High-Performance Local Area Communication With Fast Sockets

Steven H. Rodrigues, Thomas E. Anderson, and David E. Culler
University of California at Berkeley

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

Abstract

Modern switched networks such as ATM and Myrinet enable low-latency, high-bandwidth communication. This performance has not been realized by current applications, because of the high processing overheads imposed by existing communications software. These overheads are usually not hidden with large packets; most network traffic is small. We have developed Fast Sockets, a local-area communication layer that utilizes a high-performance protocol and exports the Berkeley Sockets programming interface. Fast Sockets realizes round-trip transfer times of 60 microseconds and maximum transfer bandwidth of 33 MB/second between two UltraSPARC 1s connected by a Myrinet network. Fast Sockets obtains performance by collapsing protocol layers, using simple buffer management strategies, and utilizing knowledge of packet destinations for direct transfer into user buffers. Using *receive posting*, we make the Sockets API a single-copy communications layer and enable regular Sockets programs to exploit the performance of modern networks. Fast Sockets transparently reverts to standard TCP/IP protocols for wide-area communication.

This work was supported in part by the Defense Advanced Research Projects Agency (N00600-93-C-2481, F30602-95-C-0014), the National Science Foundation (CDA 0401156), California MICRO, the AT&T Foundation, Digital Equipment Corporation, Exabyte, Hewlett Packard, Intel, IBM, Microsoft, Mitsubishi, Siemens Corporation, Sun Microsystems, and Xerox. Anderson and Culler were also supported by National Science Foundation Presidential Faculty Fellowships, and Rodrigues by a National Science Foundation Graduate Fellowship. The authors can be contacted at {`steverod, tea, culler`}`@cs.berkeley.edu`.

1 Introduction

The development and deployment of high performance local-area networks such as ATM [de Prycker 1993], Myrinet [Seitz 1994], and switched high-speed Ethernet has the potential to dramatically improve communication performance for network applications. These networks are capable of microsecond latencies and bandwidths of hundreds of megabits per second; their switched fabrics eliminate the contention seen on shared-bus networks such as traditional Ethernet. Unfortunately, this raw network capacity goes unused, due to current network communication software.

Most of these networks run the TCP/IP protocol suite [Postel 1981b, Postel 1981c, Postel 1981a]. TCP/IP is the default protocol suite for Internet traffic, and provides inter-operability among a wide variety of computing platforms and network technologies. The TCP protocol provides the abstraction of a reliable, ordered byte stream. The UDP protocol provides an unreliable, unordered datagram service. Many other application-level protocols (such as the FTP file transfer protocol, the Sun Network File System, and the X Window System) are built upon these two basic protocols.

TCP/IP's observed performance has not scaled to the ability of modern network hardware, however. While TCP is capable of sustained bandwidth close to the rated maximum of modern networks, actual bandwidth is very much implementation-dependent. The round-trip latency of commercial TCP implementations is hundreds of microseconds higher than the minimum possible on these networks. Implementations of the simpler UDP protocol, which lack the reliability and ordering mechanisms of TCP, perform little better [Keeton et al. 1995, Kay & Pasquale 1993, von Eicken et al. 1995]. This poor performance is due to the high per-packet processing costs (*processing overhead*) of the protocol implementations. In local-area environments, where on-the-wire times are small, these processing costs dominate small-packet round-trip latencies.

Local-area traffic patterns exacerbate the problems posed by high processing costs. Most LAN traffic consists of small packets [Kleinrock & Naylor 1974, Schoch & Hupp 1980, Feldmeier 1986, Amer et al. 1987, Cheriton & Williamson 1987, Gusella 1990, Caceres et al. 1991, Claffy et al. 1992]. Small packets are the rule even in applications considered bandwidth-intensive: 95% of all packets in a NFS trace performed at the Berkeley Computer Science Division carried less than 192 bytes of user data [Dahlin et al. 1994]; the mean packet size in the trace was 382 bytes. Processing overhead is the dominant transport cost for packets this small, limiting NFS performance on a high-bandwidth network. This is true for other applications: most application-level protocols in local-area use today (X11, NFS, FTP, etc.) operate 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. In the request-response model, processing overhead usually cannot be hidden through packet pipelining or through overlapping communication and computation, making round-trip latency a critical factor in protocol performance.

Traditionally, there are several methods of attacking the processing overhead problem: changing the application programming interface (API), changing the underlying network protocol, changing the implementation of the protocol, or some combination of these approaches. *Changing the API* modifies the code used by applications to access communications functionality. While this approach may yield better performance for new applications, legacy applications must be re-implemented to gain any benefit. *Changing the communications protocol* changes the "on-the-wire" format of data and the actions taken during a communications exchange — for example, modifying the TCP packet format. A new or modified protocol may improve communications performance, but at the price of incompatibility: applications communicating via the new protocol are unable to share data directly with applications using the old protocol. *Changing the protocol implementation* rewrites the software that implements a particular protocol; packet formats and protocol actions do not change, but the code that performs these actions does. While this approach provides full compatibility with existing protocols, fundamental limitations of the protocol design may limit the performance gain.

Recent systems, such as Active Messages [von Eicken et al. 1992], Remote Queues [Brewer et al. 1995], and native U-Net [von Eicken et al. 1995], have used the first two methods; they implement new protocols and new programming interfaces to obtain improved local-area network performance. The protocols and interfaces are lightweight and provide programming abstractions that are similar to the underlying hardware. All of these systems realize latencies and throughput close to the physical limits of the network. However, none of them offer compatibility with existing applications.

Other work has tried to improve performance by re-implementing TCP. 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 inter-operate with other TCP/IP implementations and improve throughput and latency relative to standard TCP/IP stacks. Both implementations can realize the full bandwidth of the network for large packets. However, both systems have round-trip latencies considerably higher than the raw network.

This paper presents our solution to the overhead problem: a new communications protocol and implementation for local-area networks that exports the Berkeley Sockets API, uses a low-overhead protocol for local-area use, and reverts to standard protocols for wide-area communication. The Sockets API is a widely used programming interface that treats network connections as files; application programs read and write network connections exactly as they read and write files. The Fast Sockets protocol has been designed and implemented to obtain a low-overhead data transmission/reception path. Should a Fast Sockets program attempt to connect with a program outside the local-area network, or to a non-Fast Sockets program, the software transparently reverts to standard TCP/IP sockets. These features enable high-performance communication through relinking existing application programs.

Fast Sockets achieves its performance through a number of strategies. It uses a lightweight protocol and efficient buffer management to minimize bookkeeping costs. The communication protocol and its programming interface are integrated to eliminate module-crossing costs. Fast Sockets eliminates copies within the protocol stack by using knowledge of packet memory destinations. Additionally, Fast Sockets was implemented without modifications to the operating system kernel.

A major portion of Fast Sockets' performance is due to *receive posting*, a technique of utilizing information from the API about packet destinations to minimize copies. This paper describes the use of receive posting in a high-performance communications stack. It also describes the design and implementation of a low-overhead 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 communication 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 communication performance. Section 6 presents our conclusions and directions for future work in this area.

2 Problems with TCP/IP

While TCP/IP can achieve good throughput on currently deployed networks, its round-trip latency is usually poor. Further, observed bandwidth and round-trip latencies on next-generation network technologies such as Myrinet and ATM do not begin to approach the raw capabilities 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 communication performance.

2.1 Built for the Wide Area

TCP/IP was originally designed, and is usually implemented, for wide-area networks. While TCP/IP is usable on a local-area network, it is not optimized for this domain. For example, TCP uses an in-packet checksum for end-to-end reliability, despite the presence of per-packet CRC's in most modern network hardware. But computing this checksum is expensive, creating a bottleneck in packet processing. IP uses header fields such as 'Time-To-Live' which are only relevant in a wide-area environment. IP also supports internetwork routing and in-flight packet fragmentation and reassembly, features which are not useful in a local-area environment. The TCP/IP model assumes communication between autonomous machines that cooperate only minimally. However, machines on a local-area network frequently share a common administrative service, a common file system, and a common user base. It should be possible to extend this commonality and cooperation into the network communication software.

2.2 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 interface is usually the topmost layer, sitting above the protocol. The protocol layer may contain sub-layers: for example, the TCP protocol code sits above the IP protocol code. Below the protocol layer is the interface layer, which communicates with the network hardware. The interface layer usually has two portions, the network programming interface, which prepares outgoing data packets, and the network device driver, which transfers data to and from the network interface card (NIC).

This multi-layer organization enables protocol stacks to be built from many combinations of protocols, programming interfaces, and network devices, but this flexibility comes at the price of performance. Layer transitions can be costly in time and programming effort. Each layer may use a different abstraction for data storage and transfer, requiring data transformation at every layer boundary. Layering also restricts information transfer. Hidden implementation details of each layer can cause large, unforeseen impacts on performance [Clark 1982, Crowcroft et al. 1992]. Mechanisms have been proposed to overcome these difficulties [Clark & Tennenhouse 1990], but existing work has focused on message throughput, rather than protocol latency [Abbott & Peterson 1993]. Also, the number of programming interfaces and protocols is small: there are two programming interfaces (Berkeley Sockets and the System V Transport Layer Interface) and only a few data transfer protocols (TCP/IP and UDP/IP) in widespread usage. This paucity of distinct layer combinations means that the generality of the multi-layer organization is wasted. Reducing the number of layers traversed in the communications stack should reduce or eliminate these layering costs for the common case of data transfer.

2.3 Complicated Memory Management

Current TCP/IP implementations use a complicated memory management mechanism. This system exists for a number of reasons. First, a multi-layered protocol stack means packet headers are added (or removed) as the packet moves downward (or upward) through the stack. This should be done easily and efficiently, without excessive copying. Second, buffer memory inside the operating system kernel is a scarce resource; it must be managed in a space-efficient fashion. This is especially true for older systems with limited physical memory.

To meet these two requirements, mechanisms such as the Berkeley Unix `mbuf` have been used. An `mbuf` can directly hold a small amount of data, and `mbufs` can be chained to manage larger data sets. Chaining makes adding and removing packet headers easy. The `mbuf` abstraction is not cheap, however: 15% of the processing time for small TCP packets is consumed by `mbuf` management [Kay & Pasquale 1993]. Additionally, to take advantage of the `mbuf` abstraction, user data must be copied into and out of `mbufs`, which consumes even more time in the data transfer critical path. This copying means that nearly one-quarter of the small-packet processing time in a commercial TCP/IP stack is spent on memory management issues. Reducing the overhead of memory management is therefore critical to improving communications performance.

3 Fast Sockets Design

Fast Sockets is an implementation of the Sockets API that provides high-performance communication and inter-operability with existing programs. It yields high-performance communication through a low-overhead protocol layered on top of a low-overhead transport mechanism (Active Messages). Interoperability with existing programs is obtained by supporting most of the Sockets API and transparently using existing protocols for communication with non-Fast Sockets programs. In this section, we describe the design decisions and consequent trade-offs of Fast Sockets.

3.1 Built For The Local Area

Fast Sockets is targeted at local-area networks of workstations, where processing overhead is the primary limitation on communications performance. For low-overhead access to the network with a defined, portable interface, Fast Sockets uses Active Messages [von Eicken et al. 1992, Martin 1994, Culler et al. 1994, Mainwaring & Culler 1995]. An *active message* is a network packet which contains the name of a *handler function* and data for that handler. When an active message arrives at its destination, the handler is looked up and invoked with the data carried in the message. While conceptually similar to a remote procedure call [Birrell & Nelson 1984], an active message is constrained in the amount and types of data that can be carried and passed to handler functions. These constraints enable the structuring of an Active Messages layer for high performance. Also, Active Messages uses protected user-level access to the network interface, removing the operating system kernel from the critical path. Active messages are reliable, but not ordered.

Using Active Messages as a network transport involves a number of trade-offs. Active Messages has its own 'on-the-wire' packet format; this makes a full implementation of TCP/IP on Active Messages infeasible, as the transmitted packets will not be comprehensible by other TCP/IP stacks. Instead, we elected to implement our own protocol for local area communication, and fall back to normal TCP/IP for wide-area communication. Active Messages operates primarily at user-level; although access to the network device is granted and revoked by the operating system kernel, data transmission and reception is performed by user-level code. For maximum performance, Fast Sockets is written as a user-level library. While this organization avoids user-kernel transitions on communications events (data transmission and reception), it makes the maintenance of shared and global state, such as the TCP and UDP port name spaces, difficult. Some global state can be maintained by simply using existing facilities. For example, the port name spaces can use the in-kernel name management functions. Other shared or global state can be maintained by using a server process to store this state, as described in [Maeda & Bershad 1993b]. Finally, using Active Messages limits Fast Sockets communication to the local-area domain. Fast Sockets supports wide-area communication by automatically switching to standard network protocols for non-local addresses. It is a reasonable trade-off, as endpoint processing overheads are generally not the limiting factor for internetwork communication.

Active Messages does have a number of benefits, however. The handler abstraction is extremely useful. A handler executes upon message reception at the destination, analogous to a network interrupt. At this time, the protocol can store packet data for later use, pass the packet data to the user program, or deal with exceptional conditions. Message handlers allow for a wide variety of control operations within a protocol without slowing down the critical path of data transmission and reception. Handler arguments enable the easy separation of packet data and metadata: packet data (that is, application data) is carried as a bulk transfer argument, and packet metadata (protocol headers) are carried in the remaining word-sized arguments. This is only possible if the headers can fit into the number of argument words provided; for our local-area protocol, the 8 words supplied by Active Messages is sufficient.

Fast Sockets further optimizes for the local area by omitting features of TCP/IP unnecessary in that environment. For example, Fast Sockets uses the checksum or CRC of the network hardware instead of one in the packet header; software checksums make little sense when packets are only traversing a single network. Fast Sockets has no equivalent of IP's 'Time-To-Live' field, or IP's internetwork routing support. Since maximum packet sizes will not change within a local area network, Fast Sockets does not support IP-style in-flight packet fragmentation.

3.2 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 communications stack together. This avoids abstraction conflicts between the programming interface and the protocol and reduces the number of conversions between layer abstractions, further reducing processing overheads.

The actual network device interface, Active Messages, remains a distinct layer from Fast Sockets. This facilitates the portability of Fast Sockets between different operating systems and network hardware. Active Messages implementations are available for the Intel Paragon [Liu & Culler 1995], FDDI [Martin 1994], Myrinet [Mainwaring & Culler 1995], and ATM [von Eicken et al. 1995]. Layering costs are kept low because Active Messages is a thin layer, and all of its implementation-dependent constants (such as maximum packet size) are exposed to higher layers.

The Fast Sockets layer stays lightweight by exploiting Active Message handlers. Handlers allow rarely-used functionality, such as connection establishment, to be implemented without affecting the critical path of data transmission and reception. There is no need to test for uncommon events when a packet arrives — this is encoded directly in the handler. Unusual data transmission events, such as *out-of-band* data, also use their own handlers to keep normal transfer costs low.

Reducing the number of layers and exploiting Active Message handlers lowers the protocol- and API-specific costs of communication. While collapsing layers means that every protocol layer–API combination has to be written anew, the number of such combinations is relatively few, and the number of distinct operations required for each API is small.

3.3 Simple Buffer Management

Fast Sockets avoids the complexities of `mbuf`-style memory management by using a single, contiguous virtual memory buffer for each socket. Data is transferred directly into this buffer via Active Message data transfer messages. The message handler places data sequentially into the buffer to maintain in-order delivery and make data transfer to a user buffer a simple memory copy. The argument words of the data transfer messages carry packet metadata; because the argument words are passed separately to the handler, there is no need for the memory management system to strip off packet headers.

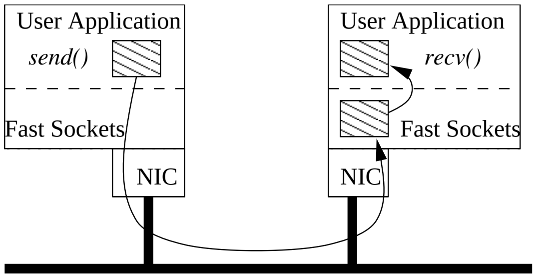

Figure 1: Data transfer in Fast Sockets. A `send()` call transmits the data directly from the user buffer into the network. When it arrives at the remote destination, the message handler places it into the socket buffer, and a subsequent `recv()` call copies it into the user buffer.

Fast Sockets eliminates send buffering. Because many user applications rely heavily on small packets and on request-response behavior, delaying packet transmission only serves to increase user-visible latency. Eliminating send-side buffering reduces protocol overhead because there are no copies on the send side of the protocol path — Active Messages already provides reliability.

Figure 1 shows Fast Sockets' send mechanism and buffering techniques.

A possible problem with this approach is that having many Fast Sockets consumes considerably more memory than the global `mbuf` pool used in traditional kernel implementations. This is not a major concern, for two reasons. First, the memory capacity of current workstations is very large; the scarce physical memory situations that the traditional mechanisms are designed for is generally not a problem. Second, the socket buffers are located in pageable virtual memory — if memory and scheduling pressures are severe enough, the buffer can be paged out. Although paging out the buffer will lead to worse performance, we expect that this is an extremely rare occurrence.

A more serious problem with placing socket buffers in user virtual memory is that it becomes extremely difficult to share the socket buffer between processes. Such sharing can arise due to a `fork()` call, for instance. Currently, Fast Sockets cannot be shared between processes (see section 3.7).

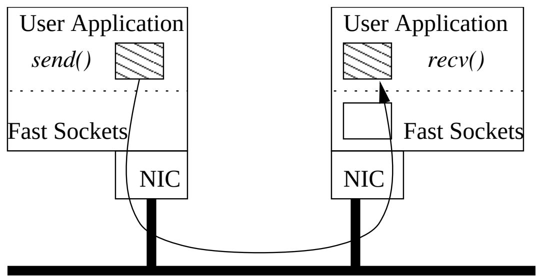

Figure 2: Data transfer via receive posting. If a `recv()` call is issued prior to the arrival of the desired data, the message handler can directly route the data into the user buffer, bypassing the socket buffer.

3.4 Copy Avoidance

The standard Fast Sockets data transfer mechanism involves two copies along the receive path, from the network interface to the socket buffer, and then from the socket buffer to the user buffer specified in a `recv()` call. While the copy from the network interface cannot be avoided, the second copy increases the processing overhead of a data packet, and consequently, round-trip latencies. A high-performance communications layer should bypass this second copy whenever possible.

It is possible, under certain circumstances, to avoid the copy through the socket buffer. If the data's final memory destination is already known upon packet arrival (through a `recv()` call), the data can be directly copied there. We call this technique *receive posting*. Figure 2 shows how receive posting operates in Fast Sockets. If the message handler determines that an incoming packet will satisfy an outstanding `recv()` call, the packet's contents are received directly into the user buffer. The socket buffer is never touched.

Receive posting in Fast Sockets is possible because of the integration of the API and the protocol, and the handler facilities of Active Messages. Protocol–API integration allows knowledge of the user's destination buffer to be passed down to the packet processing code. Using Active Message handlers means that the Fast Sockets code can decide where to place the incoming data when it arrives.

3.5 Design Issues In Receive Posting

Many high-performance communications systems are now using *sender-based memory management*, where the sending node determines the data's final memory destination. Examples of communications layers with this memory management style are Hamlyn [Buzzard et al. 1996] and the SHRIMP network interface [Blumrich et al. 1994]. Also, the initial version of Generic Active Messages [Culler et al. 1994] offered only sender-based memory management for data transfer.

Sender-based memory management has a major drawback for use in byte-stream and message-passing APIs such as Sockets. With sender-based management, the sending and receiving endpoints must synchronize and agree on the destination of each packet. This synchronization usually takes the form of a message exchange, which imposes a time cost and a resource cost (the message exchange uses network bandwidth). To minimize this synchronization cost, the original version of Fast Sockets used the socket buffer as a default destination. This meant that when a `recv()` call was made, data already in flight had to be directed through the socket buffer, as shown in Figure 1. These synchronization costs lowered Fast Sockets' throughput relative to Active Messages' on systems where the network was not a limiting factor, such as the Myrinet network.

Generic Active Messages 1.5 introduced a new data transfer message type that did not require sender-based memory management. This message type, the *medium message*, transferred data into an anonymous region of user memory and then invoked the handler with a pointer to the packet data. The handler was responsible for long-term data storage, as the buffer was deallocated upon handler completion. Keeping memory management on the receiver improves Fast Sockets performance considerably. Synchronization is now only required between the API and the protocol layers, which is simple due to their integration. The receive handler now determines the memory destination of incoming data at packet arrival time, enabling a `recv()` of in-flight data to benefit from receive posting and bypass the socket buffer. The net result is that receiver-based memory management did not significantly affect Fast Sockets' round-trip latencies and improved large-packet throughput substantially, to within 10% of the throughput of raw Active Messages.

The use of receiver-based memory management has some trade-offs relative to sender-based systems. In a true zero-copy transport layer, where packet data is transferred via DMA to user memory, transfer to an anonymous region can place an extra copy into the receive path. This is not a problem for two reasons. First, many current I/O architectures, like that on the SPARC, are limited in the memory regions that they can perform DMA operations to. Second, DMA operations usually require memory pages to be pinned in physical memory, and pinning an arbitrary page can be an expensive operation. For these reasons, current versions of Active Messages move data from the network interface into an anonymous staging area before either invoking the message handler (for medium messages) or placing the data in its final destination (for standard data transfer messages). Consequently, receiver-based memory management does not impose a large cost in our current system. For a true zero-copy system, it should be possible for a handler to be responsible for moving data from the network interface card to user memory.

Based on our experiences implementing Fast Sockets with both sender- and receiver-based memory management schemes, we believe that, for messaging layers such as Sockets, the performance increase delivered by receiver-based management schemes outweigh the implementation costs.

3.6 Other Fast Socket Operations

Fast Sockets supports the full sockets API, including socket creation, name management, and connection establishment. The `socket` call for the for the `AF_INET` address family and the "default protocol" creates a Fast Socket; this allows programs to explicitly request TCP or UDP sockets in a standard way. Fast Sockets utilizes existing name management facilities. 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. Other operations, such as `setsockopt()` and `getsockopt()`, work in a similar fashion.

Shadow sockets are also used for connection establishment. When a `connect()` call is made by the user application, Fast Sockets determines if the name is in the local subnet. For local addresses, the shadow socket performs a `connect()` to a port number that is determined through a hash function. If this `connect()` succeeds, then a handshake is performed to bootstrap the connection. Should the `connect()` to the hashed port number fail, or the connection bootstrap process fail, then a normal `connect()` call is performed. This mechanism allows a Fast Sockets program to connect to a non-Fast Sockets program without difficulties.

An `accept()` call becomes more complicated as a result of this scheme, however. Because both Fast Sockets and non-Fast Sockets programs can connect to a socket that has performed a `listen()` call, there two distinct port numbers for a given socket. The port supplied by the user accepts connection requests from programs using normal protocols. The second port is derived from hashing on the user port number, and is used to accept connection requests from Fast Sockets programs. An `accept()` call multiplexes connection requests from both ports.

The connection establishment mechanism has some trade-offs. It utilizes existing name and connection management facilities, minimizing the amount of code in Fast Sockets. Using TCP/IP to bootstrap the connection can impose a high time cost, which limits the realizable throughput of short-lived connections. Using two port numbers also introduces the potential problem of conflicts: the Fast Sockets-generated port number could conflict with a user port. We do not expect this to realistically be a problem.

3.7 Fast Sockets Limitations

Fast Sockets is a user-level library. This limits its full compatibility with the Sockets abstraction. First, applications must be relinked to use Fast Sockets, although no code changes are required. More seriously, Fast Sockets cannot currently be shared between two processes (for example, via a `fork()` call), and all Fast Sockets state is lost upon an `exec()` or `exit()` call. This poses problems for traditional Internet server daemons and for "super-server" daemons such as `inetd`, which depend on a `fork()` for each incoming request. User-level operation also causes problems for socket termination; standard TCP/IP sockets are gracefully shut down on process termination. These problems are not insurmountable. Sharing Fast Sockets requires an Active Messages layer that allows endpoint sharing and either the use of a dedicated server process [Maeda & Bershad 1993b] or the use of shared memory for every Fast Socket's state. Recovering Fast Sockets state lost during an `exec()` call can be done via a dedicated server process, where the Fast Socket's state is migrated to and from the server before and after the `exec()` — similar to the method used by the user-level Transport Layer Interface [Stevens 1990].

The Fast Sockets library is currently single-threaded. This is problematic for current versions of Active Messages because an application must explicitly touch the network to receive messages. Since a user application could engage in an arbitrarily long computation, it is difficult to implement operations such as asynchronous I/O. While multi-threading offers one solution, it makes the library less portable, and imposes synchronization costs.

4 Performance

This section presents our performance measurements of Fast Sockets, both for microbenchmarks and a few applications. The microbenchmarks assess the success of our efforts to minimize overhead and round-trip latency. We report round-trip times and sustained bandwidth available from Fast Sockets, using both our own microbenchmarks and two publicly available benchmark programs. The true test of Fast Sockets' usefulness, however, is how well its raw performance is exposed to applications. We present results from an FTP application to demonstrate the usefulness of Fast Sockets in a real-world environment.

4.1 Experimental Setup

Fast Sockets has been implemented using Generic Active Messages (GAM) [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 UltraSPARC 1's connected via a Myrinet network [Seitz 1994]. For all tests, there was no other load on the network links or switches.

Our microbenchmarks were run against a variety of TCP/IP setups. The standard HP/UX TCP/IP stack is well-tuned, but there is also a single-copy stack designed for use on the Medusa network interface. We ran our tests on both stacks. While the Solaris TCP/IP stack has reasonably good performance, the Myrinet TCP/IP drivers do not. Consequently, we also ran our microbenchmarks on a 100-Mbit Ethernet, which has an extremely well-tuned driver.

We used Generic Active Messages 1.0 on the HP/UX platform as a base for Fast Sockets and as our Active Messages layer. The Solaris tests used Generic Active Messages 1.5, which adds support for medium messages (receiver-based memory management) and client-server program images.

4.2 Microbenchmarks

4.2.1 Round-Trip Latency

Our round-trip microbenchmark is a simple ping-pong test between two machines for a given transfer size. The ping-pong is repeated until a 95% confidence interval is obtained. TCP/IP and Fast Sockets use the same program, Active Messages uses different code but the same algorithm. We used the TCP/IP `TCP_NODELAY` option to force packets to be transmitted as soon as possible (instead of the default behavior, which attempts to batch small packets together); this reduces throughput for small transfers, but yields better round-trip times. The socket buffer size was set to 64 Kbytes[1], as this also improves TCP/IP round-trip latency. We tested TCP/IP and Fast Sockets for transfers up to 64 Kbytes; Active Messages only supports 4 Kbyte messages.

Figures 3 and 4 present the results of the round-trip microbenchmark. Fast Sockets achieves low latency round-trip times, especially for small packets. Round-trip time scales linearly with increasing transfer size, reflecting the time spent in moving the data to the network card. There is a "hiccup" at 4096 bytes on the Solaris platform, which is the maximum packet size for Active Messages. This occurs because Fast Sockets' fragmentation algorithm attempts to balance packet sizes for better round-trip and bandwidth characteristics. Fast Sockets' overhead is low, staying relatively constant at 25–30 microseconds (over that of Active Messages).

Table 1 shows the results of a least-squares linear regression analysis of the Solaris round-trip microbenchmark. We show t_0, the estimated cost for a

Layer	Per-Byte (μsec)	t_0 (μsec)	Actual Startup
Fast Sockets	0.068	157.8	57.8
Active Messages	0.129	38.9	45.0
TCP/IP (Myrinet)	0.076	736.4	533.0
TCP/IP (Fast Ethernet)	0.174	366.2	326.0
Small Packets			
Fast Sockets	0.124	61.4	57.8
Active Messages	0.123	45.4	45.0
TCP/IP (Myrinet)	0.223	533.0	533.0
TCP/IP (Fast Ethernet)	0.242	325.0	326.0

Table 1: Least-squares analysis of the Solaris round-trip microbenchmark. The per-byte and estimated startup (t_0) costs are for round-trip latency, and are measured in microseconds. The actual startup costs (for a single-byte message) are also shown. Actual and estimated costs differ because round-trip latency is not strictly linear. Per-byte costs for Fast Sockets are lower than for Active Messages because Fast Sockets benefits from packet pipelining; the Active Messages test only sends a single packet at a time. "Small Packets" examines protocol behavior for packets smaller than 1K; here, Fast Sockets and Active Messages do considerably better than TCP/IP.

[1] TCP/IP is limited to a maximum buffer size of 56 Kbytes.

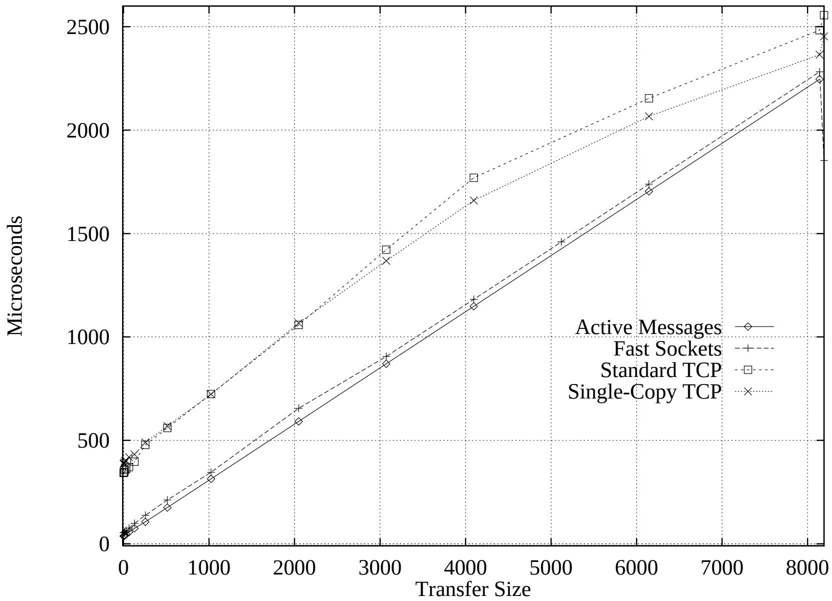

Figure 3: Round-trip performance for Fast Sockets, Active Messages, and TCP for the HP/UX/Medusa platform. Active Messages for HP/UX cannot transfer more than 8140 bytes at a time.

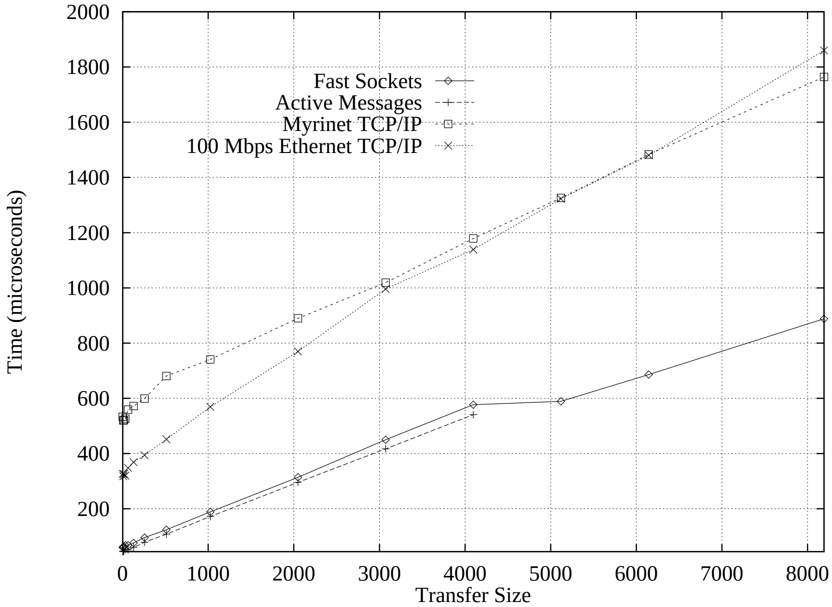

Figure 4: Round-trip performance for Fast Sockets, Active Messages, and TCP on the Solaris/Myrinet platform. Active Messages for Myrinet can only transfer up to 4096 bytes at a time.

0-byte packet, and the marginal cost for each additional data byte. Surprisingly, Fast Sockets' cost-per-byte appears to be lower than that of Active Messages. This is because the Fast Sockets per-byte cost is reported for a 64K range of transfers while Active Messages' per-byte cost is for a 4K range. The Active Messages test minimizes overhead by not implementing in-order delivery, which means only one packet can be outstanding at a time. Both Fast Sockets and TCP/IP provide in-order delivery, which enables data packets to be pipelined through the network and thus achieve a lower per-byte cost. The per-byte costs of Fast Sockets for small packets (less than 1 Kbyte) are slightly higher than Active Messages. While TCP's long-term per-byte cost is only about 15% higher than that of Fast Sockets, its performance for small packets is much worse, with per-byte costs twice that of Fast Sockets and startup costs 5–10 times higher.

Another surprising element of the analysis is that the overall t_0 and t_0 for small packets is very different, especially for Fast Sockets and Myrinet TCP/IP. Both protocols pipeline packets, which lowers round-trip latencies for multi-packet transfers. This causes a non-linear round-trip latency function, yielding different estimates of t_0 for single-packet and multi-packet transfers.

4.2.2 Bandwidth

Our bandwidth microbenchmark does 500 `send()` calls of a given size, and then waits for a response. This is repeated until a 95% confidence interval is obtained. As with the round-trip microbenchmark, the TCP/IP and Fast Sockets measurements were derived from the same program and Active Messages results were obtained using the same algorithm, but different code. Again, we used the TCP/IP `TCP_NODELAY` option to force immediate packet transmission, and a 64 Kbyte socket buffer.

Results for the bandwidth microbenchmark are shown in Figures 5 and 6. Fast Sockets is able to realize most of the available bandwidth of the network. On the UltraSPARC, the SBus is the limiting factor, rather than the network, with a maximum throughput of about 45 MB/s. Of this, Active Messages exposes 35 MB/s to user applications. Fast Sockets can realize about 90% of the Active Messages bandwidth, losing the rest to memory movement. Myrinet's TCP/IP only realizes 90% of the Fast Sockets bandwidth, limited by its high processing overhead.

Table 2 presents the results of a least-squares fit of the bandwidth curves to the equation

$$y = \frac{r_\infty x}{n_{\frac{1}{2}} + x}$$

Layer	r_∞ (MB/s)	$n_{\frac{1}{2}}$ (bytes)
Fast Sockets	32.9	441
Active Messages	39.2	372
TCP/IP (Myrinet)	29.6	2098
TCP/IP (Fast Ethernet)	11.1	530

Table 2: Least-squares regression analysis of the Solaris bandwidth microbenchmark. r_∞ is the maximum bandwidth of the network and is measured in megabytes per second. The half-power point ($n_{\frac{1}{2}}$) is the packet size that delivers half of the maximum throughput, and is reported in bytes. TCP/IP was run with the `TCP_NODELAY` option, which attempts to transmit packets as soon as possible rather than coalescing data together.

which describes an idealized bandwidth curve. r_∞ is the theoretical maximum bandwidth realizable by the communications layer, and $n_{\frac{1}{2}}$ is the *half-power point* of the curve. The half-power point is the transfer size at which the communications layer can realize half of the maximum bandwidth. A lower half-power point means a higher percentage of packets that can take advantage of the network's bandwidth. This is especially important given the frequency of small messages in network traffic. Fast Sockets' half-power point is only 18% larger than that of Active Messages, at 441 bytes. Myrinet TCP/IP realizes a maximum bandwidth 10% less than Fast Sockets but has a half-power point four times larger. Consequently, even though both protocols can realize much of the available network bandwidth, TCP/IP needs much larger packets to do so, reducing its usefulness for many applications.

4.2.3 `netperf` and `ttcp`

Two commonly used microbenchmarks for evaluating network software are `netperf` and `ttcp`. Both of these benchmarks are primarily designed to test throughput, although `netperf` also includes a test of request-response throughput, measured in transactions/second.

We used a version 1.2 of `ttcp`, modified to work under Solaris, and `netperf` 2.11 for testing. The throughput results are shown in Figure 7, and our analysis is in Table 3.

A curious result is that the half-power points for `ttcp` and `netperf` are substantially lower for TCP/IP than on our bandwidth microbenchmark. One reason for this is that the maximum throughput for

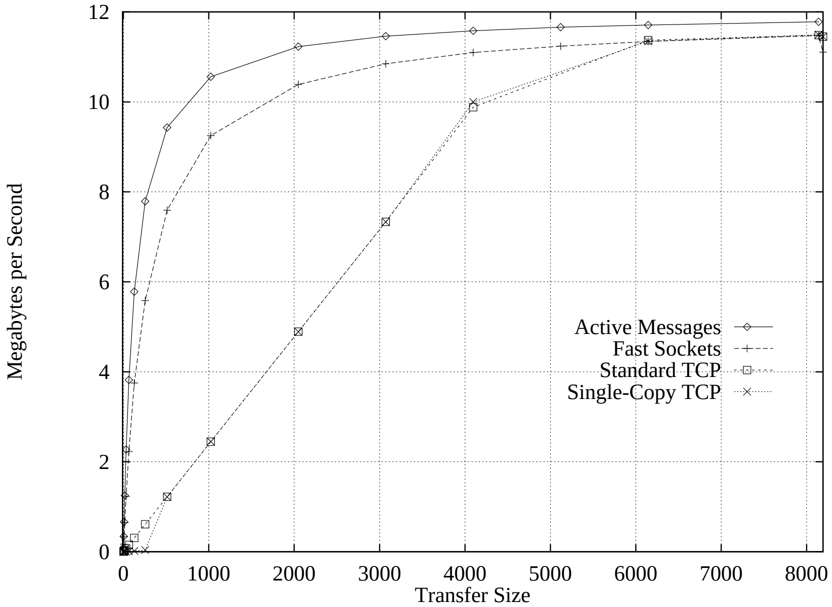

Figure 5: Observed bandwidth for Fast Sockets, TCP, and Active Messages on HP/UX with the Medusa FDDI network interface. The memory copy bandwidth of the HP735 is greater than the FDDI network bandwidth, so Active Messages and Fast Sockets can both realize close to the full bandwidth of the network.

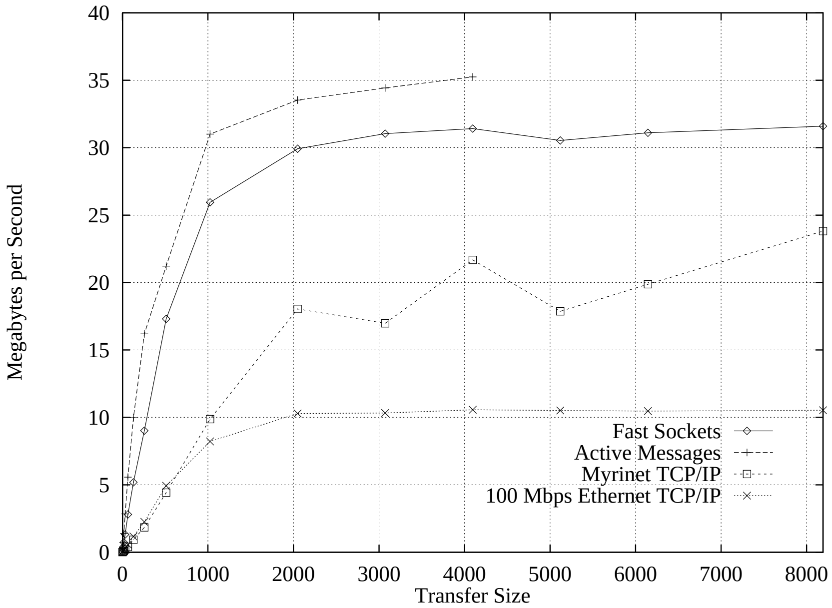

Figure 6: Observed bandwidth for Fast Sockets, TCP, and Active Messages on Solaris using the Myrinet local-area network. The bus limits the maximum throughput to 45MB/s. Fast Sockets is able to realize much of the available bandwidth of the network because of receive posting.

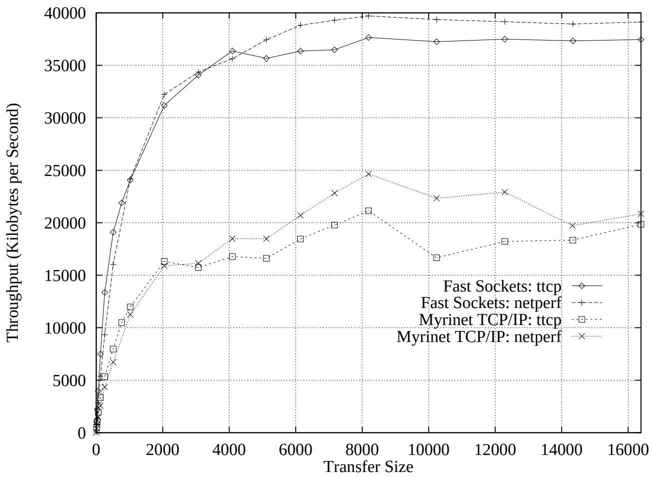

Figure 7: `Ttcp` and `netperf` bandwidth measurements on Solaris for Fast Sockets and Myrinet TCP/IP.

Program/ Communication Layer	r_∞ (MB/s)	$n_{\frac{1}{2}}$ (bytes)
`ttcp`		
Fast Sockets	38.57	560.7
Myrinet TCP/IP	19.59	687.5
`netperf`		
Fast Sockets	41.57	785.7
Myrinet TCP/IP	23.72	1189

Table 3: Least-squares regression analysis of the `ttcp` and `netperf` microbenchmarks. These tests were run on the Solaris 2.5.1/Myrinet platform. TCP/IP half-power point measures are lower than in Table 2 because both `ttcp` and `netperf` attempt to improve small-packet bandwidth at the price of small-packet latency.

TCP/IP is only about 50–60% that of Fast Sockets. Another reason is that TCP defaults to batching small data writes in order to maximize throughput (the *Nagle algorithm*) [Nagle 1984], and these tests do not disable the algorithm (unlike our microbenchmarks); however, this behavior trades off small-packet bandwidth for higher round-trip latencies, as data is held at the sender in an attempt to coalesce data.

The `netperf` microbenchmark also has a "request-response" test, which reports the number of transactions per second a communications stack is capable of for a given request and response size. There are two permutations of this test, one using an existing connection and one that establishes a connection every time; the latter closely mimics the behavior of protocols such as HTTP. The results of these tests are reported in transactions per second and shown in Figure 8.

The request-response test shows Fast Sockets' Achilles heel: its connection mechanism. While Fast Sockets does better than TCP/IP for request-response behavior with a constant connection (Figure 8(a)), introducing connection startup costs (Figure 8(b)) reduces or eliminates this advantage dramatically. This points out the need for an efficient, high-speed connection mechanism.

4.3 Applications

Microbenchmarks are useful for evaluating the raw performance characteristics of a communications implementation, but raw performance does not express the utility of a communications layer. Instead, it is important to characterize the difficulty of integrating the communications layer with existing applications, and the performance improvements realized by those applications. This section examines how well Fast Sockets supports the real-life demands of a network application.

4.3.1 File Transfer

File transfer is traditionally considered a bandwidth-intensive application. However, the FTP protocol that is commonly used for file transfer still has a request-response nature. Further, we wanted to see what improvements in performance, if any, would be realized by using Fast Sockets for an application it was not intended for.

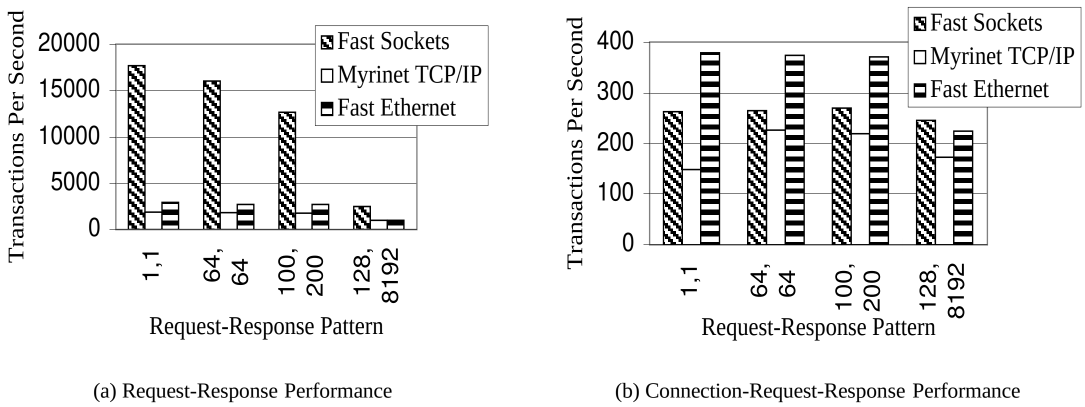

Figure 8: `Netperf` measurements of request-response transactions-per-second on the Solaris platform, for a variety of packet sizes. Connection costs significantly lower Fast Sockets' advantage relative to Myrinet TCP/IP, and render it slower than Fast Ethernet TCP/IP. This is because the current version of Fast Sockets uses the TCP/IP connection establishment mechanism.

We used the NcFTP ftp client (`ncftp`), version 2.3.0, and the Washington University ftp server (`wu-ftpd`), version 2.4.2. Because Fast Sockets currently does not support `fork()`, we modified `wu-ftpd` to wait for and accept incoming connections rather than be started from the `inetd` Internet server daemon.

Our FTP test involved the transfer of a number of ASCII files, of various sizes, and reporting the elapsed time and realized bandwidth as reported by the FTP client. On both machines, files were stored in an in-memory filesystem, to avoid the bandwidth limitations imposed by the disk.

The relative throughput for the Fast Sockets and TCP/IP versions of the FTP software is shown in Figure 9. Surprisingly, Fast Sockets and TCP/IP have roughly comparable performance for small files (1 byte to 4K bytes). This is due to the expense of connection setup — every FTP transfer involves the creation and destruction of a data connection. For mid-sized transfers, between 4 Kbytes and 2 Mbytes, Fast Sockets obtains considerably better bandwidth than normal TCP/IP. For extremely large transfers, both TCP/IP and Fast Sockets can realize a significant fraction of the network's bandwidth.

5 Related Work

Improving communications performance has long been a popular research topic. Previous work has focused on protocols, protocol and infrastructure implementations, and 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. In contrast, Fast Sockets provides the same models as TCP/IP and maintains application compatibility.

Other 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 as or better than special-purpose protocols for most applications. The measurements made in [Kay & Pasquale 1993] lend 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 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),

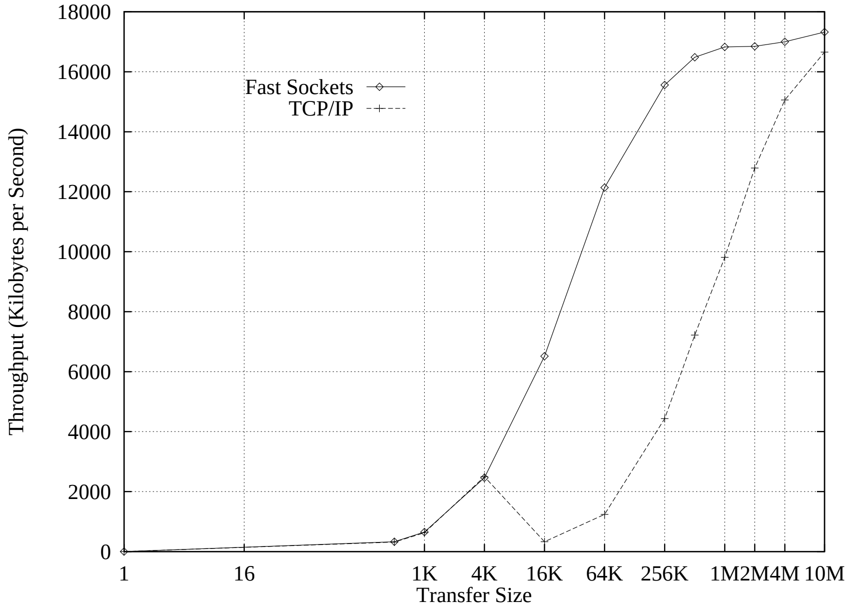

Figure 9: Relative throughput realized by Fast Sockets and TCP/IP versions of the FTP file transfer protocol. Connection setup costs dominate the transfer time for small files and the network transport serves as a limiting factor for large files. For mid-sized files, Fast Sockets is able to realize much higher bandwidth than TCP/IP.

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 were the concepts of *application device channels* (ADC), which provide 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 re-mapping, and copy-on-write pages to improve communications performance. To take full advantage of the zero-copy stack, user applications had to use page-aligned buffers and transfer sizes larger than a page. Because of these limitations, the designers focused on improving realized throughput, instead of small message latency, which Fast Sockets addresses. The resulting system achieved 32 MB/s throughput on a similar network but with a slower processor. This throughput was achieved for large transfers (16K bytes), not the small packets that make up the majority of network traffic. This work required a 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 use either in a user-level library (like Fast Sockets) or a separate user-level server. Mach 3.0 used the latter approach [Forin et al. 1991], which yielded 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 with worse performance than the in-kernel TCP stack.

[Kay & Pasquale 1993] 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] provided an interface similar to that of previous versions of Active Messages, but did not allow processes to share the network. Remote Queues [Brewer et al. 1995] provided low-overhead communications similar to that of Active Messages, but separated the arrival of messages from the invocation of handlers.

The SHRIMP project implemented a stream sockets layer that uses many of the same techniques as Fast Sockets [Damianakis et al. 1996]. SHRIMP supports communication via shared memory and the execution of handler functions on data arrival. The SHRIMP network had hardware latencies (4–9 µs one-way) much lower than the Fast Sockets Myrinet, but its maximum bandwidth (22 MB/s) was also lower than that of the Myrinet [Felten et al. 1996]. It used a custom-designed network interface for its memory-mapped communication model. The interface provided in-order, reliable delivery, which allowed for extremely low overheads (7 µs over the hardware latency); Fast Sockets incurs substantial overhead to ensure in-order delivery. Realized bandwidth of SHRIMP sockets was about half the raw capacity of the network because the SHRIMP communication model used sender-based memory management, forcing data transfers to indirect through the socket buffer. The SHRIMP communication model also deals poorly with non-word-aligned data, which required programming complexity to work around; Fast Sockets transparently handles this un-aligned data without extra data structures or other difficulties in the data path.

U-Net [von Eicken et al. 1995] is a user-level network interface developed at Cornell. It virtualized the network interface, allowing multiple processes to share the interface. U-Net emphasized improving the implementation of existing communications protocols whereas Fast Sockets uses a new protocol just for local-area use. A version of TCP (U-Net TCP) was implemented for U-Net; this protocol stack provided the full functionality of the standard TCP stack. U-Net TCP was modified for better performance; it succeeded in delivering the full bandwidth of the underlying network but still imposed more than 100 microseconds of packet processing overhead relative to the raw U-Net interface.

6 Conclusions

In this paper we have presented Fast Sockets, a communications interface which provides low-overhead, low-latency and high-bandwidth communication on local-area networks using the familiar Berkeley Sockets interface. We discussed how current implementations of the TCP/IP suite have a number of problems that contribute to poor latencies and mediocre bandwidth on modern high-speed networks, and how Fast Sockets was designed to directly address these shortcomings of TCP/IP implementations. We showed that this design delivers performance that is significantly better than TCP/IP for small transfers and at least equivalent to TCP/IP for large transfers, and that these benefits can carry over to real-life programs in everyday usage.

An important contributor to Fast Socket's performance is receive posting, which utilizes socket-layer information to influence the delivery actions of layers farther down the protocol stack. By moving destination information down into lower layers of the protocol stack, Fast Sockets bypasses copies that were previously unavoidable.

Receive posting is an effective and useful tool for avoiding copies, but its benefits vary greatly depending on the data transfer mechanism of the underlying transport layer. Sender-based memory management schemes impose high synchronization costs on messaging layers such as Sockets, which can affect realized throughput. A receiver-based system reduces the synchronization costs of receive posting and enables high throughput communication without significantly affecting round-trip latency.

In addition to receive posting, Fast Sockets also collapses multiple protocol layers together and reduces the complexity of network buffer management. The end result of combining these techniques is a system which provides high-performance, low-latency communication for existing applications.

Availability

Implementations of Fast Sockets are currently available for Generic Active Messages 1.0 and 1.5, and for Active Messages 2.0. The software and current information about the status of Fast Sockets can be found on the Web at `http://now.cs.berkeley.edu/Fastcomm/fastcomm.html`.

Acknowledgements

We would like to thank John Schimmel, our shepherd, Douglas Ghormley, Neal Cardwell, Kevin Skadron, Drew Roselli, and Eric Anderson for their advice and comments in improving the presentation of this paper. We would also like to thank Rich Martin, Remzi Arpaci, Amin Vahdat, Brent Chun, Alan Mainwaring, and the other members of the Berkeley NOW group for their suggestions, ideas, and help in putting this work together.

References

[Abbott & Peterson 1993] M. B. Abbott and L. L. Peterson. "Increasing network throughput by integrating protocol layers." *IEEE/ACM Transactions on Networking*, 1(5):600–610, October 1993.

[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.

[Blumrich et al. 1994] M. A. Blumrich, K. Li, R. D. Alpert, C. Dubnicki, E. W. Felten, and J. Sandberg. "Virtual memory mapped network interface for the SHRIMP multicomputer." In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pp. 142–153, Chicago, IL, April 1994.

[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.

[Buzzard et al. 1996] G. Buzzard, D. Jacobson, M. Mackey, S. Marovich, and J. Wilkes. "An implementation of the Hamlyn sender-managed interface architecture." In *Proceedings of the Second Symposium on Operating Systems Design and Implementation*, pp. 245–259, Seattle, WA, October 1996.

[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 distributed system." In *Proceedings of SIGMETRICS '87*, pp. 216–225, Banff, Alberta, Canada, May 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." Request For Comments 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.

[Damianakis et al. 1996] S. N. Damianakis, C. Dubnicki, and E. W. Felten. "Stream sockets on SHRIMP." Technical Report TR-513-96, Princeton University, Princeton, NJ, October 1996.

[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 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.

[Felten et al. 1996] E. W. Felten, R. D. Alpert, A. Bilas, M. A. Blumrich, D. W. Clark, S. N. Damianakis, C. Dubnicki, L. Iftode, and K. Li. "Early experience with message-passing on the SHRIMP multicomputer." In *Proceedings of the 23rd Annual International Symposium on Computer Architecture (ISCA '96)*, pp. 296–307, Philadelphia, PA, May 1996.

[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.

[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.

[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.

[Liu & Culler 1995] L. T. Liu and D. E. Culler. "Evaluation of the Intel Paragon on Active Message communication." In *Proceedings of the 1995 Intel Supercomputer Users Group Conference*, June 1995.

[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.

[Nagle 1984] J. Nagle. "Congestion control in IP/TCP internetworks." Request For Comments 896, Network Working Group, January 1984.

[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. "Internet protocol." Request For Comments 791, IETF Network Working Group, September 1981.

[Postel 1981b] J. Postel. "Transmission control protocol." Request For CommentsC 793, IETF Network Working Group, September 1981.

[Postel 1981c] J. Postel. "User datagram protocol." Request For Comments 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.