Efficient Layering for High Speed Communication: Fast Messages 2.x[*]

Mario Lauria, Scott Pakin, and Andrew A. Chien[†]
Department of Computer Science
University of Illinois at Urbana-Champaign
1304 W. Springfield Ave.
Urbana, IL 61801, USA
{lauria, pakin, achien}@cs.uiuc.edu

Abstract

We describe our experience designing, implementing, and evaluating two generations of our high performance communication library, Fast Messages (FM) for Myrinet. In FM 1.x, we designed a simple interface and provided guarantees of reliable and in-order delivery, and flow control. While this was a significant improvement over previous systems, it was not enough. Layering MPI atop FM 1.x showed that only about 20% of the FM 1.x bandwidth could be delivered to higher level communication APIs. Our second generation communication layer, FM 2.0, addresses the identified problems, providing gather-scatter, interlayer scheduling, receiver flow control, as well as some convenient API features which simplify programming. FM 2.x can deliver 70-90% to higher level APIs such as MPI. This is especially impressive as the absolute bandwidths delivered have increased nearly fourfold to 70 MB/s. We describe general issues encountered in matching two communication layers, and our solutions as embodied in FM 2.x.

[*]The research is supported in part by DARPA Order #E313 through the US Air Force Rome Laboratory Contract F30602-96-1-0286 and Contract F30602-97-2-0121, NSF grant MIP-92-23732, and NASA grant NAG 1-613. Support from Intel Corporation, Compaq/Tandem Computers, Hewlett-Packard and Microsoft is also gratefully acknowledged. Andrew Chien is supported in part by NSF Young Investigator Award CCR-94-57809. Mario Lauria is supported by a NATO-CNR Advanced Science Fellowship.

[†]In September 1998, Andrew Chien will be the Science Applications International Corporation Chair Professor at the University of California, San Diego. He can be contacted at achien@cs.ucsd.edu and 9500 Gilman Road, Dept 0114, La Jolla, CA 92093-0114

1. Introduction

Dramatic advances in low-cost computing technology have combined to make clusters of PCs an attractive alternative to massively parallel processor (MPPs) architectures. Leveraging on mass-market volumes of production, the humble PC has benefited from huge and ever increasing investments in the development of its key components (CPU, memory, disks, I/O buses, peripherals), while at the same time MPP manufacturers are coming to terms with a contraction of the market for multi-million dollar machines.

However a supercomputer is more than a collection of high performance computing nodes; it is in the way its component parts are integrated that lies the real challenge of a parallel machine design. In comparing a cluster architecture with the custom design of a contemporary MPP, it is in the interconnection technology that the latter has the largest edge over the former. For example, the Cray T3D achieves communication latencies of about 2 μs and peak bandwidth of about 300 MB/s, the IBM SP2 of about 35 μs and 100 MB/s respectively, whereas the typical values for a classical Ethernet-interconnected cluster are 1 ms and 1.2 MB/s respectively.

The new high speed Local Area Networks (LANs) available today (ATM [4], FDDI [10], Fibrechannel [1], Myrinet [2]) offer comparable hardware latency and bandwidth to the proprietary interconnect found on MPPs. The introduction of these enabling technologies shifts the focus of the MPP versus cluster comparisons from performance more general considerations of system scalability, reliability, affordability, and software availability.

New hardware technologies are only part of the communication picture, and delivering performance to

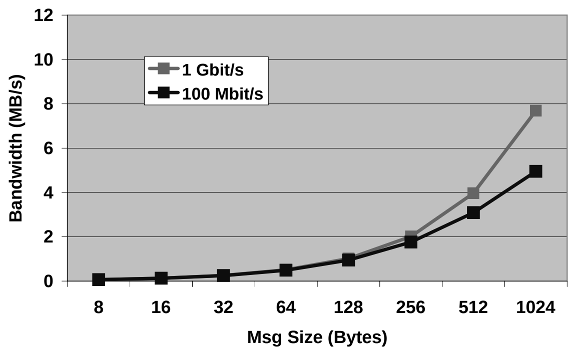


Figure 1. (a) 100 Mbit and (b) 1 Gbit Ethernet theoretical bandwidth assuming a fixed 125μs protocol processing overhead

applications requires communication software capable of delivering the network performance. Fast network hardware alone is not sufficient to speed up communication [18]. Existing communication protocols have been developed to address requirements of robustness in the presence of unreliable transport and large network latencies, and with operating system controlled access to network interfaces. As a consequence, they are characterized by a large processing overhead, which prevents them from fully exploiting the performance of the new networks (Figure 1).

Over the past few years, many research projects have studied the design of high performance communication software (Fast Messages (FM) [19], Active Messages (AM) [29], U-Net [28], VMMC-2 [9], PM [27], BIP [24]). In the Fast Messages project, we built two generations of systems optimized to deliver communication performance to the application. The first generation, FM 1.0, was based on our studies of essential communication guarantees (reliable, in-order communication with flow control) and tuned for realistic message-size distributions (mostly short messages). FM 1.0 achieved dramatically more usable communication performance, reducing the half-power message size for the Myrinet network by nearly two orders of magnitude, from over four thousand bytes to 54 bytes. We present the results of our initial experience with the implementation of user-level libraries on top of FM 1.x, which expose the critical issues and the important services required in matching two adjacent layers of the communication hierarchy.

For the second generation Fast Messages system, we used the insights gained from FM 1.x to optimize the FM API and maximize the portion of FM performance delivered to the applications. By building high-level libraries such as MPI on top of FM and analyzing the resulting performance of the entire software stack, we found a number of inefficiencies were created at the interface between libraries. The performance losses caused by the interface are remarkable, limiting network performance to a small fraction (<10%) of the hardware.

Fast Messages 2.x eliminates these interface problems, enabling over 90% of FM's performance to be delivered to higher level API's such as MPI. We describe the new elements of the FM 2.x API: gather/scatter, interlayer scheduling, receiver data pacing and their impact on usable performance. The interface efficiency obtained the FM 2.x interface is over 70%, even for sixteen byte messages and increases rapidly to 90%, a dramatic improvement. The implementation of MPI-FM atop the FM 2.x API achieves 70 MB/s peak bandwidth versus the 77 MB/s available on FM. The performance increase is even more impressive considering the nearly fourfold increase of absolute performance of FM 2.x with respect to FM 1.x as a result of the migration from a Sparc to an x86 architecture.

The remainder of the paper is organized as follows. In Section 2 we review the results that motivate the design of FM. In Section 3 we present the FM 1.x API and discuss its strengths and weaknesses. In Section 4 we present the FM 2.x API and describe its features. Related work is surveyed and contrasted to our work in Section 5. Finally, we make a few concluding remarks in Section 6.

2. Motivation for Fast Messages Design

The design of Fast Messages is motivated by the wealth of knowledge about message size distributions, the characteristics of traditional network protocols, and studies of high performance networks in parallel computers. The core of these results are summarized below.

2.1. Network Traffic Characteristics

Since the first use of computer networks, scientists have studied the size, frequency, and distributions of both for network traffic. Such studies consistently show that the majority of traffic (by packet count) consists of short messages. This property is remarkably stable across networks, time, and applications [11, 16]. In a study of traffic on a Ethernet connecting diskless workstations to file servers [11], Gusella found that the majority of packets were less than 576 bytes; of these 60% were 50 bytes or less. In another study [16], Kay et al. measured the TCP and UDP traffic on a FDDI LAN of Unix workstations in a university computer science department. They found that TCP message sizes are small: over 99% of packets are less than 200 bytes. UDP traffic was slightly larger, with 86% of messages of less than 200 bytes. NFS-generated UDP packets accounted for 90% of the traffic measured. Continuous studies at the SUNY-Buffalo campus also chronicle the predominance of short messages. For a wide variety of networks, across a wide range of time, average packet sizes of 300 to 400 bytes were recorded.

The prevalence of short messages implies that if good network performance is to be accessible, it must be delivered to short messages. Many gigabit network projects were successful in achieving Gbit/s speeds, but required megabyte-sized messages to deliver such bandwidth. In short, overhead must be minimized, as at high network speeds, there is little spooling time available to mask network overhead.

2.2. Legacy Protocols

Widely used Internet protocols such as TCP [23] and UDP [22] provide widespread interoperability and two levels of functionality – reliable byte streams and unreliable datagrams. However, these protocols incur significant overheads [7], essentially preventing the delivery of network performance to short messages. For example, the *fastest* implementations of UDP achieve per packet overheads of $\approx 125\mu$seconds. This implies that for typical packet size distributions (< 256 bytes), bandwidths of no greater than 2 megabytes/second could be sustained. Of course, the overhead for reliable protocols such as TCP are even greater.

2.3. High Performance Communication Layers

To identify crucial performance factors for high performance networks, we undertook empirical studies of communication layers inside parallel computers. These studies identified the key guarantees a communication layer must provide to avoid incurring large software overhead at higher levels of the system. Our study of CM-5 Active Messages (CMAM) [12] measured the dynamic instruction count of CMAM assembly code and identified the overhead contributions of the range of guarantees provided by the communication layer (in-order delivery, buffer management, fault tolerance). Because the network of the CM-5 provided none of these features, the software overhead can be considered the "cost" of each feature on the CM-5. In a

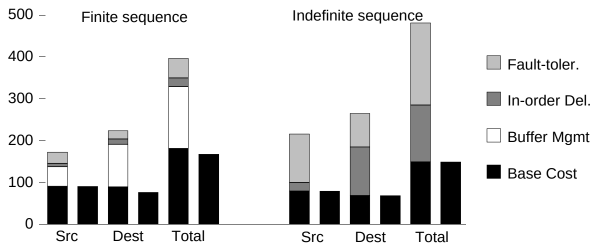


Figure 2. Breakdown of overhead for Active Messages on the CM-5

highly optimized messaging layer like the CMAM up to 50%-70% of the software messaging costs are a direct consequence of the gap between user requirements such as in-order and reliable delivery, end-to-end flow control, and actual network features like arbitrary delivery order, finite buffering, unreliable communication. For example, in one case (16-word messages, 4-word packet size, multi-packet delivery) 216 out of a total 397 cycles are spent for buffer management (148 cycles), in-order delivery (21 cycles) and fault tolerance (47 cycles) (see Figure 2).

These results imply that careful design of an interface, particularly the guarantees provided, is crucial for the low overhead essential to achieving high performance. In gigabit networks, where delivering network performance to short messages is essential to delivering usable performance, careful interface design to provide first the right guarantees and second the right functional interfaces is critical. These lessons were crucial in the design of two generations of Fast Messages systems.

3. Fast Messages 1.x

In the design of Fast Messages 1.0 for Myrinet, we applied the lessons of the networking community – designing a system with low overhead to deliver performance to short messages, and a simple interface with the right guarantees to deliver performance to the application. By providing a few key services – buffer management, reliable and in-order delivery – the FM programming interface allowed for a leaner, more efficient implementation of the higher level communication layers.

The first workstation cluster implementation of Fast Messages (FM) [20] was built atop the Myrinet network, largely due to its high performance and availability of development tools. On this network FM achieves a short message latency of only 14 μs and a peak bandwidth of 17.6 MB/s, with an Active Messages style interface. As a result of the design focus on short message performance, the value of $N_{\frac{1}{2}}$ is 54 bytes, with a bandwidth of 17.5 MB/s available for messages as small as 128 bytes (see Figure 3).

3.1. Design of Illinois Fast Messages 1.x

The FM 1.1 API consists of three functions `FM_send(.)`, `FM_send_4(.)`, and `FM_extract(.)` as shown in Table 1. `FM_send(.)` and `FM_send4(.)` inject messages in the network. FM differs from a pure message passing paradigm by not having explicit receives. Instead, each message includes the name of a handler, which is a user-defined function that is invoked upon message arrival, and that will process as required the carried data.

The `FM_extract()` primitive is used to service communication on the receive side, checking for incoming messages and executing the corresponding handlers. The user needs to call this primitive frequently to ensure the prompt processing of incoming communication in the host. However it needs not to be called for the network to make progress. FM provides buffering so that senders can make progress while their corresponding receivers are computing and not servicing the network.

The FM interface is similar to the Active Messages model [29] from which it borrows the notion of message handlers. However there are a number of key differences: the FM API offers stronger guarantees (in particular in-order delivery), uniform handling of messages with respect to size, and it does not follow a rigid request-reply scheme. Also, in contrast to Active Messages, where the send calls implicitly poll the network, FM's send calls do not normally process incoming messages, enabling a program to control when received data is processed.

In choosing which service guarantees to include during the design phase of FM, we gave careful consideration to the performance of the communications stack as a whole, not of FM as an isolated messaging layer. If a messaging layer's guarantees are too weak (i.e. they do not provide the functionality that applications expect), other messaging layers built on top will need to supply the missing functionality, incurring additional overhead in the process. On the other hand, if a messaging layer's guarantees are too strong (i.e. they provide more functionality than is generally needed), the messaging layer's common-case performance may be needlessly degraded. Analysis of the literature and our ongoing studies to support fine-grained parallel computing [5, 12, 13, 14] have led to the conclusion that a low-level messaging layer should provide the following key guarantees:

- Reliable delivery,
- In-order delivery, and
- Control over scheduling of communication work (decoupling).

As mentioned in the previous section, studies of communication software costs [12] show that implementing guarantees like reliable and in-order delivery atop a messaging library can increase communication overhead by over 200%. To reduce these costs careful consideration was given to exploiting hardware features. We found that by taking advantage of Myrinet features such as very low bit error rate, absence of buffering in the network fabric, deterministic routing, link-level flow control by means of back-pressure, we only needed to add flow control and buffer management to provide reliable and in-order delivery. FM provides these, and its performance demonstrate that these guarantees need not to be costly.

Figure 3(a) shows that the addition of buffer management and flow control does not substantially degrade performance. The different curves represent the performance measured with the simplest code needed to operate the link DMAs, then with a few more lines to move data across the I/O bus, and finally with the flow management code added. The transport of data across the I/O bus is on the critical path and adds to the overhead, while flow control if properly designed can be overlapped with other operations. Similarly, the further addition of buffer management does not add substantial overhead, and leads to the final version of the FM code (figure 3(b)). A more detailed analysis of the FM 1.x design choices is reported in [20].

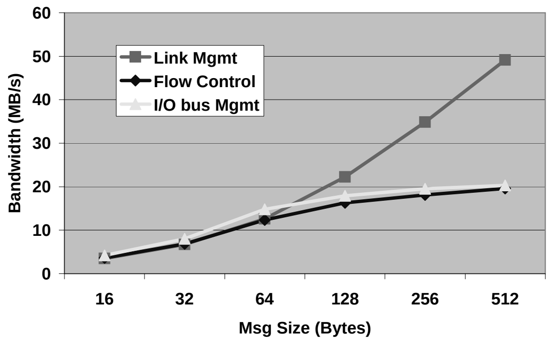
3.2. Evaluation of FM 1.x

The real measure of the effectiveness of a communication library is the level of performance that can be actually delivered to an application. Given the low-level nature of the FM interface, typical applications are language runtime supports or user level libraries. We selected MPI and BSD sockets as test applications, and experimented extensively with the former.

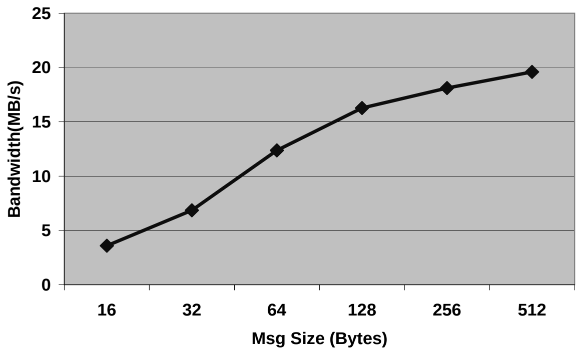
Figure 4 shows that the initial version of MPI-FM had poor performance, failing to deliver more than 35% of the underlying FM bandwidth. It was clear that the

Function	Operation
FM_send_4(dest,handler,i0,i1,i2,i3)	Send a four word message
FM_send(dest,handler,buff,size)	Send a long message
FM_extract()	Process received messages

Table 1. The primitives of the FM 1.1 API



(a)



(b)

Figure 3. FM 1.x overhead: (a) overhead break-down; (b) overall performance

FM 1.x interface lacked several key features required for efficient layer composition. So the analysis of the MPI-FM inefficiencies turned into a study on how to design an API that makes it easy to deliver performance (see [17] for details).

The overhead originates from a number of memory-to-memory copies of the data taking place at the interface between MPI-FM and FM. The service guarantees we built in FM allowed a streamlined and thin implementation of the body of MPI-FM, for example making unnecessary the source buffering, timeout, and retry that would be otherwise required to provide reliable communication. But inefficiencies arose at the interface between layers, surprisingly for different reasons for each direction of transfer.

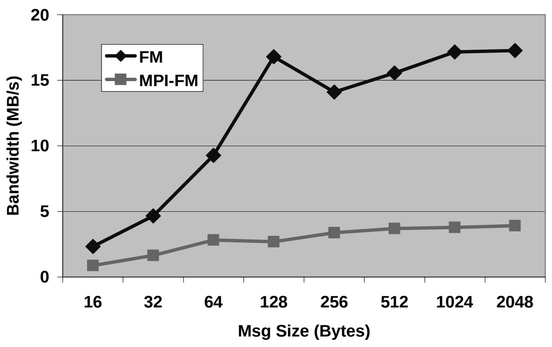
First, FM adopted its basic API from Active Message (AM) [29], and thus accepted (and presented) data as a single contiguous buffer. While sending, this approach charges the upper layers with the task of assembling/disassembling of messages. In many cases, this incurred an additional step (and copy) in performing common protocol processing operations such as packet header attachment, message encapsulation, checksumming.

Similarly to the send side, on the receive side the message is handed over to the handler as a single contiguous buffer. This required that the entire message had to be received into a staging buffer before the handler could start processing it and possibly copying it to the final destination. Such a scheme forced FM to perform an additional copy even when the availability of the destination buffer (from a pre-posted MPI receive) made it unnecessary.

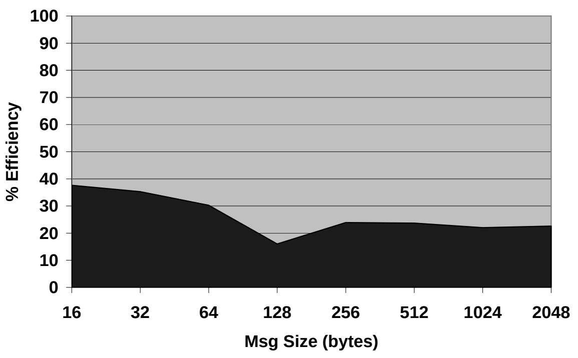
Second, FM 1.x allowed the receiving process to decide when to service the network, however, it was unable to control the quantity of data presented at that time (all the pending packets were processed). In high speed networks, data can easily be transmitted faster than a receiver can accept it. The presentation of the data before the application was prepared to accept induced additional layers of buffering and data copies.

In conclusion, the implementation of MPI-FM showed that the FM API was lacking flexibility in two crucial areas:

- presentation of data across layer boundaries
- control over interlayer scheduling



(a)



(b)

Figure 4. MPI-FM initial performance compared to FM: (a) absolute; (b) as a percentage of FM

Addressing these shortcomings required some fundamental changes to the API, and motivated the design of a new version of FM.

4. Fast Messages 2.x

4.1. Design of Illinois Fast Messages 2.x

The FM 2.x API retains the service guarantees of FM 1.x, and adds support for gather-scatter, layer interleaving, and receiver flow control. The primary vehicle for these features is the addition of the *stream abstraction*, in which messages are viewed as byte streams and primitives are provided for the piecewise manipulation of data, both on the send and the receive side.

Table 2 shows the the new FM 2.x interface. The old `FM_send(.)` primitive is replaced by `FM_send_piece(.)`, which can be called as many times as desired to send chunks of a same message of arbitrary size. Message boundaries are still honored (using the `FM_begin_message(.)` and `FM_end_message(.)` calls), but in the new API a message is a byte stream instead of a single contiguous region of memory.

Mirroring this abstraction on the receive side is the `FM_receive()` primitive, that can be called an arbitrary number of times from within a handler. Again, the notion of message is retained but it is no longer associated to the concept of a contiguous region of memory. The addition of an argument to the `FM_extract(.)` primitive allows the user to specify an upper limit on the amount of data extracted (rounded to the next packet boundary) enabling receiver flow control.

Thus, the key problems identified in studies of FM 1.x are remedied as follows:

Gather/Scatter By performing a sequence of `FM_send_piece(.)` calls, the user can compose a message on the fly using any number of pieces, each of arbitrary size. Similarly, a receiver can employ a handler with a sequence of `FM_receive()` calls, allowing the efficient decomposition of a message into any number of pieces. Each call composes/extracts as many bytes as desired, and the number and sizes of the pieces need not match on the two sides. Examples include header attachment/removal in MPI-FM, and in protocol encapsulation in general (e.g. IP and TCP headers in TCP/IP hierarchy).

Layer Interleaving A second important benefit of the stream abstraction is the controlled interleaving of FM's and the application's threads of execution on the receive side. While everything runs within one user process, conceptually there is one thread of execution for the FM primitives, and one for each of the application-specific handlers. The typical message processing scenario within the handler is illustrated below:

```
int myHandler(FM_stream *str, unsigned sender)
{
  struct header myHeader;
  int msglen;

  /* get the header */
  FM_receive(&myHeader, str,
```

Function	Operation
FM_begin_message(dest, size, handler)	Start of a message to be sent
FM_send_piece(stream, buf, bytes)	Send a chunk of message
FM_end_message(stream)	End of a message to be sent
FM_receive(stream, buf, bytes)	Get a chunk of message
FM_extract(bytes)	Process received messages

Table 2. The primitives of the FM 2.x API

```
               sizeof(struct header));

  msglen = myHeader.length;

  if (myHeader.littlemsg)
    /* short message */
    FM_receive(littlebuf++, str, msglen);
  else
    /* long message */
    FM_receive(findBuf(msglen), str, msglen);

  return FM_CONTINUE;
}
```

The first `FM_receive()` call is used to extract just the message header (`FM_receive()` is executed within the FM thread). Then the handler reads the header fields, identifies the message, and selects the buffer into which to copy the message payload (the handler is executed within its own thread). Finally, another `FM_receive()` call with the selected buffer passed as second argument extracts the payload directly into the buffer (FM thread).

The interleaving makes possible the elimination of staging buffers for incoming messages. For example, in MPI-FM, using FM 1.x, we could not deliver an incoming message directly into its destination buffer specified by the user through a pre-posted MPI receive call. The problem originated from the fact that incoming messages are handled by FM, while the buffer management occurs within MPI-FM, and the required exchange of information between the two layers (identity of message in one direction, pointer to the appropriate buffer in the other) was missing.

Receiver Flow Control The FM 2.x interface also provides receiver flow control, allowing the receiver to control the rate at which data is processed from the network. This feature is only possible because of the underlying flow control and reliable delivery provided by FM. The receiver flow control can eliminate network overruns of application buffer pools, avoiding memory copies, and for some protocols, message discarding. In many applications, the ability to intentionally delay the extraction of the message until a buffer becomes available can simplify the buffer management. For example, receiver flow control enables zero-copy transfers in a significantly larger number of cases for both our Socket-FM and MPI-FM implementations.

Transparent Handler Multithreading One of the differences between FM 1.x and FM 2.x is that handler execution is no longer delayed until the entire message has arrived, rather it is started has soon as the first packet is received. Since packets belonging to different messages can be received interleaved, the execution of several handlers can be pending at a given time. As it extracts each packet from the network, FM 2.x schedules the execution of the associated pending handler. By having the interleaved packet reception transparently drive the handler execution, a number of benefits are achieved.

First, the handler multithreading combined with the stream abstraction allows arbitrary-sized data chunks to be composed/received, without any concern for packet boundaries. Second, handler multithreading plus packetization not only simplifies resource management, it can also increase performance by increasing effective pipelining. On a long message the handler can be processing one part of the message while the sender is still sending the rest. And the interleaving means that one long message from one sender does not block other senders.

The FM 2.x interface cleanly hides the physical packetization and handler multithreading by offering a clean sequential view of message reception. Except for the possibility of being descheduled on a `FM_receive()` call, a handler can be written as if the entire message had already being received. Second, the FM 2.x interface provides a logical thread for each message, avoiding explicit management of state sharing/isolation for complex messages.

Despite the additional implementation complexity,

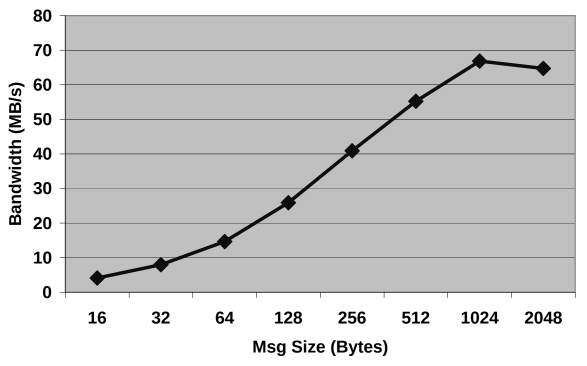


Figure 5. FM 2.1 performance on a 200 MHz PPro

the multi-threading adds only a small amount of overhead in exchange for a number of crucial services like the streaming abstraction and related benefits, sender decoupling, totally transparent packetization, in addition to the simple and clean sequential view of communication.

4.2. Evaluation of FM 2.x

Figure 5 shows the performance achieved by FM 2.1 on a 200 MHz PPro. The peak performance values are 11 μs minimum latency, 77 MB/s peak bandwidth, with $N_{\frac{1}{2}} < 256$ bytes. These values represent high absolute performance, comparing to MPP interconnect performance and internal memory bandwidth. Similar to FM 1.x, a design attentive to short message performance shows in the $N_{\frac{1}{2}}$ values and in the rapid growth of the bandwidth curve.

The graphs of Figure 6 show the improved efficiency of MPI-FM on top of FM 2.x, proving that the FM 2.x API can deliver a high percentage of its measured performance. MPI-FM achieves up to 90% of the FM bandwidth, with a minimum latency of 17 μs and a peak bandwidth of 70 MB/s. The key enhancements of FM 2.x (gather-scatter, layer interleaving, and receiver flow control) enable the MPI on FM 2.x to eliminate many buffer copies, and avoid buffer pool overruns, delivering the underlying FM (and hardware network) performance to the application. To further demonstrate FM 2.x's capabilities, we have implemented other APIs, including Shmem Put/Get and Global Arrays (both global address space interfaces). An implementation of Winsock 2 is in progress.

5. Related Work

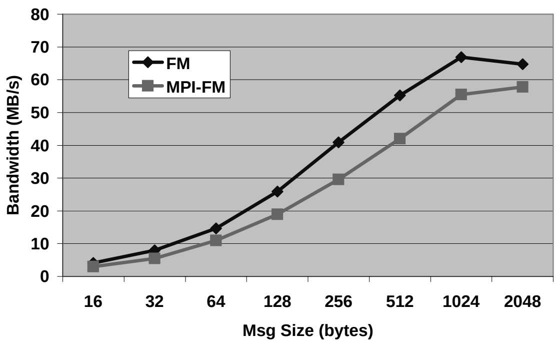
Fast Messages is not the only approach to delivering high-performance communication by efficient protocol layering. Most related efforts involve either optimized implementations of heavyweight protocols, high-performance network hardware, or other high-performance low-level messaging layers. We now discuss projects in each of these categories.

High Performance Communication Layers Active Messages (AM) [29] has been one of the first realizations of high performance messaging layers. The AM project started as a communication library for the CM-5, and today some of its new implementations retain some of the features of the original version, like the specialized primitives for short transfers. A problem with specialized primitives is that they often fall short of the practical message size of overlying applications. For example, in the implementation of MPI-FM we found that the minimum length of the header added by the MPI code is 24 bytes (6 words), while short message transfers in Active Messages style libraries are optimized for 4 or 5 words.

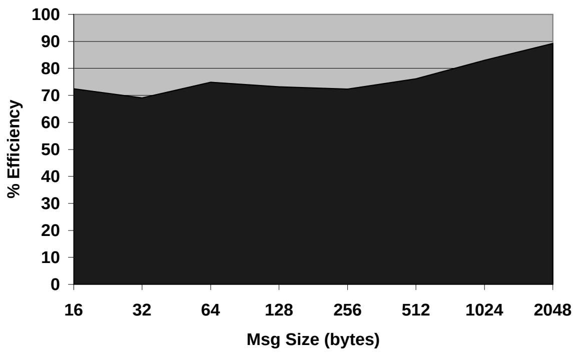
A work from the same group on the efficient realization of a high level API on top of a low level messaging layer is Fast Sockets [25], an implementation of the Berkeley Sockets on top of Active Messages. One of the issues explored by Rodrigues at al. in their work is the elimination of unnecessary copies at the layer interface. The copy avoidance technique of *receive posting* in Fast Sockets is similar to what FM 2.x achieves with the layer interleaving, in which the user handler collaborates with FM to direct the incoming data directly into the destination buffer. The main difference is that the FM model supports packetization and thus works with messages of arbitrary size.

Another high performance messaging layer is U-Net [28]. Developed originally on a ATM network, it provides buffer management, demultiplexing in hardware but no flow control, and thus data can be lost due to overflow. Contrary to FM, U-Net and other messaging layers try to avoid the passage of data through kernel memory by performing a DMA transfer directly into the user buffer. The disadvantage of such feature is that the user must declare in advance the regions of memory to be used for communication, so to allow the library to permanently pin them down.

In our experience such a scheme seems to lack the flexibility needed in building user-level libraries. In the case of MPI-FM, the buffers are provided by the MPI application and their location is not in general known in advance. A new version of U-Net called U-



(a)



(b)

Figure 6. MPI-FM 2.0 performance compared to FM 2.0: (a) absolute; (b) as a percentage of FM

Net/MM [30] is under development which addresses this limitation by including a TLB on the network interface and coordinating its operation with the operating system's virtual memory subsystem. This mechanism would allow network buffer pages to be pinned and unpinned dynamically and thus messages can be transferred to and from any part of the application's address space.

Princeton's VMMC-2 [9] interface is different from FM's in that it expects receivers to prepost buffers. This is a consequence of VMMC-2's lack of flow control. Messages arriving before a buffer is posted are stored in a staging area and copied out when the buffer is posted. If the staging area overflows, the message is dropped, and the sender retransmits it. FM, in contrast, uses flow control to ensure that no message is sent unless it can be reliably delivered, which avoids wasting network bandwidth.

In some respects similar to FM is the Real World Computing Partnership's PM [27]. Like FM, PM runs on clusters of Myrinet-connected workstations and performs flow control and buffer management. The main difference with FM is in the optimistic flow control mechanism, and variable-sized packets.

BIP [24] is another messaging layer developed for the Myrinet at the Ecole Normale Superieure de Lion. It has a more traditional message passing interface, with both blocking and non blocking send/receive primitives, and offers reliable and in-order delivery communication. It has been specifically designed to support standard message passing libraries like MPI and PVM, for which its interface represents a good match.

Optimized heavyweight protocols One approach to fast communication that a number of researchers have taken is to start with traditional, heavyweight, kernel-mode protocol stacks and tune the implementations to deliver more performance. Frequently, these projects focus on the TCP and UDP stacks, but other protocols have been optimized, as well. One of the largest performance penalties that occurs when sending large messages is memory copying, which occurs at each level in the protocol stack.[1] Hence, the most common optimization technique is to reduce the amount of data copying by sharing buffers across layers.

This is the approach taken by *fbufs* [8], which avoids data-touching overheads by remapping pages of data from one domain to another instead of copying. The Solaris operating system does something similar, but uses copy-on-write semantics to prevent wayward applications from corrupting data that are still "live" in the protocol stack [6]. *Container shipping* [21] and other protocol-stack optimizations [3] expand upon the basic fbufs technique. XTP [26] takes a different approach: It improves performance by providing high-level features such as multicast and priority control in a new, alternative heavyweight protocol.

The problem with all of these schemes, and one of the reasons that Fast Messages does not attempt a similar solution to the protocol layering performance problem, is that they perform poorly on small messages. And, for realistic message sizes—generally less than 256 bytes—memory copying is much less of a bot-

[1]In TCP, the other big penalty is computing the TCP checksum, but this cost can be eliminated in some modern network interfaces by performing the checksum in hardware.

tleneck than the various constant-time overheads [15]. Even the overhead to switch between user mode and kernel mode is too high for forthcoming networks. For perspective, note that on a gigabit network, about 1 KB of data can arrive in the time it takes just to switch modes.

6. Summary

We have described our experience with the implementation of user-level libraries on top of the FM library. Our work exposes the need for a design of the programming interface that specifically targets the efficient matching of adjacent layers, and identifies the crucial services required for such matching.

Services like gather/scatter, interlayer scheduling, receiver data pacing are key to the elimination of unnecessary copying otherwise required to perform routine protocol processing operations like header addition/removal or payload delivery.

We have then described how we redesigned the API of our second generation communication layer, FM 2.0, to add these services in a flexible and performance-conscious way. The validity of our new design is shown by the peak bandwidth of an high level library like MPI-FM that went from an initial 20% to a final 90% of the bandwidth made available by the FM layer.

References

[1] T. M. Anderson and R. S. Cornelius. High-performance switching with Fibre Channel. In *Digest of Papers Compcon 1992*, pages 261–268. IEEE Computer Society Press, 1992. Los Alamitos, Calif.

[2] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W.-K. Su. Myrinet—a gigabit-per-second local-area network. *IEEE Micro*, 15(1):29–36, February 1995. Available from `http://www.myri.com/research/publications/Hot.ps`.

[3] J. C. Brustoloni and P. Steenkiste. Effects of buffering semantics on I/O performance. In *Proceedings of the Second USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 277–291, Seattle, Washington, October 1996. Available from `http://www.cs.cmu.edu/afs/cs/user/jcb/papers/osdi96.ps`.

[4] CCITT, SG XVIII, Report R34. *Draft Recommendation I.150: B-ISDN ATM functional characteristics*, June 1990.

[5] A. Chien, J. Dolby, B. Ganguly, V. Karamcheti, and X. Zhang. Supporting high level programming with high performance: The Illinois Concert system. In *Proceedings of the Second International Workshop on High-level Parallel Programming Models and Supportive Environments*, pages 15–24, April 1997.

[6] H. Chu. Zero-copy TCP in Solaris. In *Proceedings of the USENIX Annual Technical Conference*, pages 253–264,
San Diego, California, January 1996. Available from `http://playground.sun.com/~hkchu/zc-usenix.ps`.

[7] D. D. Clark, V. Jacobson, J. Romkey, and H. Salwen. An analysis of TCP processing overhead. *IEEE Communications Magazine*, 27(6):23–29, June 1989.

[8] P. Druschel and L. L. Peterson. Fbufs: A high-bandwidth cross-domain transfer facility. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles (SOSP)*, pages 189–202, Asheville, North Carolina, December 1993. ACM SIGOPS, ACM Press. Available from `ftp://ftp.cs.arizona.edu/xkernel/Papers/fbuf.ps`.

[9] C. Dubnicki, A. Bilas, Y. Chen, S. Damianakis, and K. Li. VMMC-2: efficient support for reliable, connection-oriented commnication. In *Proceedings of Hot Interconnects V*. IEEE, August 1997. Available from `http://www.cs.princeton.edu/shrimp/Papers/hotIC97VMMC2.ps`.

[10] Fiber-distributed data interface (FDDI)—Token ring media access control (MAC). American National Standard for Information Systems ANSI X3.139-1987, July 1987. American National Standards Institute.

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

[12] V. Karamcheti and A. Chien. Software overhead in messaging layers: Where does the time go? In *Proceedings of the Sixth Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VI)*, pages 51–60, San Jose, California, October 1994. Association for Computing Machinery. Available from `http://www-csag.cs.uiuc.edu/papers/asplos94.ps`.

[13] V. Karamcheti and A. A. Chien. A comparison of architectural support for messaging on the TMC CM-5 and the Cray T3D. In *Proceedings of the International Symposium on Computer Architecture*, pages 298–307, 1995. Available from `http://www-csag.cs.uiuc.edu/papers/cm5-t3d-messaging.ps`.

[14] V. Karamcheti, J. Plevyak, and A. A. Chien. Runtime mechanisms for efficient dynamic multithreading. *Journal of Parallel and Distributed Computing*, 37(1):21–40, 1996. Available from `http://www-csag.cs.uiuc.edu/papers/rtperf.ps`.

[15] J. Kay and J. Pasquale. The importance of non-data touching processing overheads in TCP/IP. In *Proceedings of the ACM Communications Architectures and Protocols Conference (SIGCOMM)*, pages 259–269, San Francisco, California, September 1993. Available from `http://www-csl.ucsd.edu/CSL/pubs/conf/sigcomm93.ps`.

[16] J. Kay and J. Pasquale. Profiling and reducing processing overheads in TCP/IP. In

IEEE/ACM Transactions on Networking, December 1996. Available from `http://www-cse.ucsd.edu/users/pasquale/Papers/profTCP96.ps`.

[17] M. Lauria and A. Chien. MPI-FM: High performance MPI on workstation clusters. *Journal of Parallel and Distributed Computing*, 40(1):4–18, January 1997. Available from `http://www-csag.cs.uiuc.edu/papers/jpdc97-normal.ps`.

[18] M. Liu, J. Hsieh, D. Hu, J. Thomas, and J. MacDonald. Distributed network computing over Local ATM Networks. In *Supercomputing '94*, 1995.

[19] S. Pakin, V. Karamcheti, and A. A. Chien. Fast Messages: Efficient, portable communication for workstation clusters and MPPs. *IEEE Concurrency*, 5(2):60–73, April-June 1997. Available from `http://www-csag.cs.uiuc.edu/papers/fm-pdt.ps`.

[20] S. Pakin, M. Lauria, and A. Chien. High performance messaging on workstations: Illinois Fast Messages (FM) for Myrinet. In *Supercomputing '95*, December 1995. Available from `http://www-csag.cs.uiuc.edu/papers/myrinet-fm-sc95.ps`.

[21] J. Pasquale, E. W. Anderson, and K. Muller. Container Shipping: Operating system support for I/O-intensive applications. *IEEE Computer*, 27(3):84–93, March 1994.

[22] J. Postel. User datagram protocol. RFC 768, Internet Engineering Task Force, August 1980. Available from `ftp://ds.internic.net/rfc/rfc768.txt`.

[23] J. Postel. Transmission control protocol. RFC 793, Internet Engineering Task Force, September 1981. Available from `ftp://ds.internic.net/rfc/rfc793.txt`.

[24] L. Prylli and B. Tourancheau. Protocol design for high performance networking: a Myrinet experience. Technical Report N. 97-22, LIP, Ecole Normale Superieure de Lion, July 1997. Available from `http://www-bip.univ-lyon1.fr/`.

[25] S. Rodrigues, T. Anderson, and D. Culler. High-performance local-area communication using Fast Socket. In *Proceedings of the USENIX 1997 Technical Conference*, San Diego, California, January 1997. USENIX Association. Available from `http://now.cs.berkeley.edu/Papers2/`.

[26] W. T. Strayer, B. J. Dempsey, and A. C. Weaver. *XTP: The XPress Tranfer Protocol*. Addison-Wesley, 1992. ISBN 0-201-56351-7.

[27] H. Tezuka, A. Hori, and Y. Ishikawa. PM: A high-performance communication library for multi-user parallel environments. Technical Report TR-96-015, Tsukuba Research Center, Real World Computing Partnership, November 1996. Available from `http://www.rwcp.or.jp/papers/1996/mpsoft/tr96015.ps.gz`.

[28] 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 15th ACM Symposium on Operating Systems Principles*, pages 40–53, December 1995. Available from `http://www2.cs.cornell.edu/U-Net/papers/sosp.pdf`.

[29] T. von Eicken, D. Culler, S. Goldstein, and K. Schauser. Active Messages: a mechanism for integrated communication and computation. In *Proceedings of the International Symposium on Computer Architecture*, pages 256–266, 1992.

[30] M. Welsh, A. Basu, and T. von Eicken. Incorporating memory management into user-level network interfaces.
In *Hot Interconnects V*, Stanford, California, August 1997. Available from `http://www.cs.cornell.edu/U-Net/papers/hoti97.ps`.