Issues, Problems and Solutions in Sharing X Clients on Multiple Displays

Hussein Abdel-Wahab
Department of Computer Science
Old Dominion University
Norfolk, Virginia, 23529
wahab@cs.odu.edu

Kevin Jeffay
Department of Computer Science
University of North Carolina at Chapel Hill
Chapel Hill, N.C. 27599
jeffay@cs.unc.edu

SUMMARY

Many computer conferencing systems based on the X Window System have recently emerged. While these systems hold the promise for fostering collaboration among groups of geographically separated individuals, they are, at present, difficult to build because the X programming model does not support conferencing. This paper describes the salient problems that face the designers of X-based shared window systems and provides solutions and implementation principles for addressing the problems. The enumeration of issues and solutions is based on our experiences with XTV — an X-based shared window system we have developed.

KEY WORDS

X protocol, Computer Conferencing, Computer Supported Collaborative Work, Internet Protocols, UNIX Network Programming, Client/Server Model, Distributed Systems.

SHARED WINDOW SYSTEMS

Growing interest in concurrent engineering and computer-supported cooperative work has led to the development of a number of computer conferencing systems that allow geographically distributed groups of individuals to (simultaneously) view and manipulate shared images, documents, or programs, while they communicate via audio and possibly video links [1-6]. There are two basic approaches to supporting such conferences. The first involves the development of so-called *collaboration-aware* applications — special-purpose applications, that directly support multiple, simultaneously active users. Alternatively, one can leverage the large base of existing single-user applications by augmenting a window system to support the sharing of application interfaces (*e.g.*, windows) across multiple, distributed users. The resulting window system is typically called a *shared window system*. The X Window System has been a particularly popular vehicle for experimentation with, and development of, shared window systems [1,4,5]. Indeed, we have developed, and placed in the public domain, a shared window system, called XTV (X Teleconferencing and Viewing [1,2]), based on the X Window System. XTV allows a conference to be created around one or more arbitrary X applications. Conferees have the same view of shared applications and, by following a simple floor-passing protocol, may control the shared applications.

The X Window System was not originally designed to support conferencing and hence a number of technical problems confront the developer of an X-based shared window

In: *Internetworking — Research and Practice*, Vol. 5, No. 1, (March 1994) pp. 1-15.

system. Patterson provided an initial assessment of the salient difficulties encountered in the development of these systems [7]. We reiterate the difficulties Patterson describes and extend the discussion to new problems such as accommodating applications that establish multiple connections to an X server. Moreover, we describe candidate solutions for each problem. This paper can therefore be viewed as a terse guide to the developers of X-based shared window systems. By extending the classification of problems in such systems we also hope to stimulate discussion on how one might design a collaboration-aware window system to make utilities such as a conferencing system easier to build, more reliable, and more efficient.

In the following sections, we discuss eleven generic problems:

- Traffic interception and Connection Set Up.
- Floor Control, Why and When.
- Routing of Requests, Replies and Events.
- Sequence Numbers.
- Resource and Atom Translation.
- Colors.
- Accommodating Latecomers.
- Applications with Multiple X Server Connections.
- Popup Menus.
- Window Resizing.
- Handling Extension Requests and Events.

TRAFFIC INTERCEPTION AND CONNECTION SET UP

Shared window systems based on the X Window System work by intercepting the X client/server protocol stream and redistributing the client-to-server stream to the X servers of conference participants while combining the server-to-client streams into a single stream that appears to the client to come from some logically virtual X server. Figure 1 depicts the principle involved. (See reference 1 for a more detailed discussion.) A packet interception and distributor process (referred to as the PID process) listens and accepts connections from clients at TCP/IP port number $P = Q + a$, where Q is the port number used by the X server to listen and accept connections from clients, and a is a constant greater than 0. Port P is the next available port that PID can get starting from $Q + 1$. To have an X client connect to the PID instead of the X server, we change the Unix DISPLAY environment variable using the command

```
setenv DISPLAY $hostid:a.0
```

before executing the client program. Note that if we reset the DISPLAY variable back to `$hostid:0.0` the client will contact the X server instead of the PID process.

As shown in Figure 1, the PID process reads every X request message from the client and conceptually distributes each message to all displays (remote X servers) participating in the conference. The first request from the client is the `Connection Request` message [8]. The reply to this request determines the behavior of the client throughout its life. Since this request is forwarded to n displays, n replies will be sent back to the PID. The

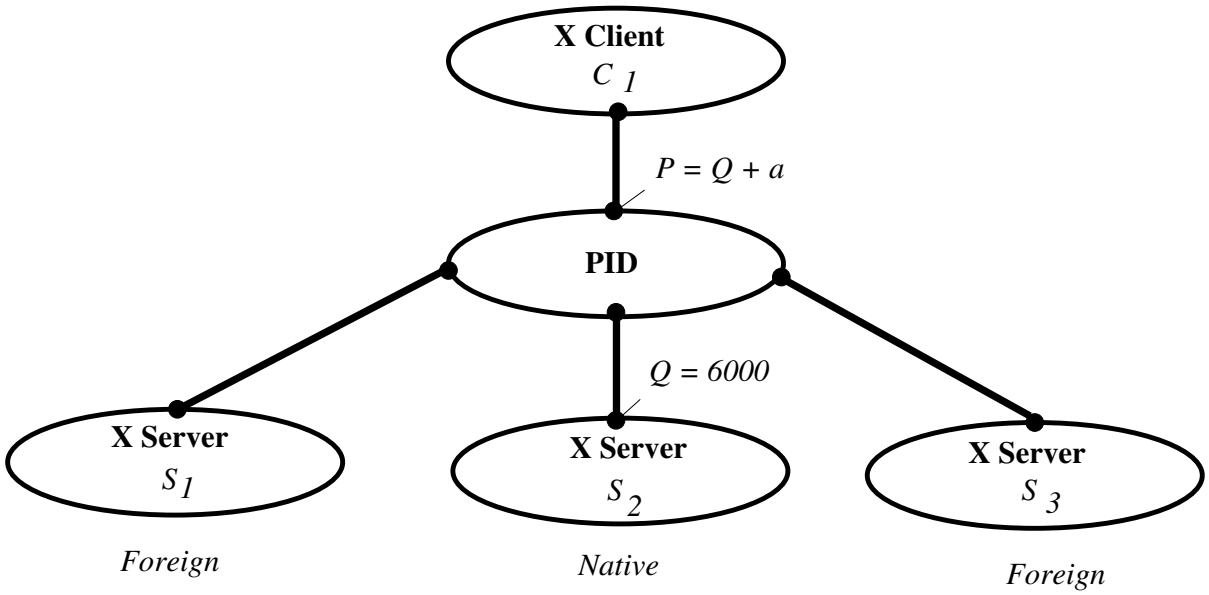


Figure 1: Interception and Distribution of Traffic

PID must select 1 out of *n* replies and send it to the client. The display corresponding to the selected reply is called the *native* display; all others are called *foreign* displays. In Figure 1, display S_2 is the native display and the other two displays are foreign.

FLOOR CONTROL: WHY AND WHEN

If we allow messages (replies, events and errors) from more than one display to be delivered to the client without any coordination, the client might behave abnormally and terminate prematurely. For this reason controlling the input to clients is required for the correct operation of most clients. There are several approaches to controlling the source of the input received by a client. The simplest approach is to use a "token" that participants must acquire in order for their input to be sent to the client. This is similar to the mechanism used for media access on a token ring network. In the following we use the term "floor" instead of token.

The central problem in floor control is deciding when the floor can be passed from one participant to another. For example, if participant *A* holds the floor for a client *T* and a request that needs a reply is sent from *T* (*e.g.*, `QueryPointer` request), then the PID must wait for a reply to the request from *A* before the floor is passed to another participant. In general, the floor is *passable* if and only if there is no pending reply for a request.

There are two extremes for floor control: *explicit* and *implicit* floor control. In explicit floor, a participant explicitly asks for the floor. If the participant's request is granted, he may provide input to the shared clients until he releases the floor. A floor-holder explicitly releases the floor when he either no longer needs it or when he is asked by another participant to release it. In implicit floor control, the floor is implicitly assigned to any participant who provides input to the client (by generating key or mouse events in one of the client's windows). If a another participant provides input to the client and the floor is passable, then the floor is implicitly released from the first participant and is assigned to the second. This is similar to switching a single CPU between multiple processes in a time sharing system.

For informal conferences between colleagues, implicit floor control is desirable because it facilitates rapid, unconstrained interaction. Implicit floor control is problematic, however, as it is possible for multiple participants to generate a sequence of events that could not have been otherwise generated by a single user. Since the majority of X clients expect to interact with only a single user at a time, such a sequence of events can cause a client to abort.

Explicit floor control is more structured and rigid and hence can slow the pace of interaction (although this may be desirable in certain instances). Explicit floor control is, however, "safe" in the sense that the sequence of events seen by the client is always consistent with the actions of single user.

A remaining question is what becomes of the inputs generated by participants when they do not hold the floor. One solution is to discard the input and inform the user that his input is being ignored. Another solution is to buffer the input until the floor is acquired and then supply the buffered input to the client. The first solution is more suitable for the explicit floor control while the second is suitable for a more practical form of implicit floor control.

Some inputs generated by non-floor holders are always sent to the client. For example, events such as `Expose` events must always be sent to the client since they can determine which portions of a client's interface is currently visible to a participant.

ROUTING OF REQUESTS, REPLIES AND EVENTS

Normally an X client forwards its requests to a single X server and the replies, events and errors generated by that server are sent back to the client. However, in a shared window system an X client may be connected to $n \geq 1$ X servers via a PID as shown in Figure 1 (although the client still thinks it is connected to only one X server). The issue here is whether or not to forward a given request to one or all n servers and, in the latter case, to decide which of the replies from the n servers to send back to the client. In the X protocol, some requests (called round-trip requests) require an immediate reply from the server before the client can continue. Other requests do not require a reply from the server unless there is an error. The "core" X protocol contains 82 requests that do not normally require or generate a reply and 37 requests that require replies. A request that does not require a reply should be sent to all n servers. Examples of such requests are `CreateWindow`, `DestroyWindow`, `PolyLine`, and `StoreColors`. The reason for sending the request to all n servers is that the request may create or change server resources.

We group requests that require replies into three groups according to how a request should be routed:

R1 — *Requests that should be sent to all n servers.* Requests that alter the state of a server by creating or changing resources must be sent to all servers. Examples of such requests are: `AllocColor` and `InternAtom`. For this type of request it makes no difference to the client where the reply comes from. Since for a given request we have n possible replies, we may arbitrarily select one reply and send it back to the client.

R2 — *Requests that may be sent to only the floor-holder's server.* These are requests that have no effect on the state of a server and hence need not be to sent to all n servers. Examples of such requests are: `QueryPointer` and

`TranslateCoords`. Naturally the reply to a such a request has to be obtained from the floor-holder's server.

R3 — *Requests that may be sent to one or all servers*. Examples of such requests are: `ListFonts`, and `ListHosts`. For simplicity, this type of request may be sent to only one server and the reply obtained from the server is sent back to the client. A conferencing system may, however, choose to send a request of this type to all *n* servers. In such case, the system may combine the *n* replies into one consensus reply and send it back to the client. For example, if `ListFonts` is sent to all servers, we may get *n* lists of fonts. The system may form a single list of the common fonts among the *n* lists and send it to the client as a reply for the request.

The events generated by the X servers are routed as follows:

E1 — Events that should be sent to the client from the floor-holder server. Examples of such events are: `KeyPress`, `KeyRelease` and `EnterNotify`.

E2 — Events that may be sent to the client from any (but only one) server. Examples of such events are: `CreateNotify` and `DestroyNotify`.

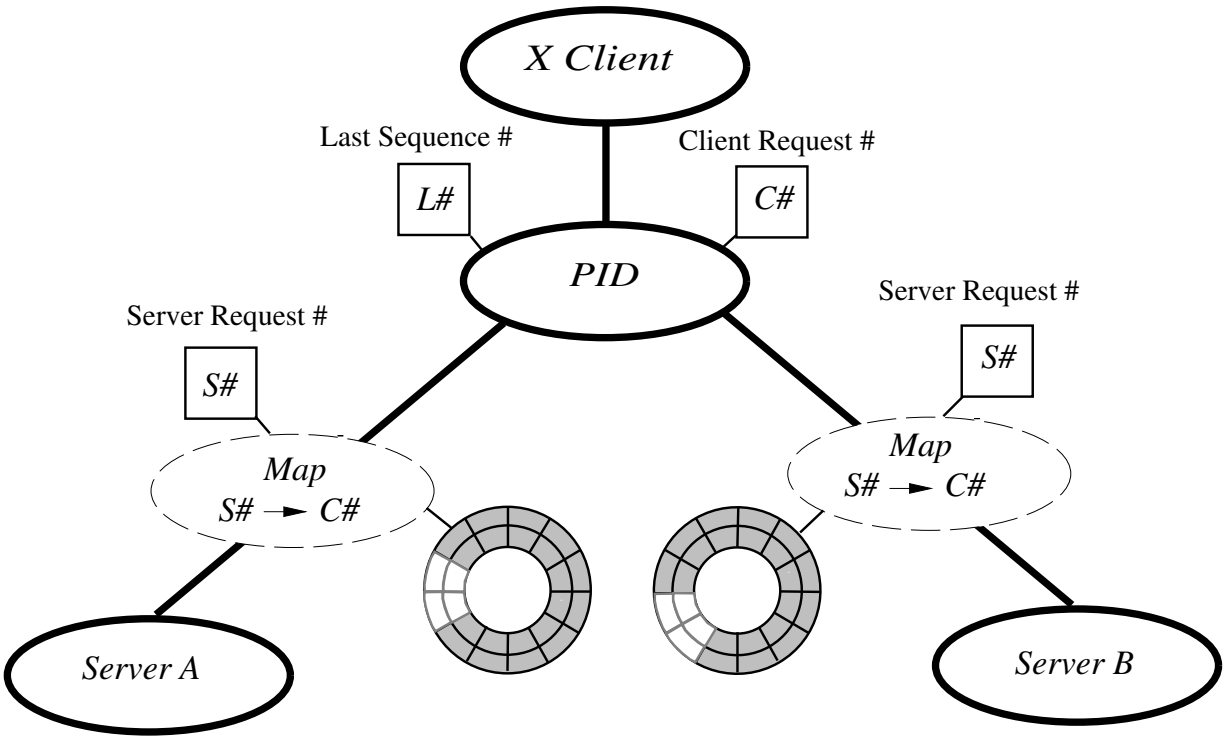
E3 — Events that may be forwarded from all servers. An example of such an event is the `Expose` event.

SEQUENCE NUMBERS

Every reply, event, and error message directed from an X server to a client has a *sequence number* field. The sequence number is a count kept by the X server of the last request successfully processed for the client. Sequence numbers are problematic since, as discussed in the previous section, a PID may not send all requests to every server and thus different servers may have different successful request counts for the same client. Problems occur when, for example, the floor is passed from one participant to another, the client may receive an unexpected sequence number in response to request of type **R2**. In this case the client will likely terminate abnormally with an Xlib error message indicating a "broken sequence number."

To address this problem, we use a sequence number mapping technique that uses a circular queue data structure shown in Figure 2. Upon receiving a request from an X client, the PID increments a variable called *C#* (Client Request Number). The variable *C#* keeps track of the total number of requests issued by the X client. As described in the previous section, the PID may forward a given client request to any subset of the X servers. For each X server, a variable called *S#* (Server Request Number) is used to keep track of the total number of requests sent to it by the PID. Before sending a request to an X server, the variable *S#* is incremented and a pair (*S#*, *C#*) containing the values of *S#* and *C#* is inserted into the circular queue *Q* in the next empty slot following the *Tail* pointer (see Figure 2(b)).

Let α be the sequence number of a message *M* (reply, event, or error) received from one of the X servers. Before forwarding *M* to the PID, the value α is replaced with another value β where β is obtained by moving the *Head* of *Q* to entry (α, β). Note that the arrival of a message *M* from a server may result in freeing some slots from its associated sequence number queue.



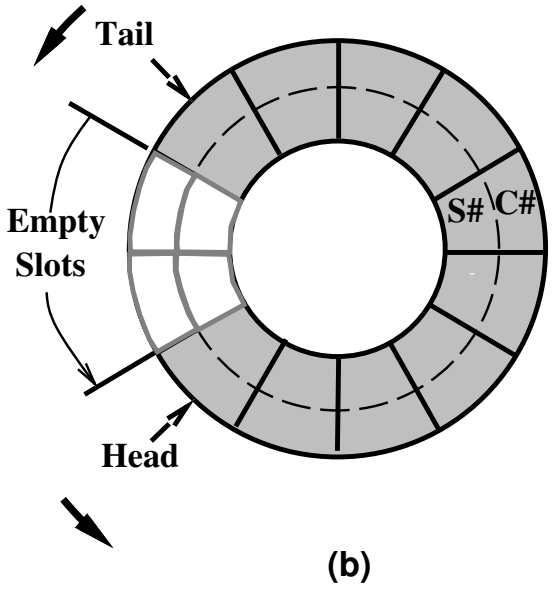


Figure 2: Sequence Number Problem

The PID uses a variable called *L#* (Last Sequence #) to store the sequence number of the last message sent to the X client from all servers. Before forwarding a message *M* to the X client, the PID may further replace the sequence number β with another value γ = MAX(β, *L#*). The value of *L#* is updated by assigning it the value γ. This ensures that

sequence numbers of all messages sent to the client are monotonically increasing. In addition, it allows us to send messages such as the `Expose` events from any X server to an X client at any time (regardless of which participant holds the floor).

An X client may send a large number of requests before any reply, event or error is generated by a server. To avoid overflowing the circular queue, we use the following technique. Let E be number of empty slots in Q. If E falls below a certain threshold T, we artificially generate a request that forces the server to send a message M to the client. The sequence number of M is then used to move the *Head* forward and thus freeing up queue slots.

In XTV, we send a request with an illegal opcode which forces the server to send an error message. (The following error message is not forwarded to the client.) From our experience with XTV, we have found that a queue size of 500 entries and a threshold value T of 100 are adequate for all the X applications we have used.

RESOURCE AND ATOM TRANSLATION

A successful reply from server S_i to the connection request by client C_j contains "general" resource information $G(S_i)$ and "private" resource information $P(S_i, C_j)$. The information $G(S_i)$ is a function of only the server S_i and it is considered general because it is known to all clients connected to S_i. Examples of such information are the *root window ID* and the *default colormap* of server S_i. The information in $P(S_i, C_j)$ is a function of both the server S_i and the client C_j. It is considered private since it is intended for use only by C_j in its subsequent communications with S_i. Examples of such information are the *resource-id-base* and *resource-id-mask.*

Atoms are unique resource IDs used to represent strings. A client creates an atom by sending an `InternAtom` request to a server. The request contains a string t. The server replies by sending the resource ID of the atom corresponding to t. Let $A(S_i, C_j)$ be the list of resource IDs for the atoms created by C_i as a result of sending `InternAtom` requests to S_i.

The set of all resource information: general, private and atoms of server S_i with respect to a client C_j is expressed as:

$$R(S_i, C_j) = G(S_i) \cup P(S_i, C_j) \cup A(S_i, C_j).$$

Let r be either a request from C_j or a reply from S_i. A field in r is said to be *R-dependent* if its value is derived from $R(S_i, C_j)$. Assume S_n is the native server for client C_j and S_f is one of C_j's foreign servers. Each request r from C_j is formulated in terms of the native resource set $R(S_n, C_j)$. If request r is sent as is without any modification to a foreign server S_f, it may not be successfully executed and might generate an error message from S_f. Therefore, we should replace the value v_n of each R-dependent field in r with another value v_f using the following function:

$$v_f = Translate(R(S_n, C_j), v_n, R(S_f, C_j)).$$

The function *Translate* examines the set $R(S_n, C_j)$ looking for v_n and maps it to a corresponding value v_f from the set $R(S_f, C_j)$. Translating v_n to v_f (*i.e.*, translating messages sent from the client to the server) is called "forward" translation. The "reverse" translation of v_f to v_n (*i.e.*, the translation of messages sent from a foreign server to the

client) is described as follows. The value v_f of each R-dependent field in replies, events and errors from foreign server S_f must be replaced with another value v_n using:

$$v_n = Translate(R(S_f, C_j), v_f, R(S_n, C_j)).$$

The function *Translate* examines $R(S_f, C_j)$ looking for v_f and returns the corresponding value v_n from $R(S_n, C_j)$.

COLORS

In the X Window System, applications refer to colors using an index into a *colormap*. The color *cells* in the colormap are dynamically allocated and freed by the X server in response to client requests. The index to a color cell is called its *pixel* value. Clients use the pixel values to refer to stored colors in the colormap. Assume that a client has sent an `AllocColor` request for color c_1 to two different X servers: the native server A and a foreign server B as shown in Figure 3. Server A may assign the color to a cell with pixel value p_1 whereas the server B may assign the same color to a cell with pixel value p_2 where $p_2 \neq p_1$. If we forward the pixel value p_1 back to the client as a reply to its `AllocColor` request, then any subsequent reference to p_1 in any request by the client (*e.g.*, `CreateGC`) must be replaced with the pixel value p_2 before sending it to server B. Without this "pixel translation," the colors appearing on display B will be different from the colors on display A.

Our solution to this problem is similar to the resource and atom translation scheme presented earlier. We maintain a *NativePixelList* of pixel values for the native server and a corresponding $ForeignPixelList_i$ of pixel values for each foreign server i. Before sending a request to foreign server i, we translate each pixel value p in the request with

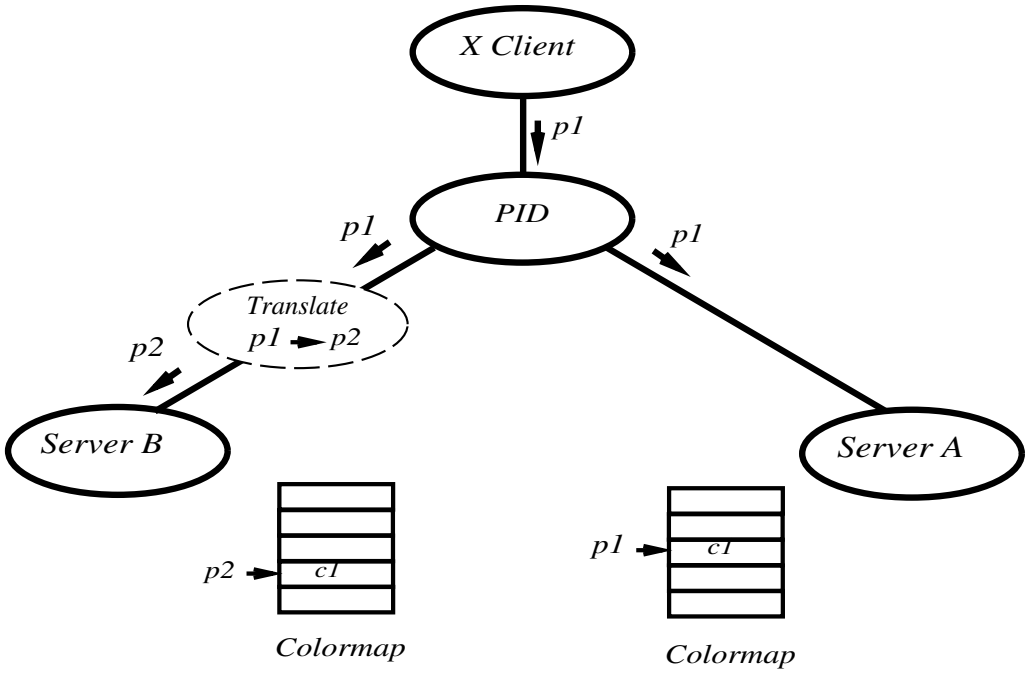


Figure 3: Same color with different pixel values

another pixel value q, where:

$$q = Translate\ (NativePixelList, p, ForeignPixelList_i).$$

The function *Translate* searches the *NativePixelList* for p and returns the corresponding value q from $ForeignPixelList_i$. The pixel lists are updated whenever an `AllocColor`, `AllocNamedColor` or `FreeColor` request is encountered.

Our color mapping approach applies most directly to consoles whose visual type is *PseudoColor*. Consoles of other visual types, most notably of type *TrueColor*, can be handled in a similar manner. (For a console of type TrueColor, pixel values are decomposed into three subfields which index the red, green and blue color maps separately.)

ACCOMMODATING LATECOMERS

To be effective, conferences must be flexible and robust. New participants should be able to join conferences already in progress and participants who become disconnected from a conference should be able to rejoin. Both issues are addressed through a message filtering and archival process that monitors the communication channel between a client and the native X server and records the minimal information necessary to generate an exact replica of a client's interface on a remote display. When either a latecomer joins the conference or a disconnected participant attempts to reconnect to the conference, the information recorded for each client in the conference is played back to the new participant's X server.

The information recorded for each client consists primarily of a list of the resources created by the client and the current attributes of each resource. The key problem here is to identify the minimal set of resources that must be created. For example, an application may create resources, such as a window, for a temporary object such as a pop-up menu or dialog box. The application destroys these resources when a user completes the interaction with the object. The resources corresponding to the window should not be created on the latecomer's X server if the window is no longer part of the application's interface. For this reason, a simple approach to accommodating latecomers such as archiving all messages sent by a shared application to its X server is undesirable. The storage required to record all messages becomes prohibitive as the conference progresses. Moreover, the time required for a latecomer to join a conference will be a function of the conference's duration and hence also becomes prohibitive. Our approach is to selectively catalog messages used to create or modify the attributes of resources currently in use by the clients in the conference.

One complication is that attributes of resources can themselves be resources and hence dependency relations can exist between resources. These relations constrain the process of logging resource creation messages. When a shared application deletes a resource on an X server, one can not delete its representation of the resource if a second resource depends on its presence. We represent dependency relations among resources with a directed graph: the nodes represent resources and the edges represent dependencies between pairs of nodes. We run a mark-and-sweep style garbage-collection algorithm on the dependency graph when resources are deleted, deleting resources only when no other resources depend on their existence. This scheme is described in greater detail in Reference 2.

APPLICATIONS WITH MULTIPLE X SERVER CONNECTIONS

If we assume that each X application establishes only one connection to the X server, then we can easily identify the messages (requests, replies events and errors) belonging to each application. Figure 4(a) shows a PID connected to *n* applications where each application makes a single connection to the X server. Application *i*, first contacts the PID on the PID's *listen socket L*. The connection is accepted on a *private socket* p_i. (See references 9 and 10 for the details of TCP/IP connections.) The PID next contacts the X server on the server's listen socket *L′*. The connection is accepted on a private socket *pi′*. Message traffic between application *i* and the X server then traverses the path of sockets *ai – pi – ai′ – pi′*.

Some X applications, like *framemaker* and *ghostview* establish more than one connection to the X server. In Figure 4(a) if an existing application tries to create a second connection, the PID will not be able to identify the application making the connection. To solve this problem, the PID may create a separate listen socket L_i for each application *i*, $n \geq i \geq 1$, as shown in Figure 4(b). This allows an application to make any number of connections to the X server. The PID will group together the connections belonging to each application. In Figure 4(b), all connections of application *i* are made through the listen socket L_i and thus can be identified as belonging to the same application. The ability to group all connections of the same application is important in sharing the application among several participants. For example, when the floor for application *i* is passed from one participant to another, the input on all its connections, pi_1, pi_2, ..., pi_k, where $k \geq 1$, is passed to the new floor holder. When the application terminates, all of its *k* connections are closed. Unlike the PID, the X server still considers each socket pi'_j as belonging to a separate application and is not able to group all sockets of the same application. For example, in Figure 4(b), the X Server considers the sockets $p1'_1$, $p1'_2$, ..., $p1'_k$, as belonging to *k* different applications.

POPUP MENUS

In some applications (like *idraw* shown in Figure 5) a popup menu is implemented as a window whose parent is the display root window and whose *override-redirect* attribute is set to TRUE [11]. Window managers do not interfere with this type of window (*e.g.*, by putting a frame around it). Since the parent window is the root window, the popup menu is drawn at an absolute position on all displays and thus may appear at an undesired location on the displays of non-floor holders as shown in Figure 5. In Figure 5 the window on the right hand side belongs to the floor-holder. The goal here is to make popup menus appear in the correct position for all participants as shown in Figure 6.

Our solution to this problem is restricted to clients with only one top level window (like idraw). We also assume that the first window created by the application is its top-level window. To detect that a window is a popup menu, the PID examines a `CreateWindow` request (*OpCode* = 1) and checks that the following two conditions are true:

1. The parent window ID is equal to the root window ID.

2. The override-redirect bit in the BITMASK is 1 and thecorresponding value in the LISTofVALUE field is TRUE (see `CreateWindow` packet format in [11]).

Consider the example shown in Figure 7 where participant A has the floor for the application. Let (X_{TA}, Y_{TA}) and (X_{PA}, Y_{PA}) be the coordinates of the upper left corners of the top level window *TA* and the popup menu window *PA* of participant A.

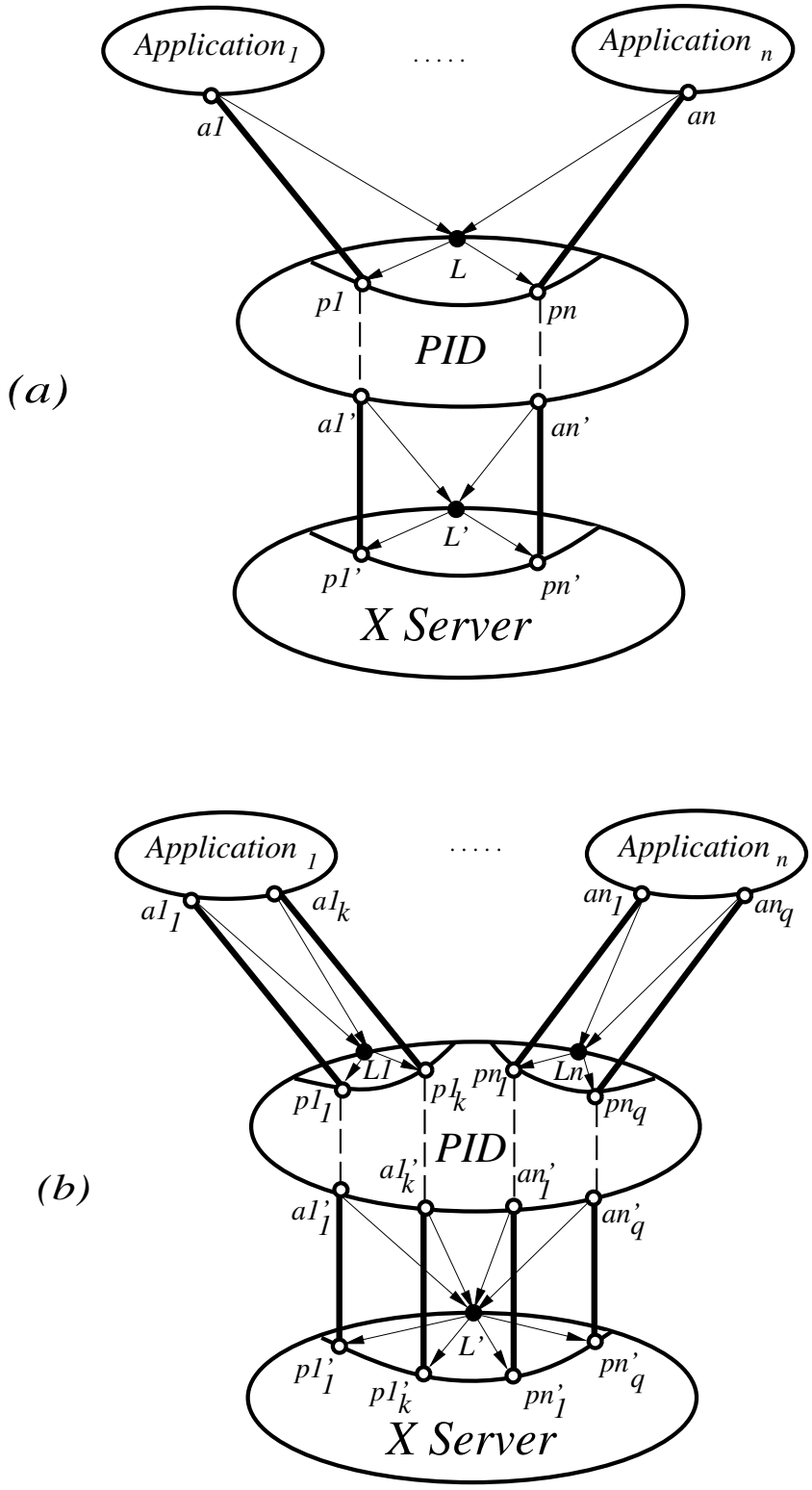


Figure 4: Applications with Multiple Connections

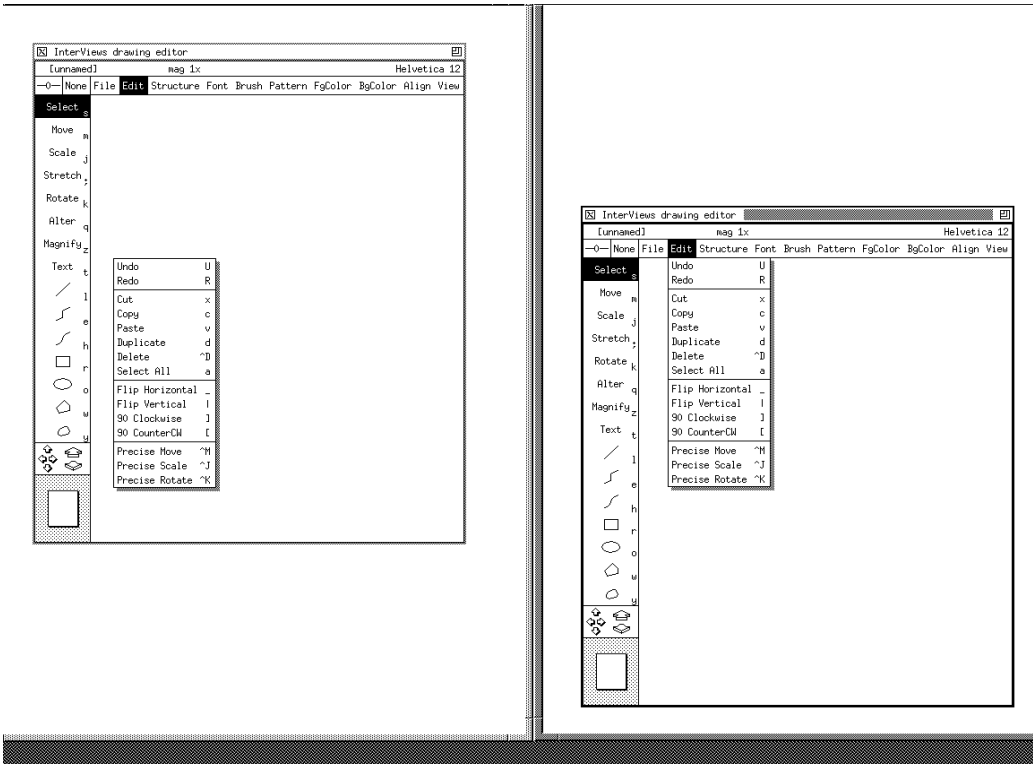


Figure 5: Displaced Popup Menus

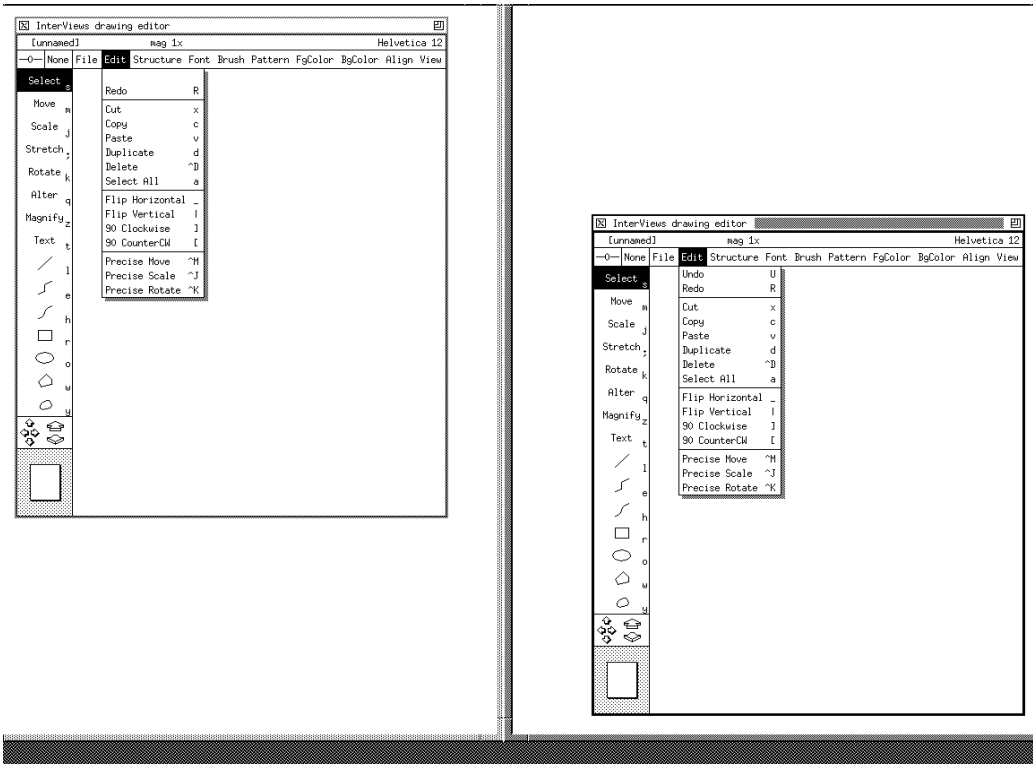


Figure 6: Correct Positions for Popup Menus

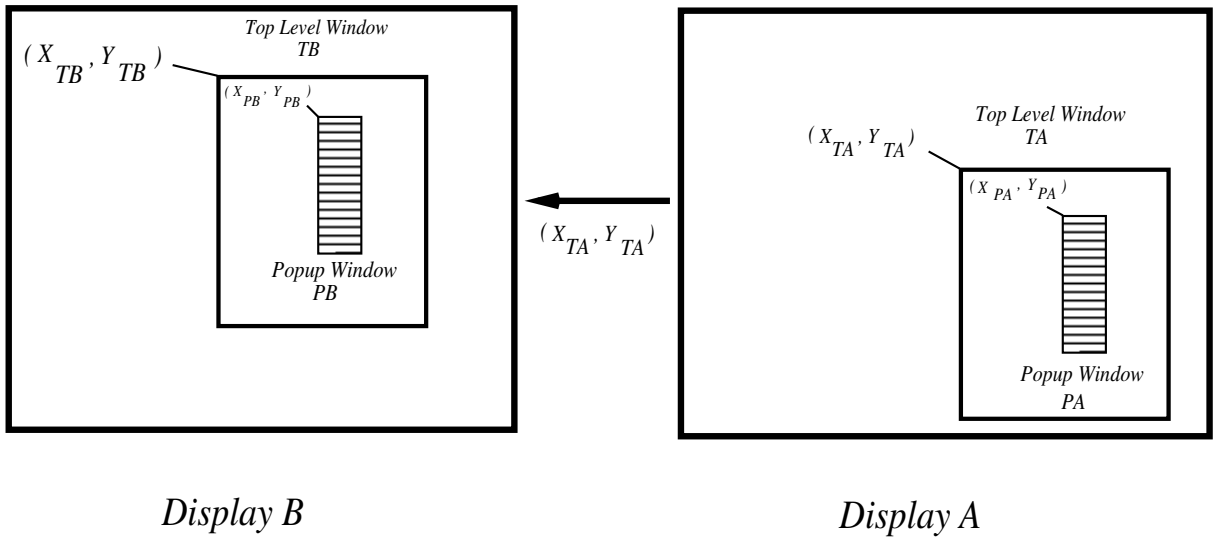


Figure 7: Correcting Popup Menus Positions

The PID will send a message *M* to each participant indicating that the following `CreateWindow` request is for a popup menu. The message *M* will also include the top level window coordinates (X_{TA}, Y_{TA}) of the floor holder. When a non-floor-holder like participant B gets message M, followed by the `CreateWindow` request *R*, it will replace the coordinates (X_{PA}, Y_{PA}) in *R* with (X_{PB}, Y_{PB}) using the following transformations:

$$X_{PB} = X_{PA} + (X_{TA} - X_{TB})$$

and

$$Y_{PB} = Y_{PA} + (Y_{TA} - Y_{TB})$$

where (X_{TB}, Y_{TB}) are the coordinates of the upper left corner of the top level window T_B.

WINDOW RESIZING

If the floor-holder of a client resizes one of the top-level windows using a window manager, it may be desirable to resize the corresponding window on all participant's displays in order to maintain consistent views of the shared clients. Figure 8 shows what happens when the floor-holder P_k of a client *T* resizes a top-level window $W_i(T)$. The X server of P_k generates a `ConfigureNotify` event as a result of resizing the window. The PID forwards this event to the client *T* and sends a message *M* to all participants to resize the corresponding windows on their displays. The message *M* contains the resource ID of $W_i(T)$ and its dimensions (width and height) obtained from the `ConfigureNotify` event. Upon receiving *M*, each participant will resize window $W_i(T)$ using the Xlib function `XResizeWindow` [11].

HANDLING EXTENSION REQUESTS AND EVENTS

There are many requests and events that are not part of the core X protocol. If the exact specification of these extension requests (*OpCode* > 127) and events (*Opcode* > 34) are known, they can likely be handled in using the techniques described in the previous

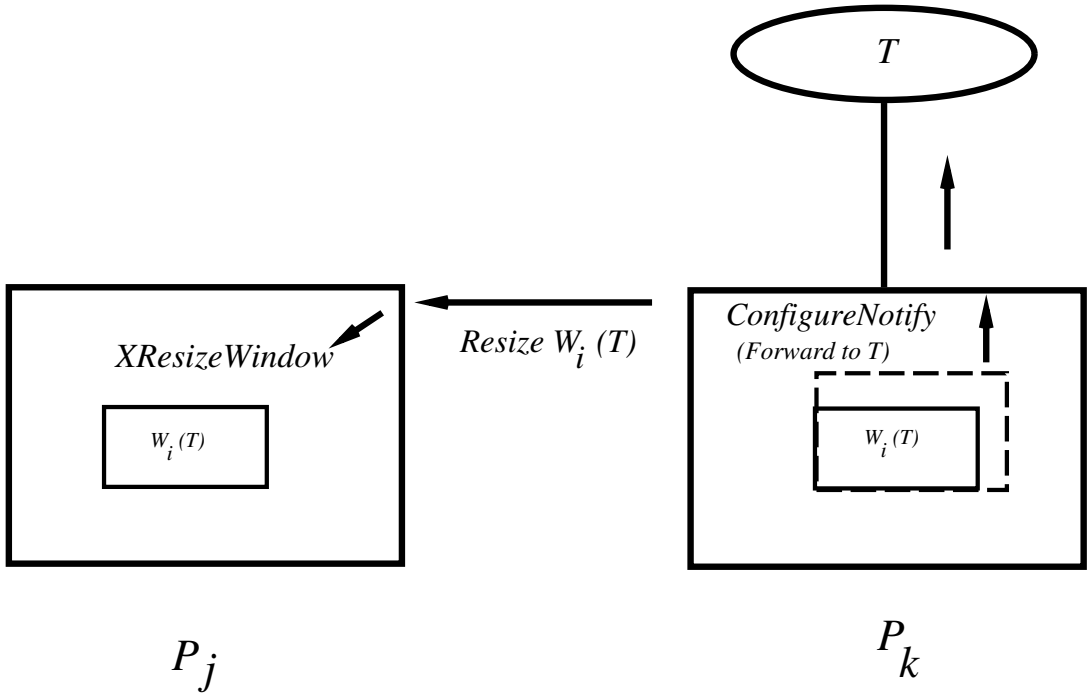


Figure 8: Window Resizing

sections. However, if we do not know the exact message format of an extension request, we can deal with it in two ways:

1. Send the extension requests only to the native display and forward the extension replies and events from the native server to the client. In this case, the foreign displays may have different views from that of the native display. For example, using this method for *xcalc*, the native display might have oval buttons while the foreign displays has rectangular buttons.

2. Discard the extension requests and do not forward them to any display. In this case all displays will have identical views. From example, if we follow this approach in *xcalc*, all displays will have rectangular buttons. Similarly, we may ignore the extension events and not forward them to the client. However, using this method may cause some clients not to function properly and terminate abnormally.

SUMMARY AND CONCLUSIONS

Many computer conferencing systems based on the X Window System have recently emerged. While these systems hold the promise for fostering collaboration among groups of geographically separated individuals, they are, at present, difficult to build. We have described the salient problems that face the designers of X-based shared window systems. In each case we have suggested a solution or outlined principles that should guide the implementation of any solution.

Much work remains. Sustaining conferences among participants with workstations and X servers with widely varying capabilities remains a serious problem. For example, servers that are heterogenious to the extent that they support different keycodes, visuals, or font

databases, cannot be seamlessly supported by our shared window system if applications use these features. Nonetheless, despite the lack of direct support for conferencing in the definition of the X client/server protocol, and despite the ad hoc nature of some of our proposed solutions, it is possible to construct a conferencing system that accommodates the majority of commonly used applications such as text editors, drawing programs, document previewers, and program debuggers with our proposed solutions.

ACKNOWLEDGEMENTS

This work was supported by the National Science Foundation (Grant # IRI-9015443), Office of Naval Research (Contract # N00014-86-K-0680), the IBM Corporation, and the Digital Equipment Corporation.

REFERENCES

[1] H. M. Abdel-Wahab and M. A. Feit, "XTV: A Framework for Sharing X Window Clients in Remote Synchronous Collaboration," *Proceedings, IEEE Conference on Communications Software: Communications for Distributed Applications & Systems*, Chapel Hill, North Carolina, 159-167 (April 1991).

[2] G. Chung, K. Jeffay and H. Abdel-Wahab, "Accommodating late-comers in shared window systems," *IEEE Computer*, Vol. 26, No. 1, pp. 72-74, (January 1993).

[3] J. R. Ensor, S. R. Ahuja, D. N. Horn and S. E. Lucco, "The Rapport Multimedia Conferencing System - A Software Overview," *Proceedings, IEEE Conference on Computer Workstations*, Santa Clara, 52-58 (March 1988).

[4] J. C. Lauwers and K. A. Lantz, "Collaboration Awareness in Support of Collaboration Transparency: Requirements for the Next Generation of Shared Window Systems," *Proceedings, Conference on Human Factors in Computer Systems*, ACM (April 1990).

[5] K. Srinivas, Y. V. Reddy, L. Chang, A. Babadi, S. Kamana, Z. Dai and V. Kumar, "MONET: A Multimedia Conferencing System for Colocating People and Programs," *Proceedings of the CALS & CE Third National Conference on Concurrent Engineering*, Washington, D.C., 433-441 (1991).

[6] K. Watabe, S. Sakata, K. Maeno, H. Fukuoka and T. Ohmori, "Distributed Multiparty Desktop Conferencing System: MERMAID," *Proceedings, CSCW 90 Conference on Computer-Supported Cooperative Work*, (October 1990).

[7] J. F. Patterson, "The Good, the Bad, and the Ugly of Window Sharing in X," *Proceedings, 4th Annual X Technical Conference*, (January 1990).

[8] A. Nye, *X Protocol Reference Manual for Version 11*, Volume 0, O'Reilly & associates, Inc., Sebastopol, CA (1989).

[9] D. E. Comer and D. L. Stevens, *Internetworking with TCP/IP Volume III: Client-Server Programming and Applications: BSD Socket Version*, Prentice-Hall, (1993).

[10] W. R. Stevens, *Unix Network Programming*, Prentice-Hall, (1990).

[11] A. Nye, *Xlib Programming Manual for Version 11*, Volume 1, O'Reilly & associates, Inc., Sebastopol, CA (1989).