Making TCP Viable as a High Performance Computing Protocol[*]

Patricia Gilfeather and Arthur B. Maccabe
Scalable Systems Lab
Department of Computer Science
University of New Mexico
pfeather@cs.unm.edu maccabe@cs.unm.edu

Abstract

Generally, TCP is reputed to be too slow and bloated to be useful in high performance computing. However, it has very definite advantages over proprietary or specialized protocols and could potentially be very useful in high performance computing. We researched different implementations of MPI over TCP to distinguish between the limitations of TCP and the limitations of implementations of MPI over TCP. Next, we describe a new approach to addressing the limitations of TCP: splintering. In contrast to OS bypass, the principle of splintering isn't that the operating system shouldn't be used, but that it should be used effectively. Finally, we describe the approach we will take in splintering processing for TCP packets.

1 Introduction

Clusters built from commodity hardware and software have several advantages over more traditional supercomputers. Commodity clusters are cheap and ubiquitous. They are easier to design, program and maintain. However, as high-speed networks reach 10Gb/s and processors reach 2–3GHz, new commodity clusters are unable to harness increases in power.

It is generally perceived that commodity protocols, like TCP, are themselves inefficient for use in large high-speed clusters. However, current implementation inefficiencies at higher level message passing layers or in the protocol stacks themselves hide any theoretical bottlenecks of the protocol. We tested several MPI implementations and found that none of the perceived difficulties with the TCP protocol were evident at the MPI layer.

Nonetheless, there are inefficiencies in the implementation of TCP that need to be addressed before TCP can be considered a good wire-level or end-to-end messaging layer for high-performance computing. Namely, we must address the memory copy from kernel to user space, the interrupt and scheduling of the kernel and the state necessary for maintenance of connections before we can consider TCP a viable solution.

One of the most effective tools for increasing the efficiency of high-performance computing has been the use of programmable Network Interface Cards (NICs). Offloading work to a programmable NIC has been an important tool in facilitating OS bypass. Improvements in networking technology have revealed the OS as a significant bottleneck in our ability to deliver low latency and high bandwidth to applications. The goal of OS bypass is to remove the OS from the communication path, thus eliminating costly interrupts and data copies between kernel space and user space. Ultimately, the OS must be involved in communication. As a minimum, the OS needs to be involved with the memory used for communication, e.g., validating the use of memory and making sure that pages are "pinned."

Instead of using bypass as a way to disengage the OS, our approach is to determine which functionality in the communication protocol stack presents the most benefits when offloaded. As we will illustrate later in the paper, some functions, like fragmentation and defragmentation or IP checksum, can be offloaded with positive results. However, other functions, like error handling, gain little from offloading and consume valuable NIC resources. We describe our proposal to *splinter* TCP, a commodity protocol stack. Splintering is the process of determining which functionality to extract from the protocol stack and distributing it. *By splintering the functionality of the TCP/IP stack, we expect to retain the advantages of commodity protocols and gain the performance efficiencies of appropriate offloading.*

Most importantly, splintering the communication stack means that communication rarely increases operating system activity. In other words, the act of communication doesn't cause the operating system to be invoked. First, we will discuss the advantages of commodity components.

[*]Los Alamos Computer Science Institute SC R71700H-29200001 and Albuquerque High Performance Computing Center through IBM SUR

Second, we will discuss operating system bypass and its successes and disadvantages. Third, we will introduce splintering and finally, we will propose a method for splintering the TCP stack.

2 Advantages of Commodity Components

Commodity-based hardware and software, including communication protocols, provide several advantages. They have been extensively developed and tested, they are highly interoperable, and they represent inexpensive alternatives to specialized solutions. The cost advantages for the commodity approach reaches far beyond the savings realized at time of purchase. Often code has already been created in the community so there is little to no development cost. In the remainder of this section, we discuss the advantages of commodity-based hardware and communication protocols.

2.1 Hardware

Fast Ethernet is a great example of the trade-offs between inexpensive commodity hardware and more expensive specialized solutions. While Fast Ethernet is very inexpensive, it is capable of only 1/10th the bandwidth of Myrinet[1]. Ethernet is, additionally, saddled with a very small transmission unit (1500 bytes) which is becoming more and more of a problem. In 1999, Gigabit Ethernet offered bandwidth that was comparable to Myrinet, but the cost for switches and NICs was high. In the last six months, the price of Gigabit Ethernet NICs has dropped from around $1000/NIC to $200/NIC[3, 5]. This is the true advantage of commodity-based hardware. While Myrinet has out-paced Gigabit Ethernet in performance, it does not promise near the reductions in costs that we see when components become a true commodity.

2.2 Network Layer Protocol

Internet Protocol (IP) is the ubiquitous network layer protocol. IP routing is absolutely necessary to remain interoperable in wide area networks. Additionally, IP routers remain the most cost-effective hardware choice and the IP routing mechanism is well-tested.

There are disadvantages to IP as well. IP is an old protocol and some of its assumptions are no longer valid. Generally, IP's hierarchical, dynamic routing is inefficient although LANs can maintain route table information and alleviate this weakness. Additionally, IP checksums reside in the header rather than at the end of the packet so computing the checksum requires maintaining additional state[9]. Also, headers are of various size requiring a check of the length field of each header.

Despite these difficulties, it is unlikely that any protocol (with the possible exception of IPv6) will replace IP as the commodity network layer protocol.

2.3 Transport Layer Protocols

TCP and its unreliable cousin User Datagram Protocol (UDP) are the most common transport layer protocols used today. TCP gives all of the advantages of a commodity protocol. Specifically, TCP is transparent to the layer above it, in this case the application layer. Thousands of applications are written based on the TCP protocol and interoperability would be severely limited without support for TCP.

On the other hand, TCP contains a large amount of communication processing overhead to administer flow control, error discovery and correction, and to maintain connections. Many of these services are unnecessary to the high-speed network application. These networks are highly reliable with little to no errors so the error detection and recovery would be less intrusive if it were not on the main path for all messages. Congestion control is maintained by the application so its existence in the transport layer is redundant.

3 Actual Bottlenecks of MPI over TCP

The perceived problems of TCP are the advertised flow control window and the slow start and three-way handshake associated with connection startup. However, currently, the implementation of high performance communication libraries hide any potential bottlenecks at the transport protocol layer.

We tested ping-pong latency over 100Mb/s Ethernet between two 933Mhz Intel Pentiums using Linux 2.4.2 and a warmed cache. We tested the MPICH 1-2.3 implementation over TCP, the LAM implementation using UDP against a raw ping-pong test with TCP sockets with the TCP_NODELAY option set.

The MPICH-1.23 implementation of MPI over TCP uses abstract device interfaces to allow for various underlying transport protocol layers. This makes the MPICH library very portable. TCP connections are established as needed meaning that there is a large variance in latency between the first message passed between two nodes and the second seen in Figure 1.

LAM MPI employs a user-level multi-process daemon that provides message-passing outside of the kernel at the user level using a UDP-based protocol. Because message-passing occurs at the user level, the daemon must be scheduled and scheduling variance will create latency variance. Figures 2 and 3 both show this latency variance.

Finally, the potential bottlenecks of TCP are not visible in Figures 2 and 3. Here, MPICH, LAM and TCP all have

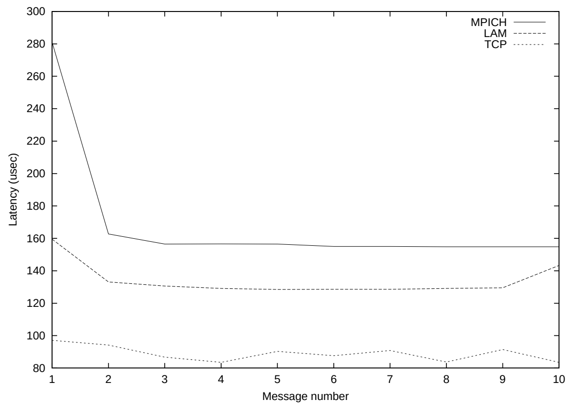

Figure 1. Ping-pong latency for MPICH, LAM and TCP on 50 byte messages

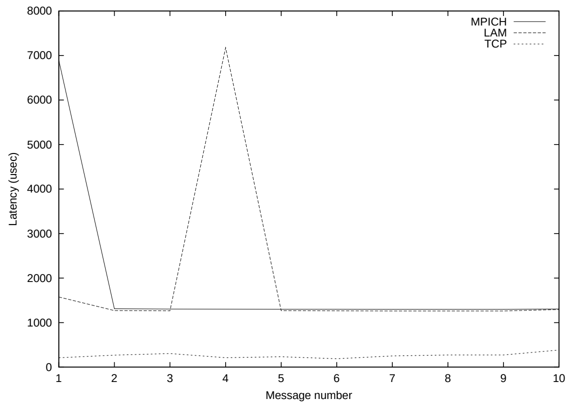

Figure 2. Ping-pong latency for MPICH, LAM and TCP on 5000 byte messages

established connections between the two nodes. The difference in latency between the first and second messages exhibited by MPICH is probably due to the library procuring kernel buffers. On the other hand, the differences in latency in LAM is probably due to scheduling of the LAM daemon.

4 Future Performance Enhancements

There are several solutions to the bottlenecks discussed in the previous section. Researchers have implemented MPI over other protocols and this has proved successful. Researchers can also implement other message-passing protocols over TCP. Finally, researchers can attempt to improve the mapping between the MPI message-passing protocol and the TCP protocol.

Implementations of MPI over other protocols like GM or

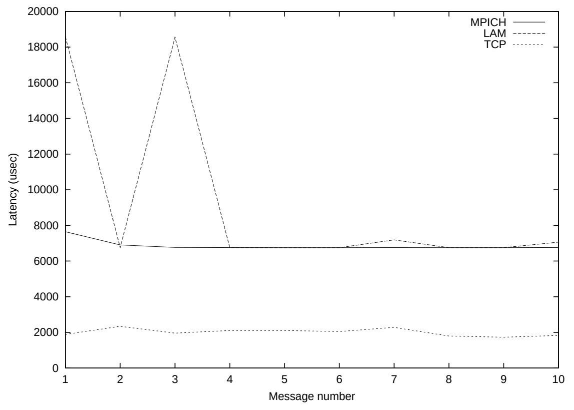

Figure 3. Ping-pong latency for MPICH, LAM and TCP on 36000 byte messages

Portals have generally been successful. However, these underlying protocols and some implementations of TCP perform similarly with respect to bandwidth and latency[2]. This leads us to believe that the increase in performance of MPI over other protocols could be due to a better implementation of MPI or to a better mapping between the message passing library and the underlying protocol.

Figures 2 and 3 show a gap in performance between the underlying protocol (TCP) and the implementation of MPI. This gap is due in part to poor implementation of the message-passing layer. However, some of this is overhead associated with the MPI library. For example, MPI matching is an expensive procedure which is performed regardless of the needs of the application. MPI may not be the message-passing paradigm of choice in the future. This will become particularly true as clusters and supercomputers grow into the hundreds of thousands of nodes.

Despite the shortcomings of MPI and its current implementations, one of the most obvious difficulties with most current TCP stacks is that they reside wholly in the kernel and contain overhead that is redundant with respect to MPI. The mapping between MPI and TCP is poorly conceived for high-performance computing. Namely, not only is the socket API which assumes a pull protocol (as opposed to a put protocol) inappropriate for use with sophisticated message-passing libraries, but it is the only interface library implementors have with the protocol. This does not allow the flexibility necessary to take full advantage of the perfomance opportunities of TCP.

For example, the interrupt into the kernel and the memory copy between kernel space and user space that is employed by most implementations of TCP may add too much overhead to CPU sensitive applications, whereas the variance in latency that exists if connections are setup as needed may be too much jitter for latency-sensitive applications. Connection management implementations create too much maintenance overhead, specifically, kernel buffers attached to each open connection create drastic inefficiencies for applications running on extremely large clusters.

Our research does not include creating more efficient implementations of MPI. Nor do we wish to enter the debate regarding MPI's efficacy as a message-passing protocol. Our research concentrates on pushing the capacities of commodity protocols with respect to high-performance computing. The research community needs a better implementation of the TCP protocol itself. As we design a new implementation of TCP, we must take into consideration the various needs of applications. The design must allow for hooks into the protocol stack so that applications and message-passing libraries can tailor the TCP stack in accordance to their particular performance needs.

Our goal is to create a flexible implementation of TCP that allows applications or subsystems to tune TCP so that its protocol and implementation bottlenecks are minimized. We propose to do this by isolating or *splintering* small parts of the functionality of the TCP stack and optimizing the functionality by offloading it or moving it away from the kernel. We begin our look at splintering by discussing operating system bypass and then contrasting this method with splintering. Finally, we propose splintering specific functionality of the TCP stack.

5 Operating System Bypass

Operating system bypass (OS bypass) was developed to reduce latency by removing the host processor from the common path of communication. Additionally, OS bypass addresses the bottlenecks associated with memory copies and frequent interrupts. In most instances, OS bypass is achieved by moving OS policy onto the NIC and protocol processing into user space.

OS bypass achieves lower latency by eliminating interrupts and all copying of network data, including the final copy from kernel memory to user memory. Although this technique has been demonstrated successfully in some cases, zero-copy *to* user space has yet to be proven generally useful in an operating system standard release. The primary concern is that the overhead and special-casing in the page cache necessary to manage the transition between the two address spaces may exceed the overhead of a single memory copy into a user buffer.

5.1 VIA

The virtual interface architecture (VIA) is one of the best-known OS bypass solutions. VIA assigns a virtual NIC in user space. A virtual interface is created for each connection (much like a socket) and each virtual interface contains a send queue and a receive queue. The receive queue contains descriptors with physical addresses. These physical addresses are translated at initialization time and the memory pages are locked[8].

VIA decreases latency especially for small messages since the overhead involved with managing buffers is not counted in the latency. If the virtual interface is implemented in user space, CPU overhead associated with communication remains very high. If the virtual interface is implemented on the NIC, the CPU overhead surprisingly remains high since the application must still be invoked to pull messages off of the queue.

OS bypass is achieved by moving data directly from the NIC to application space. If the NIC knows where the application expects a message, it can DMA data directly into application space and avoid all memory copies. The application either needs to tell the kernel where it wants a message so the kernel can translate addresses and tell the NIC, or the application must ask the kernel for an address translation and tell the NIC directly [8, 7].

No matter how the NIC gets information about memory addresses, both the application and the kernel are involved. First, the application must become active in order to control addressing and this requires a context switch. Second, the operating system must be active in order to perform address translation which requires a trap into the kernel. Using OS bypass, communication traffic still increases operating system activity.

6 Splintering

The philosophy of splintering isn't that the operating system shouldn't be used, but that it should be used effectively. In the case of communication offload, the goal is to minimize the overhead associated with invoking the OS while still enabling the OS to control communication.

6.1 Splintering IP

The IP stack has been successfully splintered in the past. A great example of the splintering of the IP stack is checksum offloading. Many NICs will compute the checksum of an IP packet during the DMA of data from the NIC to the host processor.

Fragmentation and reassembly are tasks that are extremely well-suited for splintering. Their implementation is possible on a reasonably powerful NIC such as the Acenic and they are easy to separate from the rest of the IP stack. This is accomplished by transparently accepting packets larger than the real MTU of the link and fragmenting on the card. Almost no modification of the driver is necessary. Moreover, splintering IP fragmentation and reassembly minimizes the overhead associated with invoking the OS by increasing the size of the packet and should increase the CPU availability for the application without sacrificing bandwidth.

Figure 4 shows that offloading fragmentation and reassembly onto the NIC significantly increases the effective utilization (bandwidth * availability) of the host. Effective utilization increased because this method maintained bandwidths within about 10% of bandwidths using a 1500 byte MTU but reduced the amount of time the host spent processing communication by about 50%. In fact, offloading fragementation and reassembly was more successful in relieving the interrupt pressure bottleneck than the use of interrupt coalescing or jumbo frames[4]

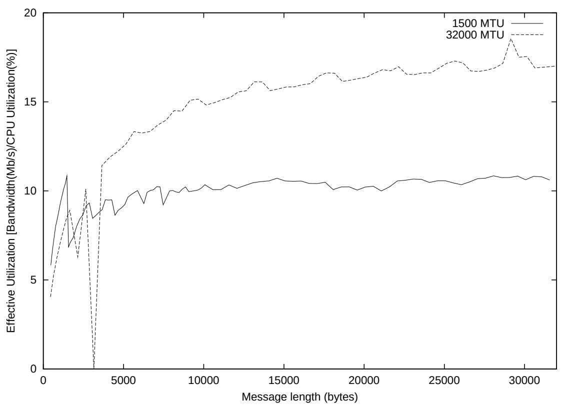

Figure 4. Effective utilization for unmodified firmware at 1500 byte MTU and offloaded fragmentation with driver MTU of 32000 bytes

We first splintered the IP stack in an attempt to relieve the interrupt pressure bottleneck associated with small frame sizes (1500 bytes) and high-speed networks (1Gb/s). We were able to implement fragmentation and reassembly on the NIC to demonstrate that it is possible to drastically impact the performance of a commodity protocol without significantly compromising its advantages[4].

6.2 Splintering RMPP

A related project in the Scalable Systems Lab splintered the reliable message passing protocol RMPP protocol (a simple RTS/CTS protocol) associated with the high-performance message-passing protocol, Portals[6]. Fig 5 shows that offloading the receive portion of RMPP (essentially, a DMA to user space rather than a DMA to kernel space with no OS interrupt) increases the effective utilization (bandwidth * availability). Furthermore, offloading part of the send portion of the protocol (specifically the reception of the CTS and the DMA of data from user space rather than from kernel space) further increases the effective utilization.

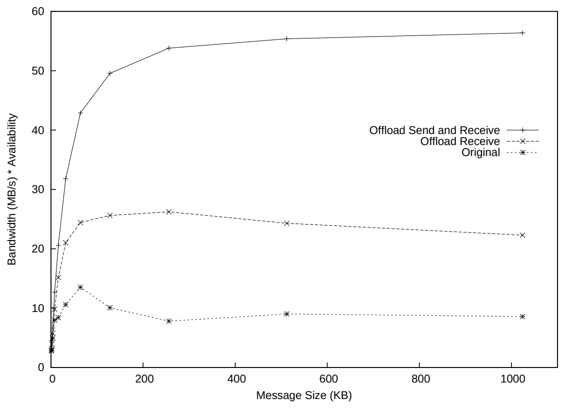

Figure 5. Effective utilization for RMPP

Regardless of the protocol, it is possible to allow the OS control over resources. Here, the RTS and the sending of the CTS remain in the OS. Appropriate offloading of small parts of a protocol can increase CPU availability without sacrificing bandwidth.

6.3 Splintering TCP

We have successfully splintered and offloaded part of the IP stack to reduce the interrupt pressures and to increase the effective utilization of compute nodes. Next, we will propose to splinter and offload part of the TCP stack in order to reduce the communication costs associated with the protocol. Because we want to allow applications to tailor TCP to fit their needs, we will propose three uses of the splintering method to increase the efficiency of TCP. In addition, we would like to see hooks or optimization levels that allow for higher layers to choose between speed and safety or speed and interoperability.

Figure 6 presents a graphical illustration of our approach to splintering the processing associated with the TCP protocol. In this case we only illustrate the processing done while receiving datagrams. In this illustration, solid lines indicate the paths taken by datagrams, while dashed lines represent control activities.

We start by considering the control path. In particular, we assume that the application issues a socket read before the data has arrived, i.e., a "pre-posted" read. Eventually, after traversing several layers of libraries, this read is transformed into a request that is passed to the operating system. The OS can then build a descriptor for the application

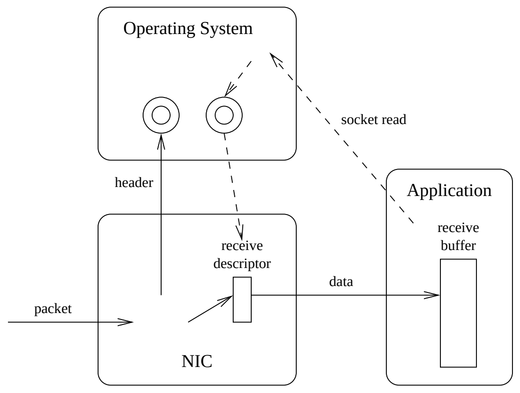

Figure 6. The Splintered TCP Architecture

buffer. This descriptor will include physical page addresses for the buffer. Moreover, as the OS builds this descriptor, it can easily ensure that these pages will remain in memory (i.e., they will be pinned) as long as the descriptor is active in the NIC.

Now, we consider the handling of datagrams. When a datagram arrives, the NIC first checks to see if the incoming datagram is associated with a descriptor previously provided by the OS. If it finds such a descriptor, the NIC will DMA the data portion of the datagram directly to the application buffer, providing a true zero copy, and make the header available to the OS. If the NIC does not find the needed descriptor, it will simply make the entire datagram available to the OS for "normal" processing.

Perhaps more interesting than the functionality that we intend to put on the NIC is the functionality that we plan to leave in the OS. As we have described, we will leave memory management in the OS and only provide the NIC with the mapping information that it needs to move data between the network and the application. We also plan to leave all failure detection and recovery in the OS.

High-performance applications employ various communication patterns. Generally, three types of messages are likely to be sent. Very large messages are used for bulk data transfer, small messages are used for communicating progress or upcoming data transfers, and very small messages with no (useful) data are used for synchronization. Applications using splintered TCP can tune the stack to decrease the CPU overhead for bulk data transfers, drastically decrease the latency of synchronization messages, or decrease the kernel resources necessary to maintain a connection allowing for applications to scale to more nodes regardless of the types of messages they send.

6.3.1 Decreasing CPU Overhead

Applications often perform error checking on data and can handle corruption of messages (either short or long) through resends at the application level, but want to avoid extra CPU overhead. Applications with these needs can consider TCP in which congestion control is splintered and offloaded onto a NIC.

Splintered TCP provides true zero copy for any preposted receives. This allows an application to receive and process data without unnecessarily interrupting the OS. The less often the OS is interrupted, the lower the CPU overhead associated with communication.

However, with splintering it is still necessary to invoke the OS to process the TCP headers, acknowledgements, and "unexpected" datagrams that have been queued by the NIC. Traditionally, the OS is invoked by the NIC, using an interrupt, for every datagram. This is needed to ensure the timely processing of datagrams. To avoid overrunning the processor with interrupts, many high-performance networks coalesce interrupts. That is, they wait until an number of packets have arrived or until a timer expires before they generate an interrupt. Because data expected by the application is being delivered in a timely manner, we can employ coalescing of interrupts without adding variance of latency to these messages.

Successful offloading of congestion control along with true zero-copy should substantially decrease the amount of CPU overhead associated with communication. Additionally, splintering allows the operating system to maintain appropriate control over resource management of the host processor and the NIC.

6.3.2 Decreasing Latency

Some applications are more concerned with the latency of synchronization message whereas the accuracy of data passed is less important (or not important at all as the sending of the message is the synchronization event). Applications with these needs can consider TCP in which acknowledgement generation, in addition to congestion control is splintered and offloaded onto the NIC.

In addition to moving data to the application and maintaining congestion information, the NIC will generate and send an acknowledgement, including the needed flow-control information, for a datagram associated with a preposted receive. The TCP headers are still made available to the OS, but because the NIC generates the acknowledgement and moves the data, latency is very short.

If, on the other hand, an application wants TCP to be an end-to-end messaging layer (as opposed to a wire layer), then the OS will calculate and validate checksums when processing the TCP headers. Acknowledgments will not be sent until after the checksum is completed, and the application will not have access to the data until the checksumming is completed. The price to pay for end-to-end reliability is an increase in latency.

6.3.3 Decreasing OS Resource Usage

Applications that require communication between hundreds of thousands of nodes and are sensitive to latency variance cannot use current implementation of TCP because there are not enough OS resources (specifically buffers) to open n^2 connections. Applications with these needs can consider TCP where connection maintenance is splintered and active connections are offloaded onto the NIC. The control schema is very similar to the control schema presented in the previous sections. In fact, splintering to relieve overhead, splintering to decrease latency and splintering to decrease OS resource usage can be used in conjunction.

All connections will be opened at the beginning of the application. However, connections will only be assigned resources if the OS receives either a message or a pre-posted receive. The OS only needs to keep a pointer to congestion control information and a sequence number for each inactive connection. As connections become active, they will be assigned buffers (either user memory in the case of pre-posted receives, or OS buffers) and congestion control windows. Because we can take advantage of the homogeneity of the SAN, we need only cache equivalence classes of congestion control information rather than keep individual congestion information for each connection.

7 Related Work

Splintered TCP is similar to the EMP protocol[7] in that both push descriptors onto the NIC and achieve true zero-copy. EMP is different in that it's purpose is OS bypass for MPI whereas our work uses the TCP protocol. Additionally, EMP includes error handling on the NIC which potentially pushes too much processing onto the slower processor on the NIC. In our view, error handling should be treated as a special case and should not consume the limited resources that we would like to dedicate for high-performance activities.

Like splintered TCP, Trapeze[2] separates TCP headers from data. However, Trapeze sends both data and headers through the OS and uses page remapping to achieve zero-copy. Splintered TCP DMA's data directly to the application, thereby achieving a true zero copy. Also, splintered TCP avoids costly interrupts by offloading congestion control and ack generation to the NIC.

Splintered TCP is most similar to Wenbin Zhu's work on offloading parts of the RTS/CTS protocol of Portals[6]. While that work addressed the implementation of a specialized API, Portals, and a specialized protocol, RMPP, our work addresses a commodity API, sockets, and a commodity protocol, TCP. Moreover, we intend to extend the earlier work, by pushing descriptors onto the NIC on pre-posted reads.

8 Summary

OS bypass has been a popular philosophy aimed at relieving the performance pressures associated with high-performance communication. However, OS bypass attempts to fully disengage the operating system from the process of communication. This is not possible to achieve as the operating system must manage resources like memory and scheduling. Also, if work is pushed to the application, the application must be invoked, which is no more efficient than invoking the OS.

Splintering, on the other hand, attempts to more efficiently use the operating system to control communication. By moving select functions of communication onto the NIC, we can decrease the number of interrupts to the operating system while still allowing the operating system to manage resources.

There has long been a trend toward improving the performance of TCP/IP in high-performance computing while still maintaining its tremendous advantages. By creating a true zero-copy TCP, we can show that we can reduce overhead enough that TCP/IP becomes a viable protocol in the world of cluster computing. Furthermore, by appropriately offloading small parts of the protocols' functionality onto a NIC we have demonstrated the methods of splintering that will become more and more prevalent as we move into a distributed computing environment.

Here we have found that splintering the protocol stacks and offloading some implementation aspects of TCP/IP will offer application programmers more flexibility in determining whether they want to tune the TCP stack to increase safety or increase performance by lowering communication overhead, lowering latency, or increasing the number of connections or all three. Additionally, the splintered TCP stack provides this increase in performance without sacrificing the interoperability and cost advantages of the protocol.

References

[1] 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, 1995.

[2] J. Chase, A. Gallatin, and K. Yocum. End-system optimizations for highspeed TCP. In *IEEE Communications, special issue on TCP Performance in Future Networking Environments*, volume 39, page 8, 2000.

[3] S. Elbert, Q. Snell, A. Mikler, G. Helmer, C. Csandy, K. Stearns, B. MacLeod, M. Johnson, B. Osborn, and I. Veri-

gin. Gigabit ethernet and low cost supercomputing. Technical Report 5126, Ames Laboratory and Packet Engines, Inc., 1997.

[4] P. Gilfeather and T. Underwood. Fragmentation and high performance ip. In *Proc. of the 15th International Parallel and Distributed Processing Symposium*, April 2001.

[5] P. Hochmuth. Vendors lower gigabit ethernet price bar. Web: 'http://www.nwfusion.com/archive/2001/127651_11-26-2001.html', 2001.

[6] A. Maccabe, W. Zhu, J. Otto, and R. Riesen. Experience offloading protocol processing to a programmable nic. Technical Report TR-CS-2002-12, University of New Mexico, 2002.

[7] P. Shivam, P. Wyckoff, and D. Panda. EMP: Zero-copy OS-bypass NIC-driven Gigabit Ethernet message passing. In ACM, editor, *SC2001: High Performance Networking and Computing. Denver, CO, November 10–16, 2001*, New York, NY 10036, USA and 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 2001. ACM Press and IEEE Computer Society Press.

[8] E. Speight, H. Abdel-Shafi, and J. K. Bennett. Realizing the performance potential of the virtual interface architecture. In *Proceedings of the 1999 Conference on Supercomputing*, ACM SIGARCH, pages 184–192, N.Y., June 20–25 1999. ACM Press.

[9] W. R. Stevens. *TCP/IP Illustrated, Volume 1; The Protocols*. Addison Wesley, Reading, 1994.