High-Performance Local Area Communication With Fast Sockets

Steven H. Rodrigues Thomas E. Anderson David E. Culler

University of California at Berkeley
Berkeley, CA 94720
{steverod,tea,culler}@cs.berkeley.edu

June 24, 1996

Abstract

This is the abstract.

1 Introduction

The development and deployment of high performance networks such as ATM [de Prycker 1993], Myrinet [Seitz 1994], and switched high-speed Ethernet [?] has promised improved communications performance for network applications. These networks are capable of microsecond latencies and hundreds of megabits per second bandwidths. The network fabric is switched, to eliminate the contention seen on shared-fabric networks like traditional Ethernets. Network interfaces for these networks usually contain a megabyte or more of buffer memory, and often have a dedicated communications processor.

Most of these networks are used to run the TCP/IP protocol suite [Postel 1981a, Postel 1981b, ISI 1981]. TCP/IP is the default protocol for Internet traffic, and is available on a wide variety of computing platforms and network technologies. The TCP protocol, which provides the abstraction of a reliable, ordered byte stream, can deliver bandwidth close to the rated maximum of many of these networks. TCP's round-trip latency, however, is well above the minimum level of these networks. The UDP protocol, which lacks the reliability and ordering mechanisms of TCP, does little better [Keeton et al. 1995, Kay & Pasquale 1993, von Eicken et al. 1995]. Both protocols have high per-packet processing costs.

A common interface to the TCP/IP protocol suite is the Berkeley Sockets API [Stevens 1990]. The Sockets API extends the Berkeley Unix operating system metaphor of file abstractions to network communication. A process can read from and write to a network connection in the same way that it reads and writes files. The sockets interface adds a number of additional operations to access features of the TCP/IP protocol suite. This interface adds another layer of code and additional processing costs on top of the high costs of the protocol suite.

These high processing overheads limit overall communications performance for many user applications. Most network traffic consists of small packets [Gusella 1990, Caceres et al. 1991, Claffy et al. 1992, Feldmeier 1986, Amer et al. 1987, Kleinrock & Naylor 1974, Schoch & Hupp 1980, Cheriton & Williamson 1987]. Small packets are the rule even in applications considered bandwidth-intensive: 95% of all packets in a NFS trace performed at the Berkeley CS division carried less than 192 bytes of user data [Dahlin et al. 1994]; the average packet size in the trace was 382 bytes. This small average size means that processing overheads are a dominant factor in communications cost. Round-trip performance is also important, however; a cursory examination of the application-level protocols in use today (HTTP, NFS, FTP, etc.) reveals that network applications communicate in a *request-response*, or *client-server* manner: a client machine sends a small request message to a server, and awaits a response from the server. Protocol overhead thus cannot be hidden through packet pipelining or overlapping communication and computation.

This paper presents our solution for this problem. Our work attempts to minimize protocol overhead and expose the full performance of the underlying network hardware to the application. In particular, we have attempted to optimize network performance for low round-trip latencies, especially for small packets, and for realizing the full bandwidth of the network without having to use full or nearly-full packets. We have accomplished this while maintaining compatibility with existing applications.

Lowering communications overhead requires changing the application programming interface (API), the communications protocol, the implementation of the protocol, or a combination of these methods. Changing the programming interface involves modifying the code used by applications to access communications functionality. While this approach may yield better performance, it does not provide a method to improve the performance of legacy applications. Changing the communications protocol involves changing the "on-the-wire" format of data and the actions taken in supporting a communications exchange; for example, modifying the TCP packet format. Changing the protocol may improve communications performance, but at the cost of direct incompatibility: applications communicating via the new protocol are unable to share data directly with applications using the old protocol. Changing the protocol implementation simply involves rewriting the software that implements a particular protocol; packet formats and events do not change. While this approach provides full compatibility with existing protocols, fundamental limitations of the protocol design may limit the performance gain possible.

Recent systems such as Active Messages [von Eicken et al. 1992], Remote Queues [Brewer et al. 1995], and U-Net [von Eicken et al. 1995] have taken these methods to an extreme. These systems implement new protocols and new programming interfaces for improved network performance. The protocols and interfaces are very lightweight, closer to the network hardware than the abstractions provided by TCP/IP. All of these systems realize latencies and throughputs close to the physical limits of the network. None of these systems offer compatibility with existing applications, and require applications to be rethought and rewritten to achieve good performance.

At the other extreme are attempts to improve existing TCP performance, by changing existing implementations. Recent work includes zero-copy TCP for Solaris [Chu 1996] and a TCP interface for the U-Net interface [von Eicken et al. 1995]. These implementations can interact with other implementations of TCP/IP, and provide improved throughput and performance, realizing the full bandwidth of the network. Both systems still have round-trip latencies considerably higher than the raw network, because of high processing overheads.

Our work attempts to change the existing communication protocol and provide a high-performance protocol implementation while maintaining compatibility with existing software. We have written Fast Sockets, a high-performance communications stack which exports the Berkeley Sockets API for local-area communication and which reverts to standard TCP/IP protocols for wide-area communication. Fast Sockets attempts to minimize round-trip latency and maximize communications bandwidth through a lightweight protocol and efficient buffer management. It integrates the communication protocol and the interface to the protocol to eliminate module-crossing costs. Fast Sockets also exploits knowledge of packet destinations to eliminate copies within the stack, without requiring knowledge of or modifications to the operating system kernel.

The major contribution of this paper is its introduction of receive posting, a technique of utilizing user-supplied information about packet destinations to minimize copies. It also describes the design and implementation of a high-performance protocol for local-area communication.

The rest of the paper is organized as follows. Section 2 describes problems of current TCP/IP and sockets implementations and how these problems affect communications performance. Section 3 describes how the Fast Sockets design attempts to overcome these problems. Section 4 describes the performance of the resultant system, and section 5 compares Fast Sockets to other attempts to improve communications performance. Section **??** presents our conclusions and directions for future work in this direction.

2 Problems with TCP/IP

While TCP provides good throughput on most networks, its round-trip latency is poor. Further, observed bandwidth on workstation networks such as Myrinet and ATM does not begin to approach the raw capacity of these networks [Keeton et al. 1995]. In this section, we describe a number of features and problems of commercial TCP implementations, and how these features affect network performance.

2.1 Multiple Layers

Standard implementations of the sockets interface and the TCP/IP protocol suite separate the protocol and interface stack into multiple layers. The sockets layer and interface is usually a single layer, and sits on top of the protocol layer. The protocol layer itself contains multiple sublayers: for example, the TCP sublayer sits on top of the IP sublayer. Below the protocol layer lies the interface layer, which communicates with the network hardware. The interface layer usually has two portions, the network interface layer (which takes input from higher layers) and the network device driver, which communicates with the underlying layer.

Layering imposes a number of costs on the implementation, however. First, switching layers involves multiple function calls and possibly context switches to follow the code path. Second, layered systems may involve a change in the programming abstraction; for example, from contiguous buffers to packetized segments, and performing this conversion can be expensive. Finally, layering hides important implementation details within each layer, which makes tuning difficult without knowledge of all adjacent layers [Clark 1982, Crowcroft et al. 1992, Clark & Tennenhouse 1990].

2.2 General-Purpose Buffering

TCP/IP manages buffer space for network packets to allow data to be easily passed between layers. To do this easily, Berkeley Unix and its derivatives use *mbufs*. Mbufs are data structures used to hold portions of a data packet. A single mbuf can hold up to 108 bytes of user data, or it can point to a 4096-byte page. Mbufs can be linked together to hold large packets. Allocation of mbufs is a complex algorithm, attempting to balance allocation costs and memory usage [Wright & Stevens 1995].

Mbufs are an expensive abstraction. Mbuf management consumes over 100 microseconds on a DECstation 5000for small (single-packet) TCP messages, growing to 300 microseconds for 512-byte messages [Kay & Pasquale 1993]. Managing incoming and outgoing data involves walking chains of mbufs, a memory-intensive process. The allocation scheme is particularly bad for small packets, using three variants of mbuf for packets between 0–207 bytes.

2.3 Multiple Copies

While using a generalized abstraction such as mbufs can reduce copies, the semantics of the socket model and the TCP protocol force a number of copies to be made. In the socket model, when a `send` call is issued, the data is assumed to be copied into a network buffer and sent later. It crosses the network, and is placed into a buffer on the receiving socket. Once a `recv` call is issued, the data is copied from the socket buffer into the user buffer. This path is followed whether the final destination (the user buffer) is known at send time or not. Additionally, TCP requires that the data be copied for reliability. In the best case, the data is copied a minimum of three times: user-space to kernel-space and a copy for TCP reliability on the sender side and a copy from kernel-space to user-space on the receive side.

With high-performance networks, the network is no longer the limiting factor in protocol implementation; instead, memory and bus bandwidth are. Modern high-performance network bandwidth ranges from 20MB/s (ATM) to 80MB/s (Myrinet) or higher, whereas memory copy bandwidth on the SPARCstation 10 is 28MB/s and SBus bandwidth is 25MB/s. Any copies incurred in the critical path of the communications software will consequently tend to limit performance, regardless of the speed of the network.

3 Fast Sockets Design

Fast Sockets was intended to provide the sockets programming interface for local-area communication while yielding better performance than existing TCP/IP implementations. Fast Sockets is built upon Active Messages [von Eicken et al. 1992, Martin 1994, Culler et al. 1994, Mainwaring & Culler 1995], a high-performance, low-overhead communications interface. This section first describes Active Messages, the enabling technology for Fast Sockets, and then describes our solutions for the problems outlined in section 2. We close the section by describing current limitations of the Fast Sockets implementation.

3.1 Active Messages

Active Messages is a high-performance interface to the network card that is optimized for low-latency, high-bandwidth communication. Active Messages provides an efficient memory-to-memory copy abstraction for large data transfers and low-cost small messages for low-latency control operations. An *active message* consists of the name of a *handler function* and arguments for that function. On message arrival, the handler is looked up and invoked with the message arguments. Active messages are reliable but not ordered. Small messages carry up to 8 data words to be used as arguments for the specified handler. Bulk transfer messages carry up to 8 data words as individual handler arguments and up to 8Kbytes of additional data to be copied. The bulk transfer messages require a memory destination to be specified for the additional data at transmission time.

3.2 Fast Sockets Solutions

3.2.1 Collapsing Layers

To avoid the performance and structuring problems created by the multi-layered implementation of Unix TCP/IP, Fast Sockets collapses the API and protocol layers of the protocol stack together. This technique avoids abstraction conflicts between the programming interface and the internal protocol, and allows for easy sharing of information between the layers. It also reduces the number of conversions between layer abstractions, further reducing processing overheads.

Fast Sockets does not integrate the network interface layer into its processing. This allows Fast Sockets to be portable across network architectures, as long as the interface layer provides the same API. Fast Sockets uses Active Messages as its network interface. Active Messages implementations currently exist for the Intel Paragon [Brewer et al. 1995], HP735 and Medusa network interface [Martin 1994], Sun SPARC and Myrinet, and Sun SPARC and FORE ATM SBA-200 [von Eicken et al. 1995], meeting our goal of portability.

The combined API and protocol layers are written to be extremely lightweight. The use of Active Message handlers in Fast Sockets enables this by removing non-essential operations from the critical path of the data transfer routines. Rarely used functionality, such as connection establishment or out-of-band data transmission, uses different handlers than the normal send/receive path. A normal data transfer packet never needs to be examined for unusual situations such as connection establishment or shutdown; this is transparently handled by the message handler dispatch. Fast Sockets also attempts to reuse existing facilities when possible. To ensure consistent name management (for the `bind` call), Fast Sockets utilizes the existing name management code in the normal sockets layer.

These optimizations combine to reduce the protocol-specific and API-specific costs of communication.

3.2.2 Simple Buffer Management

To avoid the complexities of mbuf-style memory management, Fast Sockets uses a contiguous virtual memory buffer for intermediate data storage. Data is transferred directly into this buffer via Active Message bulk transfer calls. Successive bulk transfer calls place data at consecutive addresses within the buffer, automatically maintaining the in-order delivery guarantee of stream sockets. Packet metadata is carried in the argument words of the bulk transfer messages, eliminating the need for mechanisms to efficiently strip off packet headers.

Send buffering is eliminated in Fast Sockets. Because user applications rely heavily on small packets and on request-response behavior, delaying packet transmission only increases user-visible latency. This is an undesirable result. Eliminating send-side buffering has the additional benefit of reducing protol overhead by eliminating copies on the send side of the protocol path.

Fast Socket's send mechanism and buffering techniques are shown in Figure 1. These techniques eliminate the memory-allocation and many of the copy costs traditionally associated with the socket interface.

3.2.3 Copy Avoidance

The standard Fast Sockets data transfer mechanism results in a single copy on the data transfer path, from the contiguous socket buffer to the user buffer specified in the `recv` call. This copy increases the processing overhead of a data packet and consequently, round-trip latencies. A high-performance communications layer should bypass this copy whenever possible.

It is possible, under certain circumstances, to avoid the copy. If the final destination of the data is known at send time, the data can be redirected into the user buffer, entirely bypassing the socket buffer. Fast Sockets uses a technique called *receive posting* to do this. In receive posting, when a `recv` call is issued and no data is waiting in the socket buffer, a request message is sent to the socket's connection partner. This message informs the connection partner that a `recv` call is awaiting data and provides the location of the destination buffer. If the appropriate data has not been sent, its transmission will go direct to the user buffer, rather than through the socket buffer. Receive posting is graphically depicted in Figure 2.

Receive posting is a powerful tool. It provides the mechanism for a true, low-latency, zero-copy communications stack without modifications to the operating system or to user applications. It is particularly useful for applications using an RPC or request-response programming model [Birrell & Nelson 1984]. In this model, a client machine generates a request that is sent to a server machine, and blocks waiting on the request from the server. This type of situation is exploited by receive posting, reducing overhead on the client and decreasing overall response time.

Receive posting exploits features of the stream sockets model, of Active Messages, and of the integration of the API and the protocol. First, it presumes that the origin of data coming to a socket can be known in advance. This is true of stream sockets because of their connection-oriented nature; it is not true of connectionless endpoints, like datagram sockets, Mach

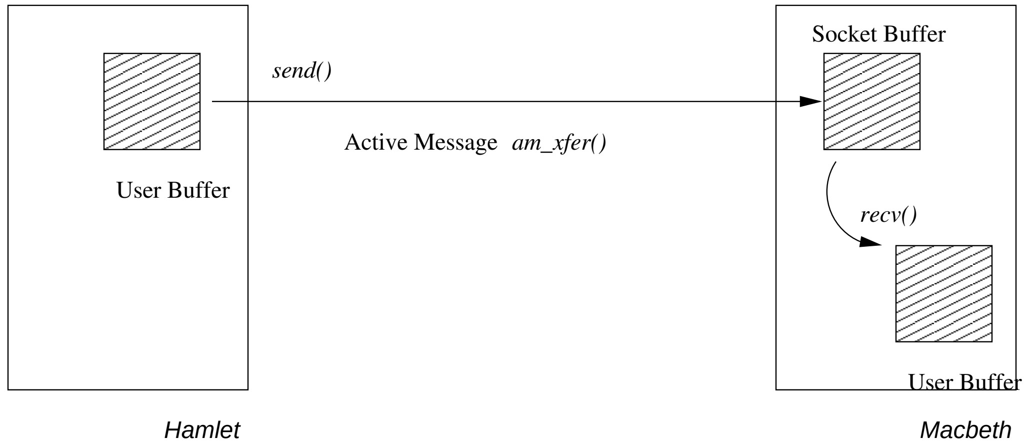

Figure 1: Data transfer via a dedicated socket buffer. Data is sent asynchronously via the `send` call, arrives in the socket buffer, and is copied out of the socket buffer via a `recv` call.

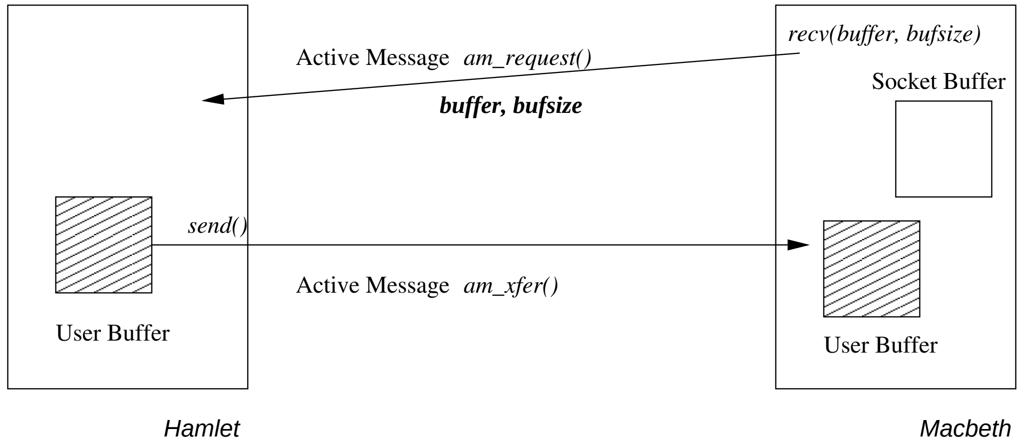

Figure 2: Data transfer via receive posting. A `recv` call is issued and, upon finding no data in the socket buffer, notifies the connection partner that it is awaiting data. A subsequent `send` to the socket goes directly to the user buffer, rather than indirecting through the socket buffer.

ports, or bare Active Message endpoints. Second, it requires that the memory destination of a given packet can be specified in advance. This is true for Active Messages, as an arbitrary memory region can be used to hold incoming data. Third, receive posting utilizes cheap communication, assuming that the cost of notifying the connection partner is less than the cost of a copy through the socket buffer. This is true for Active Messages, once the size of the transfer exceeds approximately 256 bytes. Finally, receive posting requires that the API layer be able to convey information to lower layers of the communication stack. This is possible in Fast Sockets through the integration of the API and protocol layers. Receive posting would not be possible without these features.

While receive posting reduces the number of copies involved in the communications critical path and improves performance (see section 4.1.1), it is not always desirable. Small messages incur low copy costs and so do not benefit from receive posting. Currently, receive posting is not performed for messages smaller than 256 bytes. When data is ready to be sent, but no post message has arrived, Fast Sockets performs the transfer, for correctness and performance. A key assumption of the socket model is that a `send` call is normally asynchronous; barring a lack of buffer space, computation can proceed immediately after a `send` is issued, rather than waiting for a matching `receive` call. Additionally, asynchronous `send` calls enable the overlap of communication and computation; requiring receive posting can reduce this overlap significantly.

Receive posting can eliminate all unnecessary copies from the data transfer path. It is particularly useful in programs using an RPC model of communication.

3.3 Other Fast Socket Operations

Fast Sockets supports the full sockets API, including socket creation, name management, and connection establishment. A Fast Socket is created via the `socket` call for the `AF_INET` address family and using the "default protocol" for that type of socket; this allows programs which explicitly request TCP or UDP sockets to use those protocols. For name management, Fast Sockets utilizes the facilities in the existing sockets implementation. Every Fast Socket has a *shadow socket* associated with it; this shadow socket is of the same type and shares the same file descriptor as the Fast Socket. Whenever a Fast Socket requests to `bind` to an `AF_INET` address, the operation is first performed on the shadow socket to determine if the operation is legal and the name is available. If the shadow socket `bind` succeeds, the `AF_INET` name is bound to the Fast Socket's Active Message endpoint name via an external name service.

Connection establishment proceeds in a fashion similar to that of TCP/IP. When the name of a remote socket is presented to the `connect` call, Fast Sockets performs a lookup of the name to find any associated Active Messages address (analagous to an ARP lookup of a physical network address), and contacts the remote endpoint if it exists. The initial message contains information necessary to set up the connection, such as socket buffer size and the name of the endpoint performing the `connect`. Once the connection request has been accepted and a connection partner created via the `accept` call, a message containing similar data is returned to the original socket.

3.4 Fast Sockets Limitations

Fast Sockets is implemented as a user-level library. This places a number of limitations on the level of compatibility it can provide. Fast Sockets are not currently shared between two processes via a `fork` call, and they do not survive an `exec`. This poses problems for traditional Internet server daemons and for "super-server" daemons such as `inetd`. Existing at user-level also poses problems for socket termination; in-kernel sockets are gracefully shut down when the owning process exits, either normally or via a `SIGKILL`. These problems can be circumvented either through the use of a dedicated server for exceptional conditions [Maeda & Bershad 1993b] or the use of dynamic interposition code to capture all system calls in a protected user-level process [Ghormley et al. 1996].

Another limitation arises from the reliance of current implementations of Active Messages on polling.. Because an application must explicitly check the network to see if messages have arrived, it is difficult to perform operations such as asynchronous I/O, where events on a socket generate a signal. The current implementation of Fast Sockets is single-threaded and so "asynchronous" events can only occur when a Fast Sockets call is made or when the application explicitly polls for data. This difficulty can be avoided by multithreading the Fast Sockets library, but this hinders portability and may cause difficulties and performance bottlenecks because of synchronization difficulties. As future versions of Active Messages will implement interrupts, we do not see this as a fundamental limitation.

4 Performance

The key measure of Fast Sockets' usefulness is its performance. In this section, we present the raw performance of Fast Sockets, obtained from microbenchmarks, and we present performance results for applications.

Fast Sockets has been implemented using the Generic Active Messages (GAM) specification [Culler et al. 1994] on both HP/UX 9.0.x and Solaris 2.5. The HP/UX platform consists of two 99Mhz HP735's interconnected by an FDDI network and using the Medusa network adapter [Banks & Prudence 1993]. The Solaris platform is a collection of SPARCstation 10 and 20's interconnected by a Myrinet network [Seitz 1994].

4.1 Microbenchmarks

4.1.1 Round-Trip Times

Fast Sockets was intended to lower communications overheads and allow for very low-latency communication, especially for small packets. Our round-trip benchmark does a ping-pong of a given packet size 500 times per iteration, until a 95% confidence interval has been obtained. The results for HP/UX and Solaris are presented in figures 3 and 4.

Fast Sockets consistently outperforms TCP/IP on both HP/UX and Solaris at all packet sizes up to around 7Kbytes. Receive posting is extremely useful, as it keeps Fast Sockets overhead close to that of Active Messages. This can be seen by the "Fast Sockets without receive posting" line on the graph, which is more steeply sloped than the standard Fast Sockets or Active Messages line. The effect is most noticeable on HP/UX, as Fast Sockets without receive posting becomes more expensive than TCP for large packet sizes. This is due to HP/UX using a smaller maximum transmission unit (MTU) than Fast Sockets. On Solaris, Fast Sockets actually outperforms Active Messages between 4Kbytes and 8Kbytes; this is due to the Active Messages maximum transfer unit being 4Kbytes for Myrinet. Although `am_xfer` will handle larger transfers and automatically fragment the packets, Fast Sockets has a better fragmentation heuristic than does Active Messages.

It is interesting to note that TCP overhead for very small packets is lower on Solaris for Ethernet than for Myrinet. This suggests that the Myrinet device driver could be substantially improved, although still unlikely to beat Fast Sockets.

4.1.2 Bandwidth

A secondary design goal of Fast Sockets was to maintain or better the high bandwidths achieved by TCP/IP and by raw Active Messages. Our bandwidth microbenchmark transmitted 500 packets of a given size in one direction, and a response was then generated. This was repeated until a 95% confidence interval was obtained. The results for HP/UX and Solaris are shown in figures 5 and 6, respectively.

Again, Fast Sockets betters TCP, especially for smaller packet sizes. On the HP's, Fast Sockets stays close to Active Messages and well above TCP. On Solaris, however, Fast Sockets performance levels out at 11MB/s, approximately half that of raw Active Messages. One key to this behavior is that Fast Sockets without receive posting achieves approximately the same performance. The microbenchmark demonstrates a situation where receive posting is not useful: data is constantly streaming into the socket, faster than it can be processed, so data is always in the socket buffer or en route to the socket buffer, forcing a copy through the buffer. Fast Sockets is constrained by the copy bandwidth of the SPARCstation 20, which is 28MB/s, significantly lower than the 80MB/s of the Myrinet. Although the same situation arises on the HP/Medusas, the HP has a memory copy bandwidth eqivalent to the SPARC10 (26MB/s), but the network has a much lower bandwidth (12.5MB/s), and so Fast Sockets can keep up with incoming data.

5 Related Work

This section has not yet been rewritten.

Improving communications performance has long been a popular research topic. Work has focused on protocols, on protocol and infrastructure implementations, and on the underlying network device software.

The VMTP protocol [Cheriton & Williamson 1989] attempted to provide a general-purpose protocol optimized for small packets and request-response traffic. It performed quite well for the hardware it was implemented on, but never became widely established; VMTP's design target was request-response and bulk transfer traffic, rather than the byte stream and datagram models provided by the TCP/IP protocol suite.

More recent work [Clark et al. 1989, Watson & Mamrak 1987] argued that protocol implementations, rather than protocol designs, were to blame for poor performance, and that efficient implementations of general-purpose protocols could do as well

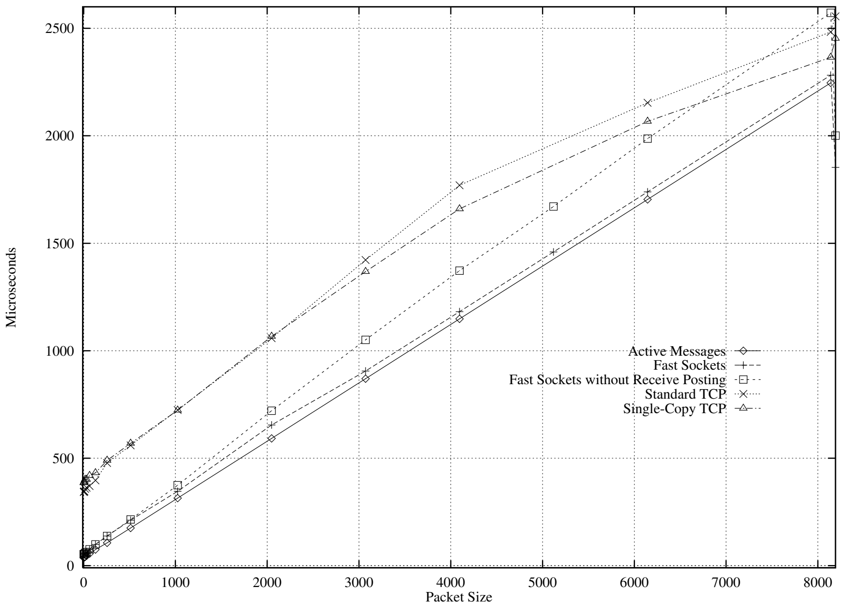

Figure 3: Round-trip performance for Fast Sockets, Active Messages, and TCP for the HP/UX/Medusa platform.

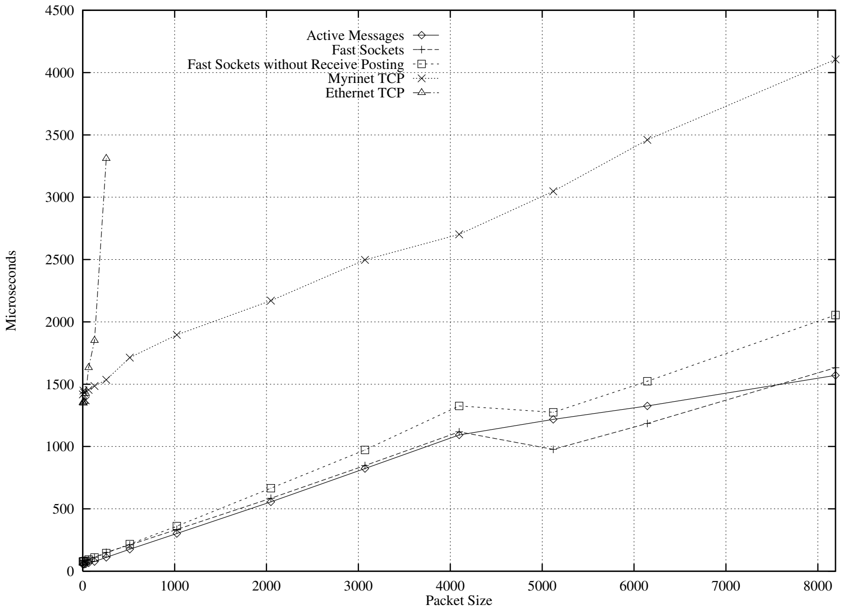

Figure 4: Round-trip performance for Fast Sockets, Active Messages, and TCP on the Solaris/Myrinet platform.

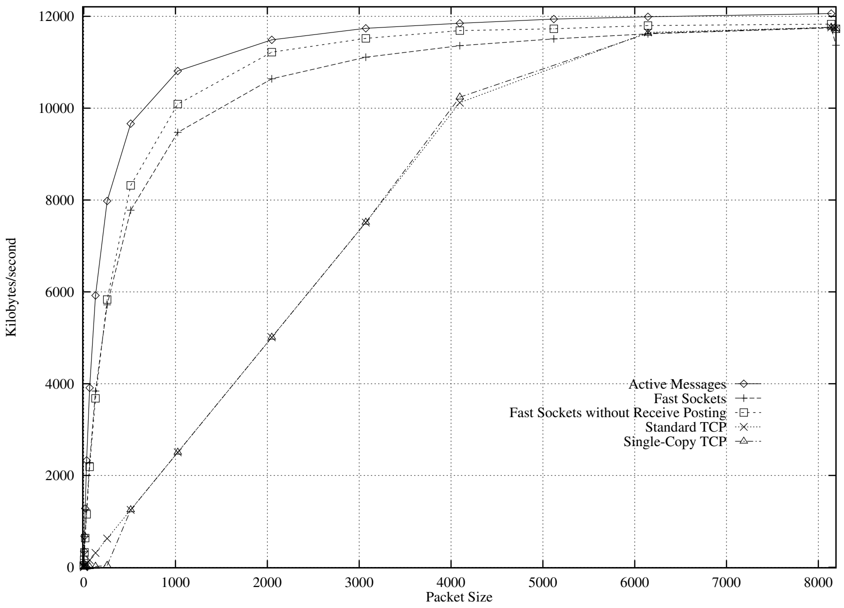

Figure 5: Observed bandwidth for Fast Sockets, TCP, and Active Messages on HP/UX with the Medusa FDDI network interface.

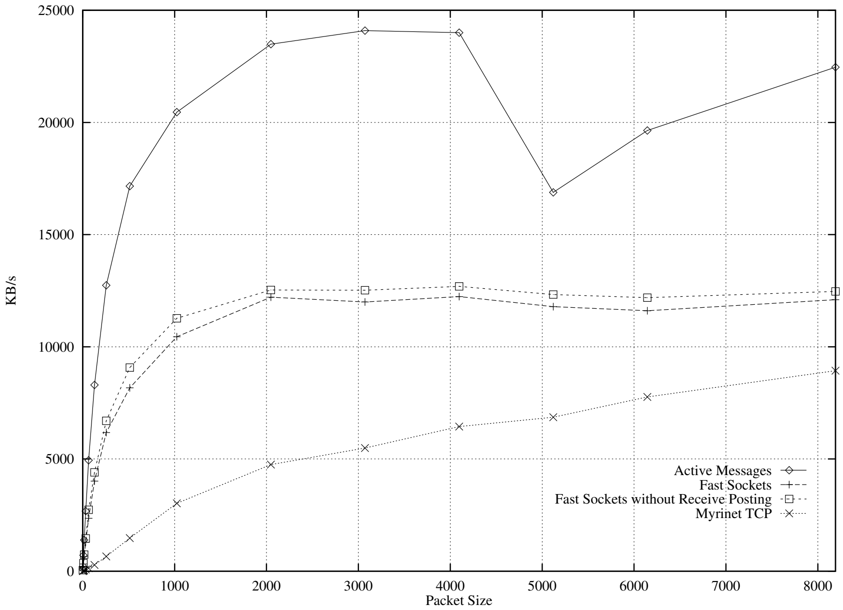

Figure 6: Observed bandwidth for Fast Sockets, TCP, and Active Messages on Solaris using the Myrinet local-area network.

as or better than special-purpose protocols for most applications. The measurements made in [Kay & Pasquale 1992] lend some credence to these arguments; they found that memory operations and operating system overheads played a dominant role in the cost of large packets. For small packets, however, protocol costs were still significant, amounting for up to 33% of processing time for single-byte messages.

The concept of reducing infrastructure costs was explored further in the *x*-kernel [Hutchinson & Peterson 1991, Peterson 1993], an operating system designed for high-performance communications. The original, stand-alone version of the *x*-kernel performed significantly better at communication tasks than did BSD Unix on the same hardware (Sun 3's), using similar implementations of the communications protocols. Later work [Druschel et al. 1994, Pagels et al. 1994, Druschel et al. 1993, Druschel & Peterson 1993] focused on hardware design issues relating to network communication and the use of software techniques to exploit hardware features. Key contributions from this work are the concept of an *application device channel*(ADC), which provides protected user-level access to a network device, and *fbufs*, which provide a mechanism for rapid transfer of data from the network subsystem to the user application. While Active Messages provides the equivalent of an ADC for Fast Sockets, fbufs are not needed, as receive posting allows for data transfer directly into the user application.

Recently, the development of a zero-copy TCP stack in Solaris [Chu 1996] aggressively utilized hardware and operating system features such as direct memory access (DMA), page remapping, and copy-on-write pages to improve communications performance. The result was higher latencies, but higher throughputs, than Fast Sockets on a similar platform (SPARCstation 20, using a 622Mb/s ATM adapter). The higher throughput is a result of the zero-copy implementation, versus Fast Sockets' forced copy (see section 4.1.2). However, this work required an thorough understanding of the Solaris virtual memory subsystem, and changes to the operating system kernel; Fast Sockets is an entirely user-level solution.

An alternative to reducing internal operating system costs is to bypass the operating system altogether, and work either in a user-level library (like Fast Sockets) or a separate user-level server. While Mach 3.0 uses the latter approach [Forin et al. 1991], it yields poor networking performance [Maeda & Bershad 1992]. Both [Maeda & Bershad 1993a] and [Thekkath et al. 1993] explored building TCP into a user-level library linked with existing applications. Both systems, however, attempted only to match in-kernel performance, rather than better it. Further, both systems utilized in-kernel facilities for message transmission, limiting the possible performance improvement. Edwards and Muir [Edwards & Muir 1995] attempted to build an entirely user-level solution, but utilized a TCP stack that had been built for the HP/UX kernel. Their solution replicated the organization of the kernel at user-level and achieved performance worse than the in-kernel version of the TCP stack.

[Kay & Pasquale 1992] showed that interfacing to the network card itself was a major cost for small packets. Recent work has focused on reducing this portion of the protocol cost, and on utilizing the message coprocessors that are appearing on high-performance network controllers such as Myrinet [Seitz 1994]. Active Messages [von Eicken et al. 1992] is the base upon which Fast Sockets is built and is discussed above. Illinois Fast Messages [Pakin et al. 1995] provides an interface similar to that of previous versions of Active Messages, but does not allow processes to share the network. Remote Queues [Brewer et al. 1995] provide low-overhead communications similar to that of Active Messages, but seperates the arrival of messages from the invocation of handlers. Remote Queues also do not require bulk transfer messages to specify a destination.

These next two sections are here because I haven't thought hard about where they "really" belong; they can — and probably should — be moved up, at least the one on TCP. More important is whether they are cogent and coherent.

5.1 TCP/IP

Fast Sockets distinguishes itself from TCP/IP in two ways: its simplicity, and its exploitation of connection-level state. TCP/IP is a wide-area, full-featured network protocol: it is designed to work across network links of varying reliability, latency, and bandwdith, between a wide variety of machines, and to provide a wide array of functionality. Fast Sockets is intended to work within a high-reliability, high-performance local-area network between modern workstations. The functionality of Fast Sockets is intentionally limited, relative to TCP, in order to minimize overhead. Fast Sockets does not attempt to deal with internetwork routing, with congestion control, or with any of the TCP message options. Fast Sockets also collapses the messaging protocol and its interface together, to avoid overhead from excessive modularity [Clark 1982]. Fast Sockets also exploits connection-level information for stream sockets for better performance, for example, using receive posting to improve round-trip latencies. TCP abstractions stop at the port level, which forces mechanisms like network storage.

5.2 U-Net

U-Net [von Eicken et al. 1995] is a user-level network interface developed at Cornell. It virtualizes the network interface, allowing for multiple processes to share the card. U-Net provides a device-level interface to the network and does not place any restrictions on packet formats. This allows versions of TCP/IP and UDP/IP to be run on a U-Net system. The implementation of TCP/IP on U-Net was improved over a standard implementation of TCP/IP through streamlining of the data buffering mechanism.

I am beginning to believe that the TCP benchmarks did not implement the full functionality of the sockets interface; that is, they got high bandwidth by having the user buffer for data reception specified well in advance – i.e., like the Fast Streams work I did for the thesis.

References

[Amer et al. 1987] P. D. Amer, R. N. Kumar, R. bin Kao, J. T. Phillips, and L. N. Cassel. "Local area broadcast network measurement: Traffic characterization". In *Spring COMPCON*, pp. 64–70, San Francisco, California, February 1987.

[Banks & Prudence 1993] D. Banks and M. Prudence. "A high-performance network architecture for a PA-RISC workstation". *IEEE Journal on Selected Areas in Communications*, 11(2):191–202, February 1993.

[Birrell & Nelson 1984] A. D. Birrell and B. J. Nelson. "Implementing remote procedure calls". *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.

[Brewer et al. 1995] E. A. Brewer, F. T. Chong, L. T. Liu, S. D. Sharma, and J. D. Kubiatowicz. "Remote Queues: Exposing message queues for optimization and atomicity". In *Proceedings of SPAA '95*, Santa Barbara, CA, June 1995.

[Caceres et al. 1991] R. Caceres, P. B. Danzig, S. Jamin, and D. J. Mitzel. "Characteristics of wide-area TCP/IP conversations". In *Proceedings of ACM SIGCOMM '91*, September 1991.

[Cheriton & Williamson 1987] D. R. Cheriton and C. L. Williamson. "Network measurement of the VMTP request-response protocol in the V distribued system". In *Proceedings of SIGMETRICS '87*, pp. 216–225, Banff, Alberta, Canada, May 11–14 1987.

[Cheriton & Williamson 1989] D. Cheriton and C. Williamson. "VMTP: A transport layer for high-performance distributed computing". *IEEE Communications*, pp. 37–44, June 1989.

[Chu 1996] H.-K. J. Chu. "Zero-copy TCP in Solaris". In *Proceedings of the 1996* USENIX *Annual Technical Conference*, San Diego, CA, January 1996.

[Claffy et al. 1992] K. C. Claffy, G. C. Polyzos, and H.-W. Braun. "Traffic characteristics of the T1 NSFNET backbone". Technical Report CS92–270, University of California, San Diego, July 1992.

[Clark & Tennenhouse 1990] D. D. Clark and D. L. Tennenhouse. "Architectural considerations for a new generation of protocols". In *Proceedings of SIGCOMM '90*, pp. 200–208, Philadelphia, Pennsylvania, September 1990.

[Clark 1982] D. D. Clark. "Modularity and efficiency in protocol implementation". RFC 817, IETF, July 1982.

[Clark et al. 1989] D. D. Clark, V. Jacobson, J. Romkey, and H. Salwen. "An analysis of TCP processing overhead". *IEEE Communications*, pp. 23–29, June 1989.

[Crowcroft et al. 1992] J. Crowcroft, I. Wakeman, Z. Wang, and D. Sirovica. "Is layering harmful?". *IEEE Network*, 6(1):20–24, January 1992.

[Culler et al. 1994] D. E. Culler, K. Keeton, L. T. Liu, A. Mainwaring, R. P. Martin, S. H. Rodrigues, and K. Wright. *The Generic Active Message Interface Specification*, February 1994.

[Dahlin et al. 1994] M. D. Dahlin, R. Y. Wang, D. A. Patterson, and T. E. Anderson. "Cooperative caching: Using remote client memory to improve file system performance". In *Proceedings of the First Conference on Operating Systems Design and Implementation (OSDI)*, pp. 267–280, October 1994.

[de Prycker 1993] M. de Prycker. *Asynchronous Transfer Mode: Solution for Broadband ISDN*. Ellis Horwood Publishers, second edition, 1993.

[Druschel & Peterson 1993] P. Druschel and L. L. Peterson. "Fbufs: A high-bandwith cross-domain data transfer facility". In *Proceedings of the Fourteenth Annual Symposium on Operating Systems Principles*, pp. 189–202, Asheville, NC, December 1993.

[Druschel et al. 1993] P. Druschel, M. B. Abbott, M. A. Pagels, and L. L. Peterson. "Network subsystem design: A case for an integrated data path". *IEEE Network (Special Issue on End-System Support for High Speed Networks)*, 7(4):8–17, July 1993.

[Druschel et al. 1994] P. Druschel, L. L. Peterson, and B. S. Davie. "Experiences with a high-speed network adapter: A software perspective". In *Proceedings of ACM SIGCOMM '94*, August 1994.

[Edwards & Muir 1995] A. Edwards and S. Muir. "Experiences in implementing a high performance TCP in user-space". In *ACM SIGCOMM '95*, Cambridge, MA, August 30–September 1 1995.

[Feldmeier 1986] D. C. Feldmeier. "Traffic measurement on a token-ring network". In *Proceedings of the 1986 Computer Networking Conference*, pp. 236–243, November 1986.

[Forin et al. 1991] A. Forin, D. Golub, and B. N. Bershad. "An I/O system for Mach 3.0". In *Proceedings of the* USENIX *Mach Symposium*, pp. 163–176, Monterey, CA, November 1991.

[Ghormley et al. 1996] D. P. Ghormley, D. Petrou, and T. E. Anderson. "SLIC: Secure loadable interposition code". In *Proceedings of the Second* USENIX *Symposium on Operating Systems Design and Implementation (OSDI)*, Seattle, WA, October 1996. Submitted for publication.

[Gusella 1990] R. Gusella. "A measurement study of diskless workstation traffic on an Ethernet". *IEEE Transactions on Communications*, 38(9):1557–1568, September 1990.

[Hutchinson & Peterson 1991] N. Hutchinson and L. L. Peterson. "The *x*-kernel: An architecture for implementing network protocols". *IEEE Transactions on Software Engineering*, 17(1):64–76, January 1991.

[ISI 1981] I. S. Institute. "Internet protocol". RFC 791, IETF Network Working Group, September 1981.

[Kay & Pasquale 1992] J. Kay and J. Pasquale. "A performance analysis of TCP/IP and UDP/IP networking software for the DECstation 5000". Technical Report 92/18, Sequoia 2000 Project, 1992.

[Kay & Pasquale 1993] J. Kay and J. Pasquale. "The importance of non-data-touching overheads in TCP/IP". In *Proceedings of the 1993 SIGCOMM*, pp. 259–268, San Francisco, CA, September 1993.

[Keeton et al. 1995] K. Keeton, D. A. Patterson, and T. E. Anderson. "LogP quantified: The case for low-overhead local area networks". In *Hot Interconnects III*, Stanford University, Stanford, CA, August 1995.

[Kleinrock & Naylor 1974] L. Kleinrock and W. E. Naylor. "On measured behavior of the ARPA network". In *AFIPS Proceedings*, volume 43, pp. 767–780, 1974.

[Maeda & Bershad 1992] C. Maeda and B. N. Bershad. "Networking performance for microkernels". In *Proceedings of the Third Workshop on Workstation Operating Systems*, pp. 154–159, 1992.

[Maeda & Bershad 1993a] C. Maeda and B. Bershad. "Protocol service decomposition for high-performance networking". In *Proceedings of the Fourteenth Symposium on Operating Systems Principles*, pp. 244–255, Asheville, NC, December 1993.

[Maeda & Bershad 1993b] C. Maeda and B. Bershad. "Service without servers". In *Workshop on Workstation Operating Systems IV*, October 1993.

[Mainwaring & Culler 1995] A. Mainwaring and D. Culler. *Active Messages: Organization and Applications Programming Interface*, September 1995.

[Martin 1994] R. P. Martin. "HPAM: An Active Message layer for a network of HP workstations". In *Hot Interconnects II*, pp. 40–58, Stanford University, Stanford, CA, August 1994.

[Pagels et al. 1994] M. A. Pagels, P. Druschel, and L. L. Peterson. "Cache and TLB effectiveness in processing network I/O". Technical Report 94-08, University of Arizona, March 1994.

[Pakin et al. 1995] S. Pakin, M. Lauria, and A. Chien. "High-performance messaging on workstations: Illinois Fast Messages(FM) for Myrinet". In *Supercomputing '95*, San Diego, CA, 1995.

[Peterson 1993] L. L. Peterson. "Life on the OS/network boundary". *Operating Systems Review*, 27(2):94–98, April 1993.

[Postel 1981a] J. Postel. "Transmission control protocol". RFC 793, IETF Network Working Group, September 1981.

[Postel 1981b] J. Postel. "User datagram protocol". RFC 768, IETF Network Working Group, August 1981.

[Schoch & Hupp 1980] J. F. Schoch and J. A. Hupp. "Performance of an Ethernet local network". *Communications of the ACM*, 23(12):711–720, December 1980.

[Seitz 1994] C. Seitz. "Myrinet: A gigabit-per-second local area network". In *Hot Interconnects II*, Stanford University, Stanford, CA, August 1994.

[Stevens 1990] W. R. Stevens. *UNIX Network Programming*. Prentice Hall, Englewood Cliffs, NJ, 1990.

[Thekkath et al. 1993] C. A. Thekkath, T. Nguyen, E. Moy, and E. D. Lazowska. "Implementing network protocols at user-level". *IEEE/ACM Transactions on Networking*, pp. 554–565, October 1993.

[von Eicken et al. 1992] 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 the Nineteenth ISCA*, Gold Coast, Australia, May 1992.

[von Eicken et al. 1995] 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 the Fifteenth SOSP*, pp. 40–53, Copper Mountain, CO, December 1995.

[Watson & Mamrak 1987] R. M. Watson and S. A. Mamrak. "Gaining efficiency in transport services by appropriate design and implementation choices". *ACM Transactions on Computer Systems*, 5(2):97–120, May 1987.

[Wright & Stevens 1995] G. R. Wright and W. R. Stevens. *TCP/IP Illustrated: The Implementation*, volume 2. Addison-Wesley, Menlo Park, CA, 1995.