Problems Implementing Window Systems in UNIX®

James Gettys

Digital Equipment Corporation
Project Athena
Massachusetts Institute of Technology
Cambridge, Massachusetts 02139

ABSTRACT

Over that last 18 months the X window system has been implemented under 4.2BSD UNIX at MIT at the Laboratory for Computer Science and Project Athena. While on the whole the result X's design is reasonably clean and pleasing, UNIX strongly limited the possible implementation strategies. This paper discusses the problems encountered, how they influenced our design decisions, and suggests issues for future study and development as UNIX evolves toward a distributed environment.

X Window System Design

While this paper is not specifically about the X window system, X will be used as an example for much of the discussion. X is best described using a client/server model. X consists of a collection of client programs which communicate with the window system server. They are implemented entirely in user code. All communications with the window system occur over a stream connection to the window system. X is completely network transparent; i.e. a client program can be running on any machine in a network, and the clients and the server need not be executing on the same machine architecture. The block diagram shown in Figure 1 describes the structure of the system.

X supports overlapping possibly transparent windows and subwindows to arbitrary depth. Client programs can create, destroy, move, resize, and restack windows. X will inform clients on request of key presses, key releases, button presses, button releases, mouse window entry and exit, mouse motion, a number of types of window exposure, unmap (removal of a window from the screen), and input focus change. Cut and paste buffers are provided in the server as untyped bytes. Graphic primitives provided include dashed and dotted multi-width lines, and splines. There is a full complement of raster operations available. The implementation supports color, though the current protocol limits the depth of a display to 16 bits/pixel.

The X window system consists of a collection of UNIX programs providing various services on a bitmap display. There is only a minimal device driver to field interrupts from mouse, keyboard, and potentially a display engine. The X server accepts connections from client applications programs. Window, text and graphics commands are all multiplexed over (usually) a single connection per client. Multiple clients may connect to the server.

The X protocol is the only way to communicate to the window system. The X server enforces clipping, does resource allocation, multiplexes output from multiple clients, performs hit detection for mouse and keyboard events, and performs graphics primitives in windows. The protocol is entirely based on a stream. The current implementation uses TCP as its stream transport layer; though it has been run experimentally using DECNET stream connections. A client program may run on any machine in a network. On a local net, performance is the same or better when run remotely as when run locally given two identical unloaded processors.

The X server is a single threaded server program. Requests from clients are processed in a round robin fashion to provide apparently simultaneous output. This has proven to be sufficient, and vastly simplified the design and implementation. Single threading provides all locking and synchronization without any complex structure. The X server must therefore be very careful never to block waiting on a client, and exploits the observation that each individual graphics operation is very fast on a human time scale (though it may be slow on a systems time scale). The 4.2BSD facilities that make this easy to implement include select(2), non-blocking I/O, and the network mechanism (IPC to unrelated processes).

The current X server implementation does NOT maintain the contents of windows. Refresh of a damaged window is the responsibility of the client. The server will inform a client if the contents of a window has been damaged. This was motivated by a number of observations: 1) clients often have their own backing store, and this must be maintained by most programs when resized anyway; if the window system provides backing store, it is often duplicating existing facilities. 2) keep the window system simple and FAST. 3) the amount of data that would have to be stored for bitmap backing store on color displays is very large. Naive UNIX applications are run under a terminal emulator which provides the refresh function for them.

X delegates as much to a client as possible. It provides low level ‘‘hooks’’ for window management. No less than three window manager programs (a separate client program in the X design from the window system) have been written to date, and provide quite different user interfaces. Menus are left to client code to implement, using the flexible primitives X provides. There have been four different styles of menus implemented to date, including a quite sophisticated ‘‘deck of cards’’ menu package.

Figure 1: Block Diagram Structure of X

X runs as of this writing on four quite different types of hardware, from very intelligent to very simple. An example of a very intelligent (and reasonably well designed) piece of hardware from the programmers point of view is the DEC Vs100, though it suffers due to the nature of its interface to a VAX, which adds overhead and latencies to each operation. A QVSS display on a MicroVAX (VS1 and VS2) is at the opposite end of the spectrum, being a simple bitmap with no hardware assist. Other ports are in progress.

Alternatives to User Process Window Systems

As currently implemented on most machines, the UNIX kernel does not permit preemption once a user process has started executing a system call unless the system call explicitly blocks. Any asynchrony occurs at device driver interrupt level. UNIX presumes either that system calls are very fast or quickly block waiting for I/O completion.

This has strong implications for kernel window system implementations. While window system requests do not take very long (if they did, the presumptions made in X would be unacceptable), they may take very long relative to normal system calls. If a system call is compute bound for a ''long period'', interactive response of other processes suffers badly, as the offending process will tend to monopolize the CPU. One might argue that this is not offensive on a single user machine but it is a disaster on a multiuser machine. If graphics code and data is in the kernel for sharing, it permanently uses that much of kernel memory, incurs system call overhead for access, and cannot be paged out when not in use.

Similarly, in X as well as most other window systems, if a window system request takes too long, other clients will not get their fair share of the display. This is currently somewhat of a problem during complex fill or polyline primitives on slow displays. The concept of interrupting a graphics primitive is so difficult that we have chosen to ignore the problem, which is seldom noticeable. If such graphics primitives occur in system calls, they have a much greater impact on process scheduling.

An alternative to a strictly kernel window system implementation splits responsibility between the kernel and user processes. Synchronization, window creation and manipulation primitives are put in the kernel, and clients are relied on to be well behaved for clipping. Output to the window is then performed in each user process. This has several disadvantages (presuming no shared libraries, not available on most current UNIX implementations). Each client of the window system must must then have a full copy of graphics code. This can be quite large on some hardware, replicated in each client of the window system. For example, the current bit blit, graphics and clipping code for QVSS is approximately 90kbytes, or 18000 lines of C source code. Fill algorithms may also require a large amount of buffer space.

Even worse (as the number of different display hardware proliferates with time on a single machine architecture) is that this split approach requires the inclusion in your image of code for hardware you do not currently have. Upward compatibility to new display hardware is also impossible without shared libraries, but dynamic linking is really required for the general solution.

With much existing hardware it is hard to synchronize requests from multiple processes if the hardware has not been designed to efficiently support context switching. There are sometimes work arounds for these problems by ''shadowing'' the write only state in the hardware. We have seen displays which incur additional hardware cost to allow for such multiprocess access. One must also then face the locking of critical sections of window system data structures if the window system is interruptible.

UNIX internal kernel structuring currently provides most services directly to user processes. It would be difficult to provide network access to the window system if it were in the kernel due to this horizontal structure but a better ability to layer one facility on another would improve this situation. Again, this is a failure of the kernel to be sufficiently modular to anticipate the evolving environment.

X finesses all of these problems: 1) X and client applications are user processes; ergo no scheduling biases. 2) There is only one copy of display code required, in the server, which can be paged since it is completely user code. This also saves swap space, in short supply on most current workstations. The resulting client code is thus small. Minimal X applications are as small as 16k bytes. No graphics code is in an application program. 3) Client code can potentially work with new hardware without relinking, as no display specific code appears in a client program image. 4) Network access to the window system comes at no additional cost, and no performance penalty (in practice, performance is often gained). 5) X avoids system call overhead by buffering requests into a single buffer and delaying writing in a fashion similar to the standard I/O library. The system call overhead for output is therefore reduced by well over an order of magnitude per X operation. 6) User process code is easy to debug. Some complications can arise due to the distributed nature of the system. In practice, this has rarely been a problem. 7) Applications requiring a ''compute server'' can be run from the user's workstation.

Kernel lightweight processes could be used to solve the non-preemptable nature of system calls and would create more options for window system implementations. Since raster operations can be quite long lived, performing these in the current structure allows one process to monopolize the system to the detriment of other processes. Since all context in the system call layer of the kernel is associated with a user process, there is currently no way to divorce such operations from a process and schedule them independently.

While lightweight processes would unnecessarily complicate the X server design (requiring us to lock data structures and perform synchronization), they could be used prevent the most common X programming mistake. Programmers new to X invariably forget to flush the output buffer when testing their first program. A timer driven lightweight process in clients would be useful to guarantee automatic flushing of the buffers.

Shared Memory

On a fast display and processor, X may be performing more than one thousand operations (X requests) per second. If every access to the device requires a system call, the overhead rapidly predominates all other costs. X uses a shared memory structure with the device driver for two purposes: 1) to get mouse and keyboard input and 2) to access the device or write into a memory bitmap.

As pointed out before, X is a single threaded server. Since client programs should be able to overlap with the window system as much as possible (remember that you may be running applications on other machines), it is particularly important to send input events to the correct client as soon as possible. It is therefore desirable to test if there is input after each graphic output operation. This test can be performed in only a couple of instructions given shared memory, and would otherwise require either one system call/output operation (to check for new input) or a compromise in how quickly input would be handled.

All input events are put into a shared memory circular buffer; since the driver only inserts into the buffer, and X only removes from the buffer, synchronization is easy to provide with separate head and tail indices (presuming a write to shared memory is atomic).

Output on the QVSS is directly to a mapped bitmap. In the case of the Vs100, a piece of the UNIBUS† and a shared DMA buffer are statically mapped where both the driver and the X server can access them. Output requests to the Vs100 are directly formated into this buffer, minimizing copying of data.‡ This permits the device dependent routines to start I/O transfers without system call overhead (by directly accessing device CSR registers), and avoids UNIBUS map setup overhead that DMA from user space requires.

These changes dramatically increased performance and improved interactive feel when implemented, while greatly reducting CPU overhead. Since proper memory sharing primitives are lacking in 4.2BSD, it was implemented by making pages readable and writable in system space, where they are accessible to any process. In theory, any program on the machine could cause a Vs100 implementation to machine check (odd byte access in the UNIBUS space), though in practice it has never happened. None the less, it is the ugliest piece of the current X implementation. We are more willing to allow a server process to access hardware directly than kernel code, as it is much easier to debug user processes than kernel code.

The current X implementation uses a TCP stream both locally and remotely, though one could easily use UNIX domain sockets for the local case at the cost of a file descriptor. For current applications, the bandwidth limitations (of approximately 1 million bits/second on 780 class processor) is not major, though faster devices (and image processing applications) would probably benefit from implementation of a shared memory path between the X server and client applications.

Current shared memory implementations in variants of UNIX are not sufficient. Memory sharing primitives should allow appropriately privileged programs to both share memory with other processes and map to both kernel space and I/O space. Shared libraries (available in some versions of UNIX) would also increase the options available to window system designers (see below).

File Descriptors

Andrew, the window system developed at the ITC at CMU [1] uses one connection (file descriptor) per window. While simple from a conceptual level, also allowing naive applications to do output to a window, it ties an expensive resource (file descriptor and connection) to what should be a cheap resource (a window or sub-window). It requires more kernel resources in the form of socket buffers for each file

† UNIBUS is a trademark of Digital Equipment Corporation.

‡ Our thanks go to Phil Karlton, of Digital's Western Research Lab, for the first implementation of this mechanism.

descriptor. In addition, the handshaking required for opening a connection is expensive in terms of time and will become more so once connections become authenticated. The attraction of having a simple stream interface to a window can be had by other means [2]. In addition, if a window is tied to a file descriptor, the application loses the implicit time sequencing provided by the event stream coming over a single connection.

One X application uses more than 120 subwindows, all multiplexed over a single connection. One could postulate a single connection per client for input, and a single connection per window for output; with the limited number of file descriptors in 4.2BSD and other current versions of UNIX, this was eliminated as a possibility. Sixteen client programs seems to be sufficient for most people, (this is limited by 20 file descriptors on standard UNIX, with four file descriptors needed for X; one for the display, keyboard and mouse, two to listen for incoming connections, and one for reading fonts). Sixteen is not a tolerable limit on the total number of (sub)windows, however.

4.3BSD lifts this limit to sixty four. (It can be configured to any size.) While this increase in the number of file descriptors is beneficial, it is still too expensive a resource to use one per window.

Terminal Emulation

The current terminal emulator for X (*xterm*) is a client application, in principle similar to any other application. In practice, *xterm* is probably the most complex and least graceful part of the package. Pseudo teletypes (hereafter called pty's) are used to implement this in 4.2BSD. As currently implemented, ptys consist of a device driver which presents a terminal on one side and a master controlling device on the other side. Data is looped back from one side to the other, with full terminal processing occurring (tab expansion, cooked/raw mode, etc.)

These present a number of problems: 1) pty's are a limited resource. Typical systems have 16 or 32 ptys. On a single user machine, this limit is seldom reached, but on a timesharing machine it can be inconvenient. 2) Since they appear statically in the file system, protection on the tty/pty pairs can be a problem. A previous process that terminates unexpectedly can leave the pty in an incorrect state. *Xterm* is the only application that must run set user id to root to guarantee it can make the tty/pty pair properly accessible and to set ownership on the slave to the user.

The net result is that *xterm* is the most UNIX dependent (and least likely to port between UNIX implementations) of any of the X clients currently existing. Dennis Richie's [3] stream mechanism appears to eliminate most of these problems by allowing stacking of terminal processing on IPC.

Window System Initialization

Most displays capable of running a window system bear little resemblance to UNIX´s model of a terminal connected by a serial line. Current display hardware may require involved initialization before it is usable as a terminal, and may have an interface that looks nothing like the conventional view of a serial device. As soon as the window system is running, however, it is easy to provide a terminal emulator to a user.

Unix currently realizes someone has logged out by the eventual termination of the process started by *init*(8). *Init* is also the only process which can detect when an orphan process terminates, so the restart of a terminal line (or window system) after logout can only be performed by *init*.

The solution taken to support X (or Andrew, which has a similar structure) was to generalize *init*. *Getty*(8) or (in X's case, *xterm*) now opens and revokes access to a terminal or pty rather than *init*. The format of the /etc/ttys file, already extended at Berkeley, was further extended to allow the specification of an arbitrary command to be run as *getty*. For X, this would normally be the terminal emulator. *Init* will also restart an optional window system server process associated with the pty. *Init* must start this process, since *init* is the only process in UNIX that can detect its exit. The initial *xterm* can not be started from a window system server, since the server must exist all the time, and *init* has to know the process id in order for it to detect the *login* process has terminated. The X server process itself opens the display device and performs whatever initialization may be required (for example, the Vs100 requires loading with firmware stored in a file).

Once *xterm* starts execution, it exec's *getty* on the slave side of the pty, and a user can log in normally. When the user's shell exits, *xterm* exits, and *init* can then detect the user has logged out normally.

Init can now be used to guarantee that a process will be kept running despite failures as long as the system is multi-user. Another approach not seriously examined would have made it possible for an orphan process to have a parent other than *init*.

Resource Location and Authentication

At this time, UNIX lacks good network authentication and resource location. The only example of a real name server in widespread use is the internet name server. As UNIX moves toward a distributed systems environment, questions of distributed resource location become important. X at this time does little to solve this problem, relying on either command line arguments or an environment variable to specify the host and display you want the application to use. In reality, it should be closely tied to the user's name, since the name of a machine is basically irrelevant as users often move. X seems to highlight some issues in the future design of such servers that may not be widely appreciated.

The model used to best describe distributed computing goes under the name of the ''client/server'' model. That is, a client program connects to a ''server'' which provides a service somewhere in the network. The additional twist is that the window system is a ''server'' in this model, and other network services may become ''clients'' of the X server. For example, one can envision using services that want to interact with the user's display. The result is that the ''name'' of the X server must somehow propagate through such service requests, along with whatever authentication information may be required to connect the X server in the future. This ''cascaded'' services problem has not been well explored.

The access control currently in X requires no authentication, but is only adequate for workstations, and fails badly in an environment which includes timesharing systems. X can be told to only accept connections from a list of machines. Unfortunately, if any of them are timesharing machines, and you allow access from that machine, then anyone on that machine may manipulate your display arbitrarily. This has the unfortunate side effect of making it trivial to write password grabbers (across the net!) or otherwise disturb the display if access is left open.

The ''name'' of the user's display server also comes and goes with some frequency, as each time you log out, any previously authenticated connection information needs to be invalidated, so no background process from a previous user will disturb the user's display. It is also not uncommon that a single user may use multiple displays, possibly on multiple machines simultaneously. This might be common, for example, in a laboratory environment. Interesting questions arise as to which display to use on what machine. (For example, the user may initiate a request on a black and white display that really works better on a color display; which display on what machine should be used?) We do not believe these issues, in particular the transient and cascading nature of such display services and authentication information, have been properly taken into account in the design of resource location and authentication servers.

Stub Generators and the X Protocol

The X protocol is not a remote procedure call protocol as defined in the literature [4,5], as client calls are not given the same guarantee of completion and error handling that an RPC protocol provides. The X protocol transports fairly large amounts of data and executes many more requests than typically seen in true RPC systems. Given this generation of display hardware and processors, X may handle greater than 1000 requests/second from client applications to a fast display.

X clients only block when they need information from the server. Performance would be unacceptable if X were a synchronous RPC protocol, both because of round trip times and because of system call overhead. This is the most significant difference between X and its predecessor W, written by Paul Asente of Stanford University. On the other hand, a procedural interface to the window system is essential for easy use. We spent much time crafting the procedure stubs for the several library interfaces built during X development.

The original implementation of the client library would always write each request at the time the request was made. This implies a write system call per X request. There was implicit buffering from the start in the connection to the server due to the stream connection. Over a year ago, we received new

firmware for the Vs100, and were no longer able to keep up with the display. We changed the client library to buffer the requests in a manner similar to the standard I/O library; this improved performance dramatically, as the client library performs many fewer write system calls.

Many current RPC [6] argument martialing mechanisms perform at least one procedure call per procedure argument to martial that argument. This is almost certainly too expensive to use for this application. Even if martialing the argument took no time in the procedure, the call overhead would account for ˜10% of the CPU. Stub generators need to be able to emit direct assignment code for simple argument types. Complex argument types can probably afford a procedure call, but these are not common in the current X design.

Proper stub generation tools would have saved several months over the course of the project, had they been available at the proper time. Arguments could be made that the hand-crafted stubs in the X client library are more efficient than machine generated stubs would have been. On the other hand, to keep the protocol simple, X often sends requests with unused data, for which it pays with higher communications cost. It would be instructive to reimplement X using such a stub generator and see the relative performance between it and the current mechanism.

Machine dependencies in such transport mechanisms need further work. The protocol design deserves careful study. Issues such as byte swapping cannot be ignored. With strictly blocking RPC, the overhead per request is already so high that network byte order is probably not too expensive, given the current implementation of RPC systems on UNIX. With the higher performance of the X protocol, this issue becomes significant. It is desirable that two machines of the same architecture pay no penalty in performance in the transport protocols. Our solution was to define two ports that the X server listens at, one for VAX byte order connections, and one for 68000 byte order connections. At a late stage of X development, after X client code had already been ported to a Sun workstation and would interoperate with a VAX display, another different machine architecture showed that the protocol was not as conservatively designed as we would like. Care should be taken in protocol design that all data be aligned naturally (words on word boundaries, longwords on longword boundaries, and so on) to ensure portability of code implementing them.

X would not be feasible if round trip process to process times over TCP were too long. On a MicroVAX† II running Ultrix‡, or on a VAX 11/780 running 4.2, these times have been measured between 20 and 25 milliseconds using TCP. As this time degrades, interactive "feel" becomes worse, as we have chosen to put as much as possible in client code. Birrell and Nelson report much lower times using carefully crafted and tuned RPC protocols on faster hardware; even extrapolating for differences in hardware, UNIX may be several times slower than it could be. Given a much faster kernel message interface, one should be able to improve on the current times substantially. The X protocol requires reliable in order delivery of messages.

The argument against using such specific message mechanisms are: 1) the buffering provided by the stream layer is used to good advantage at the server and client ends of the transmissions. 2) Lless interoperability. X has been run over both TCP and DECNET, and would be simple to build a forwarder between the domains if needed. This reduces the number of system calls required to get the data from the kernel at either end, particularly when loaded.

These times have been improved somewhat by optimizing the local TCP connection, and could be further improved by using UNIX domain connections in the local case.

In general UNIX needs a much cheaper message passing transport mechanism than can currently be built on top of existing 4.2BSD facilities. Stub generators need serious work both for RPC systems and other message systems particularly in light of some of the issues discussed above. We would make a plea that there be further serious study of non-blocking protocols[7]. There should be some way to read multiple packets from the kernel in a single system call for efficient implementation of RPC and other protocols.

---

† VAX is a trademark of Digital Equipment Corporation.

‡ Ultrix is a trademark of Digital Equipment Corporation.

Select and Non-blocking I/O

Without select(2), building X would have been very difficult. It provides the only mechanism in UNIX for multiplexing many requests in a single process. It is essential for the X server to be able to block while testing for work to do on any client connection and on the keyboard device. X will then wake up with the information required to determine which device or connection needs service.

In actual interactive use of X on a very fast display, select accounts for both the most CPU time and the most subroutine calls. Over an afternoon's use on this display, it accounted for more than 20% of the CPU time used. This is not surprising, since most use of the window system is generated by input events going to editors (in our environment), and output character echoing as well as clock and load monitor graphics calls. When not loaded, one would expect on the order of one select call per X request performed. In fact, there are approximately two X requests performed per select call.

One should remember that select's overhead diminishes as the load on the window system increases, both because you are likely to have many requests on a single connection, and because multiple connections may be processed on a single call. Profiling of the server when the display is loaded shows select using a much smaller percentage of the total CPU time.

Note that for the typical case under normal use, TWO system calls will be occurring where one might potentially do. In the output case (from a client), X will be blocked in select awaiting input (one call). It must then read the data from the client and process it (second call). Due to the shared memory described above, we are avoiding a write system call to the display. On input (keyboard or mouse), X will be blocked in select (one call). It then gets the input event out of the queue, determines which client should get the event, and writes it (second call). Again, we have saved a system call to read the data. Note that since buffering may occur on both input and output, the overhead per graphics operation performed will diminish as the load on the server goes up, since the server will perform more work for the same amount of overhead.

Optimally, select should be very cheap. On fairness grounds, one would like to see if more input from a different client is available after each X request. The original X request handler would check after every request for more requests. The current scheduler only checks for more input when all previously read data has been processed, and provides an approximately 30% reduction in X server overhead (all in the select and read system calls).

Summary

The current UNIX kernel implementation is quite inflexible, closing off what might be interesting design choices. Lightweight processes both in the kernel and in user processes could be used to good advantage. The kernel is not properly structured to allow easy use of different facilities together. Streams may be a decent first step in this direction.

Stub generators, message passing and RPC transport protocols all need substantial work as UNIX moves into the distributed world. Using these protocols without stub generators is like a day without sunshine.

Resource location, authentication and naming are issues UNIX has not faced in the distributed environment. Cascaded services present another level of issues which need to be faced in their design.

UNIX has ascii terminals ingrained into its very nature. It will take much more work to smooth the rough edges emerging from the forced marriage of workstation displays with UNIX.

If a system resource is in short supply (as file descriptors are), the correct solution is to lift the limit entirely. Doubling or tripling a limit on a resource only delays the day of reckoning, while still preventing those design strategies that found them in short supply originally.

Shared memory should allow sharing of memory between processes, between the kernel and a process, and between a process and hardware. Shared libraries would open up design opportunities.

More work needs to be done on performance of some of the new kernel facilities. The X server uses select more heavily than any other system call, accounting for the largest single component of CPU time used, though select is not the limit in absolute performance.

Acknowledgements

Without Bob Scheifler of MIT's Laboratory for Computer Science, there would be no X window system.

The list of contributors is now too long for an exhaustive list, and includes Paul Asente, of Stanford University, Mark Vandevoorde, Tony Della Fera, working for Digital at Project Athena, Ron Newman of Project Athena, the UNIX Engineering Group and the Workstations group of Digital. My thanks also go to Sam Leffler, Steve Miller and Noah Mendelsohn for helpful comments during the writing of this paper. My thanks also go to Branco Gerovac for Figure 1.

References

[1] Gosling, J. and Rosenthal, D. ''A Window-Manager for Bitmapped Displays and UNIX,'' to be published in *Methodology of Window-Managers*, F. R. A. Hopgood et al (eds) North-Holland.

[2] Newman, R., Rosenthal, D., Gettys, J. ''User Extensible Streams,'' In preparation.

[3] Richie, D. M., ''A Stream Input-Output System,'' *AT&T Bell Laboratories Technical Journal*, Vol. 63, No. 8, Part 2, pp. 1897, October 1984.

[4] Birrell, A. D. and Nelson, B. J., ''Implementing Remote Procedure Calls,'' *Transactions on Computer Systems*, vol. 2, no. 1, February 1984.

[5] Nelson, B. J., ''Remote Procedure Call,'' *Technical Report CSL-81-9*, Xerox Palo Alto Research Center, 1981.

[6] ''Sun Remote Procedure Call Specification,'' Sun Microsystems, Inc. Technical Report 1984.

[7] Souza, R. J. and Miller, S. P., ''UNIX and Remote Procedure Calls: A Peaceful Coexistence?,'' Project Athena Internal Paper, 1985.