Design and Implementation of a Distributed X-Multiplexor*

Alan T. Krantz *Sarah E. Chodrow* *Michael D. Hirsch*†
Injong Rhee *Julie Sult* *Vaidy S. Sunderam*
Department of Mathematics and Computer Science
Emory University
Atlanta, GA 30322
{atk,sal,hirsch,rhee,jules,vss}@mathcs.emory.edu

Abstract

Application sharing is an important aspect of collaborative computing. One mechanism for sharing applications is a multiplexor, a tool that broadcasts input/output messages from a single client to multiple displays. We describe the design and implementation of a multiplexor for X windows that supports a heterogeneous environment and works effectively across both local and wide area networks. This X-multiplexor is composed of two distinct components, the Collaborative Computing Frameworks X-Multiplexor (CCFX) *which acts as a pseudo-server for the client, and the* Collaborative Computing Session Manager (CCSM) *which renders the client images and supplies the user interface to the system. In effect, CCSM acts as a distributed window manager, ensuring that all participants have the same view of the shared clients.*

This two-headed approach offers significant gains in flexibility and resource usage. CCFX and CCSM can communicate using any reliable transport protocol. This can result in significant reduction of bandwidth, decreased latency, and avoids some TCP/IP communication required by the X protocol. In addition, this architecture facilitates the creation of a virtual X environment. Using this virtual environment, instead of an actual X server's environment, and then translating the virtual properties to a specific server's properties, allows for more robust support of heterogeneous systems.

1 Introduction

One aspect of collaborative work is the sharing of applications, enabling researchers to view, discuss and make interactive changes to computer generated data and images during their study. While performance benefits can be realized by using applications designed for sharing, because many researchers have preferred applications that are often proprietary it is essential to provide a method to share existing applications. Therefore, in order to perform collaborative work using these applications, a method of sharing them without modification must be found. On the other hand, the method used should not preclude collaboration-aware tools from coexisting with the unaware applications in the shared workspace.

Project Objective. The objective in undertaking the Collaborative Computing Frameworks (CCF) project is to develop an environment suitable for collaborative work among researchers in the natural sciences. A subgoal of this project is to support the sharing of existing applications, such as those used for modeling and presentation of graphical results, in an efficient scalable fashion. To realize this objective we designed and implemented a back-end/front-end X-multiplexor consisting of the Collaborative Computing Frameworks X-Multiplexor (CCFX) and Collaborative Computing Session Manager (CCSM). The CCSM contains two functions, rendering the image of CCFX multiplexed applications and providing the user interface to the CCF system. The CCFX is described in Section 2 while the CCSM is described in Section 3.

Background and Related Work. Though application sharing in the Unix environment is not new, motivation for developing CCFX/CCSM stems from limitations of previous work. Some previous work has been proprietary (e.g., Sun Microsystems' ShowMe), others used a centralized architecture (e.g., XMX [1]), or failed to function in a heterogeneous environment (e.g., XTV [2]). Furthermore, many systems for writing collaboration-aware applications (e.g., GroupKit [3] and MMconf[4]) have no facility for sharing collaboration-unaware applications.

The windowing system most commonly found on UNIX platforms is called X. X uses a client server

*Research supported in part by NSF grant ASC-9527186.
†Research also supported by NSF grant DMS-9404261.

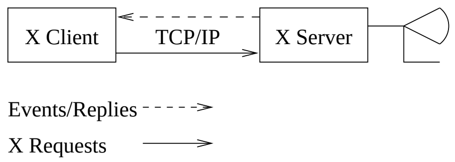

Figure 1: The X client/server model.

model in which the user's application, the client, renders images on the user's display via sending messages to a special program, the server, using TCP/IP (see Figure 1).

Utilizing this model, a 'fake' server can be inserted between the client and X-server (see Figure 2). This *pseudo-server* can then send the instructions for rendering the client's image to multiple X-servers. This is known as X-multiplexing.

In most X-multiplexors[2, 1, 5], a client interacts with a central pseudo-server that then sends the X-requests to a number of X servers, as illustrated in Figure 2. In this centralized design, all requests are passed to a single process, the X-multiplexor, via TCP/IP, and then are forwarded by this process to the appropriate X servers, again using TCP/IP [6]. The pseudo-server often chooses one X server as the "master" X server and then uses the master X server to supply characteristics such as Visual (type of graphical display supported), keyboard mapping (mapping of scan code to key type), and ID space for the client. However, in a heterogeneous environment this approach results in the pseudo-server having to support an $N \times N$ mapping among the N different types of X servers in the system.

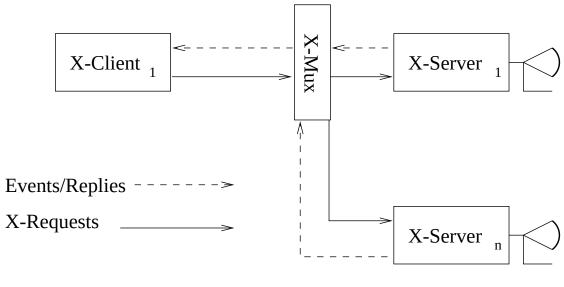

Figure 2: Model of a centralized X-multiplexor.

The primary disadvantage of a centralized pseudo-server is that the pseudo-server is a potential bottleneck. Long delays can arise if the pseudo-server and the X client are not running on the same platform. This is particularly apparent over a WAN or a high-latency network. The pseudo-server makes multiple TCP/IP connections (one per client per display) to carry the multiplexed X traffic with no ability to use alternative network protocols, such as reliable multicast, which may prove to be more efficient.

To address these limitations, we have designed and implemented the Collaborative Computing Frameworks X-Multiplexor (CCFX) and Collaborative Computing Session Manager (CCSM) as a distributed X-multiplexor that allows the sharing of existing X applications without modification on heterogeneous hardware. CCFX/CCSM also supports the development of collaboration-aware tools [7], but that is beyond the scope of this paper.

Implementation Objectives. In designing the CCSM and CCFX we focused on three objectives: the ability to share well-behaved[1] applications; support for heterogeneity; and efficiency over a WAN.

We emphasize support of a heterogeneous environment. Heterogeneity affects X-multiplexing in three areas. First, differing display types induce the complexity of converting client instructions intended for one type of display to be compatible with another type. The X protocol refers to these different display types as *Visuals*. Unfortunately this difference extends beyond pixel size and aspect ratio to the client's interaction with the display hardware (e.g., color manipulation). Our solution to this problem is oriented towards making a shared client functional rather than exactly reproducing the client's output. Second, heterogeneity introduces additional complexity in the handling of keyboards. As with displays, X presents a hardware-dependent keyboard model to a client. The third difficulty arises from different representation of values on heterogeneous architectures, e.g., byte order. While CCFX does not yet support byte-order conversion, the X protocol has a well defined protocol for handling byte-order conversion and hooks are provided by the Xlib and X server to enable byte-order conversion in a painless fashion.

Another objective in designing the CCFX/CCSM X-multiplexor is network efficiency. We achieve this through the CCFX/CCSM architecture, which allows us to transmit data between CCFX (local to the client) and CCSM using methods other than TCP/IP (such as reliable multicast). While this structure induces additional overhead since an additional layer has been added between the client and the multiplexor (CCFX $\leftrightarrow$ CCSM communication), we believe that this overhead (a factor of 2-4, see Section 5.2) is not significant, especially over a WAN, and that the benefits

[1] A well-behaved application is loosely defined as an application that does not manipulates resources assigned to other applications.

are worth the cost. We should note that this overhead is avoidable if one moves the CCFX functionality into a customized Xlib. This approach would either require that existing applications be relinked, which is not always possible, or the use of shared libraries which are not supported on all systems. To take further advantage of the CCFX/CCSM architecture we are exploring solutions for dynamic compression schemes similar to LBX[8] and FHBX[9] to reduce the overhead of WAN communications. We have implemented CCFX/CCSM on top of CCTL[10], a reliable multicast transport protocol developed in conjunction with the Collaborative Computing Frameworks project[7]. CCFX/CCSM could utilize any reliable multicast communication protocol.

WYSIWIS. CCSM performs two major tasks. The first deals with translation of X protocol stream between the virtual machine (defined by CCFX) and the user's X server. The second function is session and window management for the collaborative session. In designing CCSM, a fundamental decision, and one adding great complexity, was that users sharing applications should have an identical view of the shared applications, windows, and data (subject, of course, to hardware limitations). This property is often called "what you see is what I see," or WYSIWIS.

The target user group for our project is natural scientists. We found while experimenting with other systems, systems that did not preserve WYSIWIS were significantly more confusing to these users than systems that did. With WYSIWIS, one user can arrange the windows into a meaningful arrangement for all the other participants, who can then manipulate and interpret the various applications and data. We have found that even when all the users are equally well versed in both computer use and application/interpretation of the programs used, WYSIWIS is a great aid to usability, since users can refer to applications by both name and window position.

Architecture Overview. The CCFX/CCSM X-multiplexor consists of two major components–CCFX (collaborative computing X-multiplexor) and CCSM (collaborative computing session manager)–implemented on top of a reliable multicast layer. CCFX is responsible for acting as a proxy server for X clients (usually run on the same machine or network as CCFX), accepting connections, describing the virtual X environment, responding, when possible, to X-requests locally, sending X-requests to CCSM in an efficient fashion, and collecting responses to X-requests and X events. CCFX forces the client to operate in a well understood state by describing to connecting clients a virtual X environment with known properties. However, because the the client may then operate in a mode not compatible with the native server being used, a front end, CCSM, is needed to convert the client's X-requests so they are compatible with the user's X server.

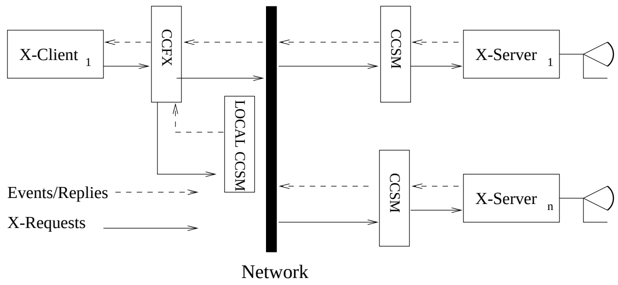

Figure 3: CCFX/CCSM communication architecture.

CCSM is responsible for translating information between the virtual X server and the local display engine (which is usually an X server). CCSM also creates a "desktop" window on the local display. CCSM draws all windows, supplies a user interface for starting applications, and ensures that all users' desktops have an identical appearance. CCFX creates an *input token* for each client and this token is passed among the CCSMs. The CCSM with the token is permitted to send input to the client. All CCSMs in a session function collectively as a distributed window manager, maintaining WYSIWIS throughout the session. In essence, CCFX acts as a server to the client while CCSM renders the images on the user workstation. It is important to note that a one-to-one relationship does not exist between CCFX and CCSM. (See Figure 3.) The system supports an arbitrary number of CCFXs and CCSMs with each CCSM representing a user's display and each CCFX serving local clients. This architecture allows for CCFX and CCSM to be dynamically added and removed from the system, representing users that joins or leaves an active session. While no machine boundary is defined for the various programs in use (CCFX, CCSM, user clients) a typical usage of this system would find one CCSM, one CCFX and none, one or more clients running on a user's local workstation.

The current transport layer used by CCFX/CCSM is provided by the CCTL (collaborative computing transport layer) library [10], which allows efficient communication among CCFXs and CCSMs in a session. CCTL uses reliable multicast when possible, and otherwise uses fast reliable unicast. The underlying transport mechanism is hidden from CCFX/CCSM through a programming model that allows both point-to-point and broadcast communication.

As an optimization to reduce latency, CCFX will accept X-replies from one of two CCSMs per client under its control. X-replies are accepted from the CCSM that holds the input token for a particular client, or a CCSM designated as "local" (which may or may not reside on the same platform), as illustrated in Figure 3. There are a few exceptions, such as `QueryPointer` in which an X-reply represents user input and therefore can only be accepted from the token holder. Also, X-events, which usually represent user interaction, are only accepted from the token holder.

2 CCFX

The X protocol was designed to support a variety of display and keyboard hardware. The hardware in use dictates the abstraction X presents to the clients. Likewise, this abstraction influences an X client's behavior. In order to reduce the effort of supporting many different servers, CCFX creates a virtual X environment that forces the client to operate in a well understood mode, independent of the actual hardware in use.

Creating the Virtual Environment. When an X client first connects to an X server, the server sends the client a description of the environment, including the type of Visual available, the range of scan codes that the keyboard might generate and how ids can be assigned. When a client connects to CCFX, CCFX returns a description of the *virtual* environment. CCFX currently supports two virtual X servers differing in Visual type–24 bit TrueColor and 8 bit PseudoColor–that emulate real server environments with varying degrees of accuracy. 24 bit TrueColor was chosen for ease of implementation and image verisimilitude, while 8-bit PseudoColor was chosen to experiment with the X protocol and for bandwidth efficiency. The selection of the virtual X server and the type of Visual a client uses is determined by the port that the client uses when connecting to CCFX. A complete description of TrueColor and PseudoColor modes can be found in [5]. The set of supported keycodes is static and arbitrarily chosen to be in the range of 8 to 167 to correspond to the virtual keyboard table we have constructed.

Visual Translation. Unlike other X-multiplexors (e.g., XMX) that have a "master" server and use the visuals supported by that server, CCFX dictates to the client what Visuals are available at the time of the client's connection. CCFX allocates two sockets for client connections, one for 8 bit PseudoColor and one for 24 bit TrueColor. CCFX directly processes all X-requests regarding color manipulation, including `AllocColorCells`, `LookupColor`, `QueryColors`, `AllocColor`, `AllocNamedColor` and `CreateColormap`. In TrueColor mode the processing of these requests is straightforward. In PseudoColor mode the processing of the X-request is somewhat more complicated. CCFX initially constructs a color cube [5]. ¿From this color cube, it performs a best fit for `AllocColor`, `AllocNamedColor`, `LookupColor` and `QueryColors` by taking the requested rgb and finding the pixel with the closest preallocated rgb value. In essence we have implemented StaticColor but we choose to report this visual as PseudoColor in order to support a greater number of existing applications.

Keyboard Translation. Different machines have different hardware keyboards (different sets of scan codes), creating a potential source of incompatibility in a heterogeneous environment. Our solution is for CCFX to present clients with a virtual keyboard type independent of the actual hardware. Each CCSM is responsible for translating local, hardware dependent, scan codes to CCFX scan codes.

Atom Translation. In X, atoms are used to attach meta-data to a client. They are often used for interclient communication–either for making hints available to other clients, such as a window manager, or for cut-and-paste operations. `InternAtom` takes a symbolic name and assigns it an integer. CCFX assigns a value to the atom, then passes the value on to each CCSM. Each CCSM is responsible for building one table per CCFX to map the local X server's atom values to the CCFX's values. The current implementation supports limited cut-and-paste operations and other forms of interclient communication.

3 CCSM

The Collaborative Computing Session Manager (CCSM) is the user interface to a distributed collaboration environment. It has two primary roles. First, CCSM is responsible for translating between the local server environment and the virtual machine environment. Second, CCSM creates a virtual desktop upon which shared windows appear. This desktop enables application launching, token control, job control, and ensures a WYSIWIS graphical environment. CCSM *is* the shared environment, with CCFX hidden from the user's view. CCSM also manages windows from collaboration-aware applications that bypass CCFX.

3.1 Translation from the Virtual Machine

One function of CCSM is to translate X messages between a CCFX and the local server. A separate connection for communication between a CCSM and its local X server is established for each client CCSM handles. CCSM translates window identifiers in a fashion similar to [2], by replacing the base portion of the ID assigned by CCFX with the one assigned by the local X server while keeping the client's offset. Visual types in X requests are translated to the local environment and references to the root window are replaced with references to the CCSM desktop. CCSM also translates pixel values from the virtual machine to the local X server, and translates local event and reply packets to the virtual machine.

Visual Translation. CCSM translates Visuals by converting those fields of an X request from the virtual machine that refer to pixel values to values which are compatible with the local machine. If CCSM is running on an 8 bit PseudoColor machine a colormap is created as described in Section 2. Using one colormap for all 8 bit PseudoColor clients prevents "flashing" when switching among clients and allows straightforward translation between 8 bit PseudoColor and 24 bit TrueColor. By using the color cube, all colors used by the client will have a reasonable approximation on the desktop. Furthermore, if the client is running in 8 bit PseudoColor mode, it will be using the same color cube as CCSM, and pixel translation from CCFX to CCSM will be trivial.

Keyboard. CCSM translates the keycodes in keyboard events from the system-dependent keycodes to CCFX's system-independent keycodes before events are passed to the client. If an application is machine-specific, and expects a given key to generate a specific keycode, then the application will not perform as expected. However, hardware-dependent applications are unlikely to perform well in any foreign environment, and are thus unsuitable for sharing.

3.2 The User Interface

CCSM is not merely an interface for application multiplexed with CCFX. It is designed to support shared applications in several modes by allowing, for example, input multiplex application to share the virtual machine[11] as well as CCF aware applications to place windows on the desktop. However, we concentrate in this paper on X-multiplexed applications.

The Virtual Desktop. When CCSM begins, it appears as a standard X application. The CCSM window has a menu and button bar at the top and a large window below. This window provides a view of the shared "virtual desktop". Any shared applications open windows inside this desktop. Since collaborators may have different size screens available to them, the full virtual desktop may be larger than the physical screen on which CCSM is displayed. The CCSM window provides a resizable, scrolling *viewport* into the virtual desktop. (See Figure 4.)

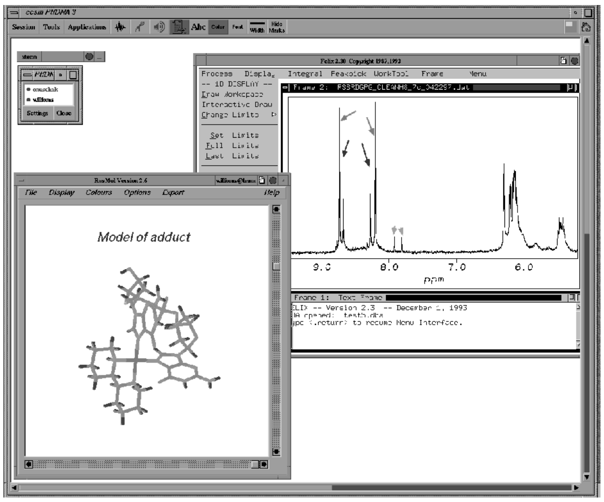

Figure 4: CCSM with session in progress.

CCSM presents the user with dialogue boxes to launch or configure shared applications. The boxes prompt the user for the application name, command-line options, type of sharing, desired Visual, and whether to add this configuration to the user's menu of shareable applications.

Window Management. One of the requirements of CCSM is that it act as a distributed window manager. CCSM is not a "true" window manager in the X sense of the word, but it *de facto* acts as one. It decorates each window in the virtual desktop with a title bar, a boundary, a window control button (with the usual window operations), an iconify button, a refresh button and an input token button. Since CCSM preserves WYSIWIS, when one collaborator modifies a window the same modification occurs to the corresponding window on every virtual desktop in the session.

Input Control. The input token button serializes accesses to a an application. Each displayed client has an associated input-control token, held by only one CCSM at a time. The button is green when the local CCSM holds the token, yellow when another user has requested the token, and red otherwise. Only the token-holder can perform window operations[2], or pass mouse or keyboard events to the application running in that window. A user requests the token by clicking

[2]Any user can request a refresh of a window through the refresh button.

on the red button. The current token control mechanism supports both a "snatch" scheme, in which a requester is immediately granted the token, and a "request-grant" scheme, in which the token holder chooses whether to release the token to a requester.

4 The Latecomers Problem

The latecomers problem arises when a user wishes to join a collaboration session already in progress. Ideally, a latecomer should be able to interact with clients that were initiated before he joined the session.

In CCFX/CCSM we take a "filtering" approach rather than an optimal approach to solving the latecomers problem. We log all X packets that define or modify resources. When a latecomer joins the session, each CCFX sends a message to the latecomer's CCSM indicating which clients are active, and transmits the saved X-requests with a control message indicating the current sequence number. Finally, an expose event is sent to each client.

Our approach is non-optimal with regards to the amount of information saved; the state information required by the latecomer's CCSM consists of the final state of each resource, not its transitions. However, we have found in practice that simple applications, like `xterm`, only require about 1 Kbyte of storage for the saved packets and very complex applications, those that define many pixmaps, like `xv`, require about 50 Kbytes in 8 bit mode. How much memory an optimized solution using dependency graphs [12] would save is unclear, although one could construct an artificial application that would require an optimized solution to function.

5 Experimental Results

In this section, we present the results from two classes of experiments: qualitative experiments in which we test the usability of CCFX/CCSM, and quantitative experiments in which we compare the performance of CCFX/CCSM to XMX.

5.1 Qualitative Experiments

As the CCFX and CCSM were developed, we ran a series of experiments with both the computer scientist designers and the natural scientists as users. In these experiments, we tested subjective criteria: usability of shared applications and utility of the system as a whole. Our observations follow.

The first experiment compared, for a single user, the *perceived* performance of a graphics-intensive application run through CCFX/CCSM vs. the performance of the same application run remotely and displayed on the local desktop. As expected, the application run through CCFX/CCSM was noticeably slower, because of network latency between the CCFX, CCSM and X server and translation from the virtual environment to the actual environment.

In the second set of experiments, we set up sessions with different numbers of collaborators (ranging from two to eight per session), and X-multiplexed various test applications through the CCFX/CCSM. The test applications included `xterm`, `emacs`, `rasmol` (a molecule viewer), `xv`, `tgif` (a drawing tool), `Xess` (a spreadsheet), and `xtetris` (a real-time, nonproductivity tool), run both singly and simultaneously. These applications were not modified in any fashion for use with CCFX/CCSM. The virtual desktops were run on various Sun displays (8 bit PseudoColor, 24 bit TrueColor) and SGI Indy's (24-bit TrueColor) within a LAN, and across subnets within the Emory campus. The collaboration tools that we provide with the system (audio, text-based chat, and clearboard) were also used.

We concentrated on user-interface issues and performance in these experiments. We noted that in a collaboration, WYSIWIS makes application sharing observably simpler to users. The learning curve for novice users of an application was more easily overcome with WYSIWIS than without. We experimented with various types of input control, and determined that having token management and refresh control available as single-press buttons ornamenting each window made the system much more usable.

Performance-wise, the applications were quite usable, although they were visibly slower than when run stand-alone. With multiple desktops, any translation slowdown was subsumed by the overhead of the communication layer. Optimizations to the transport layer made the most visible difference to collaboration performance. At this point, transport efficiency remains the main performance bottleneck.

We have also run several smaller scale WAN experiments. We have had limited success with an earlier version of the transport layer establishing a usable collaboration session between Emory and Japan. More recently, we have tested CCFX/CCSM between Emory and the University of Tennessee at Knoxville, with fewer participants per session and a smaller number of tools than in the previous experiments. As the transport layer performance improves, collaboration performance improves considerably. More substantive user-based experiments are underway.

5.2 Quantitative Experiments

Below we describe our preliminary results in evaluating the performance of CCFX/CCSM. Note that both CCFX/CCSM and the CCTL transport layer[10] over which the system is implemented are works in progress, and that these benchmarks do not systematically cover all aspects of the system.

We ran a series of four benchmarks based on a slightly modified version of the standard X benchmark XBENCH[13]. We compared CCFX/CCSM performance to the performance of a local client, a remote client, and XMX on a LAN, and measured CCFX/CCSM scalability. We chose to compare our system to XMX for two reasons. First, XMX is a simple, centralized X-multiplexor which presumably has little overhead in multiplexing the X stream; thus it serves well as a baseline. Second, XTV, another freely available X-multiplexor, was not stable enough on our Sparc-stations to run X benchmarks.

For our experiments, we chose three benchmarks from XBENCH–*line2*, *line400*, and *arcs400*–and a fourth locally implemented benchmark *atom*. *line2*, *line400* and *arcs400* are standard XBENCH tests which draw a series of lines of width 2, lines of width 400, and arcs of width 400. These experiments give some indication of performance as the system becomes less dependent on network bandwidth and more dependent on local processing time. *atom* sends a GetAtomName request and waits for a reply. In the CCFX/CCSM architecture, the request is handled by the local CCFX.

We ran the benchmarks on one quad-processor UltraSparc and 8 uniprocessor UltraSparcs, connected via a 100 Mbit Ethernet LAN. The quad-processor machine was used to execute the client and (where relevant) the CCFX. Individual CCSMs and XMX were executed on the uniprocessor machines.

For reference, we performed two standalone experiments–one in which the client was run on the same uniprocessor host as the X server, and one in which the client was run on the quad-processor machine, and its output was displayed on a uniprocessor host. The results of these reference experiments are displayed in Figure 5, along with the corresponding results for XMX and CCFX/CCSM. The table also shows an interesting result in row 4 where performance is degraded by having the client compete for resources (cpu time) with the X server.

Expr.	Local Client	Remote Client	XMX	CCFX/ CCSM
line2	888313	5671805	438630	45114
line400	84479	82470	46166	24527
arcs400	13058	12488	7272	4064
atom	371263	543864	545646	543544

Figure 5: Performance of local client, remote client, XMX, and CCFX/CCSM on one display.

The drawing experiments (*line2*, *line400* and *arcs400*) were chosen to illustrate the performance of the system as increased load is placed on the X server. These experiments do not test the effects of network bandwidth and latency, in which CCFX/CCSM should excel. The *atom* experiment shows the advantages of handling request with the CCFX (local to the client) when possible. Figure 6 shows the results of these experiments for one client displayed on one to eight machines.

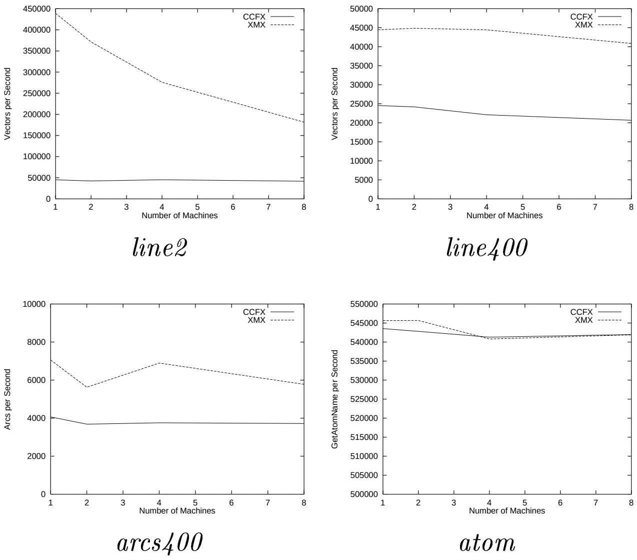

Figure 6: Results of experiments comparing CCFX/CCSM performance to XMX.

The CCFX/CCSM architecture favors WAN operations and moderate scalability. CCFX/CCSM touches the X stream several times to translate from the client environment to the CCFX to the local X server for display. XMX does not manipulate the X stream beyond the multiplexing. Therefore, when translation overhead dominates, we would expect CCFX/CCSM to be slower than XMX in most cases by at least a factor of two.

The underlying transport layer provides group communications (reliable and/or atomic multicasting) at the cost of higher communications overhead. The transport layer effects are most easily seen in the graph of the *line2* experiment. However, note that the CCFX/CCSM lines are close to flat in the *line2* and *arcs400* graphs, demonstrating the benefits of multicasting, whereas XMX degrades.

The performance of CCFX/CCSM in *line400* and *arcs400* is more encouraging. Much of the difference in performance between CCFX and XMX can be attributed to the overhead of sending data from the

CCFX to the CCSM. As discussed earlier, this overhead could be removed by using a "custom" Xlib. The performance of XMX is relatively flat in *line400* and *arcs400* because network bandwidth is less of a bottleneck. The *atoms* benchmarks illustrates the benefit of handling request locally. CCFX/CCSM's performance is almost identical to XMX, which also handles GetAtomByName requests locally, and to "Remote Client" entry in Figure 5.

In the future we hope to have results which are both more systematic and examines the performance over a wider range of conditions including WAN behavior.

While the XBENCH results for CCFX/CCSM are less than we had hoped, our users generally are not affected by the translation overhead, except in very high-bandwidth, bitmap intensive applications such as `rasmol` or `xv`. Vector graphics applications such as `xmaze` and `xtetris` are quite responsive. We have not found a benchmark suite for interactive X applications, which would better emulate our users' interaction modes.

6 Conclusions

We have designed and implemented a distributed X-multiplexor to support collaboration across a heterogeneous environment. This X-multiplexor consists of two components, CCFX and CCSM, that communicate using the CCTL transport layer [10]. CCFX behaves as a pseudo-server to the client applications, translating local requests and events to the virtual environment. CCSM acts as a front end to each user's X server, translating the virtual environment events to the local display, and provides the desktop through which the collaboration occurs. The two subsystems cooperate to support WYSIWIS and serialized input control to X-multiplexed applications.

This two headed approach offers significant gains in flexibility and resource usage: using a virtual environment instead of any actual X server's environment, and then translating the virtual properties to actual server properties, allows for more robust support of heterogeneous systems; by splitting translation responsibilities between CCFX and CCSM, we can use CCTL (or any reliable multicast protocol) to communicate, resulting in significant reduction of bandwidth.

Our experiments show that using CCSM as a distributed window manager to preserve WYSIWIS increases usability, especially for novice users. Implementing control mechanisms (token, refresh) as window decorations allows more natural user interactions than menu-based controls. We expect performance to continue to improve as advances are made in the underlying transport layer.

References

[1] O. Jones, "Multidisplay software in X: A survey of architectures," in *The X Resource*, vol. 6, pp. 97–113, O'Reilly & Associates, Inc., 1993.

[2] H. Abdel-Wahab and M. Feit, "XTV: A framework for sharing X window clients in remote synchronous collaboration," in *IEEE TriComm '91*, pp. 159–167, April 1991.

[3] M. Roseman and S. Greenberg, "Building real time groupware with GroupKit, a groupware toolkit," *ACM Transactions on CHI*, March 1996.

[4] T. Crowley, P. Milazzo, E. Baker, H. Forsdick, and R. Tomlinson, "MMConf: An infrastructure for building shared multimedia applications," *Proceedings, CSCW 90*, pp. 329–342, 1990.

[5] A. Nye, *Volume One: Xlib Programming Manual for Version 11.* Sebastopol, CA: O'Reilly & Associates, Inc., 1993.

[6] H. Abdel-Wahab and K. Jeffay, "Issues, problems and solutions in sharing X clients on multiple displays," *Journal of Internetworking Research & Experience*, pp. 1–15, March 1994.

[7] S. Cheung, S. Chodrow, T. Goddard, P. Gray, M. Grigni, M. Hirsch, P. Hutto, A. Krantz, S. Olesen, I. Rhee, J. Sult, and V. Sunderam, "CCF: Collaborative computing frameworks." In preparation., 1997.

[8] J. Fulton and C. Kantarjiev, "An update on low bandwidth X (LBX)," in *Proceedings of the 7th Annual X Technical Conference*, January 1993.

[9] J. Danskin, Q. Zhang, and D. Abrahams-Gessel, "Fast higher bandwidth X," in *International Multimedia Networking Conference*, September 1995.

[10] I. Rhee, S. Y. Cheung, P. W. Hutto, and V. S. Sunderam, "Group communication support for distributed multimedia and CSCW systems," in *17th IEEE International Conference on Distributed Computing Systems*, June 1997.

[11] A. Krantz, I. Rhee, C. Breuker, S. Chodrow, and V. Sunderam, "Supporting input multiplexing in a heterogeneous environment," in *Third Joint Conference of Information Sciences*, 1997.

[12] G. Chung, "Accommodating latecomers in a system for synchronous collaboration," Tech. Rep. TR91-038, Department of Computer Science, University of North Carolina at Chapel Hill, 1991.

[13] C. Gittinger, "`http://tecfa.unige.ch/pub-/software/unix/xbench/`," April 1995.