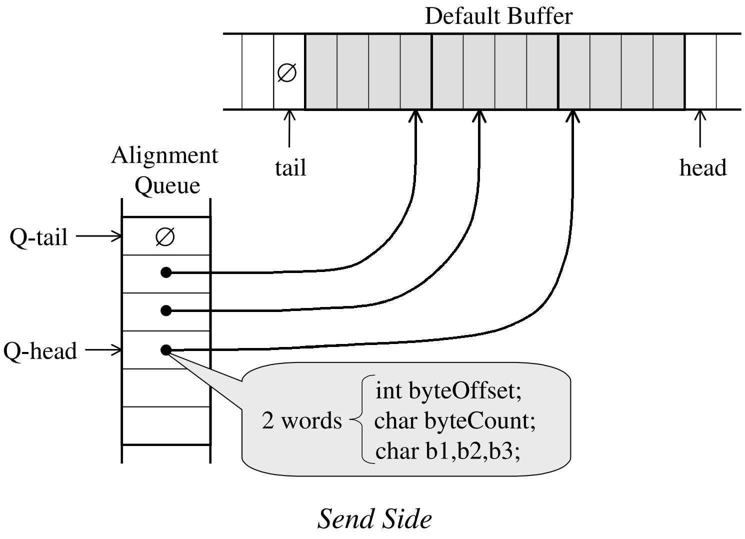


Figure 2.8: Unidirectional byte stream with an alignment queue

receive

receive must copy data from the incoming circular buffer and alignment queue to the user-specified buffer. Once it consumes the data, it updates the head pointer on the sending node. Each call costs one copy (or two, if the data wraps around the circular buffer) and a one-word message to update the head pointer. The pseudo code for receive is shown in Figure 2.11. Note that the alignment queue as made receive significantly more complicated.

select

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

When a VMMC 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

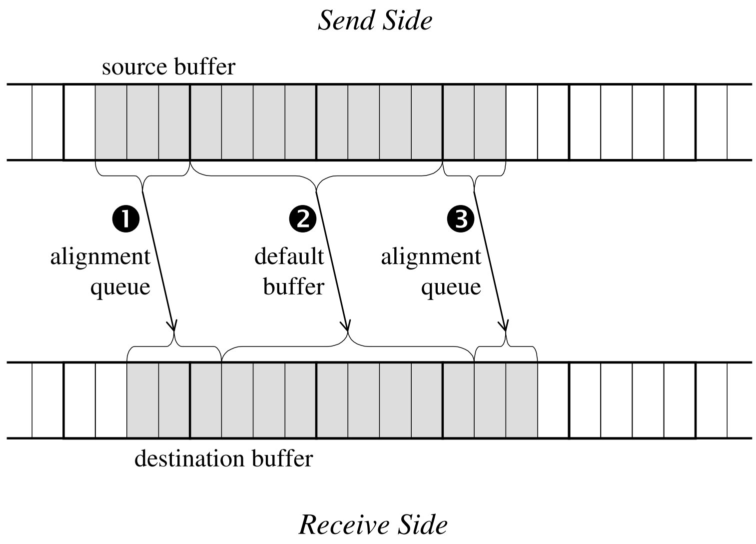


Figure 2.9: The byte alignment solution using an alignment queue

- *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, all of the socket library's buffers initially *ignore* notifications. select must first accumulate notifications for the sockets in question.

The algorithm for user-level select uses the kernel version of select, which is denoted 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 user-level select.

```
send (buffer)
{
    check that socket is valid and alive
    if the outgoing circular buffer is full
        wait until space becomes available
    if buffer does not start on word boundary
        transfer alignment bytes via the alignment queue
    transfer integer number of words remaining buffer
    if buffer does not end on word boundary
        transfer alignment bytes via the alignment queue
    update and transfer the outgoing circular buffer tail index
    return the number of bytes transfered
}
```

Figure 2.10: Pseudo-code for send()

Connection Setup and Shutdown

During connection establishment, VMMC uses a regular TCP socket connection to exchange the data required to establish two 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 VMMC 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 using an internal routine, Setup_VMMC, which does the following:

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

```
recv (buffer, nbytes)
{
    check that socket is valid and alive
    if both incoming circular buffer AND incoming alignment queue are empty
        wait for a message to arrive
    avail = total bytes available in circular buffer and alignment queue
    left = min (avail, nbytes)
    Qoffset = offset of the first alignment queue entry
    while (left != 0)
    {
        if alignment queue bytes are the next in sequence
        {
            count = number of bytes in queue entry
            B = min (count, left)
            copy B bytes from alignment queue to user buffer
            if top queue entry is empty {
                remove top alignment queue entry
                transfer queue update to remote node
            } else {
                update top alignment queue entry
            }
        }
        else if the circular buffer bytes are next in sequenece
        {
            count = number of bytes in circular buffer until next queue entry
            B = min (count, left)
            copy B bytes from circular buffer to user buffer
        }
        left -= B
    }
    update and transfer the incoming circular buffer head index
    return number of bytes received
}
```

Figure 2.11: Pseudo-code for recv()

```
select (message)
{
    deliver notifications: turn on notifications for VMMC socket data
    check for new data: check socket data structures for new messages
    loop
    {
        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
             */
        check for new data: check socket data structures for new messages
    } until data has arrived
    ignore notifications: return notifications to the fast common case
}
```

Figure 2.12: Pseudo-code for select()

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.

User-Level Threads

We developed a version of VMMC sockets that supports user-level threads. This required a change in the semantics of blocking function calls (send, receive, accept, select) so that they only block active threads. Threads allow our sockets applications to overlap communication and computation more effectively.

```
Setup_VMMC ()
{
    pack node and process ID into ID structure
    allocate buffers for incoming and outgoing data
    export the receive buffer
    if (client process) {
        TCP-send the node's local ID structure
        TCP-receive the remote node's ID structure
        import the receive buffer (from the server)
    }
    else /* server process */
    {
        TCP-receive the remote node's ID structure
        TCP-send the node's local ID structure
        import the receive buffer (from the client)
    }
}
```

Figure 2.13: Pseudo-code for Setup_VMMC()

Deliberate Update vs. Automatic Update

We implemented three versions of VMMC sockets for Shrimp-C: *AU-1copy* uses automatic update transfers, *DU-1copy* uses deliberate update transfers, and *hybrid-1copy* uses automatic update for flags and deliberate update for data. AU-1copy is clearly faster for small messages because there is no software overhead to initiate transfers. DU-1copy is faster for large transfers because of DMA. Hybrid-1copy's performance is identical to DU-1copy; it offers no advantage.

There is one variation we did not implement: automatic update for messages smaller than 512 bytes and deliberate update otherwise. This should produce the best performance for all message sizes.

The choice of AU or DU is also influenced by Shrimp-C's hardware characteristics. When the send message data is in the cache, automatic update performs better; when the data is in memory, deliberate update is better [7]. This choice is application specific. With extensions to support hints from applications, sockets can dynamically select between AU and DU.

Caching Effects

Our experience with Shrimp-C shows that communication performance is affected by the caching strategy used for send and receive buffer memory. Communication performance is better with write-through caching [6, 7]. For DU, if part of the send buffer is modified in the cache, DMA will stall so that the cache line can be written to memory. A similar situation occurs when DMAing the incoming data on the receive side. Using a write-through cache eliminates this performance penalty.

We do not want to change the caching strategy for the application's data because it may degrade the application's performance. However, we can safely change the caching mode to write-through for pages that consist entirely of communication-library data buffers.

2.4.4 Strategy Implementation

Next we describe the implementation of the message detection strategies described in Section 2.3.3 on page 18.

Some of the strategies are easy to implement. For example spin is a simple polling loop that waits for data to arrive. Implementations of the other strategies are either straightforward, or closely related to the implementation of spin-block. We focus here on the spin-block strategy.

spin-block is implemented using the following three components:

- *user-level function*: called by the user-level communication library; it spins as required by the spin/blocking strategy, and then makes a system call to block.
- *kernel system call*: if data has not arrived, this system call enables interrupts for the specified pages and then blocks the process until a suitable message has arrived.

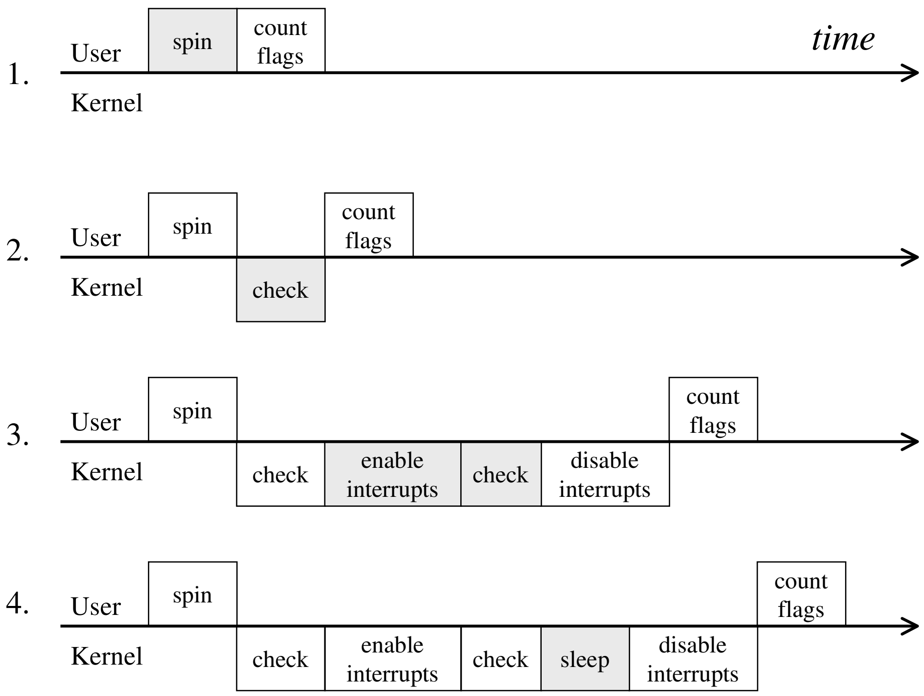


Figure 2.14: The spin-blocking implementation at work. The shading illustrates the four possible data arrival points.

- *device driver interrupt handler*: we added a hook to the standard SHRIMP interrupt handler to check whether data has arrived for a page that a process is waiting on. If so, the interrupt handler clears the interrupt enable bit for all of the pages the process is waiting on, and wakes up the process. Otherwise, it proceeds to service the interrupt normally.

Figure 2.14 illustrates how these three pieces work together by showing what happens when the data arrives at various times. The four possibilities are:

1. data arrives while the user-level library is spin-waiting.

2. data arrives during entry to the kernel; a kernel-level poll detects it and returns control to the user library.

3. data arrives while interrupts are being enabled; a kernel-level check detects and returns control to the user library.

4. data arrives while the process is asleep; an interrupt handler reawakens the process and returns control to the user library.

New messages are detected by comparing explicit flags (counters).[3] To detect the number of sources (i.e. sockets) that sent messages, we count the number of flags that have changed.

User-level Library

All seven of the policies are implemented in the user-level library function named SpinBlock_Flags(). Block_Flags(), the system call that blocks a process is used only when the spinning strategy ends before the data arrives.

The arguments to SpinBlock_Flags() are:

- count: specifies the number of flags
- pointerArray: an array of pointers to flags
- valueArray: an array that specifies an initial value for each flag
- strategy: specifies which spinning strategy to use
- timeout: limits the time a process will sleep

When a flag's value is different from the one specified in valueArray, it indicates that a suitable message has arrived. SpinBlock_Flags returns as soon as any of the flags have changed; that is, when some valueArray[i] has a value different from pointerArray[i]. Figure 2.15 shows the algorithm for SpinBlock_Flags, which returns a lower bound on the number of flags that have changed.

[3] In this case, the counters are actually the head and tail index that implement the socket circular buffers

```
int SpinBlock_Flags (count, pointerArray, valueArray, strategy, timeout)
{
    while (strategy says to keep spinning)
    {
        check flags
        if (any flag changed)
            return (number of changed flags)
        if (strategy says to yield the CPU)
            yieldCPU()
    }
    CountChangedFlags = SBF_SysCall (count, pointerArray, valueArray)
    return CountChangedFlags
}
```

Figure 2.15: Pseudo-code for SpinBlock_Flags()

Kernel System Call

SBF_SysCall() is the kernel system call that puts a process to sleep until new data arrives (see Figure 2.16). It has the same semantics as SpinBlock_Flags, except that it blocks the process rather than spinning.

Line 5 turns on the network interface's incoming interrupts, but only for those pages containing at least one of the pointerArray flags.[4] Line 6 is an optimization that keeps a bit vector for each page and allows the system to ignore interrupts caused by incoming messages to other locations on the same page as the flag(s). Line 7 sets the flag pointer (i.e. address) to its initial value. This is needed to detect if an incoming message changes the value of a flag. Line 9 makes the process not runnable. Lines 10–12 check to see if any flags have changed, to avoid a race condition between the setting of the interrupt bits and the arrival of new data. Lines 13–14 put the process to sleep if no flags have changed.

We reach line 17 one of two ways: (i) the flag checking in lines 10-12 detected a change and the process was not blocked, or (ii) the process was blocked on line 14 and the

[4]Normally, the user program runs with interrupts turned off.

```
int SBF_SysCall (count, pointerArray, valueArray)
{
    for each flag pointer
    {
        enable page interrupt
        set page bit vector
        insert (pointer,value) pair into hash table
    }
    set process state to BLOCKED
    for each flag pointer
        if data has changed
            set process state to RUNNING
    if process state is BLOCKED
        call the process scheduler
    /* cleanup */
    for each pointer {
        count number of changed flags
        disable page interrupt
        clear page bit vector
    }
    set process state to RUNNING
    return count of changed flags
}
```

Figure 2.16: Pseudo-code for SBF_SysCall()

incoming interrupt woke up the process. Either way, we must now disable interrupts again and clean up, lines 17–20. Line 22 makes the process runnable again and line 23 returns the appropriate flag count.

Device Driver Interrupt Handler

We added a simple hook to the interrupt handler in the device driver for the SHRIMP network interface. When an interrupt occurs, the hook checks whether the interrupting message has changed a flag that some process is waiting for. If so, the process is made runnable and the interrupt bits are cleared for all pages the process was waiting for. The result of this hook is that the process reawakens whenever one of the flags changes.

Together, the three pieces of code implement an efficient mechanism by which a user-level process may wait for one of a set of flags to be changed by the arrival of a message.

2.5 Experiments

This section describes the experimental results obtained for the spin-block mechanism as well as the user-level implementation of stream sockets.

2.5.1 Strategies

The goal of these experiments was to understand the performance of the various message detection strategies. We would like to know which spinning strategy performs best and under what conditions. Of particular interest is how the strategies perform in a multi-programming environment.

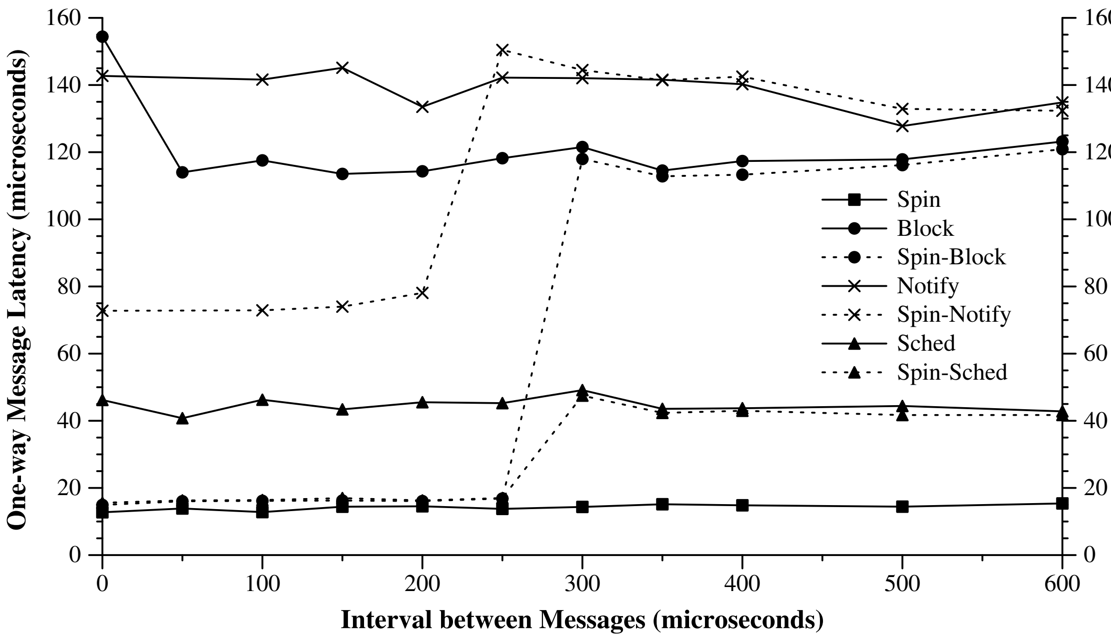


Figure 2.17: Latency of a single ping-pong program for each strategy with a spin delay of 300 μs and a message delay ranging from 0 to 600 μs.

Micro-Benchmarks

Our first set of experiments uses a simple client/server program which ping-pongs one word between two processes, running on two different Shrimp-C nodes. The ping-pong program was written using the VMMC API [31] directly. In these experiments we inserted a fixed delay (the "message delay") before the send operation in the ping-pong program. This simulates a case where some amount of computation is required before replying to each arriving message. By varying this delay time, we can see how hybrid strategies such as spin-block perform compared to the pure strategies. The graph in Figure 2.17 shows the performance for all of the spinning strategies when the message delay varies from 0 μs to 600 μs.

As expected, the pure spin strategy has the lowest message latency. This occurs because the hosts' only runnable process is ping-pong and therefore it is always ready to detect incoming messages. As a variation of pure spinning, sched performs not much worse, with some overhead due to making the system calls and calling the kernel scheduler. The two pure blocking strategies, notify and block show higher waiting time per message due to the cost of fielding the interrupts, performing the specialized system call in the case of block, and invoking the user-level signal handler in the case of notify.

The hybrid strategies—spin-sched, spin-block, and spin-notify—each spin for a fixed delay of 300 μs (our approximation of T_{block} described in Section 2.3.2) before switching to the respective pure strategy. As a result, they perform very well when the message delay is smaller than the spin delay. spin-notify has a constant overhead for handling user-level notifications even though they are not used during the spin-waiting phase. When the message delay gets larger than the spin delay, the hybrid strategies cannot catch the ping-pong messages within their spinning phase and have to call on their respective pure strategies for receiving the message.

This experiment does not show the advantage of sched, block and notify: giving up the CPU and allowing another process to make progress. Sharing the CPU is especially important when several processes on a node are actively communicating.

This experiment uncovered a performance bug in the Linux kernel. There was a race condition between the code that made a process runnable when a signal was delivered to it, and the code that halted the machine until the next interrupt when there were no runnable processes. The result was that the machine would sometimes be spuriously halted, and nothing would happen until the next timer interrupt, even though there was a runnable process. This bug showed up as a spike in the notify curve for a particular value of the message delay. Removing the race condition eliminated the performance surprise.

Effects of multiprogramming To determine the effects of multiprogramming we executed five pairs of ping-pong processes simultaneously using a message delay with a uniform random distribution between 0 and 300 μs. The results shown in Figure 2.18

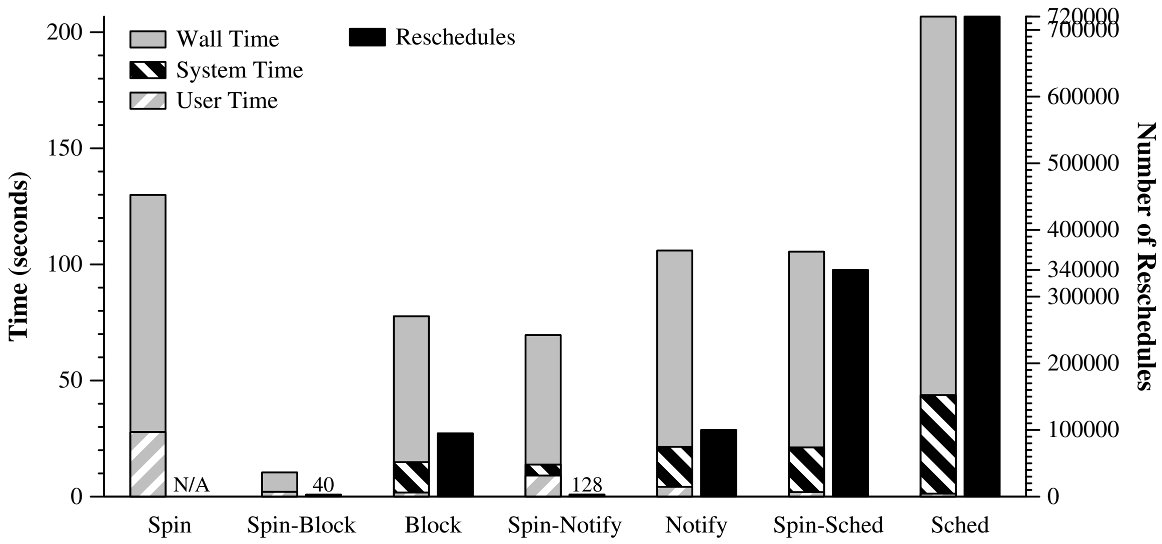


Figure 2.18: Performance for 5 pairs of processes ping-ponging 100,000 messages with a random delay of 0 - 300 μs between messages. The spinning strategies use a spin delay of 300 μs.

represent the one process average over at least four runs of each experiment. The left bars indicate the average per-process wall-clock time, and how much of it was user and system time. The right bar measures the average number of times the kernel scheduler was invoked.

Our starting point is the pure spin strategy. Each process spins at the user level waiting for data to arrive. This is not a good idea because each process hogs the CPU, which does not allow for overlapping computation and communication. One process may be spinning uselessly while another process has a message waiting.

Clearly we should give up the CPU when we are not using it. This leads to the three basic strategies: block, notify, and sched. If data is not present sched simply gives up the CPU, notify puts itself to sleep, and block turns on interrupts and then sleeps. In Figure 2.18 we see that block does better than notify despite an equal number of sleeps. The 36% increase in running time is due to the additional overhead required for notifications. That is, the additional time required to invoke the user-level handler versus simply waking up a process. The sched strategy is still worse because the kernel scheduler is thrashing; without interrupts, there is no other way to detect the arrival of new messages.

Each of these basic strategies can be improved by adding a user-level spin (i.e. a check for new data) before yielding the CPU. Roughly speaking, the hybrid strategies perform like spin when the message delay is less than the spin time, and like sched, block, or notify (for spin-sched, spin-block, and spin-notify, respectively) when the message delay is larger. In this way, each hybrid approach improves performance. The spin-sched strategy reduces sched's CPU yields and, correspondingly, the running time by 50%. The spin-block and spin-notify strategies virtually eliminate all CPU yields. The difference in their running time is because spin-notify must still service one notification with each message while spin-block, in the common case, runs without interrupts enabled.

The spin-block strategy performs particularly well in this experiment because it tends to cause processes that communicate with each other to run at the same time. When communicating processes are co-scheduled in this way, they can exchange many messages without rescheduling. The co-scheduled processes receive messages while they are still spinning. If a quantum expires and one of the processes is de-scheduled, the other process quickly blocks and allows the system to find its way back into a co-scheduled state.

Ttcp

To corroborate the results of our custom benchmarks we ran a similar set of experiments using ttcp, a popular public-domain benchmark [67]. Ttcp measures network performance using a one-way communication test in which the sender continuously pumps data to the receiver.

We ran several independent instances of each benchmark at the same time on the same pair of nodes, and summed the bandwidth reported by the instances. Each instance transfered 2,048 messages of size 8 KBytes. If contention were not a factor, the total

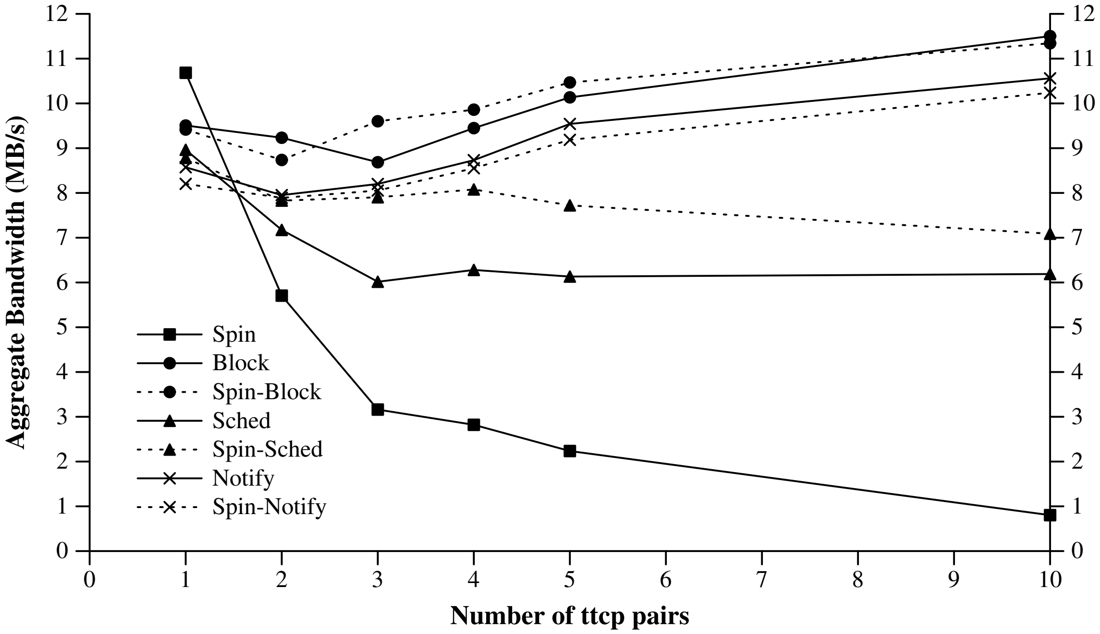


Figure 2.19: Performance of several concurrent instances of ttcp. The graph shows the total bandwidth achieved as a function of the number of ttcp instances running concurrently.

bandwidth reported by *k* benchmarks running concurrently would be the same bandwidth reported by a single benchmark running alone. The results of our experiment are shown in Figure 2.19.

With one instance of ttcp, spin performs best since there is no reason to ever block with only one process per node. However, as the number of instances increases, spin's performance degrades quickly as it becomes less and less likely that communicating processes are co-scheduled. The block strategy exhibits fairly constant bandwidth. It is the right strategy in this case since the message delay is large due to the sheer size of the messages being transferred. As usual, spin-block performs roughly the same as the better of spin and block (in this case block). sched and notify, along with their spin hybrids, perform as expected: better than spin but worse than block and spin-block.

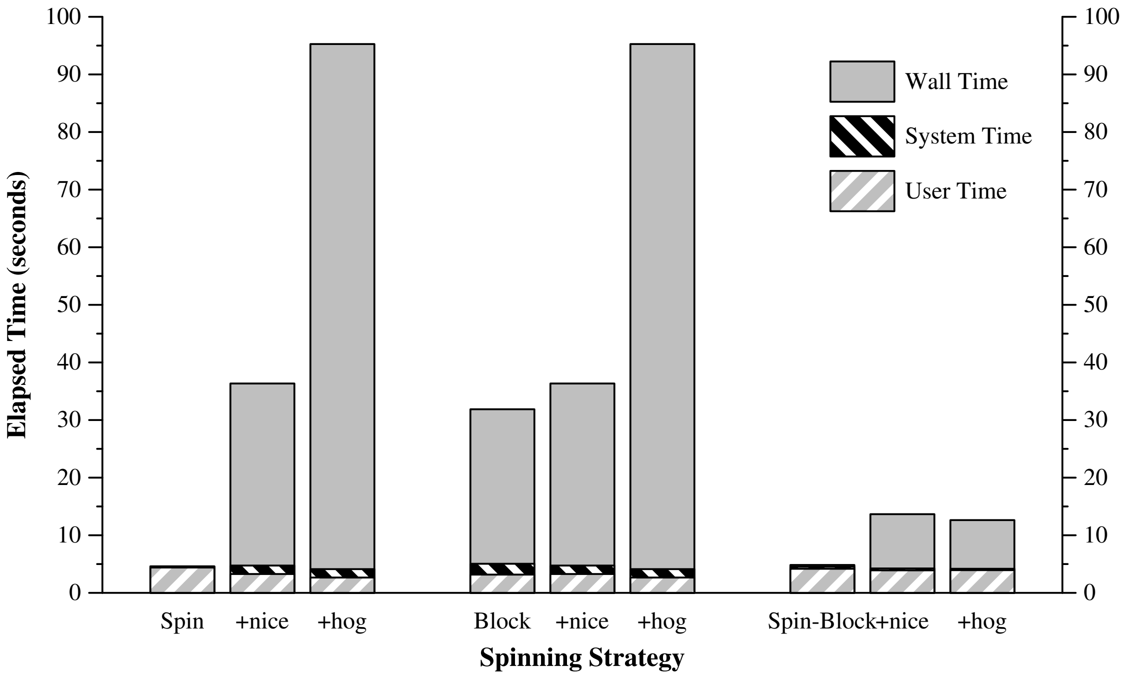


Figure 2.20: Performance of a distributed file system benchmark on two nodes

Distributed File System

Our next set of experiments uses a distributed file system (DFS) [61] implemented on top of our user-level sockets library. The distributed file system runs one process on each node; each process runs several user-level threads internally, and the processes communicate via our user-level sockets library. The distributed file system makes heavy use of select to wait on several sockets at the same time; this translates, in our sockets library, into waiting for one of several flags to change.

We ran a simple file system benchmark in which the file system process on one node accesses a sequence of file blocks that are initially cached in the memory of the file system process on the other node. This causes a sequence of data transfer messages and cache consistency protocol messages between the two nodes.

We measured the distributed file system performance for the three policies that had a chance of having the best performance: spin, block, and spin-block.

First, we used each policy to run the file system benchmark with no other processes running on the nodes. Next, we ran the file system benchmark concurrently with one of two different "CPU hog" processes running in the background on each node:

- hog: a process that simply spins in a tight loop consuming CPU cycles
- nice: identical to the hog process except with the lowest possible CPU-scheduling priority

Note that due to the process scheduler, the scheduling priority difference between hog and nice only matters when their quantum expires. Since they both spin at the user-level, once they get assigned a time quantum they will use all of it. Furthermore, we ignored their performance and only measured the performance of the file system. Specifically, we measured the user and system time used by the file system processes, both with and without the additional processes. The difference between these times is the extra cost incurred by the file system processes because of interference from these additional processes. We then measured the wall-clock time and user and system time for the file system processes, both with and without the nice or hog background processes. This experiment allows us to determine how much time is lost by communicating high-priority processes because of the existence of low-priority processes.

The results are shown in Figure 2.20. The spin-block strategy achieves the best performance in all three test cases, while the other two perform badly when either the nice or hog process is running. In the first case, where the file system process is the only running process on a node, spin is obviously the optimal strategy (see Figure 2.17). Blocking in this case would just add unnecessary overhead. Therefore both spin and spin-block outperform the block strategy.

When nice or hog are also run, the file system process has to wait for the background process to complete a full scheduling quantum before being able to do any useful work. In particular, if the file system process is not running when a message arrives, it has to wait for a full scheduling quantum before it can check for the data. Meanwhile, the file system process on the other node would either exhaust its scheduling quantum, or decide to use block strategy and go to sleep. Hence, a vicious cycle of context switch thrashing arises. This explains why both the spin and block strategies perform badly in the presence of a background process. In contrast, the spin-block strategy has the tendency to co-schedule communicating processes. This property allows spin-block to avoid, or quickly get out of, a thrashing cycle resulting in the excellent performance of spin-block in the presence of the hog and nice processes.

2.5.2 VMMC Stream Sockets

This section describes the performance measurements obtained using our implementation of sockets. All tests are performed on a prototype sixteen-node Shrimp-C 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 two different implementations of the sockets library that 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.

- *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 causes the actual data transfer.

We measure the performance of the sockets implementation using micro-benchmarks and the benchmarks ttcp and netperf. Finally, we use the following three metrics [56] to characterize performance:

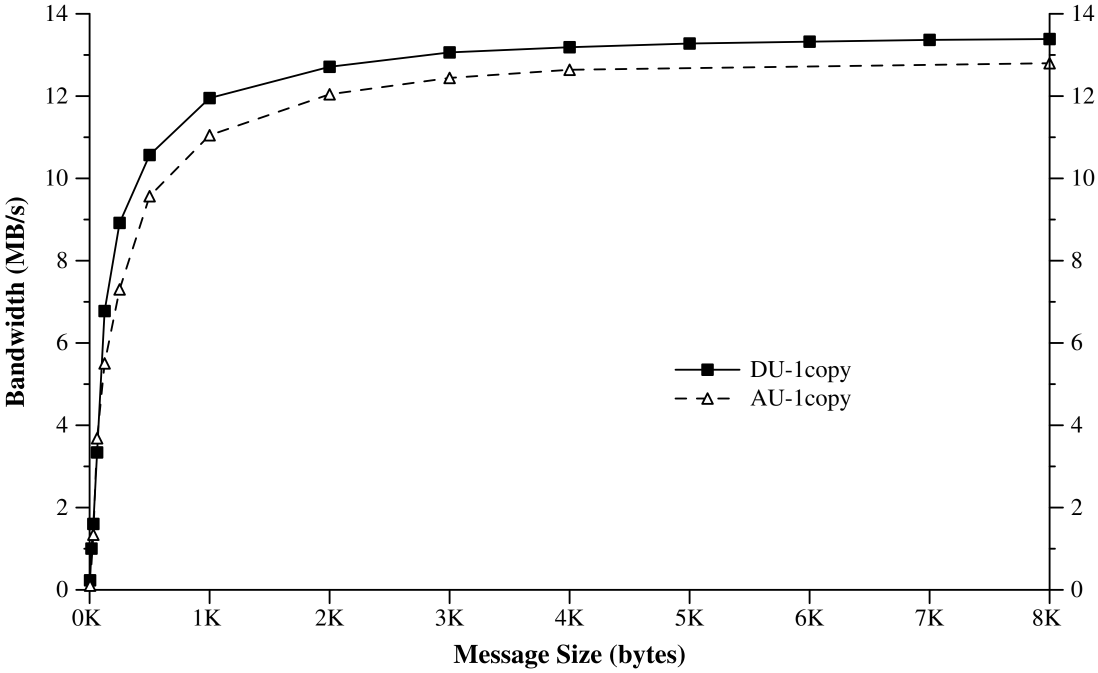


Figure 2.21: Shrimp-C sockets bandwidth using micro-benchmarks

r_∞ = peak bandwidth for infinitely large packets

$n_{\frac{1}{2}}$ = packet size to achieve $\frac{r_\infty}{2}$ bandwidth

l = one way packet latency

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 2.21 shows the results. Both implementations have similar performance. For larger 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 2.22 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

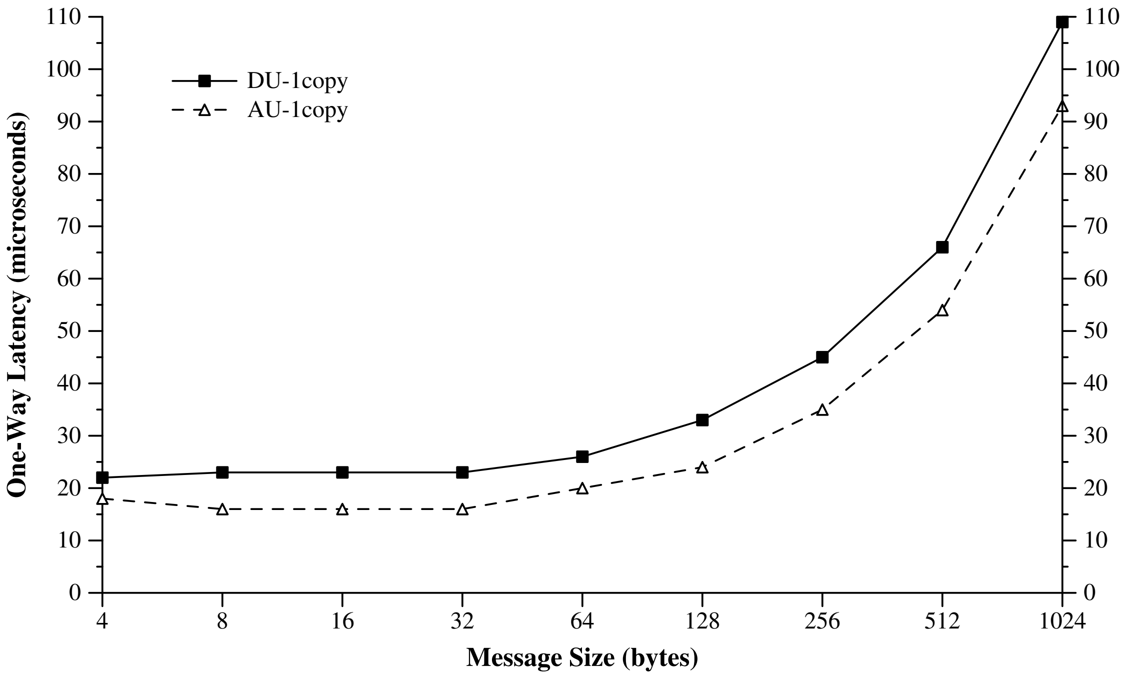


Figure 2.22: Shrimp-C sockets latency using micro-benchmarks

hardware limit. This extra time is spent equally between the sender and receiver performing procedure calls, checking for errors, and accessing the socket data structures.

Ttcp

Ttcp is a public-domain benchmark originally written at the Army Ballistics Research Lab [67]. 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 2.23. 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.

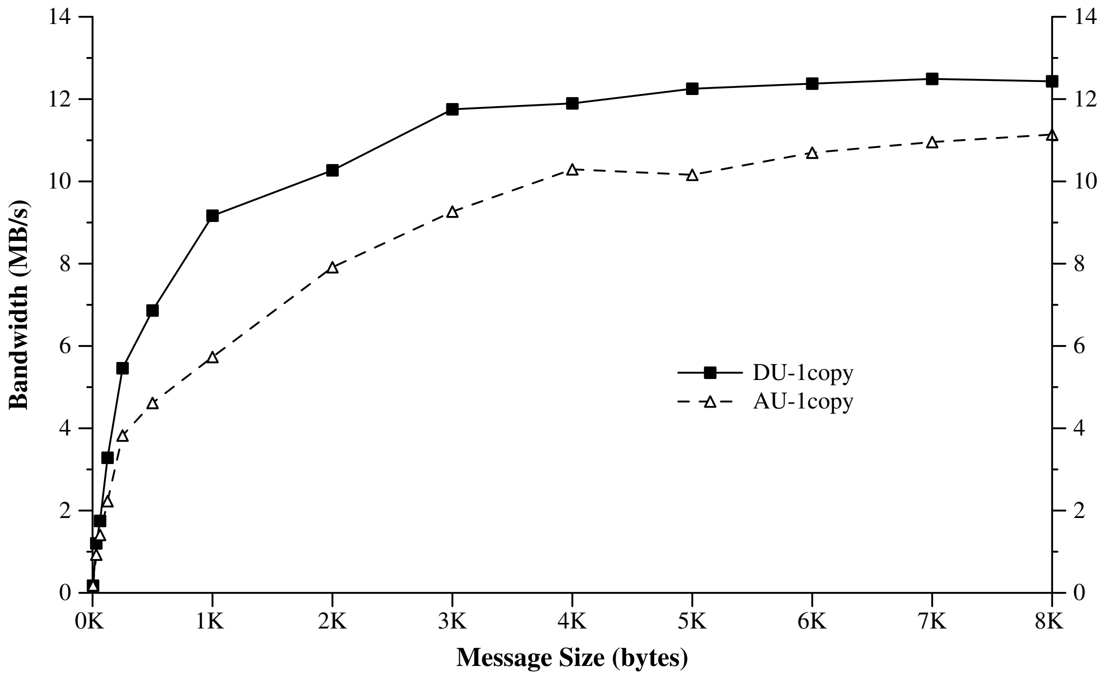


Figure 2.23: Ttcp Socket Bandwidth

Netperf

Netperf (Revision 1.7) is a public domain benchmark from Hewlett-Packard [39]. Figure 2.24 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.

2.6 Related Work

Active messages [69] invoke a receiver-side handler whenever a message arrives. Control information at the head of each message specifies the address of a user-level routine that is responsible for extracting the message from the network. This approach often requires servicing an interrupt for each message received.

The Myrinet [12] network interface has a network co-processor which implements packet switching message passing. The co-processor gives the network interface flexibility

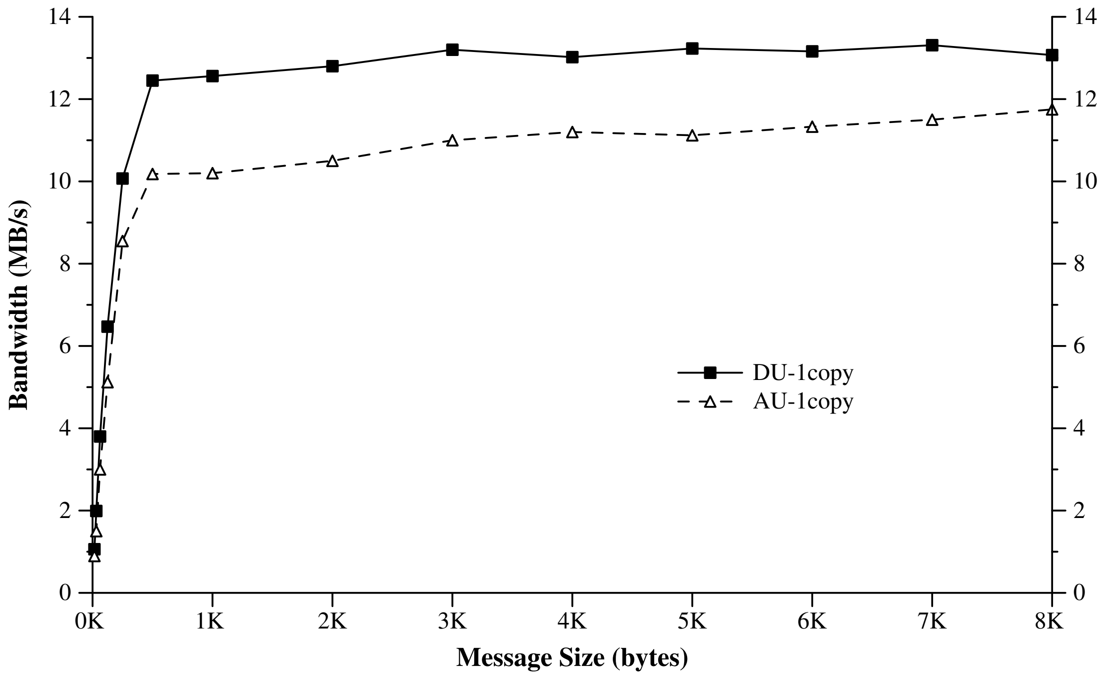


Figure 2.24: Netperf Socket Bandwidth

to implement a variety of receive-side interfaces, but the Myrinet system as shipped does not support any policies as aggressive as ours.

Several experimental network interfaces allow a sending process to deposit data directly into the memory of a receiving process [11, 16, 34, 37, 52, 70]. Though these systems differ in several important aspects, they all provide the basic functionality to support efficient user-level communication, so they all face the problem we are trying to solve. None of them has yet provided a solution, but our solution strategy should work on all of them.

The *polling watchdog* [51] is technique that tries to minimize the number of interrupts by delaying interrupting the receiving process in case it reads the incoming data in time and avoids needing to be interrupted. This work was done on a different architecture which requires the receiver to explicitly extract the incoming message.

Karlin et al [44, 43] studied the effects of spinning vs. blocking tradeoffs on multiprocessor performance in the context of locks. We applied some of their ideas in our work.

The spin-block co-scheduling side-effect is further investigated with simulations of "bulk-synchronous" SPMD applications [32]. Similar work uses *dynamic co-scheduling* [63] as part of a working system that explicitly treats all message arrivals as scheduling requests.

2.7 Conclusions

The main thing we learned from this research is that communication interrupt overhead has significant impact on application performance. The competitive spin-block strategy is very effective at reducing this overhead in a multiprogrammed environment. The main reason is that it takes advantage of temporal locality in communication. A small spin delay is able to significantly reduce the number of interrupts and hence improve overall performance. Another reason is that the mechanism has the effect of co-scheduling processes that are communicating. The co-scheduled processes receive messages while they are still spinning. If a quantum expires and one of the processes is de-scheduled, the other process quickly blocks and allows the system to find its way back into a co-scheduled state.

We performed three different experiments using three different programs in a multiprogrammed fashion. The first experiment used five instances of the ping-pong benchmark with random delays before message sends. In this experiment spin-block outperformed all other message detection strategies. The best interrupt based approach required 7.5 times more runtime than spin-block which eliminated almost all interrupts. The pure spin strategy that completely eliminates interrupts required an order of magnitude more runtime than spin-block. The same experiment exposed the benefit of the fast interrupt mechanism. The spin-block and spin-notify strategies are only different by the interrupt mechanism they use. The slower interrupt, spin-notify, requires 7 times more runtime than the fast interrupt mechanism.

The second experiment measured the impact of multiprogramming using ttcp, benchmark that measures one-way communication. The co-scheduling effects of spin-block are reduced with one-way communication because there no feedback (i.e. messages) from the receiver to the sender. Despite this, spin-block performs roughly the same as the best strategy. On the other hand, the fast interrupt mechanism translates to a consistant 1 MByte/s (10-20%) performance improvement.

The final experiment used a user-level distributed file system to measure the impact of multiprogramming in the presence of compute intensive processes. With only the DFS communicating, adding a low-priority compute intensive process more than doubles the running time of the pure spin and pure block strategies compared to spin-blocking. Making the compute process high-priority increases this the difference to almost an order of magnitude. This indicates that spin-block is very effective at sharing resources even when other processes are not using the communication subsystem.

In summary, interrupts have a significant impact on user-level communication system. Controlling the number of interrupts and minimizing their cost are paramount to obtaining good performance in a multiprogrammed environment.

The implementation of VMMC sockets exposed some weaknesses in the VMMC model. First, VMMC restricts connection-oriented communication models to one-copy protocols. Second, explicit flags must be used to indicate the arrival of new messages. This forces connection-oriented protocols to add a second message transfer for the explicit flag. Both of these issues are addressed in the next chapter.

Chapter 3

Copy Avoidance

3.1 Introduction

In Chapter 1 we briefly outlined the three sources of software overhead that limit message passing communication performance on new-generation networks: (i) interrupts, (ii) system calls, and (iii) message copying. In Chapter 2 we described VMMC, a user-level communication model that eliminates system calls from message transfers. We also discussed techniques to minimize the impact of interrupts on message reception. In this chapter we focus on the third issue, message copying.

The two communication characteristics that force message copying in connection-oriented communication are *trust* and *timing*. In connection-oriented communication, processes communicate in pairs but they do not trust each other. As a result each process' memory address space is kept separate; all communication occurs through named connections. Processes use named connections to send and receive messages.

The timing between sender and receiver plays a critical role. Let t_r be the time the receiver decides the destination of a message (i.e., the receiver reaches the appropriate place in the instruction stream). If a message is sent *before* t_r its destination is not known and the message must therefore be buffered. At some time after t_r it will be moved from the temporary buffer to the receiver-specified destination. As a result either the sender or the receiver is forced to copy the message (depending on the implementation).

These two communication characteristics make connection-oriented communication fundamentally different from other message passing models. In this chapter we address the issues of trust and timing in the context of message copying. We describe how to design and implement the low-level data transport protocol so that high-level connection-oriented protocols can be implemented with little overhead and without copying.

3.1.1 Model Mismatch

A memory-mapped communication model, as implemented by SHRIMP VMMC [11] and DEC memory channel [36], requires the least software overhead because data transfers avoid intervention by the receiver's CPU (Chapter 1). These and other implementations are *sender based* models: the sender specifies the ultimate destination of a message.

However, our early experience has shown that using such a low-level API to implement a high-level connection-oriented API like stream sockets often requires a copy on the receiver side [24]. This is because the basic memory-mapped communication model requires the sender to know the destination address before it sends data. On the other hand, connection-oriented high-level APIs such as stream sockets do not have destination addresses on the sending side; a sender knows only the name of the connection. To accomplish a zero-copy protocol often requires a scout message or a handshake in the implementation [1, 24, 59]. Neither option is satisfactory for high performance system area networks.

The Cost of Handshaking. Handshaking overhead includes the cost of a round-trip message and the cost of implementing the handshaking protocol by both the sender and receiver. Further, a handshake forces an explicit synchronization that may not occur otherwise. For example, the sender may wait unnecessarily long for a handshake acknowledgment instead of sending the data and continuing to do useful work, especially in the presence of multiprogramming where the sender and receiver may not co-scheduled.

The Cost of Copying. Without a handshaking protocol, the model mismatch forces the copying of messages. Who copies depends on the implementation; it can be either the sender or the receiver. When the receiver is responsible for copying, the sender sends messages to a *default* user buffer from where the receiver copies messages into the *destination* user buffer (Figure 3.1a). This approach is called *receive-side copy.*

The alternative, *send-side copy*, is for the sender to copy the message to a local *default* user buffer before it is transfered to the receiver (Figure 3.1b). The message is transfered into the receiver's *destination* user buffer either by a remote fetch or an explicit request from the receiver to the sender.

A send-side copy used in conjunction with copy-on-write may reduce some, but not all, of the copying. In fact, when an application uses the same source buffer to send consecutive messages, copy-on-write does not eliminate any copying. Further, copy-on-write requires system calls and page protection manipulations that are expensive.

We want to quantify the impact of copying on performance. Equation 3.1 represents the efficiency percentage, or the net bandwidth which results from copying a message k times. BW_{network} is the bandwidth of the network, and BW_{copy} is the memory-to-memory copy bandwidth. $T_{network}$ is the time a message spends in the network while T_{copy} is the time it takes for the receiver to copy the message between two user memory buffers. Simplifying results in Equation 3.2, which is expressed in term of network and copy bandwidths. Note that this equation is an optimistic because it does not include any overhead for managing the buffer copying.

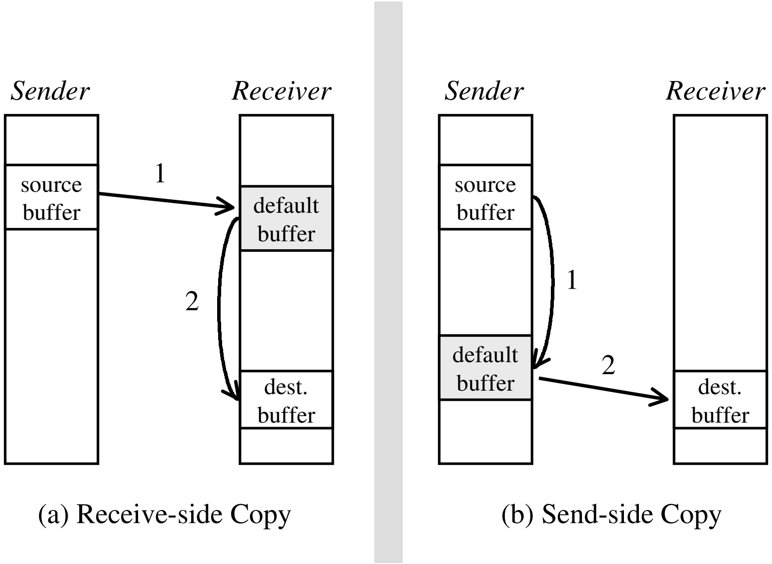


Figure 3.1: Message copying options

$$\text{Efficiency}(k) = \frac{T_{\text{network}}}{T_{\text{network}} + k \cdot T_{\text{copy}}} \quad \text{(where } k \text{ is the number of copies)} \tag{3.1}$$

$$= \frac{\frac{\text{Bytes}}{BW_{\text{network}}}}{\frac{\text{Bytes}}{BW_{\text{network}}} + k \cdot \frac{\text{Bytes}}{BW_{\text{copy}}}}$$

$$= \frac{BW_{\text{copy}}}{BW_{\text{copy}} + k \cdot BW_{\text{network}}} \tag{3.2}$$

Table 3.1 shows the results of evaluating Equation 3.2 for two different PCs and three different networks. The table shows that copying does not have a significant impact on a 1995 PC with ethernet since 94% of the bandwidth is still available to the user. The performance of current PCs in a LAN is still satisfactory with 90% of the network bandwidth

	1995 PC with 10 Mbit/s	**1998 PC with 100 Mbit/s**	**1998 PC with 1 Gbit/s**
Memory B/W	15 MB/s	100 MB/s	100 MB/s
Network B/W	1 MB/s	10 MB/s	100 MB/s
	How much of network bandwidth is seen by the application?		
1 Copy	94 %	90 %	50 %
2 Copies	88 %	83 %	33 %

Table 3.1: The cost of copying

usable. The effect of copying increases dramatically when current PCs are used in a new-generation network. Only 50% of the network bandwidth is available to an application.

A connection-oriented protocol that copies is inefficient because it uses only a third of the available network bandwidth. Unfortunately, traditional kernel implementations of connection-oriented protocols use copying to move messages between kernel and user memory. To address the model mismatch and eliminate the need for copying, we propose, implement, and evaluate a user-level mechanism called *transfer redirection*. We show that this mechanism naturally extends the virtual memory-mapped communication model and can easily support connection-based protocols to achieve zero-copy without any additional messages.

3.2 Transfer Redirection

Section 2.4.2 on page 21 described the basic virtual memory-mapped communication (VMMC) model, which provides protected, user-level communication. VMMC moves data directly from a send buffer to a receive buffer without any copying. However, since it requires the sender to know the receiver buffer address, it does not provide good support for connection-oriented communication.

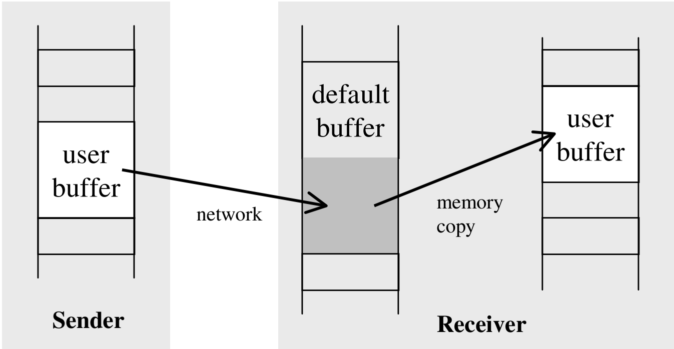
VMMC-2 extends the VMMC model with a mechanism called *transfer redirection*. The basic idea is to allow the receiver to *redirect* receive buffers. In other words, the receiver can specify a different virtual address for the next message destined for a specific receiver buffer.

We define the following terms: *default buffer* and *final buffer*. The default buffer is a receiver buffer that can be redirected, and the final buffer is the final destination of a message. A message is redirected from a default buffer to a final buffer.

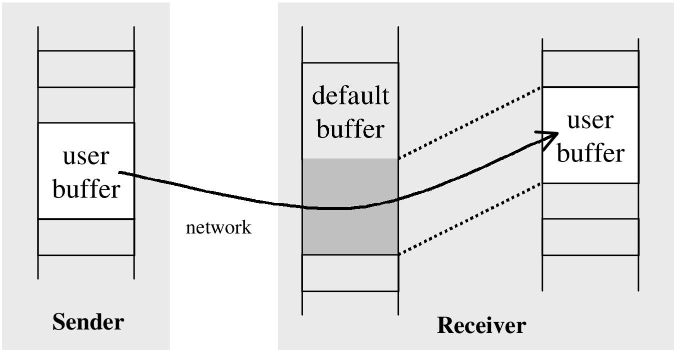
Redirecting a default buffer is a local operation performed by the receiver, independent of the sender. The sender is not aware of redirection and always sends data to the default buffer in the same way it would send to a normal receive buffer. When data arrives on the receive side, the redirection mechanism checks to see whether a redirection address has been posted. If not the data will be placed in the default buffer. Later, when the receiver determines the final buffer the message must be copied from the default buffer to the final buffer, as shown in Figure 3.2(a).

If the receiver posts the final buffer address before the message arrives, the message will be put into the final buffer directly from the network without any copying, as shown in Figure 3.2(b). If the receiver posts the final buffer address during message arrival, the message will be placed partially in the default buffer and partially in the final buffer. The redirection mechanism tells the receiver exactly how much and what part of a message was redirected. When partial redirection occurs, this information allows the receiver to copy the part of the message that is placed in the default buffer.

The redirection mechanism is accessed through two function calls. PostRedir turns on the mechanism while CancelRedir turns it off and returns redirection status information.



(a) A copy takes place when the receiver posts its buffer address too late



(b) Transfer redirection moves data directly from the network to the user buffer

Figure 3.2: Transfer redirection uses a default buffer to store message in when the receiver posts no buffer address or posts it too late, and moves data directly from the network to the user buffer if the receiver posts its buffer address before the data arrives

- PostRedir (defaultBuffer, requestOffset, requestSize, finalBuffer)

  where defaultBuffer is the default buffer for redirection, requestOffset is the offset within this buffer indicating the starting address of the redirection, requestSize is the number of bytes to be redirected, and finalBuffer is the address of the final buffer to redirect to. This call posts a user-level transfer redirection.

- CancelRedir (defaultBuffer, &redirSize, &redirOffset)

  where defaultBuffer is the default buffer for redirection, redirSize returns the number of bytes redirected and redirOffset returns the offset of the first byte redirected in the default buffer. This call cancels a transfer redirection posted to the default buffer. If no data has been redirected, the returned number of redirected bytes is zero. If a redirection did happen, the redirected data is in the contiguous user address space area starting at address defaultBuffer + redirOffset - requestOffset.

Transfer redirection is designed to function for sender/receiver pairs, i.e. one-to-one communication. The current implementation functions for many-to-one communication but its usefulness is bounded because of the implementation restriction that redirections are limited to one per buffer. This design decision is based on the fact that most connection-oriented high-level APIs implement a one-to-one communication model.

Transfer redirection naturally extends the VMMC communication model. It is controlled entirely by the receiver; the sender does not need to know what the actual destination address is nor whether a transfer redirection takes place.

Giving the receiver exclusive control over redirection addresses the trust issue in connection-oriented communication. Final buffers are the way a receiver specifies regions in virtual memory that can receive messages directly from the network interface without copying and CPU intervention. For example, a receiver can redirect to a buffer in the call stack (i.e., a local variable) without worrying that the sender will corrupt the stack.

3.2.1 Using Transfer Redirection

The amount of data that is actually redirected depends on the relative timing of the PostRedir call and that data arrival time (assuming CancelRedir does not interfere with data redirection). If a redirection is posted before message arrival, the data will be redirected. If a redirection is posted after data starts arriving but before the entire transfer finishes, the remainder of the arriving message will be redirected.

In the connection-oriented model, the receiving process specifies the final destination of incoming data, while the sending process is oblivious to what is happening at the receiver.

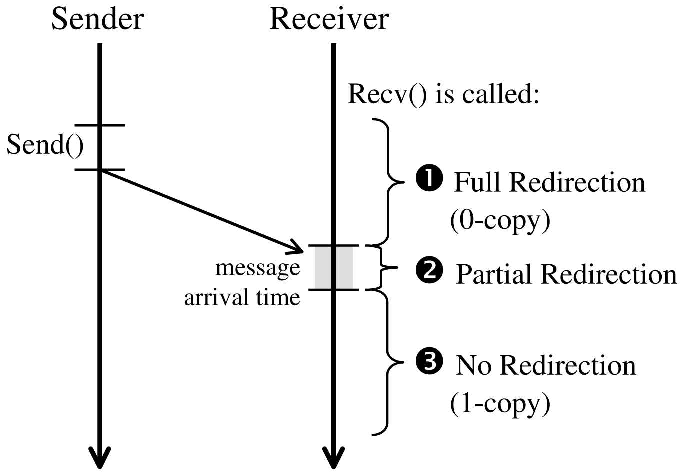


Figure 3.3: Redirection timing issues

This communication characteristic can limit the effectiveness of transfer redirection. In order to avoid copying, the redirection request must be issued by the receiver before or during message arrival. A redirection request results in one of three outcomes, illustrated in Figure 3.3:

- *full redirection:* all requested data was redirected because the request was issued before the message arrived.
- *partial redirection:* some contiguous part of the data was redirected because the request was issued while the message was arriving.
- *no redirection:* no data was redirected because the request was issued after the entire message arrived.

The receiving process does all the work associated with redirection. First, it checks the default buffer for new data. If no new data has arrived, it calls PostRedir to redirect the default buffer into the user buffer at the appropriate offset. The receiver then waits for data to arrive. A multi-threaded process can block the current thread and schedule another.

When data arrives, the receiver calls CancelRedir and checks the result of redirection. In the case of full redirection the receiver does not need to do any copying. If the redirection request partially completes, or fails, the receiver must copy some, or all, of the data from the default buffer to the user buffer. The exact amount of data copied is determined from the values returned by CancelRedir.

3.2.2 Other Approaches

A zero-copy protocol can be implemented with an additional network communication. In this approach the receiver informs the sender about the receive buffer, after which the sender is able to transfer data directly into it. Variants of this protocol have been used by NX implementation on Paragon [58] as well as an early NX implementation on VMMC [1].

Active Messages [33] is a user-level communication mechanism based on a fast remote execution facility. Recently, a fast sockets facility has been built with active messages handlers [59]. They used a receiver-based approach called *receive posting*. The arriving data is first placed in an anonymous staging area in memory and the message handler is invoked. If a receive has been posted, the handler moves the data to the final destination bypassing the socket buffer. Similar to redirection, this technique avoids the socket buffer copy. Unlike redirection, receive posting requires a copy through anonymous memory, but avoids the cost of pinning and unpinning user buffer pages.

Many methods have been proposed to avoid copying between I/O devices and application memory. IBM's OS/360 can specify a file I/O buffer area for applications to avoid copying. Page remapping has been used to avoid copying in several past and recent projects [18, 28, 42, 46, 49]. Brustoloni and Steenkiste proposed *emulated copy* [14] which combines page remapping and copying of sub-page data chunks. In this technique, the total amount of data copied is not more than one page, regardless of the user buffer size and its alignment. However, this technique requires expensive system calls and page protection manipulations.

Lately, several groups have proposed letting applications directly access communication interface memory [19, 23, 62]. However, network interface memory is usually small and non-cacheable, which limits its direct use by applications.

3.3 Receive-side versus Send-side Copying

In Section 3.1 we outlined the two approaches that addressed the connection-oriented model mismatch without using a handshaking protocol. We could copy either on the send-side or the receive-side (Figure 3.1 on page 58).

The transfer redirection mechanism is designed to work as part of the receive-side-copy solution. It is fair to question why we chose this approach and not the other. There is no reason why transfer redirection could not be designed to work with the send-side-copy solution. In such a system, the receiver would issue the redirection request to the sender who would transfer messages directly to the user-buffer. In the absence of a redirection request, the sender would copy messages into the send-side default buffer.

To decide which of the two approaches to use, we compared them to find if one was "better" then the other. It turns out that receive-side copy has concrete advantages over send-side copy. Let λ represent the one-way, user-to-user, small message latency.

The first adavantage is small message latency. Since redirection is not needed for small messages, we ignore it. We first consider what happens when the send occurs before the receive. For receive-side copy (Figure 3.1a), the sender immediately sends the message over the network to the default buffer on the receiver. This costs λ. Send-side copy (Figure 3.1b) first copies the message to the local default buffer from where the receiver will fetch it. The receiver can directly use a remote fetch which is faster than issuing an explicit request to the sender. Ideally, using remote fetch requires only λ because the fetch

Approach	Send ↘ Receive	Receive ↘ Send
Receive-side Copy	λ	λ
Send-side Copy	2λ	1λ to 2λ

Table 3.2: Small message transfers for receive-side vs. send-side

has to travel between the sender and receiver. Then, one more λ is needed to bring the small message back to the receiver. Therefore a total of 2λ is needed for send-side copy.

Next we consider the costs when the receive occurs before the send. The cost for receive-side copy is again λ since the message is immediately sent to the receiver. Send-side copy can hide some, or all, of the λ cost for issuing the remote fetch if the receive is issued early enough. This is due to the sender not having the data ready for transfer. One more λ is still needed to bring the small message back to the receiver. In this case, send-side copy requires between 1λ and 2λ, depending on how much of the cost can be hidden.

The small message costs are summarized in Table 3.2 which show that send-side copy can add up to 100% more overhead than receive-side copy. Clearly this is a significant advantage for receive-side copy.

The second advantage is subtle. Let t_s be the time the sender initiates the send and t_r the time the receiver initiates the redirection request. For send-side copy, the receiver's redirection request is "delayed" by λ, the time needed for the redirection request to arrive. Therefore, the sender see the request at time $t_r + \lambda$. To completely eliminate copying, the following must hold:

$$t_r - \lambda \leq t_s \tag{3.3}$$

On the other hand, receive-side copy benefits from the λ delay of incoming messages. This gives the receiver λ more time to issue the redirection request. Therefore, to completely eliminate copying, the following must hold:

$$t_r + \lambda \leq t_s \tag{3.4}$$

Comparing Equations 3.3 and 3.4 we see that there is a 2λ increase in the interval during which message can be completely redirected favoring receive-side copy.

There are other tradeoffs to consider, like software complexity. Receive-side copy requires a very simple send and complex receive, while the opposite is needed by send-side copy. Nevertheless, the two advantages outlined above clearly show that receive-side copy is the right approach to build on.

3.4 Adaptive Send Delay

Transfer redirection allows messages to be placed directly in the user-specified buffer without copying. In order to minimize copying we want to redirect as much of a message as possible. Ideally, copying is completely eliminated. Unfortunately this is not always possible since the message may arrive before the redirection request is issued.

In this section we present a mechanism called *adaptive send delay* (ASD) that complements transfer redirection. The goal of ASD is to improve the effectiveness of transfer redirection. The idea is to delay sending a message to allow the receiver to invoke redirection. The send delay, t_d, is the amount of time the sender waits before initiating the send.

The key question is how to determine the best value for t_d. We start by analyzing the cost of ASD (Figure 3.4). For our analysis, we assume that compute cycles on the sender and receiver are equally valuable (i.e. neither has cycles to "waste."). Time starts when

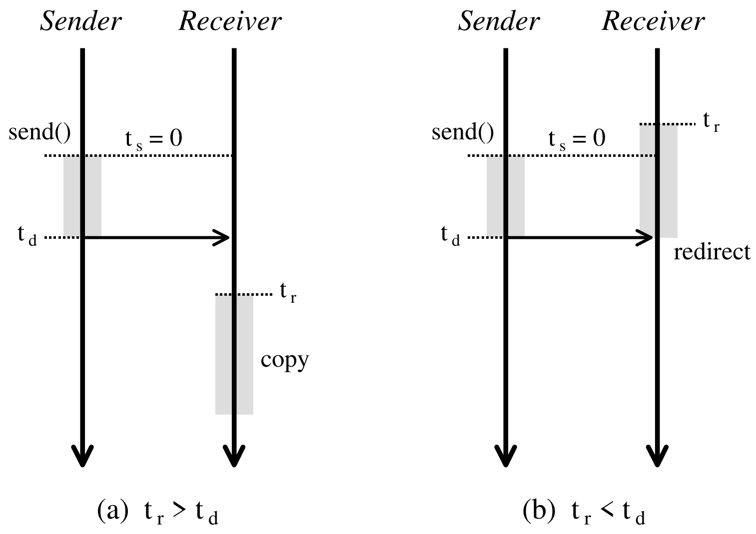


Figure 3.4: Send delay cost

the sender is ready to send ($t_s = 0$). t_r represents the time at which the receiver knows the redirection address, and can be positive or negative (i.e. it can occur before or after the sender is ready).

To compute the cost of message transfers, including ASD overhead, we sum the cycles used by the sender and receiver and include, when necessary, the message copy time. We factor out the time on the network since it is common to both cases we consider, i.e. message transfers are instantaneous. To simply the analysis we do not consider partial redirection.

First, let us consider when $t_r > t_d$ (Figure 3.4a). The receive is too late to take advantage of redirection even with the help of a delay. The sender delays for t_d cycles while the receiver uses $\frac{n}{BW_{\text{copy}}}$ cycles to copy the message.

Second, when $t_r < t_d$ (Figure 3.4b), the sender is delayed until after the receiver is ready. The sender delayed t_d cycles while the receiver was idle from the time the receive was posted until the message arrived, $t_d - t_r$. Transfer redirection was successful so there was no message copy. Therefore, the total number of cycles used by both sender and receiver is $t_d + (t_d - t_r) = 2 \cdot t_d - t_r$.

These two cases result in the following equation. Given n, the number of bytes to send, we express the message transfer cost as:

$$C(n) = \begin{cases} 2 \cdot t_d - t_r & \text{if } t_r \leq t_d \\ t_d + \frac{n}{BW_{\text{copy}}} & \text{if } t_r > t_d \end{cases} \quad \text{where } t_d \geq 0 \tag{3.5}$$

The only unknown in $C(n)$ is t_d, the delay time. How can we determine the correct value for t_d? First, let us suppose that the sender has an oracle which magically returns t_r, the time the receiver will enable transfer redirection and be ready to receive. Using t_r the sender can implement the optimal delay as follows. If $t_r < 0$ the receive has happened (since $t_s = 0$) so there is no need to delay. Similarly, no delay is used if t_r is larger than the cost to copy the message because only a long delay would avoid the copy. The delay has to be longer than the time to copy, therefore copying is optimal. But if t_r is less than the message copy cost, $t_{copy}(n)$, then the sender delays for t_r before sending because the delay saves $t_{copy}(n) - t_r$ cycles. The resulting algorithm is as follows:

```
Optimal_Delay (int MessageLength)
{
    t_r = ORACLE()
    MaxDelay = t_r - CURRENTTIME
    if ( ( MaxDelay > 0 ) and ( MaxDelay ≤ n/BW_copy ) )
        loop for MaxDelay
    return
}
```

Figure 3.5: Optimal send policy

We demonstrate that this policy is optimal using the cost graph in Figure 3.6. The abscissa, t_r, is the time when the receiver is ready to redirect. Setting the delay, t_d, to zero

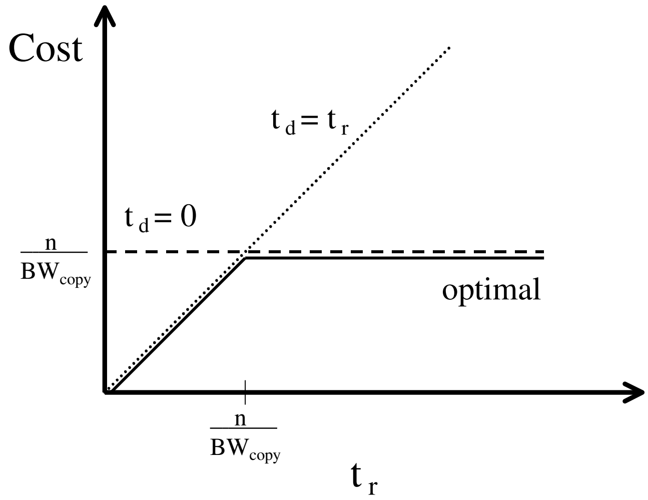


Figure 3.6: Send delay cost graph

results in a fixed cost of $\frac{n}{BW_{\text{copy}}}$, the cost of copying the message. On the other hand, the cost of always waiting (i.e. $t_d = t_r$) is linear because it is charged entirely to the sender. If t_r is small the cost is small, if t_r is large the cost is large. Setting t_d to anything else is not cost effective. Using any value such that $0 < t_d < t_r$ increases the cost in the $t_d = 0$ case to $t_d + \frac{n}{BW_{\text{copy}}}$. Similarly, for any $t_d > t_r$, the cost in the $t_d = t_r$ case increases to $t_r + (t_d - t_r) = t_d$. Therefore the algorithm in Figure 3.5 is optimal because it follows the lowest cost policy for all values of t_r. When $t_r < \frac{n}{BW_{\text{copy}}}$ it sets $t_d = t_r$ otherwise $t_d = 0$.

Now, more realistically, suppose we do not know t_r. Can we design an algorithm that is competitive with the optimal one?

We can use the groundwork established in Shared-Memory MultiProcessor Locking (SMMPL). There is a direct correspondence between our work and that of competitive spinning for SMMPL [43, 44]. Deciding between spinning and blocking a thread is analogous to deciding between spinning and sending a message. Lock spinning is analogous to the send delay and the penalty for blocking is analogous to the penalty for copying.

In SMMPL, a thread must decide between delaying and trying to acquire a lock again in the near future or blocking and allowing other threads to make progress. Similarly, a sender can send a message immediately or wait and hope to avoid a memory copy by the receiver. In both cases the decision is similar: perform a task immediately and incur a potentially high cost, or consume some cycles and hope that the cost will be lower in the near future.

In SMMPL, a thread trying to acquire a lock has additional information. For example, it knows when a lock is released, and it can keep track of how long previous lock attempts lasted. In our case, the equivalent information is not available locally to the sender. Therefore, two types of feedback are required from the receiver:

- *redirection status*: this indicates whether the receiver successfully redirected the last message. It is transfered to the sender "for free", piggy-backed on other protocol messages so that an additional send is not needed.
- *stop-spin*: it is possible that the sender ends up spinning when the receiver has already posted the redirection request. If this happens, the receiver sends a message to stop the sender from delaying.

Since there is a fixed cost associated with redirection (Section 3.6.1) we do not use it for small messages (less than 32 bytes). Simply copying such messages is more efficient.

An entire class of policies for SMMPL are described in [43, 44]. All of the policies apply equally well to our work. Next we present two of the simplest competitive delay policies.

3.4.1 Non-Adaptive Policy: Spin-Wait

Figure 3.7 presents the basic spin-wait policy where $t_d = \frac{n}{BW_{\text{copy}}}$. Intuitively, this value of t_d says that the delay should not be longer than the time it takes to copy the message. Karlin

```
SpinWait_delay (int MessageLength)
{
    /* not used for small messages */
    if (n < SmallMessageThreshold)
        return
    delay = t_copy(MessageLength)
    loop for delay
    {
        if (stop-spin bit arrives)
            break
    }
    return
}
```

Figure 3.7: Pseudo-code for Spin-wait policy

et. al. [43, 44], showed by a simple argument, that this policy is 2-competitive; its cost is never more than twice that of the optimal policy.[1]

3.4.2 Adaptive Policy: Random Walk

Figure 3.8 illustrates Adaptive Send Delay (ASD), an adaptive delay derived from a competitive, random walk algorithm [43, 44]. The send delay can vary between zero and $t_{copy}(n)$, the time to copy the message. The algorithm adjusts the delay threshold using feedback from the receiver. If the previous redirection failed, the delay is increased to improve the chances of succeeding next time. If the previous redirection was partially successful, then the delay is also increased, but by a small amount. Otherwise, the previous redirection was successful and the delay is shortened slightly in case it is too large.

[1]The 2-competitive argument is also applied to the decision by the receiver whether to spin or block when waiting for a message. See Section 2.3.2 on page 18 for a description.

```
int Adaptive_Send_Delay (int MessageLength)
{
    /* not used for small messages */
    if (n < SmallMessageThreshold)
        return
    delay = previous delay for this receive process
    /* check status of previous send */
    if (previous send was not redirected)
        delay += INCREMENT
    loop for delay
    {
        if ( t_copy(MessageLength) time has elapsed)
        {
            delay = t_copy(MessageLength)
            break
        }
        if (stop-spin bit has arrived)
        {
            delay -= INCREMENT
            break
        }
    }
    if (delay < 0)
        delay = 0;
    store delay
    return
}
```

Figure 3.8: Pseudo-code for ASD policy

3.4.3 Available CPU Cycles

The above analysis is based on the assumption that cycles are equally valuable on the sender and receiver. In other words, neither sender nor receiver have cycles to sacrifice. But there are instances where this may not be true. For example, a server's cycles may be more important, so the clients will wait until the server notifies them to send. This would allow the server to eliminate all message copying. In such cases the approach changes depending where the cycles are available. We use a scheme that assigns weights to the value of sender's cycles (W_s) and receiver's cycles (W_s). Two possible extreme weightings are:

- $W_r = 0, W_s = 1$: if the *receiver* has free cycles then the sender always uses *wait-zero* since the receiver has cycles available to copy when necessary
- $W_r = 1, W_s = 0$: if the *sender* has free cycles, then the sender always uses *wait forever* since free cycles are available

The analysis presented above assumes $W_r = 0.5, W_s = 0.5$ but we can derive an optimal policy for any weighting such that $W_r + W_s = 1$.

3.5 VMMC-2 Implementation

In this section we describe how the ideas presented in this chapter were integrated into a working system. Transfer redirection was incorporated into VMMC-2, the new version of VMMC. Adaptive send delay was added to the implementation of stream sockets described in Chapter 2.

3.5.1 Shrimp-M Platform

The VMMC-2 implementation works under Linux OS version 2.0.24 running on 166 MHz Pentium PCs connected with Myrinet network interfaces and switches [12]. Each network interface has a 33 MHz processor, named LANai, and 1 MByte of static RAM (Figure 3.9).

3.5.2 VMMC-2 Components

VMMC-2 supports multiple senders per node and runs on both uniprocessor and SMP nodes. Figure 3.10 shows the structure of VMMC-2 which consists of the following components:

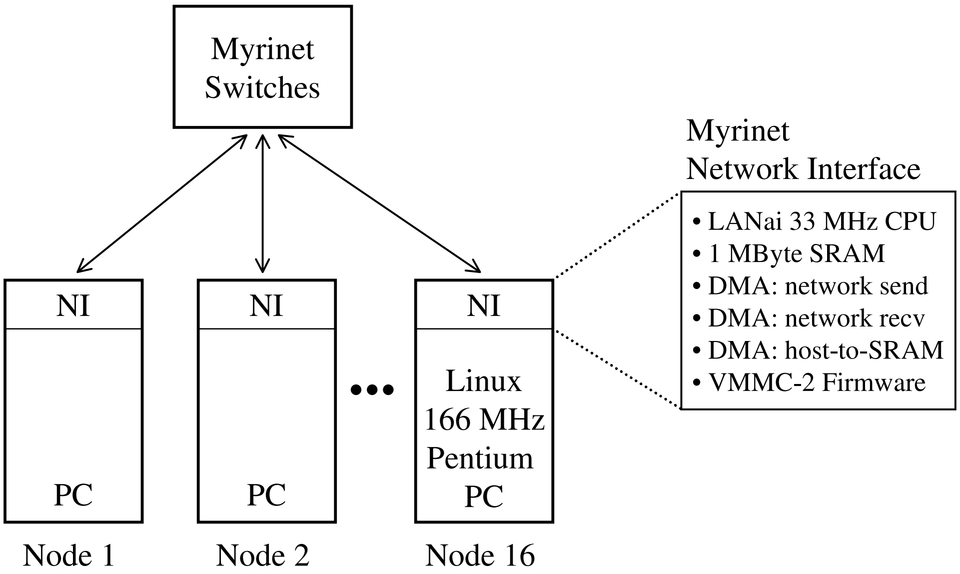


Figure 3.9: The architecture of the Shrimp-M platform

- a user-level library which implements the API of the VMMC-2 model
- firmware for the Myrinet network interface that works with the user-level library to implement data transfer and reliable communication [5]
- a device driver which provides VMMC-2 specific support. Invoking driver code requires a traditional (expensive) protection boundary crossing. We do not require any operating system modification except for this loadable driver.

Import-Export Mappings

When an application exports a buffer, it makes a call to the driver, which in turn pins the exported buffer's memory and sets up a descriptor for this buffer in the memory of the network interface. Exported receive buffers must be word aligned but they do not have to be page-aligned. On an import, the VMMC-2 user-level library passes the import request to the firmware running on the local network interface, which communicates with the remote node to match the import with an export and establish the import-export mapping.

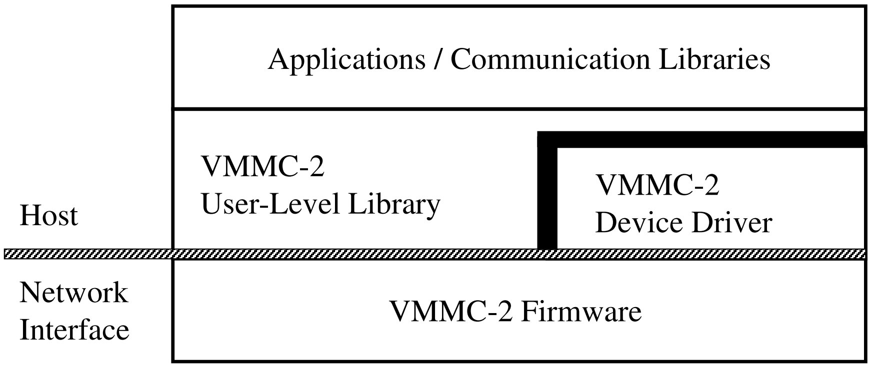


Figure 3.10: The architecture of the VMMC-2 implementation on the Myrinet platform

Data Transfer

When an application requests a data transfer, the VMMC-2 library pins the send buffer (if necessary), and obtains the physical addresses of the send buffer, which are passed to the network interface firmware.

The VMMC-2 firmware uses direct memory access (DMA) to move data—one page at time—from the host memory to the outgoing buffers of the network interface. Simultaneously, it uses DMA to transfer data from the outgoing buffers to the network. When a message arrives, the VMMC-2 firmware determines the destination address using the transfer redirection module. It then DMAs the incoming message from the network interface to the host memory.

To send data, a user process specifies a destination proxy address (Section 2.4.2), a local send address, and a number of bytes. The destination proxy address is passed directly to the network interface, which uses it, together with the outgoing table, to figure out the destination node and the physical address where the data should be deposited. Upon arrival, data is put directly into the receiver's memory by the network interface.

Transfer Redirection

The actual redirection registration and data structures reside on the network interface memory, which is mapped to the user's address space. Each user process has a set of redirection data structures allocated: one for each registered redirectable buffer. To post a transfer redirection, a user process invokes the translation library to translate the destination user buffer address into an index. Next, the process fills the translation results as well as other arguments of the PostRedir call into the redirection structure on the network interface including index, offset, a receive buffer offset where redirection should start, and the amount of data to redirect.

Upon arrival of each chunk of data from the network, the VMMC-2 firmware checks if any bytes of this chunk should be redirected by consulting the redirection structure associated with the default receive buffer of the data. Data that are not redirected are moved into the default buffer. For redirection, the data chunk may be scattered into multiple subchunks because DMA from the network interface to the host memory cannot cross page boundaries or redirection boundaries. Upon the completion of the redirection, the VMMC-2 firmware will update the redirection status by adding the number of bytes redirected. The VMMC-2 firmware will also cancel the redirection after processing the last (as indicated by a special header flag) chunk of the message.

CancelRedir cancels the posted redirection. If the redirection has been canceled already by the VMMC-2 firmware, the call simply collects redirection status from this receive buffer redirection structure in the SRAM. If the redirection has not been canceled yet, this call will set the size in the redirection structure to zero. Since both the VMMC-2 firmware and the host can cancel a redirection, race conditions are possible. Our implementation avoids the race conditions by using a redirectionActive flag in the redirection structure. This flag is set by the VMMC-2 firmware and read by the host. The VMMC-2 firmware sets the flag whenever a redirection is in progress and resets it when the transfer completes.

The host can cancel the redirection only after the flag is reset. This method avoids race conditions and ensures that there will be no more transfer of data to the user buffer after the cancellation is completed.

Our current implementation allows redirection to happen at most once for each PostRedir. As soon as a message is redirected from a redirectable buffer to a receive buffer, it cancels the redirection automatically. We chose this design to simplify CancelRedir, which would otherwise have to return multiple redirection regions.

When the last chunk of a message arrives, a receive-buffer specific memory location in user space is updated with the receive buffer offset corresponding to the last word transferred. This eliminates the need for the sending explicit flags to denote the arrival of a message. This happens even if redirection has taken place and the last word is placed outside the default receive buffer.

3.5.3 VMMC-2 Stream Sockets

The implementation of stream sockets is identical to the one described in the last chapter (Section 2.4.3) with only a few changes. First, the Shrimp-M platform does not support automatic update, so only deliberate update is used. Second, we incorporate the adaptive send delay mechanism (Section 3.4.2) with stream sockets.

Finally, stream sockets takes advantage of the redirection extension in VMMC-2 to avoid copying. Conceptually, this is shown in Figure 3.11. The sockets default buffer is redirected to the user's final buffer.

Integrating redirection with the circular and alignment queues introduces a complication (Figure 3.12). Suppose the receiver has already issued a redirection and some time later a message is sent. But the send message is "badly" aligned so the first few bytes are transfered via the alignment queue while the remaining bytes are transfered through the default buffer. These bytes trigger redirection but the result is that the receiver's final

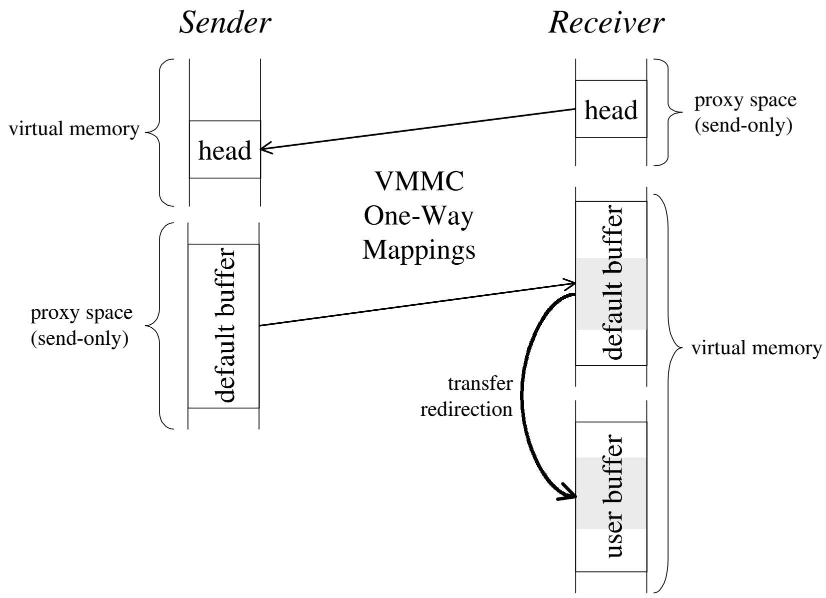


Figure 3.11: VMMC-2 mappings for a unidirectional byte stream with redirection

buffer does not contain the right data. This occurs because the first few bytes of the send message are in the alignment queue and not in the receiver buffer. The receiver must shift the redirected message and move the extra bytes from the alignment queue to the final buffer.

3.5.4 Block-Transfer Stream Sockets Extensions

The problem with this was that we did not get a sufficient amount of concurrency from the redirection mechanism. As described in Chapter 3, there can be only one outstanding redirection request per exported buffer.

The semantics of stream sockets specify that all messages are merged into a “stream” of bytes so that the receiver does not distinguish between messages. This characteristic, along with the VMMC-2 restriction of only one redirection per default buffer, led us to design an extension to the sockets API. A sockets based distributed file system (DFS) [61] drove our design. DFS transfers protocol and data messages using sockets. Because of sockets’

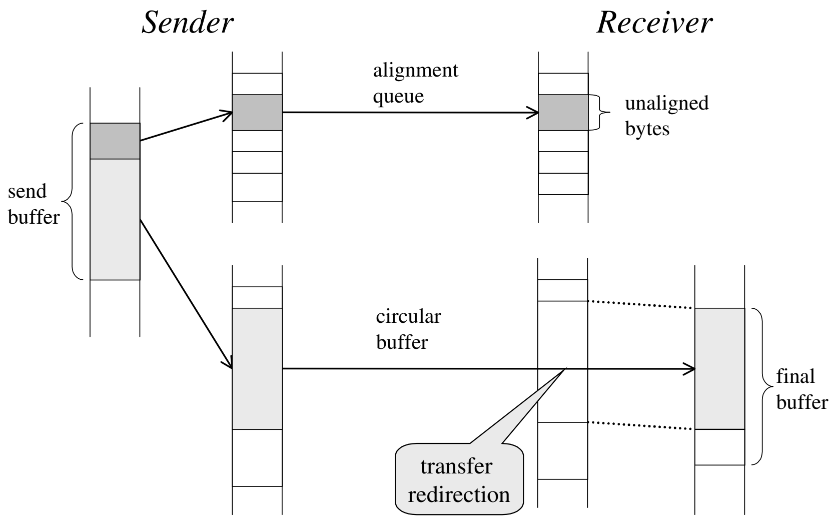


Figure 3.12: Redirection and alignment special case

message merging, DFS initially used two VMMC-2 sockets: one for protocol messages and one for data messages. The data message socket used redirection to avoid copying. This approach worked, but was limited by the VMMC-2 restriction of only one redirection per default buffer. Lifting the VMMC-2 restriction is possible, but it would not work well in the context of stream communications. This is because message boundaries are not maintained by sockets, so different messages cannot easily be redirected to different final buffers.

To address these limitations we designed a set of non-standard extensions to the sockets API. After a VMMC-2 socket connection is established, the call BlkSock_add_socket activates "block transfer" mode by specifying what size and how many additional default buffers to allocate for the socket. To use the block transfer mode a receiver "feeds" the socket with buffers (BlkSock_add_buf). These buffers are used to activate redirection for the additional default buffers. If user buffers are not specified in time, incoming messages go to the default buffers which must then be copied. After feeding the system with buffers,

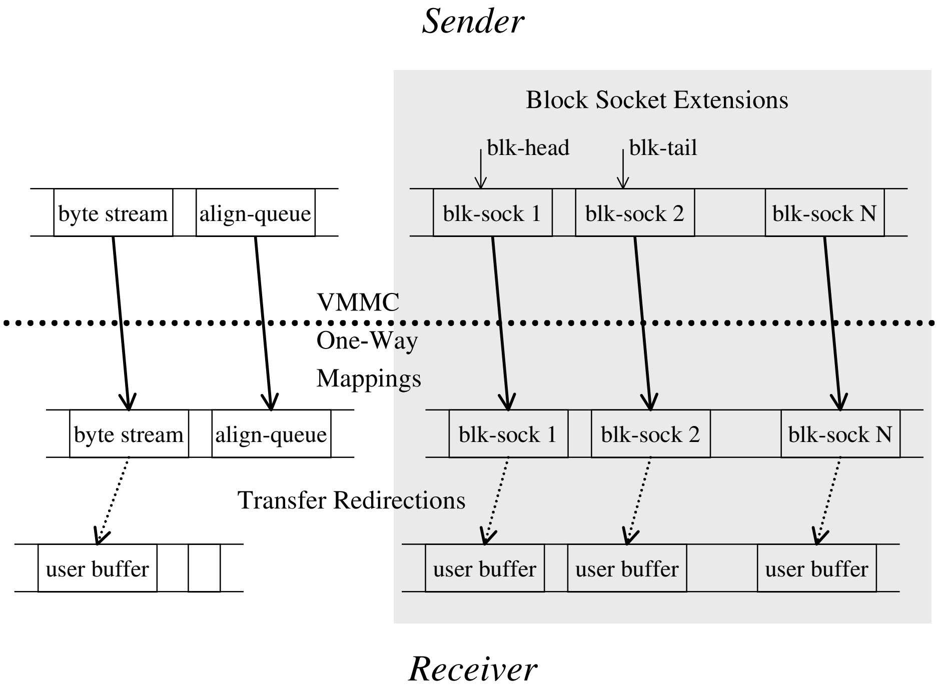


Figure 3.13: Block sockets architecture

the receiver can get messages (in the order received) by calling BlkSock_recv. This returns a handle to one of the buffers previously specified by the receiver (in first-in, first-out order). The additional default buffers are managed in circular queue fashion. Sending a block transfer message is identical to send except that a different function is required and the maximum message size cannot exceed the size specified when the block socket was created (Figure 3.13).

The block socket API is as follows:

- void BlkSock_add_socket (int sd, int num_mappings, int bufsize);

  given a VMMC-2 socket, creates the block socket extensions where num_mappings is the number of default buffers of size bufsize

- int BlkSock_add_buf (int sd, void *buf, int bufsize);

  makes buf the final buffer of the next available default buffer

- int BlkSock_send (int sd, void *buf, int bufsize);

  sends buf using the socket's next available block socket

- int BlkSock_recv (int sd, void **buf, int bufsize);

  returns buf, a handle to a buffer that contains a message. The message was either redirected or copied into the buffer.

The stream sockets extensions have three significant benefits:

- They optimize performance for block transfers. By distinguishing block transfers, the communication subsystem is able to extract better bandwidth performance from the network.
- The implementation scales well. The resource usage is significantly lower than the alternative of using multiple sockets, which require
- User code is simplified. Forcing the programmer to deal with multiple sockets is an unnecessary burden that complicates user programs.

3.6 Experiments

In order to understand the performance of VMMC-2, we measured the communication latency and bandwidth with micro-benchmarks, and analyzed VMMC-2 communication characteristics with the LogP model [20]. We also instrumented the VMMC-2 components to measure the overhead of transfer redirection.

3.6.1 Cost of Transfer Redirection

Transfer redirection incurs additional overhead, including the cost of posting a redirection request by the host, and computing new scatter lengths and addresses by the network interface. The cost to post a redirection request is about 6 μs, provided the virtual-to-physical translations for pages of the redirection destination buffer are available in the user-level translation look-aside buffer (UTLB). The UTLB allows the VMMC-2 firmware to quickly translate virtual addresses to physical addresses (see [17] for complete details). The additional work performed by the network interface takes about 2.5 μs per packet received. The Shrimp-M system can copy about 108 bytes in that time, so redirection is beneficial for all messages longer than 108 bytes (provided it is posted ahead of the message arrival).

3.6.2 VMMC-2 Micro-benchmarks

We present performance measurements for the VMMC-2 communication layer. Later, we use its performance as a basis to evaluate the efficiency of the stream sockets implementation.

The LogP Model

We characterize the performance of our system using the five parameters of the LogP model [20]: $\frac{RTT}{2}, L, O_r, O_s, and g$. $\frac{RTT}{2}$ is the one-way host to host latency for a ping-pong test. O_s, O_r are the host overheads of sending and receiving a message, respectively. L is the one-way network latency between network interfaces. Finally g is the time (or gap) between two successive sends and describes how fast messages can be pipelined through the system.

The value of g depends on the number of messages sent and the amount of buffer space on the network interface. The gap numbers presented in Figure 3.14 are for a large number of messages so the system reaches steady state. This forces messages to be consumed at the receiver before new ones can be sent.

The absence of a receive operation in VMMC-2 affects the methodology used to measure the LogP parameters [20]. O_r is almost zero, since receiving a message takes only one

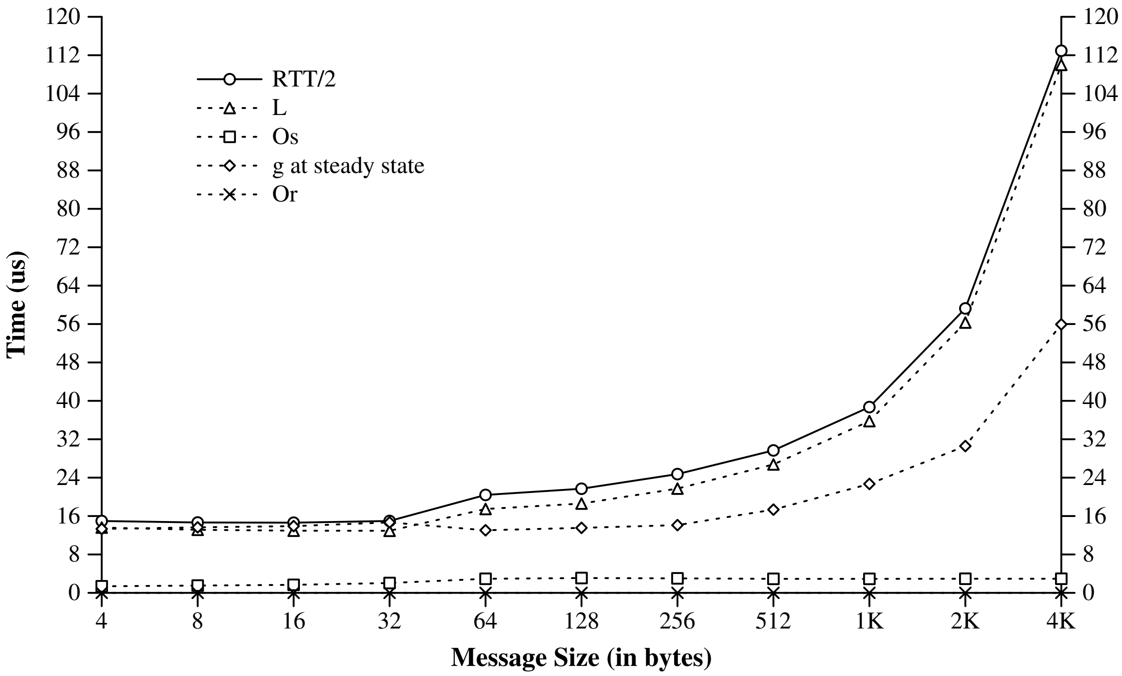


Figure 3.14: LogP parameters of VMMC-2 for various message sizes

instruction (to read a flag indicating that the message has arrived). Also, since VMMC-2 deposits data directly in remote memory, it is easy to measure the rest of the parameters, without the need to worry about consuming messages or generating replies at the receiver.

For O_s we only need to send a small number of messages to the receiver and measure the overhead at the sender. There is no need to wait for messages to be received since they will just be overwritten. If one wants to measure these parameters and have the messages received, things get more complicated because higher-level protocol needs to be used for synchronization. In VMMC-2 this higher-level protocol will be application specific, and it may depend on knowledge the programmer has for a specific application.

To measure g at steady-state, we send a large number of messages and average the overhead at the sender. As before, we do not need to worry about messages overflowing the receiver since they are not queued.

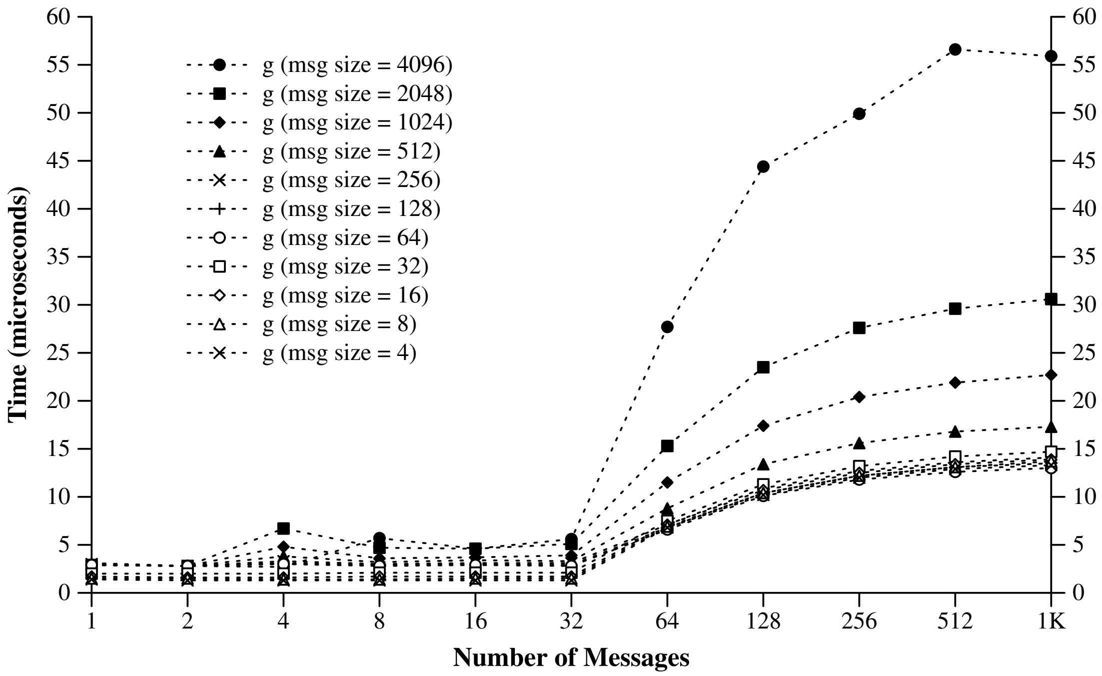
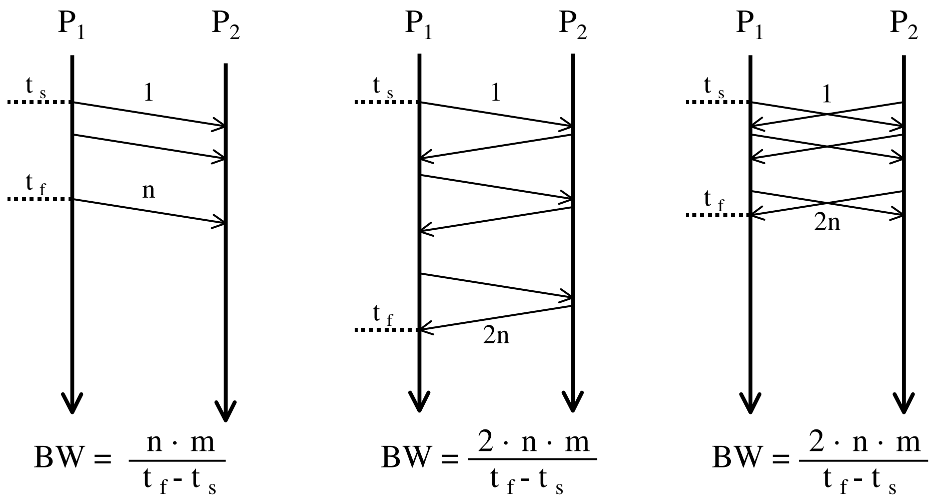


Figure 3.15: The LogP g parameter for various numbers of messages and message sizes (in bytes)

Figure 3.14 presents LogP parameters as a function of message size. When a message is sent using VMMC-2, data is written directly to the network interface send queue for sizes up to 32 bytes. For longer messages, DMA is used to transfer the data to the network interface. In both cases, the overhead to send a message is very small (below 4 μs). The one-way host-to-host latency ($\frac{RTT}{2}$) is 13.4 μs, and stays essentially flat for up to 32 bytes. Then it jumps up a lot because DMA is utilized. As expected, the latency, L, between network interfaces closely tracks $\frac{RTT}{2}$. In a steady state, g is related to the bandwidth of the slowest stage in the pipeline and is affected by the complexity of the VMMC-2 firmware.

Figure 3.15 shows how g changes as a function of the number of messages sent, until it reaches steady state. Since the size of the send queue is 32 entries, g is small (below 5 μs) when the number of messages is below this threshold. When more messages are sent, g starts growing until it reaches a steady state value that characterizes the sustained performance of the system.



(a) Unidirectional (b) Bidirectional (c) Bidirectional Simultaneous

Figure 3.16: Ping-pong communication patterns and how their bandwidths are computed using the parameters start time (t_s), finish time (t_f), message size (m), and number of messages (n)

Bandwidth Benchmarks

In addition to the LogP characterization we also ran three other ping-pong tests to measure the bandwidth of VMMC-2:

- *Unidirectional* is the bandwidth of a one way transfer. Data is flowing in one direction only and the sender does not need to wait for the receiver to acknowledge a message before it sends the next one. Figure 3.16(a)
- *Bidirectional ping-pong* benchmark data flows in both directions in a ping-pong fashion. After sending a message, each sender waits until it receives a message before sending again. One of the nodes starts as a sender and the other as a receiver. Figure 3.16(b)
- *Bidirectional simultaneous* is similar to the bidirectional ping-pong benchmark, only that both nodes start as senders. This benchmark can potentially use all three DMA

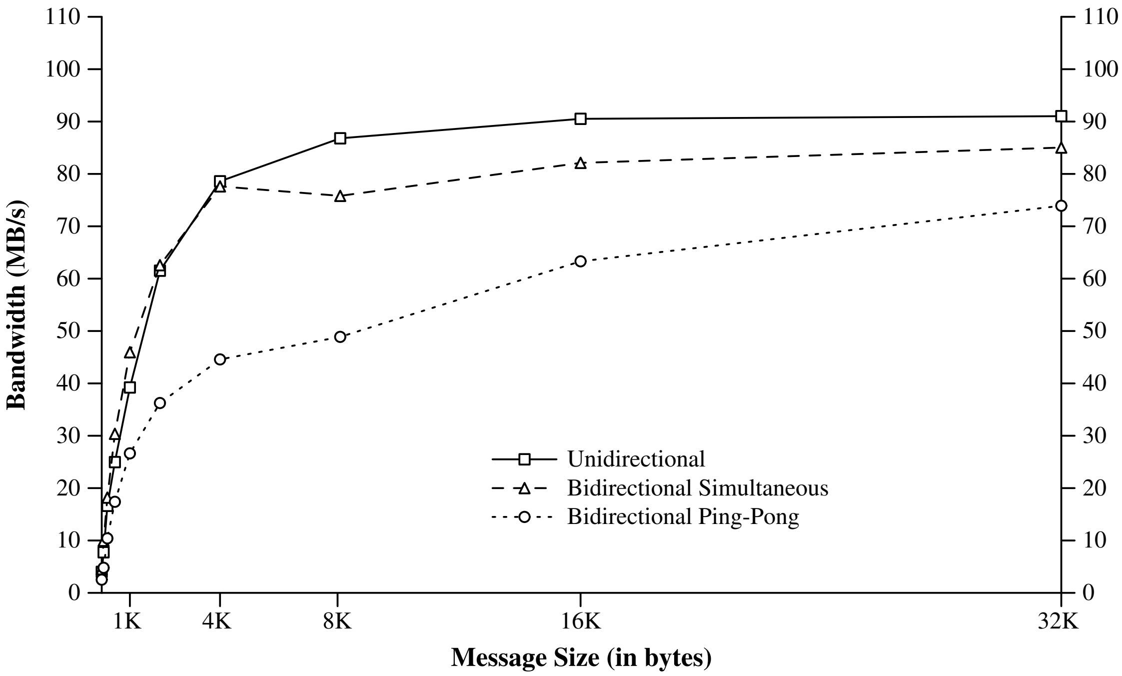


Figure 3.17: VMMC-2 bandwidth as a function of message size

engines of the network interface, depending on how the Myrinet firmware is written. Figure 3.16(c)

Figure 3.17 shows results for these three types of bandwidths. Bidirectional ping-pong bandwidth grows slower than the other two types because there is only one message in the network at any given time, so pipelining is less effective for short messages. For longer messages, data is transfered in 4 KBytes packet and bidirectional bandwidth catches up with the other two tests. The maximum bandwidth is in the range of 90-93 MBytes/s for all the tests (not shown).

3.6.3 Stream Sockets Micro-Benchmarks

In this section we present micro-benchmark results obtained using the VMMC-2 stream sockets implementation. The micro-benchmarks are similar to those in the previous section, except that they were ported to the stream sockets API. Using this API allowed us to obtain

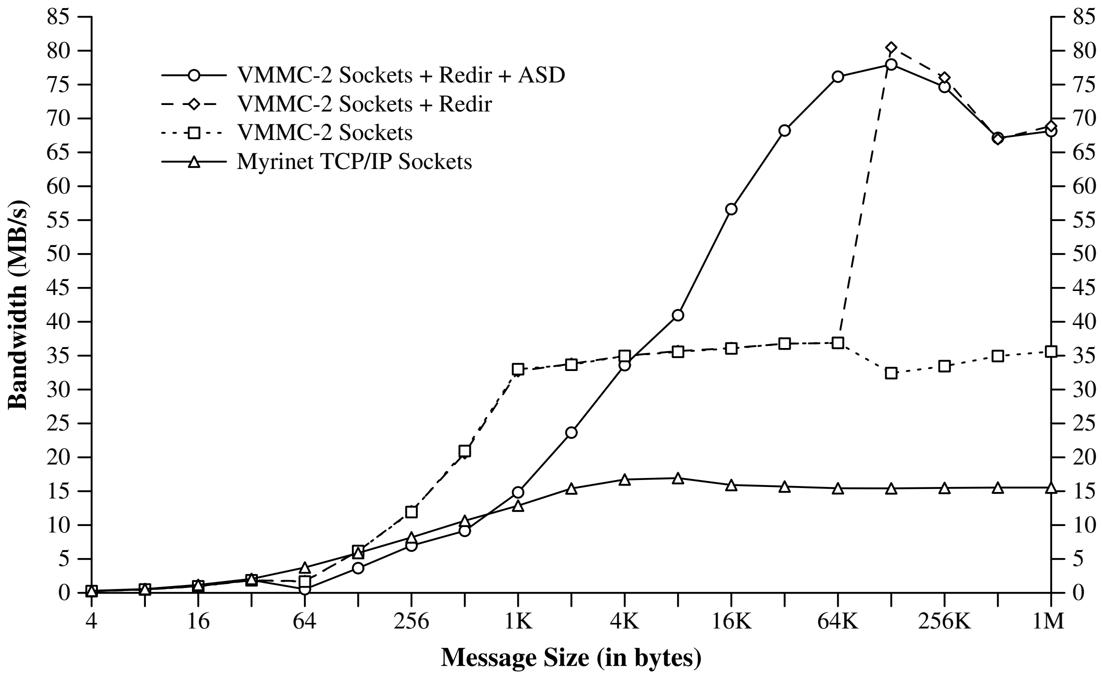


Figure 3.18: Sockets unidirectional ping-pong bandwidth

results for the kernel-level Myrinet-TCP implementation of stream sockets. We present those numbers only for reference as it is not entirely fair to compare kernel-level and user-level systems.

Figure 3.18 shows the stream sockets unidirectional bandwidth. Myrinet-TCP is limited by the kernel and can do no better than about 15 MBytes/s. Our basic VMMC-2 implementation uses 1-copy and achieve 35 MBytes/s (the copy bandwidth of our nodes). Enabling redirection results in no improvement until messages of 128 Kbytes in size, when the performance jumps to 80 Mbytes/s. This occurs due to the communication pattern of the one-way benchmark. The sender is continuously pumping messages to the receiver with no delays between send calls. This does not allow the receiver to take advantage of redirection. When a single message fills the 128 KByte internal socket circular buffer, the sockets layer throttles the sender to avoid over-running the buffer. As a result of this,

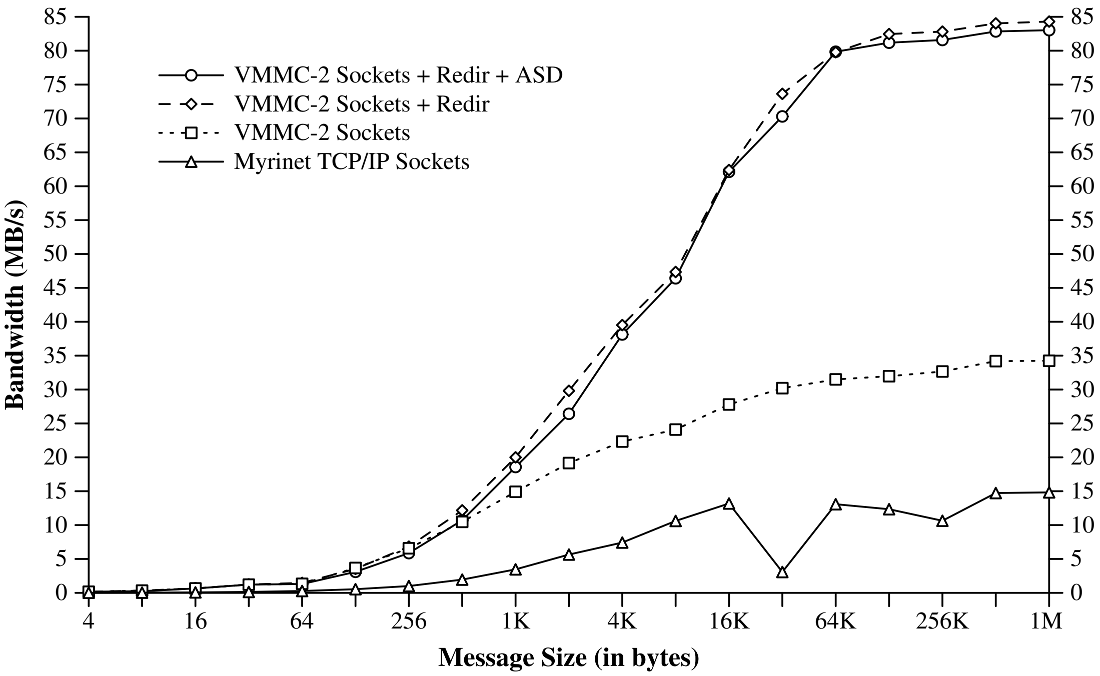


Figure 3.19: Sockets bidirectional ping-pong bandwidth

the sender waits between successive sends, and this slight delay allows the receive to take advantage of redirection in a manner similar to ASD.

The drop-off in the curve, for messages larger than 128 KBytes, is because only the first part of a very large message is redirected, while the rest of the message is copied. The reason for this is very subtle and is related to the semantics of sockets.

The send call must transfer the entire user-buffer before completing, but the receive call is supposed to receive as many bytes as it can, up to the size specified. receive returns the number of bytes received, which the application uses to decide how much more data to receive (if any at all). When the send message size is larger than the circular buffer, the semantics of send force it to fill as much of the buffer as possible and then wait for the receiver to free space before continuing to send. Further, send must transmit the buffer using multiple messages, each at most the size of the circular buffer. send does not complete until all of the data is sent. The receiver meanwhile has successfully redirected

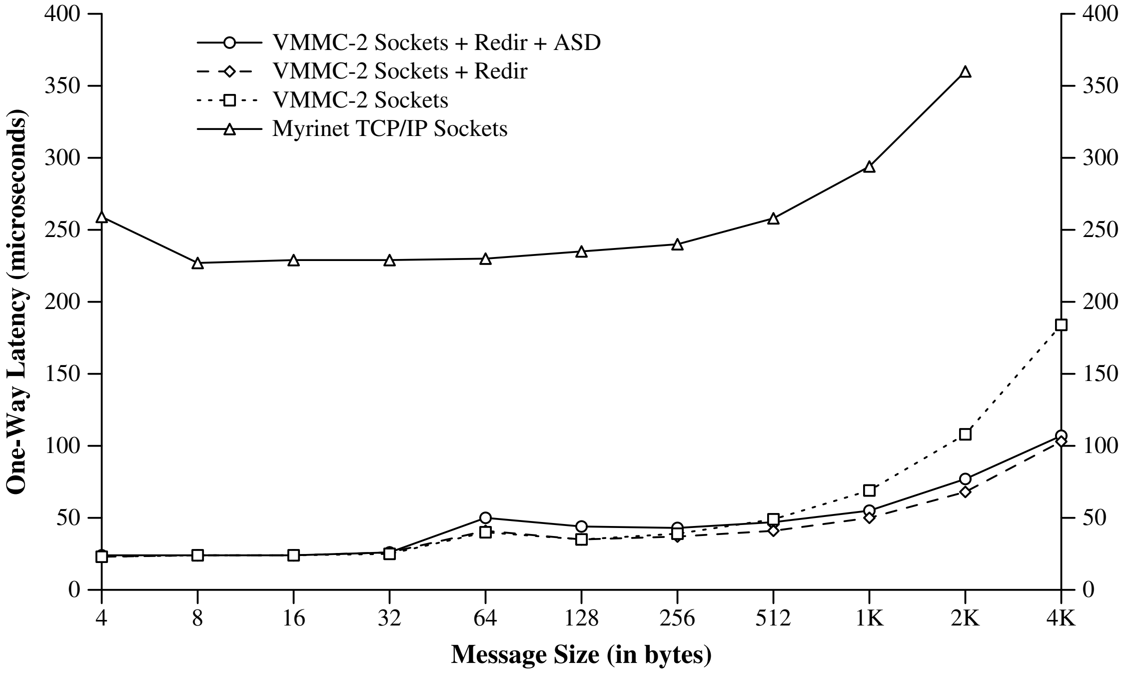


Figure 3.20: Sockets one-way latency using bidirectional ping-pong

the first message that was sent. It tells the sender that there is free space in the buffer and returns the number of bytes received—which is less than the amount requested—to the application. The application sees that not all the bytes were received and calls receive again. This time redirection fails because the sender was waiting and sent a message as soon as the buffer space was available.

The results of adding ASD to stream sockets for the unidirectional test are mixed. For messages smaller then 4 KBytes delaying the sender results in a net loss of performance. But for larger messages, the delay is offset by the avoidance of the receive side copy. The ASD implementation also suffers from the performance drop-off, for the same reasons discussed above for messages larger than the circular buffer.

Next, experiments with bidirectional ping-pong are shown in Figure 3.19. Myrinet-TCP is again limited by the kernel and can do no better than about 15 MBytes/s. Our basic VMMC-2 implementation again achieves the memory-copy bandwidth of the nodes.

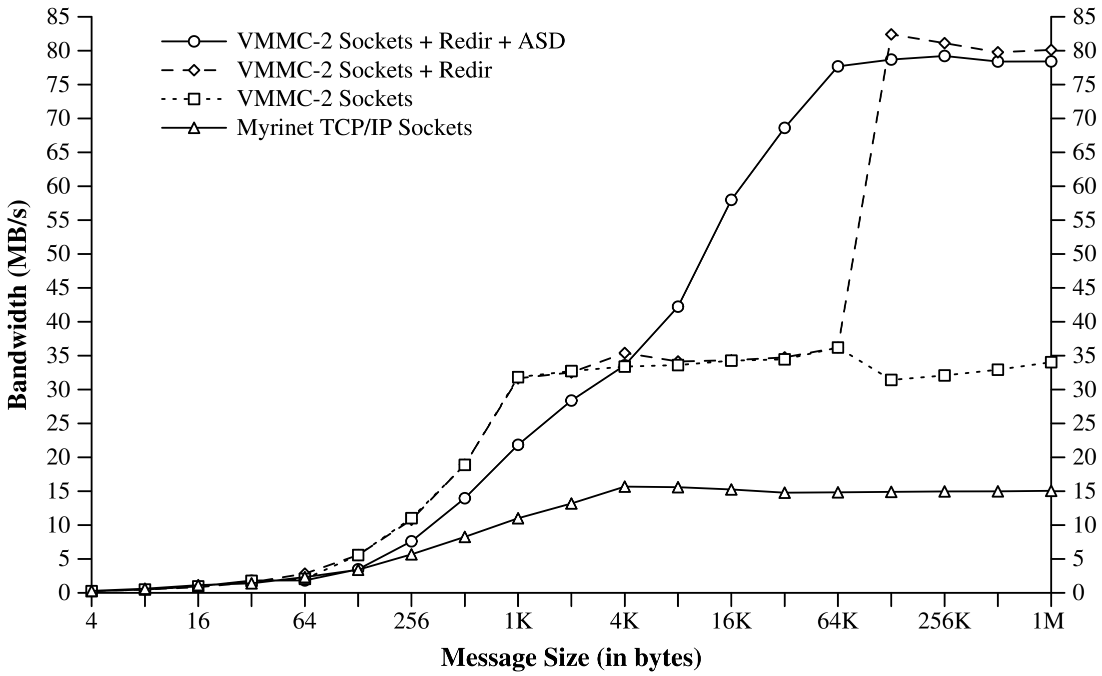


Figure 3.21: Ttcp bandwidth

We see the significant benefit of redirection since this communication pattern naturally delays senders and allows for full redirection of all messages. ASD neither helps nor hurts performance.

Using the bidirectional ping-pong test we also measured the small message latency, given in Figure 3.20. VMMC-2 sockets achieve a latency of 20 μs for short messages. Myrinet-TCP is an order of magnitude slower, primarily because it uses interrupts and system calls. The latency of all three variations of VMMC-2 sockets is identical for messages smaller than 32 bytes because both redirection an ASD are not used. From 64 to 256 bytes, the vanilla implementation performs slightly better than the other two. For messages sizes 512 bytes and larger redirection begins to show a significant advantage over the vanilla implementation. ASD performs just slightly worse than redirection.

3.6.4 Ttcp

We also measured the performance of our implementation using ttcp version 1.12, a public domain benchmark originally written at the Army Ballistics Research Lab [67]. ttcp measures network performance using a one-way communication test where the sender continuously pumps data to the receiver.

The results in Figure 3.21 are almost identical to the results of the unidirectional ping-pong micro-benchmark (Figure 3.18) with one exception: there is no significant drop-off in performance for messages larger then 128 KBytes. This is because ttcp sends large messages as a sequence of smaller messages, i.e., several send calls. This makes the large message communication pattern identical to the bidirectional micro-benchmark. Hence, for the same reason, there is no drop off in performance.

3.6.5 Tftp

Tftp (Trivial File Transfer Protocol) implements only part of the functionality provided by FTP. Tftp consists of a simple daemon and client. We run tftp on top of both Myrinet-TCP and VMMC-2 sockets.

In our experiment, the server reads a file from the file system in blocks of b bytes and sends each block to the client. The client receives each block and writes it to the file system. Since a file is broken down in blocks of size b, the operation of the server and the client are pipelined. All calls to send and receive data over the network are synchronous. However, there is some pipelining between the sender, the network, and the receiver since transfers are in progress at the same time in these stages of the data path. We set the size of blocks, b, to the value that gives the best results for each communication subsystem: 4 KBytes for Myrinet TCP and 64 KBytes for VMMC-2 sockets. In the experiments we use the standard Linux file system with local disks. File system operations (reads, writes) are cached.

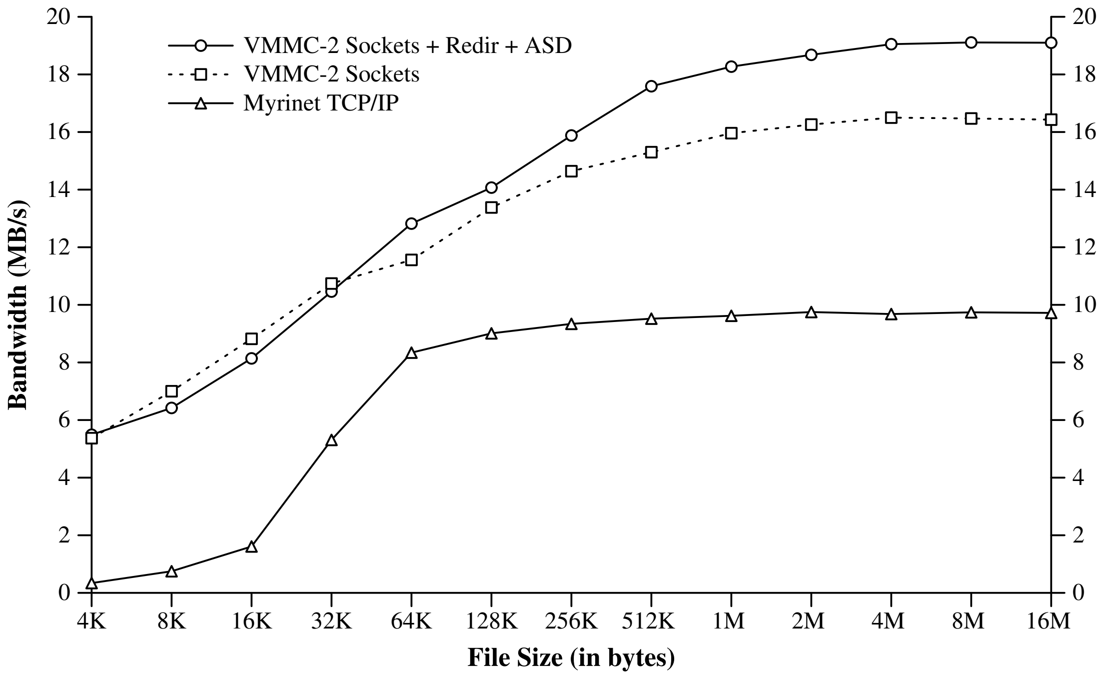


Figure 3.22: Tftp bandwidth using get operation

Figure 3.22 shows the get bandwidth for files of different sizes. The slowest phase in the pipeline is the write system call which writes a received block to the file cache. We measured the bandwidth of this operation to be 25 MBytes/s, so even with an infinitely fast network, the bandwidth of tftp would be limited to this number. The maximum bandwidth achieved by VMMC-2 sockets is 19 MBytes/s.

3.6.6 PARFUM

The PARFUM Parallel Fault Tolerant Volume Renderer [3] is a dynamically load-balanced parallel volume renderer executing in a distributed environment. PARFUM is based on a traditional ray-casting algorithm for rendering volumetric data sets. It consists of a controller processor that implements a centralized task queue, and a set of worker processors that remove tasks from the queue, process them, and send the results back to the controller

Task Size	Myrinet TCP	VMMC-2 Sockets
1,000	9.44	8.97
250	10.79	9.16
100	14.90	9.11
50	23.27	9.32
10	120.48	10.24
5	261.92	11.76
1	1,302.68	26.11

Table 3.3: PARFUM, Elapsed time in seconds for 3 worker processors and different task sizes

processor. The data set is replicated in all worker processors and is loaded at connection establishment. PARFUM is a latency sensitive application.

The task size is an important parameter of the system since it determines the tradeoff between load balance and communication. The smaller the task size, the more communication occurs and the better the load balance that can be achieved. In all of our experiments, the data set used is a three dimensional grid (167x63x34) that represents the salinity of the Mediterranean sea. The size of the output image is 640x400 pixels.

Table 3.3 presents the elapsed time for three worker processors on Myrinet TCP and VMMC-2 sockets for varying task sizes. We see that for this application good performance can be achieved with all communication systems, given that the task size is chosen appropriately. But with VMMC-2 it is never possible to be worse than a factor of three from the best case, whereas Myrinet TCP varies by 2 orders of magnitude.

The final PARFUM experiment uses a task size of one pixel. Although this small task size is not practical, it stresses the communication system by significantly increasing the

# workers	100 Mbit Eth	Myrinet TCP	VMMC-2 Sockets
1	3,835.23	3,436.90	59.90
2	1,942.99	1,280.74	33.73
3	1,032.40	1,302.68	25.68
4	978.04	989.97	22.17

Table 3.4: PARFUM, Elapsed time in seconds for different numbers of worker processors and a task size of one

amount of communication. Under these conditions, PARFUM emulates an implementation of a remote centralized queue. Table 3.4 presents the elapsed time for different number of worker processors on 100 Mbit Ethernet TCP/IP, Myrinet TCP and VMMC-2 sockets, which outperforms the other communication systems.

3.7 Related Work

There is a large body of literature on improving communication performance. Most related work is in using protected, memory-mapped communication or other user-level mechanisms to support reliable, connection-oriented high-level protocols.

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. [59] who have a user-level implementation of sockets using *active messages* [69]. 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 [68] 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. [12] 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) [56] 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 [70] 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 [13, 38, 21]. 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 [48, 35]. 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 VMMC. Bilas [6] describes two implementations of remote procedure call, one that is compatible with SunRPC [41] 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 [57] message-passing library for VMMC; it also achieves latency similar to ours.

Applications using stream sockets can effectively utilize the redirection mechanism. For example, over 94% of the receives in a file system using stream sockets used the redirection mechanism. We improved the effectiveness of transfer redirection using ASD an adaptive delay derived from a competitive, random walk algorithm.

Our main conclusions are that VMMC-2 can deliver to the user performance quite close to the hardware limit and that these mechanisms support the stream sockets quite well.

Popular communication models are almost exclusively based on connection-based APIs like stream sockets or RPC. The performance advantage of new Gigabit networks like Myrinet has not been delivered to user space because previous implementations of these APIs incur high overhead and multiple data copies. This remains true even for user-level implementations on top of thin communication models like FM or AM. For example, FM's implementation of sockets requires two copies, while AM's requires one.

Spector [64] proposed a remote memory reference model to perform communication over a local area network and his implementation is programmed in a processor's microcode. This model has been revived by Thekkath et al. [65] using fast traps. They did not deal with the issues of supporting connection-oriented high-level protocols.

Druschel et al. [27] proposed the concept of application device channels which provide protected user-level access to a network interface. U-Net [4] uses a similar abstraction to support high-level protocols such as TCP/IP. Their implementation requires applications to use special, pinned buffers. Hamlyn uses a sender-based communication model and uses a network interface to support user-level message passing [15], but it requires application programs to build headers using pinned memory. They did not address the issues of supporting connection-oriented high-level protocols.

Active Messages(AM) [33] is a user-level communication mechanism based on fast remote execution facility. Recently, a fast sockets facility has been built with AM handlers [59]. They used a receiver-based approach called *receive posting*. The arriving data is first placed in an anonymous staging area in memory. Posting a receive will move the data to the final destination. FastSockets achieves 30 μs one-way latency and 33 MBytes/s maximum bandwidth between two UltraSPARC 1 connected by a Myrinet network. Our transfer redirection is different, our default buffer is used only when the receive address is not posted in time.

Fast Messages(FM) [54] is similar to AM in that each message is associated with a handler that will process the message on the receive side. FM does not allow multiple processes to share the network access. The FM sockets implementation [55] achieves one-way latency of about 34 μs and maximum bandwidth of 11 MBytes/s between two SPARCstations connected with Myrinet.

Many methods have been proposed to avoid copying between I/O devices and application memory. IBM's OS/360 can specify a file I/O buffer area for applications to avoid copying. Page remapping has been used to avoid copying in several past and recent projects [18, 28, 42, 46, 49]. Lately, several groups have proposed letting applications directly access communication interface memory [19, 23, 62].

The Nectar system [19] implements TCP on a programmable network interface, with a 16 MHz Sparc as the communication processor. S-Connect implements a retransmission protocol in hardware to provide reliable communication. They obtain 90% of the hardware bandwidth between two network interfaces, but only 30% between between user applications. The significant performance reduction is due to protocol software overhead.

Brustoloni et al [14] proposed a novel approach that avoids copying by swapping kernel pages and user pages. But this approach requires system calls and page protection manipulations that are expensive.

3.8 Conclusions

Transfer redirection is a mechanism that allows receivers to control the ultimate destination of messages without requiring a copy or a scout message. The effectiveness of this mechanism depends on the the receiver knowing the destination of a message before it arrives. When this is the case, the incoming message is not copied, otherwise all, or part, of the message must be copied.

For our user-level stream sockets implementation, redirection can completely avoid copying while delivering over 93% of the available bandwidth to applications that post redirection requests before messages arrive. By comparison, the kernel sockets implementation sees only 16% of the bandwidth while the one-copy user-level version sees only 39%. Redirection is able to almost triple the bandwidth by avoiding copying.

A ping-pong benchmark was used to study the user-level stream sockets implementation running on top of VMMC-2. In this experiment, we find some interesting effects in the unidirectional case. The bandwidth of the basic VMMC-2 mechanism never exceeds the memory copy bandwidth of about 35 MBytes/s. The transfer redirection mechanism proves to be ineffective until the message size reaches the sockets buffer size (128 KBytes) because the receiver is never able to post a redirection before a message arrives. Once the buffer size is reached (and exceeded), the redirections are posted in time and zero-copy communication is achieved. Because of this behavior, we developed the ASD mechanism which forces senders to delay in order to allow receivers to post redirections.

In evaluating ASD we find that it helps larger messages (greater than 4 KBytes), but hurts smaller ones. Unfortunately, the loss of bandwidth for small messages can be quite sever (about 50% for messages below 1 KByte in size). Although this problem has not been solved, there are some avenues to explore in the future. For instance, knowledge of applications' transfer characteristics would allow the sockets implementation to decide to use ASD of not. Also, there is additional work to be done on tuning the ASD algorithm and implementation.

In the bidirectional stream sockets ping-pong experiment, we find that the delay imposed on senders by the communication pattern is sufficient to allow receivers to post redirections. Therefore, we get good performance without ASD for the entire range of message sizes. This is encouraging because the delay between messages is still quite small in this dedicated application, so it suggests that redirection will be effective at eliminating copying even when applications communicate quite frequently.

Latency measurements show that the transfer redirection mechanism and ASD have a positive effect on message latency, especially as the message size grows. Again this occurs because the receiver avoids copying.

As expected, experiments with the standard ttcp benchmark confirm previously discussed results of the sockets unidirectional transfer experiment. Experiments with tftp show that the bandwidth of VMMC-2 sockets approaches the limit imposed by the unavoidable disk-write system call.

The block-transfer stream sockets extension presented in this chapter is a way to improve the concurrency of the redirection mechanism. A pair of communicating processes can have many outstanding redirection requests using a single socket. Further, since sockets merges all messages into a single stream it is more difficult to take advantage large transfers. By using separate call for large message transfers, the application provides the communication subsystem with information that enables high performance communication.

In summary, connection-oriented communication can be efficiently supported by user-level communication using the transfer redirection mechanism.

Chapter 4

Conclusions

This dissertation proposes two techniques for improving the performance of connection-oriented communication on fast, new-generation networks. The first technique is a competitive spin-block mechanism for detecting the arrival of user-level messages. This technique reduces receiver overhead by avoiding unnecessary interrupts and protection boundary crossings. The second technique is a user-level transfer redirection mechanism that gives the receiver dynamic control over the ultimate destination of an incoming message in order to avoid unnecessary copying in the the protocol.

Competitive spin-blocking reduces the CPU overhead required to detect data arrival in user-level communication. This dissertation focuses on two key techniques to achieve low CPU overhead: combining polling with the process-blocking mechanism, and reducing overhead for both servicing message-driven interrupts and awakening sleeping processes. The two approaches lead to the design of the spin-block mechanism that utilizes the message-driven interrupt facility and kernel scheduler to achieve low overhead for message arrival detection.

We evaluated this mechanism along with several other strategies for various workloads in a multi-programming environment. Our experiments show that the combination of polling and blocking gives consistently good performance in a wide variety of situations.

Further, reducing the interrupt service time, in addition to the number of unnecessary reschedules, results in significant performance improvement.

Chapter 3 explored the issues of trust and timing that make connection-oriented communication fundamentally different than other message passing models. Several modifications to the low level data transfer protocol were implemented in order to support efficient user-level, connection-oriented protocols. These modifications were used to eliminate unnecessary message copying and scout messages by empowering the receiver with the ability to control the ultimate destination of a direct data transfer.

We implemented transfer redirection in the user-level VMMC-2 model running on the Shrimp-M system. The effectiveness of transfer redirection is dependent upon the relative timing of the receive call and the message arrival time. When a redirection request is posted before a message arrives no copying is needed, otherwise all, or part, of the message must be copied.

We found that transfer redirection can be extremely effective for some applications, and not very effective for others. For those cases that redirection was not effective, the performance was equal to that of the traditional copy implementation.

To improve the effectiveness of transfer redirection we implemented a mechanism called adaptive send delay (ASD) that delays the sending of messages in order to allow the receiver to invoke redirection.

The overall communication performance of the VMMC-2 implementation is quite good. Its one-way latency is 13.5 μs for a small message and it achieves an one-way bandwidth of 80 Mbytes/s for messages of 4 Kbytes and over 90 Mbytes/s for messages larger or equal to 16 Kbytes. Stream sockets on top of VMMC-2 also performs quite well. Its one-way latency is 20 μs and its peak bandwidth is over 84 Mbytes/s.

The small message latency of VMMC stream sockets is 20 μs on Shrimp-M. This is an order of magnitude better than the 259 μs required by the kernel-based TCP/IP implementation on the same platform. Small message latency is significantly reduced but its overhead is still high compared to the 13.5 μs latency of VMMC-2. A future challenge is to develop an approach that reduces the 6.5 μs (50%) overhead. This might involve specialized support from next-generation network interfaces.

A significant source of software complexity in stream sockets is because byte transfers are not supported by the network interface. Providing network interface support for byte transfers would significantly reduce the overhead of unaligned transfers.

Implementing a current (legacy) connection-oriented protocol on new-generation networks exposed a factor that limits high performance. Current protocols, like WinSock [72] and stream sockets lack block data transfer primitives. In Chapter 3 we presented block transfer extensions to stream sockets that significantly improve application performance. Future connection-oriented models would benefit from incorporating block data transfer primitives.

Improving connection-oriented communication with the ideas presented in this dissertation has made it a viable communication model for current high-performance system area networks. It allows large applications to be divided into safe, manageable pieces with separate address spaces. Stream sockets is a well known, and popular, connection-oriented API that can be used to modularize large applications without sacrificing performance. For example, the distributed file system described in Chapter 2 is used by a web server[1]. The web server and filesystem communicate with, and between, each other using user-level stream sockets with the block transfer extensions.

[1] Currently under development

Bibliography

[1] R. Alpert, C. Dubnicki, E. W. Felten, and K. Li. Design and implementation of NX message passing using shrimp virtual memory mapped communication. Technical Report TR-507-96, Dept. of Computer Science, Princeton University, Jan. 1996.

[2] T. E. Anderson, H. M. Levy, B. N. Bershad, and E. D. Lazowska. The interaction of architecture and operating system design. In *Proceedings of 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 108–120, 1991.

[3] J. Asplin and S. Mehus. On the design and performance of the parfum parallel fault tolerant volume renderer. Technical Report 97-28, Univerity of Tromso, Norway, Jan. 1997.

[4] A. Basu, B. V, V. W, and v. E. T. U-net: A user-level network interface for parallel and distributed computing. *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP), Copper Mountain, Colorado*, pages 40–53, December 1995.

[5] A. Bilas, Y. Chen, C. Dubnicki, S. N. Damianakis, and K. Li. Firmware support for reliable communication and dynamic system configuration in system area networks. Technical Report TR-581-98, Dept. of Computer Science, Princeton University, April 1998.

[6] A. Bilas and E. W. Felten. Fast RPC on the shrimp virtual memory mapped network interface. Technical Report TR-512-96, Dept. of Computer Science, Princeton University, Feb. 1996.

[7] M. 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.

[8] M. A. Blumrich, R. D. Alpert, Y. Chen, D. W. Clark, S. N. Damianakis, C. Dubnicki, E. W. Felten, L. Iftode, K. Li, M. Martonosi, and R. A. Shillner. Design choices in the shrimp system: An empirical study. In *Proceedings of 25rd International Symposium on Computer Architecture*, June 1998.

[9] M. A. Blumrich, C. Dubnicki, E. Felten, K. Li, and M. Mesarina. Virtual-memory-mapped network interfaces. *IEEE MICRO*, 15(1):21–28, Feb. 1995.

[10] M. A. Blumrich, C. Dubnicki, E. W. Felten, and K. Li. Protected, user-level DMA for the SHRIMP network interface. In *Proc. 2nd Intl. Symposium on High-Performance Computer Architecture*, pages 154–165, Feb. 1996.

[11] M. A. Blumrich, K. Li, R. Alpert, C. Dubnicki, E. Felten, and J. Sandberg. Virtual memory mapped network interface for the shrimp multicomputer. In *Proceedings of 21st International Symposium on Computer Architecture*, pages 142–153, Apr. 1994.

[12] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W.-K. Su. Myrinet: A gigabit-per-second local area network. *IEEE Micro*, 15(1):29–36, Feb. 1995.

[13] S. Borkar, R. Cohn, G. Cox, T. Gross, H. 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.

[14] J. Brustoloni and P. Steenkiste. Effects of buffering semantics on i/o performance. In *Proceedings of the Second USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Oct. 1996.

[15] G. Buzzard, D. Jacobson, M. Mackey, S. Marovich, and J. Wilkes. An implementation of the hamlyn sender-managed interface architecture. In *Proceedings of the Second USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 245–260, Oct. 1996.

[16] G. Buzzard, D. Jacobson, S. Marovich, and J. Wilkes. Hamlyn: A high-performance network interface with sender-based memory management. In *Proceedings of Hot Interconnects '95 Symposium*, Aug. 1995. Also available as HP Laboratories technical report HPL-95-86, July, 1995.

[17] Y. Chen, C. Dubnicki, S. N. Damianakis, A. Bilas, and K. Li. Utlb: An address translation mechanism for network interfaces. Technical Report TR-580-97, Dept. of Computer Science, Princeton University, April 1998.

[18] D. R. Cheriton. The unified management of memory in the v distributed system. Technical Report STAN–CS–88–1192, Stanford University, Aug. 1988.

[19] E. Cooper, P. Steenkiste, R. Sansom, and B. Zill. Protocol implmentation on the nectar communication processor. In *Proceedings of the ACM SIGCOMM'90 Symposium*, Sept. 1990.

[20] D. E. Culler, L. T. Liu, R. P. Martin, and C. Yoshikawa. LogP Performance Assessment of Fast Network Interfaces. *IEEE Micro*, 16(1):35–43, February 1996.

[21] W. J. Dally. The j-machine system. In P. Winston and S. Shellard, editors, *Artificial Intelligence at MIT: Expanding Frontiers*, pages 550–580. MIT Press, 1990.

[22] W. J. Dally and C. L. Seitz. Deadlock free message routing in multiprocessor interconnection networks. *IEEE Transactions on Computers*, C-36(5):547–553, May 1987.

[23] C. Dalton, G.Watson, D. Banks, C. Calamvokis, A. Edwards, and J. Lumley. Afterburner. *IEEE Network*, 7(4):36–43, 1995.

[24] S. N. Damianakis, A. Bilas, C. Dubnicki, and E. W. Felten. Client-server computing on shrimp. *IEEE Micro*, 17(1):8–18, Jan. 1997.

[25] S. N. Damianakis, Y. Chen, and E. W. Felten. Reducing waiting costs in user-level communication. In *Proceedings of the 1997 International Parallel Processing Symposium*, Apr. 1997.

[26] S. N. Damianakis, C. Dubnicki, and E. W. Felten. Stream sockets on shrimp. In *Proceedings of 1st International Workshop on Communication and Architectural Support for Network-Based Parallel Computing (Proceedings available as Lecture Notes in Computer Science 1199)*, pages 16–30, Feb. 1997.

[27] P. Druschel, B. S. Davie, and L. L. Peterson. Experiences with a high-speed network adapter: A software perspective. In *Proceedings of SIGCOMM '94*, pages 2–13, September 1994.

[28] P. Druschel and L. L. Peterson. Fbufs: A high-bandwidth cross-domain transfer facility. In *Proceedings of the 14th Symposium on Operating Systems Principles*, pages 189–202, December 1993.

[29] C. Dubnicki, A. Bilas, Y. Chen, S. N. Damianakis, and K. Li. Vmmc-2: Efficient support for reliable, connnection-oriented communication. In *IEEE Hot Interconnects V*, Aug. 1997.

[30] C. Dubnicki, A. Bilas, K. Li, and J. Philbin. Design and implementation of virtual memory-mapped communication on myrinet. In *Proceedings of the 1997 International Parallel Processing Symposium*, pages 388–396, Apr. 1997.

[31] C. Dubnicki, L. Iftode, E. W. Felten, and K. Li. Software support for virtual memory-mapped communication. In *Proceedings of the IEEE 8th International Parallel Processing Symposium*, Apr. 1996.

[32] A. C. Dusseau, R. H. Arpaci, and D. E. Culler. Effective distributed scheduling of parallel workloads. In *Proceedings of 1996 ACM SIGMETRICS*, May 1996.

[33] T. Eicken, D. Culler, S. Goldstein, and K. Schauser. Active messages: A mechanism for integrated communication and computation. In *Proceedings of 19th International Symposium on Computer Architecture*, pages 256–266, May 1992.

[34] E. W. Felten, R. Alpert, A. Bilas, M. A. Blumrich, D. W. Clark, S. N. Damianakis, C. Dubnicki, L. Iftode, and K. Li. Early experience with message-passing on the shrimp multicomputer. In *Proceedings of 23rd International Symposium on Computer Architecture*, May 1996.

[35] FORE Systems. *TCA-100 TURBOchannel ATM Computer Interface, User's Manual*, 1992.

[36] R. B. Gillet, M. Collins, and D. Pimm. Overview of network memory channel for pci. In *Proceedings of the IEEE COMPCON '96*, pages 244–249, 1996.

[37] R. B. Gillett. Memory channel network for PCI. *IEEE Micro*, 16(1):12–18, Feb. 1996.

[38] D. S. Henry and C. 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, Oct. 1992.

[39] Hewlett-Packard Company. *Public Netperf Homepage and Database*. http://www.cup.hp.com/netperf/NetperfPage.html.

[40] Intel Corporation. *Paragon XP/S Product Overview*, 1991.

[41] Internet Request for Comments RFC 1057, Internet Network Working Group. *RPC: Remote Procedure Call Protocol Specification, Version 2*, June 1988.

[42] D. Johnson and W. Zwaenepoel. The peragrine high-performance rpc system. *Software Practice and Experience*, 23(2):201–221, Feb. 1993.

[43] A. Karlin, K. Li, M. Manasse, and S. Owicki. Emprical studies of competitive spinning for a shared-memory multiprocessor. In *Proceedings of 13th ACM Symposium on Operating Systems Principles*, pages 41–55, Oct. 1991.

[44] A. Karlin, M. Manasse, L. McGeoch, and S. Owicki. Competitive randomized algorithms for non-uniform problems. In *1st Annual ACM Symposium on Discrete Algorithms*, pages 301–309, 1989.

[45] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy. The stanford flash multiprocessor. In *Proceedings of 21st International Symposium on Computer Architecture*, pages 302–313, Apr. 1994.

[46] P. Leach, P. Levine, B. Douros, J. Hamilton, D. Nelson, and B. Stumpf. The architecture of an integrated local network. *IEEE Journal on Selected Areas in Communications*, SAC-1(5), 1983.

[47] S. J. Leffler, M. K. McKusick, M. J. Karels, and J. S. Quarterman. *The Design and Implementation of the 4.3BSD Unix Operating System*. Addison Wesley, 1989.

[48] C. Leiserson, Z. Abuhamdeh, D. Douglas, C. Feynman, M. Ganmukhi, J. Hill, D. Hillis, B. Kuszmaul, M. S. Pierre, D. Wells, M. 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.

[49] J. Lumley. A high-throughput network interface for a risc workstation. Technical Report Technical Report HPL-92-7, Hewlett-Packard Laboratories, Jan. 1992.

[50] C. Maeda and B. N. Bershad. Protocol service decomposition for high-performance networking. In *Proceedings of 14th ACM Symposium on Operating Systems Principles*, pages 244–255, Dec. 1993.

[51] O. Maquelin, G. R. Gao, H. H. Hum, K. B. Theobald, and X.-M. Tia. Polling watchdog: Combining polling and interrupts for efficient message handling. In *Proceedings of 23rd International Symposium on Computer Architecture*, pages 179–188, May 1996.

[52] E. P. Markatos and M. G. H. Katevenis. Telegraphos: High-performance networking for parallel processing on workstation clusters. In *Proceedings of IEEE 2nd International Symposium on High-Performance Computer Architecture*, pages 144–153, Feb. 1996.

[53] J. K. Ousterhout. Why aren't operating systems getting faster as fast as hardware? In *Proceedings of Summer 1990 USENIX Conference*, pages 247–256, June 1990.

[54] S. Pakin, M. Buchanan, M. Lauria, and A. Chien. The fast messages (fm) 2.0 streaming interface. Usenix'97, 1996.

[55] S. Pakin, V. Karamcheti, and A. Chien. Fast messages (fm): Efficient, portable communication for workstatin clusters and massively-parallel processors. *IEEE Concurrency*, 1997.

[56] S. Pakin, M. Lauria, and A. Chien. High performance messaging on workstations: Illinois fast message (fm) for myrinet. In *Supercomputing '95*, Nov. 1995.

[57] P. Pierce. The NX/2 operating system. In *Proceedings of Third Conference on Hypercube Concurrent Computers and Applications*, pages 384–390, Jan. 1988.

[58] P. Pierce. The paragon implementation of the nx message passing interface. In *Proceedings of 9th Conference on Hypercube Concurrent Computers and Applications*, 1994.

[59] S. H. Rodrigues, T. E. Anderson, and D. E. Culler. High-performance local-area communication with fast sockets. In *Proc. of Winter 1997 USENIX Symposium*, Jan. 1997.

[60] A. Saini. An overview of the intel pentium processor. In *Proceedings of the IEEE COMPCON'93 Conference*, pages 60–62, Feb. 1993.

[61] R. A. Shillner and E. W. Felten. Simplifying distributed file systems using a shared logical disk. Technical Report TR-524-96, Dept. of Computer Science, Princeton University, Sept. 1996.

[62] J. Smith and C. Traw. Giving applications access to gb/s networking. *IEEE Network*, 7(4):44–52, July 1993.

[63] P. Sobalvarro, S. Pakin, W. Weihl, and A. Chien. Dynamic coscheduling on workstation clusters. In *Proceedings of the International Parallel Processing Symposium*, Mar. 1998.

[64] A. Z. Spector. Performing remote operations efficiently on a local computer network. *Communications of the ACM*, 25(4):260–273, Apr. 1982.

[65] C. A. Thekkath, H. M. Levy, and E. D. Lazowska. Separating data and control transfer in distributed operating systems. In *Proceedings of 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 2–11, Nov. 1994.

[66] R. Traylor and D. Dunning. Routing chip set for Intel Paragon parallel supercomputer. In *Proceedings of Hot Chips '92 Symposium*, Aug. 1992.

[67] US Army Ballistics Research Lab (BRL). *TTCP: A Test of TCP and UDP Performance*, Dec. 1984.

[68] T. von Eicken, A. Basu, V. Buch, and W. 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, Dec. 1995.

[69] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active messages: A mechanism for integrated communication and computation. In *Proceedings of 19th International Symposium on Computer Architecture*, pages 256–266, May 1992.

[70] J. Wilkes. Hamlyn – an interface for sender-based communications. Technical Report HPL-OSR-92-13, Hewlett-Packard Laboratories, Nov. 1992.

[71] A. W. Wilson Jr., R. P. LaRowe Jr., and M. J. Teller. Hardware assist for distributed shared memory. In *Proceedings of 13th International Conference on Distributed Computing Systems*, pages 246–255, May 1993.

[72] *Windows Sockets: An Open Interface for Network Programming under Microsoft Windows (Version 1.1)*, Jan. 1993.