Seekable Sockets: A Mechanism to Reduce Copy Overheads in TCP-based Messaging

Chase Douglas and Vijay S. Pai

Purdue University
West Lafayette, IN 47907
{cndougla, vpai}@purdue.edu

Abstract

This paper extends the traditional socket interface to TCP/IP communication with the ability to seek rather than simply receive data in order. Seeking on a TCP socket allows a user program to receive data without first receiving all previous data on the connection. Through repeated use of seeking, a messaging application or library can treat a TCP socket as a list of messages with the potential to receive and remove data from any arbitrary point rather than simply the head of the socket buffer. Seeking facilitates copy-avoidance between a messaging library and user code by eliminating the need to first copy unwanted data into a library buffer before receiving desired data that appears later in the socket buffer.

The seekable sockets interface is implemented in the Linux 2.6.13 kernel. Experimental results are gathered using a simple microbenchmark that receives data out-of-order from a given socket, yielding up to a 40% reduction in processing time. The code for seekable sockets is now available for patching into existing Linux kernels and for further development into messaging libraries.

1 Introduction

Cluster computing is often used to perform time consuming calculations which can be divided into portions of work and assigned to multiple computers. The computers send messages and data to each other frequently to update status information or provide other computers with data and information needed for other parts of the calculation. While a clustered design allows for many computers to be fully utilized to decrease computation time, clustered systems experience overheads when the computers need to communicate. The overhead of this communication varies with the computation being performed, the number of computers, the libraries used to facilitate the communication, and the choice of interface between the computers, among others. Although a variety of proprietary solutions exist for cluster interconnects, recent studies have shown that efficient messaging library design can allow clusters using commodity TCP/IP and Ethernet components to achieve performance comparable to proprietary interconnects [9, 10]. Consequently, this study focuses on TCP/IP-based messaging.

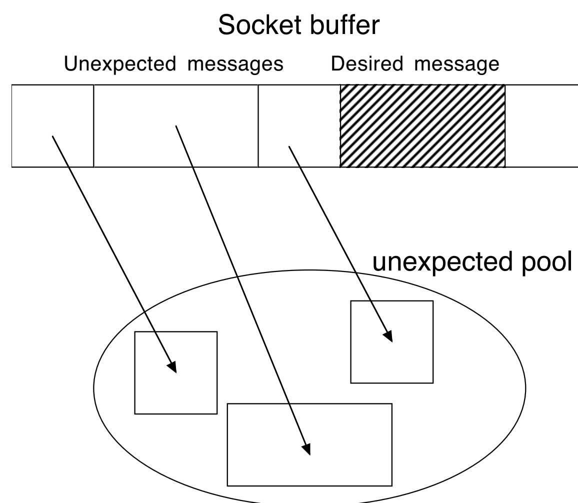

Figure 1. Performance impact of receiving unexpected messages before a desired message

The overheads experienced using TCP/IP-based messaging depend not only on the size and frequency of messages, but also on whether a message is expected or unexpected. A message is unexpected if its data arrives at the receiver before that system has invoked the library call to

This work is supported in part by the National Science Foundation under Grant Nos. CCF-0532448 and CNS-0532452.

1-4244-0054-6/06/$20.00 ©2006 IEEE

receive that message into an application-level buffer. Thus, unexpected data is typically copied first into a temporary library buffer [10, 14]. For TCP/IP-based messaging, a message can be considered to have arrived when the data appears on the network and the TCP stack places it into the socket buffer of a connected socket between two hosts. Figure 1 represents a situation where unexpected messages have arrived at a system that is waiting on a specific message (for example, an MPI message with a specific type of tag [6]). The operating system socket interface for TCP receives bytes from a given connection in-order using the `recv` or `read` system call. Thus, accesing the desired message first requires emptying the socket of the earlier unexpected messages. Since these will most likely be needed later on, each of these messages should be copied aside into an unexpected receive pool, incurring substantial overhead. Later application-level receives will first check the unexpected pool before calling the socket-level receive function.

This paper proposes an extension to the operating system socket interface to allow random-access receives from the socket buffer. This new interface, titled *seekable sockets*, allows the socket buffer to be searched for the desired content and allows the user program to receive the desired message without copying prior data. Through repeated use of seeking, a messaging application or library can treat a TCP socket as a list of messages with the potential to receive and remove data from any arbitrary point rather than simply the head of the socket buffer.

The seekable sockets interface is orthogonal to many other works that improve the performance of TCP-based messaging. Examples include those that use more efficient library design, such as event-driven architectures, or that use hardware support at the network interface card, such as TCP splintering [7, 10]. This work differs from both categories by focusing on the performance impact of an operating system interface: the socket layer. Consequently, this work should be able to work synergistically with ideas that focus on NIC support or library architecture to improve other components of TCP messaging performance.

The seekable sockets interface has been implemented in the Linux 2.6.13 kernel and tested using a simple microbenchmark that receives data out-of-order from a given socket. Experimental results show up to a 40% reduction in processing time using seekable sockets, with the greatest benefits arising from larger messages or a larger number of out-of-order messages. The seekable sockets patch is available for download from `http://www.ece.purdue.edu/~vpai/ssocks/`.

For UDP/IP-based messaging, as used in the IP path of LA-MPI, a single receiver socket could even be shared across multiple senders, and the desired message could be one from a specific host.

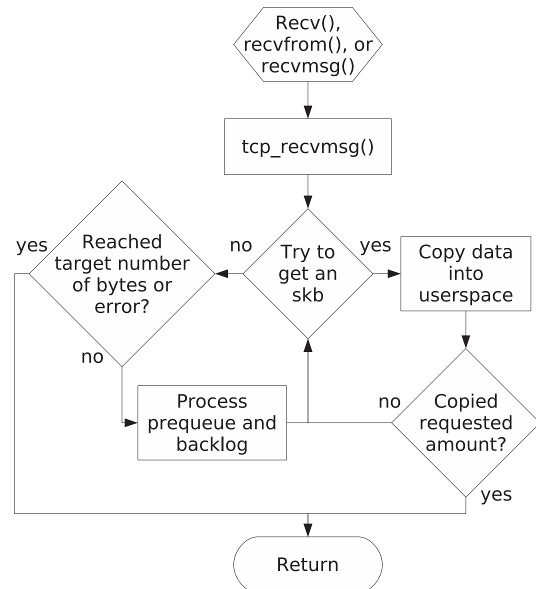

Figure 2. Flowchart of Linux `tcp_recvmsg` function.

2 Background on Linux TCP

The TCP stack of the linux kernel operates on socket buffers, known as `sk_buff`'s or `skb`'s. As packets of data are received by the network device, the data is placed in a ring buffer and an `sk_buff` is allocated and associated with the data. This `sk_buff` holds the metadata for the packet and the linux TCP/IP stack interacts with the `sk_buff` to process the packet. Eventually, the `sk_buff` is placed on the socket buffer queue for a given connection.

To facilitate packet control and reception in the correct order, each TCP packet has a sequence number, which gives the number of bytes sent on the connection prior to that particular packet. This allows for recovery from network-level reordering of packets on the receiving side and for data retrieval from the socket buffers. A user program does not know or need to know how data is arranged in packets or how it arrives on the network. An application's only concerns are the order in which the data elements were sent from the source and the length of the data.

When a user program invokes the `recv()`, `recvfrom()`, or `recvmsg()` functions on a TCP socket, `tcp_recvmsg()` is invoked within the kernel (`net/ipv4/tcp.c`). Figure 2 is a flowchart of these functions. `tcp_recvmsg()` begins copying data from the `sk_buff`'s in the socket queue which point to the actual data in the ring buffer. The function inspects the first `skb` in the socket buffer and then copies data to the user-space buffer. If the `skb` has more data than requested,

tcp_recvmsg() leaves the remainder on the socket buffer queue. If the user has requested more data than the contents of the first skb, the skb is deallocated along with its corresponding data on the ring buffer, and the above steps continue with the next skb in the queue. By default, tcp_recvmsg() returns after reading all the requested data from the socket buffer, deallocating all the skb's that have had their data completely received, and updating the sequence number of the first byte to read on the socket.

TCP uses sequence numbers to keep track of what has been read from the socket buffer and what should be read next. The per-connection copied_seq variable holds the sequence number that should be read next; this value defines what has already been copied and what will be read next from the receive queue. If copied_seq is larger than the first skb's base sequence number, then part of the skb has already been read and the requested length of data will be copied starting from the sequence number specified by copied_seq. Thus, the use of sequence numbers determines the data that a receive call will read from the socket.

3 Adding Seek to Sockets

The goal of seekable sockets is to receive data that has been placed on the socket receive buffer in any order. When data is copied from the socket, the corresponding skb's that hold the data need to be deallocated, and the linked list of skb's needs to be patched. Also, since the data at the sequence numbers that was read is no longer available, subsequent receive calls on the socket need to be aware that this data is no longer available. This is implemented through a linked list that holds the starting and ending sequence numbers of each "hole" in the socket receive buffer; the creation of a hole frees up space in the socket buffer (and allows normal TCP flow control behavior) regardless of the point in the socket buffer from which data has been removed. When a receive call begins copying data to the user, it will skip over any hole that it encounters and continue receiving normally from the sequence number after the hole. During the receive, the list of holes grows, coalesces, and is pruned dynamically.

The implementation developed creates a new pseudo-protocol, SOCK_SEEK_STREAM. This protocol uses the same tcp stack that normal SOCK_STREAM sockets use; however, the functions have been modified so that a socket of type SOCK_SEEK_STREAM may be seeked upon. If a socket call is on a non-seekable socket, or if the call is not a seeking receive, then the path through the TCP stack and its functions is nearly identical to a stock linux kernel. However, if a seeking call takes place on a seekable socket, then the path through the TCP stack is the same while the path through individual functions may vary. The main changes are in the tcp_recvmsg() function. The changes to this function implement all the modifications necessary to manage the holes list, sequence numbers, and already read skb's. The one notable change to the tcp_recvmsg() function during a seekable socket receive is that the TCP prequeue mechanism is disabled. The TCP prequeue mechanism allows for better management of stream resources in exchange for a slight decrease in performance. Also, the prequeue cannot be easily modified to allow for seeking. Therefore, the prequeue mechanism has been disabled whenever receiving on a seekable socket.

Once a socket has been created as type SOCK_SEEK_STREAM, the normal recv() and recvmsg() functions can be used as usual. A new function, seek_recv(), is implemented as a syscall and takes the following arguments:

```
ssize_t seek_recv(int s, void
*buf, size_t len, int flags,
size_t offset);
```

The arguments are identical to the recv() function, with the offset variable added to specify the number of bytes to offset into the stream. This offset is always relative to the first byte which would be received through a recv() call.

Since seek_recv() modifies the msghdr structure and then calls the generic sock_recvmsg function, it is also possible to use the standard recvmsg() library function for seeking receives. The msg_seek variable has been added to the msghdr to specify the seek offset; by modifying the msghdr structure passed into recvmsg(), it is possible to do a seeking receive without invoking the new function.

To be able to seek past large messages, the maximum receive buffer size must be increased. This is controlled by the net.core.rmem_max system variable (sysctl). This will set the maximum receive buffer that can be specified using the setsockopt() function. Therefore, the sysctl should be set to a reasonably large number of bytes, and the receive buffer should be increased as necessary within the user-space program. When the receive buffer becomes full, normal TCP actions are performed, and the seeking receive call returns an error. If new packets are received while the socket buffer is full, they will be dropped and retransmitted by the sender according to normal TCP flow control [1]. The program must respond to the socket buffer full error by removing some of the data left in the buffer to free up more space in the kernel.

4 Experimental Results and Discussion

The seekable sockets interface discussed in Section 3 is implemented on the Linux 2.6.13-rc3 kernel and tested with a microbenchmark. The microbenchmark uses two hosts:

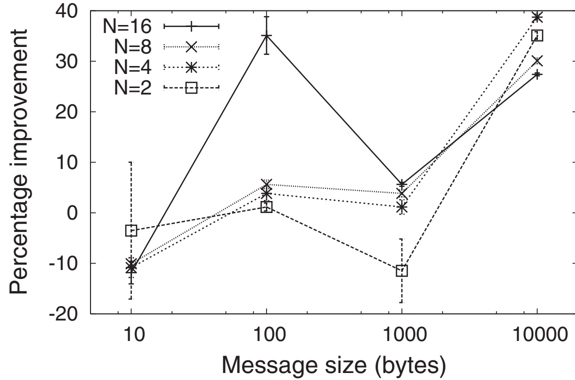

Figure 3. Performance impact of seekable sockets for varying message sizes (X-axis) and out-of-order message counts (N value).

a sender and a receiver. The sender repeatedly sends messages of a constant, configurable size on an established connection. The receiver repeatedly skips past N such messages, reads the desired message, and then reads the N skipped messages. The receiver is implemented using a non-seeking and a seeking approach. In the non-seeking version, the N skipped messages are first copied from the socket buffer into application memory. In the seeking version, all of the skipping is done using the new `seek_recv` system call. The performance of the system is evaluated using the sum of system and user time seen at the receiver system. The percentage of this active CPU time reduced by the seeking version is the metric used to evaluate the effectiveness of seeking. Each test consists of 100 repetitions of the following: opening a socket, transferring 1000 messages on it according to the receive policy described above, and closing that socket. The system only has one active socket at a time.

Figure 3 shows the active CPU time reduction achieved using the seekable sockets interface. The different curves represent N values ranging from 2 to 16, while the X-axis of the graph represents the message size in bytes. Note that the X axis is logarithmically scaled. The Y-axis represents the percentage of CPU time reduced using seekable sockets: numbers below 0% indicate situations where the new interface degraded performance. Each data point was tested 10 times; the curves show the average values with standard error bars to show the experimental variation across tests; these experimental errors decrease for larger message sizes because the tests run for longer and are less subject to random OS and cache behavior.

Seeking sometimes degrades performance for low N values or small messages because of the greater overhead in processing the socket buffers (e.g., managing the holes list). Some degradations arise for 10-byte messages; these are an unrealistic data point in well-tuned applications but may be useful for considering applications for which the communication is difficult to coordinate or for benchmarking the base latency of a communication channel. In general, however, the curves indicate better performance benefits from larger messages or larger N values. These results are not surprising given the greater amount of copying (and expense of copying) required in such tests. At best, the results indicate a 40% reduction in CPU execution time after adding the new interface.

One interesting result is the dip in performance at message sizes of around 1000 bytes. This message size is closest in size to the TCP packet payload size (1460 bytes). The relative overhead of managing the holes list is consequently greater in this case than in the others.

This work shows that the seekable socket interface is useful for TCP-based data communication, but does not implement a full-scale messaging library around this interface. Experience with microbenchmark code suggests that few changes are required for using seekable sockets. However, integrating the interface into an MPI library may yield further insights on the usability of this interface. For example, applications will first need to inspect message headers using traditional receives (possibly peeking) before knowing the amount required for seeking. The actual performance will likely vary based on the amount to which messages are received out-of-order; since our microbenchmark results only consider computation time (and not its overlap with communication latencies), these results should be composable to describe overall system performance according to the methodology of Saavedra and Smith [15]. The results should also be largely independent of the physical medium used or the number of sockets since the operating system changes are confined to the socket layer and are strictly on a per-socket basis.

Seekable sockets may also yield performance improvements outside of the cluster computing domain. For example, modern HTTP implementations pass multiple simultaneous requests and responses on a single pipelined persistent connection for more efficient TCP performance [5]. Seekable sockets would allow for parallel processing of a pipelined persistent connection, with different threads reading, parsing, validating, and rendering content in different portions of the connection.

5 Related Work

The introduction cites the work that is most closely related. While Majumder et al. focused on library design to improve the performance of TCP-based messaging, Gilfeather and Macabe have used hardware support from the network interface card to offload critical portions of pro-

tocol processing [7]. Many other works focus on hardware support for message-passing using proprietary interconnects, user-level interfaces, and customized protocols [2, 3, 4, 8, 11, 12, 13, 16, 19]. This work differs from all of those by focusing on the performance impact of a specific operating system interface: the socket layer. Consequently, this work is orthogonal to works that focus on NIC support or library architecture and should be able to work synergistically with ideas to improve other components of TCP messaging performance.

SCTP, the Stream Control Transmission Protocol, is a reliable transport-level protocol under development that is similar to TCP in that it has a notion of a flow-controlled connection between two machines [17]. However, unlike TCP, this connection (called an association) may consist of several independent message streams. SCTP enforces ordering within a message stream, but not across the entire association. This feature could be used to provide the benefits of seekable sockets for specific circumstances in which data may be processed out-of-order; for example, tagged MPI receives could be implemented as separate streams in an association, as could independent web responses on a pipelined connection. Seekable sockets are more flexible, however, because they allow arbitrary out-of-order receives. Additionally, the user-level API for seekable sockets is only a single receive-side extension to the traditional sockets interface, while SCTP requires more complex extensions and modifications to sending, receiving, and connection management functions [18]. SCTP also allows for multiple IP addresses per machine involved in a given association, potentially allowing the use of multiple network paths for greater reliability; this feature is orthogonal to seekable sockets.

6 Conclusions

This paper proposes a new extension to the socket layer that allows for random-access receives from a single TCP connection. This new seekable socket interface has been implemented in Linux 2.6.13 and tested using a simple microbenchmark that receives data out-of-order. Experimental results show up to a 40% reduction in processing time, with benefits increasing for larger messages or a greater number of out-of-order messages. The seekable sockets patch is available for download from `http://www.ece.purdue.edu/~vpai/ssocks/`, and may be applied to currently-available Linux kernels. Experience in using this new interface suggests that few changes are required in application-level source code, but this interface will need to be integrated into a fully-functional messaging or communication library before further conclusions can be drawn.

References

[1] M. Allman, V. Paxson, and W. Stevens. TCP Congestion Control. IETF RFC 2581, April 1999.

[2] 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.

[3] P. Buonadonna and D. Culler. Queue Pair IP: A Hybrid Architecture for System Area Networks. In *Proceedings of the 29th International Symposium on Computer Architecture*, pages 247–256, May 2002.

[4] D. Dunning, G. Regnier, G. McAlpine, D. Cameron, B. Shubert, F. Berry, A. Merritt, E. Gronke, and C. Dodd. The Virtual Interface Architecture. *IEEE MICRO*, 18(2):66–76, March 1998.

[5] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol - HTTP 1.1. IETF RFC 2616, June 1999.

[6] T. M. P. I. Forum. MPI: A Message-Passing Interface Standard. *International Journal of Supercomputer Applications*, 8(3/4), 1994.

[7] P. Gilfeather and A. B. Macabe. Making TCP Viable as a High Performance Computing Protocol. In *Proceedings of the Third LACSI Symposium*, October 2002.

[8] M. Lin, J. Hsieh, D. H. C. Du, and J. A. MacDonald. Performance of High-Speed Network I/O Subsystems: Case Study of a Fibre Channel Network. In *Proceedings of the 1994 conference on Supercomputing*, pages 174–183. IEEE Computer Society Press, 1994.

[9] S. Majumder and S. Rixner. Comparing Ethernet and Myrinet for MPI Communication. In *Proceedings of the Seventh Workshop on Languages, Compilers, and Run-time Support for Scalable Systems (LCR 2004)*, pages 83–89, October 2004.

[10] S. Majumder, S. Rixner, and V. S. Pai. An Event-driven Architecture for MPI Libraries. In *Proceedings of the 2004 Los Alamos Computer Science Institute Symposium*, October 2004.

[11] S. S. Mukherjee, B. Falsafi, M. D. Hill, and D. A. Wood. Coherent Network Interfaces for Fine-Grain Communication. In *Proceedings of the 23rd International Symposium on Computer Architecture*, pages 247–258, 1996.

[12] F. Petrini, W.-C. Feng, A. Hoisie, S. Coll, and E. Frachtenberg. The Quadrics Network: High-Performance Clustering Technology. *IEEE MICRO*, 22(1):46–57, January 2002.

[13] I. Pratt and K. Fraser. Arsenic: A User-Accessible Gigabit Ethernet Interface. In *Proceedings of IEEE INFOCOM '01*, pages 67–76, 2001.

[14] S. H. Rodrigues, T. E. Anderson, and D. E. Culler. High-Performance Local Area Communication With Fast Sockets. In *Proceedings of the 1997 USENIX Technical Conference*, pages 257–274, January 1997.

[15] R. H. Saavedra and A. J. Smith. Analysis of Benchmark Characteristics and Benchmark Performance Prediction. *ACM Transactions on Computer Systems*, 14(4):344–384, November 1996.

[16] P. Shivam, P. Wyckoff, and D. Panda. EMP: Zero-copy OS-bypass NIC-driven Gigabit Ethernet Message Passing. In *Proceedings of the 2001 ACM/IEEE Conference on Supercomputing (SC2001)*, November 2001.

[17] R. Stewart, Q. Xie, K. Morneault, C. Sharp, H. Schwarzbauer, T. Taylor, I. Rytina, M. Kalla, L. Zhang, and V. Paxson. Stream Control Transmission Protocol. IETF RFC 2960, October 2000.

[18] R. Stewart, Q. Xie, L. Yarroll, J. Wood, K. Poon, and M. Tuexen. Sockets API Extensions for Stream Control Transmission Protocol (SCTP). IETF Internet Draft (`draft-ietf-tsvwg-sctpsocket-11.txt`), September 7 2005.

[19] M. Welsh, A. Basu, and T. von Eicken. ATM and Fast Ethernet Network Interfaces for User-level Communications. In *Proceedings of the Third International Symposium on High Performance Computer Architecture*, pages 332–342, February 1997.