Using the Mach Communication Primitives in X11

Michael Ginsberg and Robert V. Baron and Brian N. Bershad

mikegins@microsoft.com, rvb@cs.cmu.edu, bershad@cs.cmu.edu

School of Computer Science
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213

Abstract

We have modified the X11 windowing system to use the native communication facilities of the Mach 3.0 microkernel. Our new implementation can rely on Mach's low-overhead IPC facility as a direct replacement for sockets, or it can use shared memory as a transport between X11 clients and the server. On conventional BSD Unix systems, X11 communication is done through sockets. Because a user-level process implements Unix functionality on top of Mach 3.0, a socket-based version of X11 performs substantially worse than when running on a monolithic Unix kernel. Using Mach IPC as the transport between X11 clients and the server, X11 performance is slightly better than that of a monolithic system in which sockets are implemented inside the kernel as opposed to within a user level process. Using Mach's shared memory facilities as the transport, we have measured performance improvements of over 40%.

1 Introduction

Mach is a microkernel-based operating system that provides complete 4.3 BSD Unix emulation through a user-level Unix server [Golub et al. 90]. This approach allows existing Unix applications to run unmodified on top of the Mach microkernel. In many cases, the

This research was sponsored in part by The Defense Advanced Research Projects Agency, Information Science and Technology Office, under the title "Research on Parallel Computing", ARPA Order No. 7330, issued by DARPA/CMO under Contract MDA972-90-C-0035, and by the Open Software Foundation (OSF). Bershad was partially supported by a National Science Foundation Presidential Young Investigator Award.

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of DARPA, OSF, the NSF, or the U.S. government.

system has comparable performance. In some cases, though, Unix applications run more slowly on top of an emulated Unix than they do on top of an "in-kernel" version of Unix. Applications that currently suffer most are those that use the Unix socket interface. For Mach 3.0, Unix sockets can incur a great deal of overhead since a socket call must perform an RPC through the kernel to the Unix server. Since X11 depends on the socket interface, X11 applications can run noticeably more slowly on top of Mach 3.0 than on a conventional Unix system.

We have restructured the protocol-dependent layers of X11 [Gettys et al. 90] to rely directly on the communication mechanisms provided by Mach 3.0, rather than those provided by the Unix server. This approach allows us to improve window-system performance because it removes the Unix server from X11 client/server communication.

Our implementation has followed two different paths. In one case, we have implemented the X11 transport protocols using Mach IPC [Draves 90]. This yields performance slightly better, or comparable to that of an in-kernel implementation because it eliminates the context-switching overhead incurred by the out-of-kernel Unix system. In the second case, we have used shared memory for communication between X11 clients and servers reducing the system's reliance on kernel communication primitives [Bershad et al. 91]. This approach yields substantial performance improvements.

1.1 Motivation

Mach is a microkernel designed to provide a base operating system on which other operating systems such as Unix can be built. Two versions of Mach will be discussed in this paper. The first, Mach 2.5, includes the Mach microkernel and the Unix emulation code in the kernel's address space. This combination is comparable in speed to other Unix implementations such as Ultrix and BSD 4.3, since all the Unix code is in the monolithic kernel. Mach 2.5 is compatible with standard Unix, but also provides the added functionality of Mach. In contrast, Mach 3.0 contains just the Mach microkernel. A separate program, the Unix emulator, runs as a user-level process. Mach offers two kinds of communications channels, message passing and shared memory. Message passing is provided using the Mach IPC interface. Shared memory is provided using Mach's VM interface [Rashid et al. 87].

Under Mach 3.0, communication through a socket goes from the user code through the Mach microkernel, then to the Unix emulator. Therefore, more overhead is incurred for Unix socket system calls using Mach 3.0 than when using an in-kernel implementation of Unix. For example, on a 33Mhz 80486 system, a small (64 bytes) round-trip message using sockets takes 709 microseconds on Mach 2.5, and 2319 microseconds on Mach 3.0. Under Mach 3.0, using Mach IPC rather than socket code, the time drops to about 150 microseconds. With shared memory, the time to send a message can be as low as the time to format the message in a shared buffer.

Our goal was to retarget X11 to use the less expensive communication facilities provided by the Mach 3.0 operating system. We experimented with two different implementations, one using kernel-based IPC and one using shared memory.

X11 is a client-server windowing system that runs under many different operating systems and on many different hardware platforms. It provides a server that controls the machine hardware, including displays and input devices such as keyboards and mice. Programs access the server through a library of client functions. This library allows clients to request actions to be taken by the X server, as well as to request notification of events such as keyboard and mouse events. We were willing to make changes to the server, since there are relatively few servers, but were unwilling to do so to clients. We wanted to be able to take existing clients and use them under our modified transport, without having to edit or rewrite them. Therefore, our client-side changes are restricted to the X11 library, and require only that X11 clients be relinked.

1.2 The rest of this paper

The rest of this paper describes our experiences with restructuring the X11 protocols to use the native communication facilities of Mach. In Section 2 we discuss the use of Mach IPC as a replacement for Unix sockets in the context of the X11 protocols. In Section 3 we describe the use of a user-level communication protocol based on shared memory. In Section 4 we discuss the performance of the two approaches. Finally in Section 5 we present our conclusions.

2 Using Mach IPC

In our initial implementation, we replaced socket calls in the original X11 system with calls to Mach's IPC facilities. Structurally, this approach is similar to the original socket-based implementation on a monolithic kernel in that it uses kernel routines to pass messages between address spaces.

Most client requests in X are asynchronous, and are buffered at both the client and server end. For example, when a client requests that the server draw a circle on the screen, the client library may buffer the request. On the server side, the processing of the request may be delayed as well. There are two main ways that the client can synchronize its outstanding requests with the server's. The first is `Xflush`, which instructs the client to flush its buffers of any unsent requests to the server. The `Xflush` function returns after sending the buffered requests to the server, so it does not guarantee that they have been processed by the server. Stronger guarantees are provided with `Xsync`. `Xsync`, like `Xflush`, flushes the client buffer, but then requests the server to acknowledge that it has finished all pending requests. The client blocks until the acknowledgement arrives. Most clients generally send several display drawing requests, and then block waiting for user input, which causes an `Xflush`, or they make several display requests, `Xsync` to ensure that the display looks correct, and then make more display requests.

2.1 The X11 server

We modified the server to use the Mach nameserver for establishing a client rendezvous. The server creates a Mach port for client connections, and makes it available to clients through the name server. The server listens on the initial port for incoming messages. Upon receipt of a connection message, it creates a pair of Mach ports, and sends them to the connecting client. One of these ports is monitored by the server for requests from the client, and the other is monitored by the client for responses and events from the server.

An X11 server must monitor the activity of a potentially large number of clients. In the original version of the server, the clients were represented by a set of file descriptors representing sockets. The socket-based server monitored the connections using the Unix `select` call. The X server's `select` call also monitors Unix file descriptors pertaining to keyboard and mouse I/O. These I/O channels are relatively low-bandwidth and are not latency-critical. Consequently, we left their management to the Unix server even in the IPC-based implementation.

Under Mach 3.0, our IPC-based server maintains and listens on a port set, with one port per client. Because the slower I/O channels are accessed with Unix file descriptors, we could not simply replace the original server's `select` call with a call to check the status of the port set. On the server side, we introduced a second thread to handle the blocking I/O Unix call. The primary thread performs blocking receives on the port set, while another monitors I/O activity on the Unix file descriptors. When the second thread learns of pending I/O, it alerts the primary primary thread through a "back door" port. The primary thread wakes up and handles the I/O.

2.2 X11 clients

On the client side, the X11 connection code was replaced by a nameserver lookup, followed by a message to the connection port of the server. Socket writes were replaced with Mach port sends, sending a variable length array instead of writing to the socket in a stream format. Socket reads were replaced with receives from Mach ports.

X11 allows access to the descriptor that is used to access the X11 server. With sockets, this is a file descriptor, and is made accessible so that single-threaded clients can multiplex their activity across several I/O channels. For example, xterm uses the descriptor to perform system calls such as `select` in a set with TTY descriptors. When using Mach IPC, however, there is no real Unix file descriptor through which client/server communication passes. The mixing of file descriptors, which are a strictly Unix mechanism, with Mach ports, which are a strictly Mach mechanism created some difficulties for the implementation. Unlike the server, where all blocking I/O could be controlled since it occurred within a single "program," client behavior is unconfined.

We solved the "mixed metaphor" problem by exporting a pseudo-descriptor from the X libraries. The pseudo-descriptor is simply a small integer that represents the Mach port on which server communication is occurring. We provided library stubs for system calls to watch for the pseudo-descriptor. For example, `select` was written as a client-side stub

to check if the pseudo-descriptor was in the argument set. If not, a Unix `select` call is performed. If the pseudo-descriptor is the only element in the set, a Mach system call is performed to find the number of pending messages, and a result based on this number is returned. If there is a mixture of the port and other file descriptors in the `select` call, we alternate between a `select` with a small timeout and the Mach call until either one returns a status that would be consistent with `select` terminating, or the time value specified by the caller had passed.

Our strategy of slow polling in the client for a multi-way `select` is less than optimal, and is asymmetric with respect to the server. But, given our constraints of not modifying clients, we had few other options. One option, for example, would have been to spawn off another thread in the client to handle the blocking Mach call, and to write to a special notifier descriptor that was being monitored in all `select` calls (as is done on the server side). We chose not to do this because it would have required building all X11 clients as multithreaded applications. Many X11 clients and libraries though assume that they are running in a single-threaded address space. This affects their use of Unix signal mechanisms. By rewriting the `select` call, we were able to pass a handle back to the clients that they could treat as a socket descriptor without impacting the single-threaded assumptions.

3 Mach Shared Memory

Our second approach uses Mach's shared memory facilities for communication between X11 clients and the server. While X11 is a network-extensible window system, it is commonly used for communicating between programs and servers running on the same machine. In such cases, shared memory, rather than kernel-level IPC, can be used as an extremely low-latency communication channel. Through the use of external pagers, Mach allows memory to be mapped into multiple address spaces at once. Once the pages have been mapped, no additional kernel interaction is necessary when accessing the memory.

We use shared memory only for communicating from the client to the server. Communication in the other direction is implemented as in the previous section. There are several reasons for this asymmetry. First, the majority of data is communicated from clients to servers, and not the other way around. Second, most messages from the server to a client result in the client becoming the next process to run, for example, the server's response to an `Xsync` request. On the other hand, most requests from the client to the server can be buffered, allowing the client to run ahead of the server, thereby eliminating context switches.

We modified the server so that during the connection phase it allocates memory for a shared buffer, and then returns that shared buffer to the connecting client. Although the server has access to the shared buffers of all clients, clients themselves do not have access to other clients' shared buffers. Since only the client writes new data into the buffer and only the server reads it, there are no race conditions and no need for explicit synchronization. The client moves a head pointer forward when making a request, and the server moves a tail pointer forward when processing a request.

With the socket-based and IPC-based implementations, each data transfer operation between the client and server, which went through the kernel, could also result in a context switch, allowing the server to perform the outstanding requests. In contrast, the shared-memory implementation avoids the kernel on data transfer, eliminating the possibility of an "accidental" context switch. For example, flushing no longer results in the server being the next process to run, as it generally does with kernel-based communication. All context switches must be forced by applications, which run until they are either descheduled due to quantum expiration, until they block waiting for input, or their shared buffer fills.

The server thread is normally blocked on a message receive for a port set that includes a port used by clients to alert the server that there are X11 requests outstanding. Clients only notify the server by way of this port if they have issued X11 requests that have not yet been processed, *and* the X server has not yet already been notified that there are outstanding requests.

The server shares with all clients a single page of memory that contains a bit indicating whether or not the server is suspended. The server sets the bit before it blocks on the port set. Clients, subsequent to posting a message to the server's wakeup port, clear the bit. In this way, multiple clients may post requests to the server while requiring only one IPC message.[1] When the server wakes up, it scans the buffers that it shares with clients to find and service requests. When all requests have been satisfied, the server goes back to sleep.

4 Performance

We used several programs to benchmark the various X11 configurations. All benchmarks were run on a Gateway 80486 system running at 33 megahertz. The motherboard had 16 megabytes of 70ns ram, with 64 kilobytes of 25ns cache, and a Diamond Speedstar video board. All tests were run with no other users logged in, and no other processes running that were not part of the benchmarks.

4.1 Microbenchmark performance

Our first benchmark is *muncher*, which is a program that comes with the MIT distribution. Muncher repeatedly does xors to a 256x256 window, and flushes its buffers to the server after each complete 256x256 xor (through the `Xsync` command).

We modified the client and server so that we could tell how much time was spent in each during a run of the program. We first ran the program 2500 and 10000 times. This allowed us to subtract off program startup and cleanup overhead. We then modified the server so that it could process each request either once or twice to determine the screen and server

[1] We are aware that the fact that clients can write the bit creates a potential denial-of-service situation (a client erroneously clearing the bit without sending a wakeup message would keep other clients from notifying the server). This has not yet become a problem, but if it does, we will implement one of several obvious solutions, including having the server, rather than the client, write the bit, or having the server periodically wakeup and check the input queues regardless of message traffic.

	2.5 sockets	3.0 sockets	3.0 ports	3.0 shared memory
Client time	1.1	1.1	1.1	1.1
IPC time	1.6	5.0	1.3	0.3
Server time	1.7	1.7	1.7	1.7
Screen time	0.3	0.3	0.3	0.3
Total time	4.7	8.1	4.4	3.4

Table 1: *Muncher performance (ms/iteration)*

overhead. We modified the library to send each request to the server either once or twice to determine IPC overhead. Lastly, we modified the server to write to a fixed single page in memory all requests that would otherwise have gone to video memory. This allowed us to approximate how much of the server time was spent in the server code, and how much was spent controlling the video board.

Table 1 summarizes the results of the measurements using four different implementations of the communication system: Mach 2.5 with sockets, Mach 3.0 using sockets, Mach 3.0 using Mach IPC, and Mach 3.0 using shared memory. The component and total times are for one full 256x256 xor. Taking the Mach 2.5 with sockets implementation as a baseline, 3.0 sockets take almost 75% more time to execute, while 3.0 ports offer about a 6% improvement, and the shared memory implementation offers a 25% improvement.

Any further speedup would have to come from a major rewrite of the client or server internals, or from faster hardware. The client and server each take a fixed amount of time regardless of the transfer protocol used. The data transfer time using shared memory is less than the time for this particular system to actually modify video memory. Any further gains beyond this shared memory version would be negligible, since removing the data transfer completely would only yield about an 8% gain over the shared memory version.

4.2 Macrobenchmark performance

We measured two macrobenchmarks to evaluate the impact that our changes had on application behavior. One macrobenchmark measured the time to send a large file (15000+ lines) to an *xterm* that had jump scrolling activated. Using sockets, this test took 1 minute and 19 seconds to complete under Mach 2.5, and 2 minutes and 15 seconds under Mach 3.0. Using Mach 3.0 ports, the test took only 1 minute and 17 seconds to complete, slightly better than the baseline and much better than the Mach 3.0 socket-based implementation. The Mach 3.0 shared memory version took 1 minute and 5 seconds (85% of baseline).

For applications that do not explicitly synchronize with the server, there can be an even greater improvement in performance. For example, the *maze* program distributed in the MIT release draws and solves a random maze as fast as it can. It only synchronizes with the server (for display update) when it has generated a solution. Using sockets, a draw-solve cycle took an average of 1.91 seconds with Mach 2.5, and 3.97 seconds under Mach

3.0. Under Mach 3.0 with ports, the time drops to 1.88 seconds, for an improvement of only about 2%. However, when run under shared memory, it takes only 1.10 seconds per iteration, for a savings of about 42%.

5 Conclusions

Mach's Unix server can successfully emulate Unix as a user-level application. In some cases, performance may not be as good as with a monolithic Unix system. In these cases, the Mach primitives can be used directly to attack performance bottlenecks. The work described in this paper demonstrates that ports and shared memory can be used by applications on Mach 3.0 to give performance comparable to or better than Mach 2.5.

References

[Bershad et al. 91] Bershad, B., Anderson, T., Lazowska, E., and Levy, H. User-Level Interprocess Communication for Shared-Memory Multiprocessors. *ACM Transactions on Computer Systems*, 9(2), May 1991.

[Draves 90] Draves, R. P. A Revised IPC Interface. In *Proceedings of the First Mach USENIX Workshop*, pages 101–121, October 1990.

[Gettys et al. 90] Gettys, J., Karlton, P., and McGregor, S. The X Window System, version 11. *Software – Practice and Experience*, 20(S2):35–67, October 1990.

[Golub et al. 90] Golub, D., Dean, R., Forin, A., and Rashid, R. Unix as an Application Program. In *Proceedings of the Summer 1990 USENIX Conference*, pages 87–95, June 1990.

[Rashid et al. 87] Rashid, R., Tevanian, Jr., A., Young, M., Golub, D., Baron, R., Black, D., Bolosky, W., and Chew, J. Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures. In *Proceedings of the 2nd Symposium on Architectural Support for Programming Languages and Operating Systems*, April 1987.