Dynamic participation in a computer-based conferencing system

Goopeel Chung*, Kevin Jeffay* and Hussein Abdel-Wahab†

This paper deals with the problem of allowing a participant to dynamically join and leave a computer-based conference that is already in progress. A conference is a synchronous collaboration session where participants at remote locations are cooperating through identical copies of windows generated by applications shared by all conference participants. In this paper, an efficient solution to the problem of accommodating a latecomer is found by recording the modifications to the window system's state implied in the series of commands generated by the applications, and later imposing these state modifications on a latecomer's window system.

Keywords: computer-supported cooperative work, computer-based conferencing, distributed systems, X protocol, shared windows

In conventional (i.e. non-computer-based) conferences, the cast of people interacting may change as the conference proceeds. An initial group starts the conference, others arrive late, and some may leave early. Similar behaviour can be expected to occur during a conference using networked computers. For example, one or more persons may start a conference to work on a program. Each conferee would have the same view of the program displayed on her workstation. At some point they may encounter a problem debugging a piece of program. They can ask a 'guru' for help by allowing her to dynamically join their conference even while the guru is in a remotely located office.

Therefore, it is very important that conference systems provide facilities to accommodate these kinds of spontaneous interactions in a conference. Specifically, allowing a latecomer to join an existing conference and enabling the new conferee to provide input to the conference is considered to be a very important feature of any conference system. The new conferee should be able to share windows that the shared applications create. Specifically, she should be able to see the output of the applications and provide input to the applications through the window system.

While special-purpose ('collaboration-aware') applications[1–4] provide the functionalities to accommodate dynamic joining, single-user ('collaboration-transparent') applications[5–8] are not designed to provide these functions. It is the conference system itself that should provide them.

This paper deals with an efficient method to provide this functionality in a shared window system; the ability to enable a latecomer to dynamically join a conference and share applications used in the conference.

Shared window systems[9–12] consist of application processes and window systems. Connecting all these processes in the centre is an additional process called a conference agent. The conference agent is responsible for distributing output from applications to window systems, and for relaying input from window systems to associated applications. By sharing the output from applications, the users working at workstations can share the same copy of windows associated with the applications. It is also possible for the users to provide input to the applications.

Applications usually set up their user interface environments by creating resources they need to interact with the users. For example, they create windows to draw on, or to take input from the users. Window systems generally interact with several applications simultaneously. Therefore, the resources created by multiple applications and their attributes form a certain state the window systems maintain. Applications can be considered to change the state of the window systems by

*Department of Computer Science, University of North Carolina at Chapel Hill, Sitterson Hall, Chapel Hill, NC 27599, USA
†Department of Computer Science, Old Dominion University, Norfolk, VA 23529, USA
Paper received: 22 November 1991; revised paper received 12 December 1992

creating resources and dynamically modifying the attributes of these resources.

These state modifications are expressed in the output generated by each application. If these state modifications can be projected to a new window system for a latecomer, the new participant can share the applications in the conference. Naturally, the best place to monitor these modifications is in the conference agent, which handles all traffic between application and window systems.

This paper describes an efficient method implemented in the conference agent process to record the state modifications made by the applications to the window systems, and to project the modifications to a latecomer's window system.

In the next section, some background information is provided on XTV (X Teleconferencing and Viewing), on which our solution is based. The section following discusses how we solved the problem of accommodating a latecomer. Some performance figures are then presented to demonstrate the efficiency of our solution, and finally, our conclusions are drawn.

OVERVIEW OF XTV

We begin by describing some background to the X Window System, the window system used by XTV. Then we describe XTV, on which the feature to accommodate latecomers is based.

X Window System

The X Window System is a window-based User Interface Management System (UIMS) providing capabilities to easily create graphical interfaces for distributed applications independent of the architectures on which the applications will be running[13]. X Window System applications such as terminal emulators and drawing programs are widely available. Because the X Window System is standardized and is largely independent of the host architecture, applications can be compiled and run on any machine that supports X.

Client/server model

The X Window System is built on a familiar distributed system model: the client/server model (see *Figure 1*).

The X Window System is a server that accepts requests to manipulate the display on the computer's console while reading input from the console's keyboard and mouse devices. An X server's clients are user applications such as terminal emulators and editors. X clients send request messages to the server asking it to perform operations such as create a window, draw a line on a window or destroy a window. All output on the display is generated by the server in response to requests from clients. Similarly, all input to X clients is provided

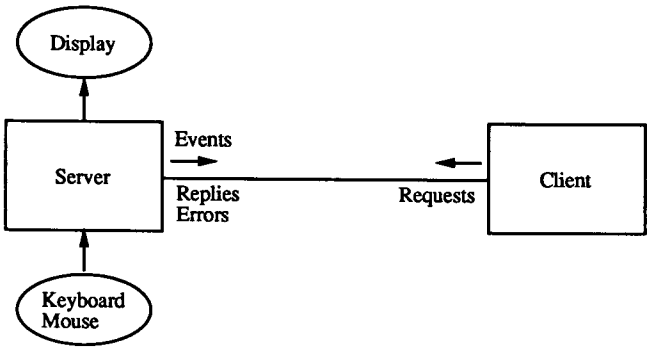

Figure 1 Client/server model

by the server. The user interacts directly with the server using keyboard and mouse.

Communication between an X client and an X server is via message passing. There are four classes of messages exchanged between the server and the client:

- *Request messages:* are sent from a client to a server. X servers handle a wide variety of requests including requests that affect the display (e.g. drawing on a window), requests that affect internal data structures (e.g. changing the keyboard mapping), and requests that return data (e.g. returning the image contents in a window).
- *Reply messages:* are sent from a server to a client. These messages contain information requested by clients in previous request messages to the server.
- *Event messages:* are sent from a server to a client whenever there is user input from the keyboard or mouse that is germane to the client.
- *Error messages:* sent from a server to a client tell a client that a previous request was invalid, e.g. the client specified a window that does not exist in a request for drawing.

Resources

The X application programming model presents six basic abstractions: *window, cursor, graphics context, pixmap, colourmap* and *font*. Windows and pixmaps are referred to collectively as *drawables*. Instances of these abstractions are referred to as *resources* in X. Resources are created, manipulated and destroyed by the server in response to requests by clients. The following is a brief description of these resources.

A Window is like a canvas on which the client may draw objects by sending pertinent request messages to the server.

Cursors are small pointers that move on the display according to movement of the mouse.

Requests to draw graphics such as dots, lines, texts or images are called graphics requests. Much information is required to fully specify how a particular graphic should be drawn. For example, when drawing a line, we may want to specify its colour, its width or the style (e.g. solid or dashed). To simplify the specification process,

X provides a Graphics Context (GC), a set of values for many of the variables. The appearance of everything that is drawn within a drawable is controlled by a GC that is specified with each graphics request.

Pixmaps are like windows in that a client can draw on them using a set of graphics requests similar to those used for drawing on a window. However, pixmaps themselves are not visible. The contents of a pixmap can be seen only when it is copied into a window. Pixmaps are used for several purposes. For example, a pixmap can be used as the source or mask of a cursor. The source of a cursor defines the cursor's pattern, and the mask is used to confine the pattern to a certain shape.

Each pixel in a window is associated with a pixel value that represents the colour of the pixel. This pixel value is an index into a list of entries called *colourcells*. A colourcell holds three values; one for each RGB (Red, Green, Blue) value. The list of colourcells constitutes a *colourmap*.

Fonts are usually used as an attribute of graphics contexts. Each text-writing graphics request always specifies a graphics context with appropriate font set in it.

Most of these resources are created by clients sending appropriate messages to the server. Some resources are created by the server by default (e.g. the default colourmap and the root window). All the resources are referred to by resource IDs (unique integers).

XTV

Most X applications (clients) have been written to interact with a single user. When such an application is used, there is a connection between an X server and a client through which input and output messages are exchanged. Most shared window systems are built by inserting a process in the connection between a server and a client. This process intercepts all message traffic and distributes it properly to make window sharing possible.

XTV is a distributed system for allowing multiple remote users to view and interact with X applications in real-time[14]. XTV looks like a client from the remote servers' points of view, and like an X server from the shared X applications' points of view.

In XTV (see *Figure 2*) a process – called the *packet switch process* (PSP) – is responsible for distributing the output of the shared applications to all of the remote servers. XTV refers to shared applications as 'tools'. The packet switch process opens a connection to the server for its own interface, and one for each shared application.

The packet switch process cannot simply distribute each request message to the remote servers without any modification to it. Many request messages contain resource IDs that refer to the resources on the local server. The ID for a resource on the local server may be

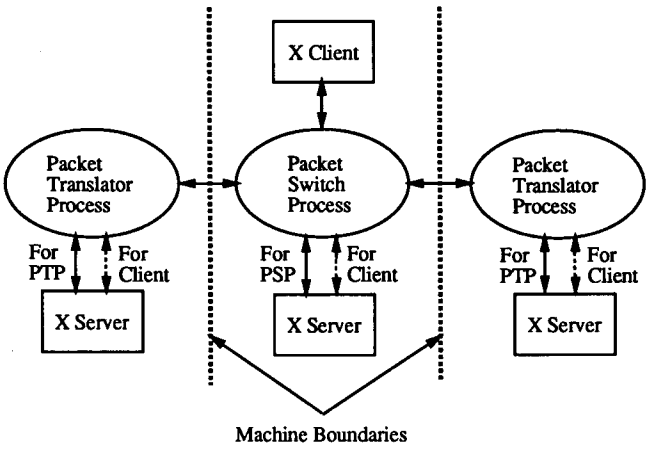

Figure 2 XTV architecture

either invalid or refer to a different resource on a remote server. The resource IDs in a request message destined for a remote server should be corrected to refer to the corresponding resource on the remote server. Another process, called the *packet translator process* (PTP), is run on every remote workstation. The packet switch process sends the request messages received from the X applications to these translator processes rather than directly to the servers. The packet translator process modifies the messages so that they contain correct resource IDs, and then sends them to the server. The packet translator process also opens separate connections for its own interface and for each shared application.

The user Interface to XTV is shown in *Figures 3* and

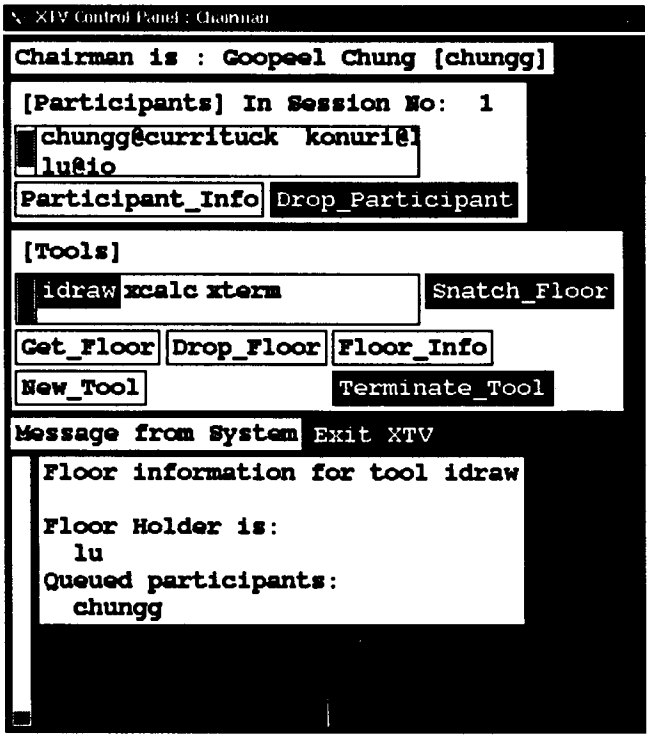

Figure 3 XTV user interface for the chairman

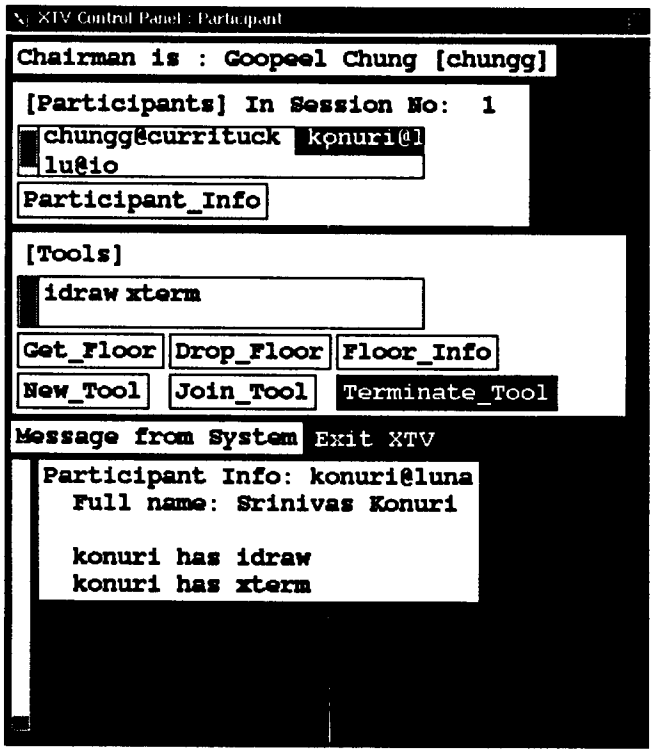

Figure 4 XTV user interface for a participant

4. A participant who starts a new XTV session is designated as the *chairman* of that session. A panel such as that shown in *Figure 3* appears in the chairman's display. The panel has three main areas:

1. **Participants area:** the scrollbar window of this area contains a list of all current participants of the session. Detailed information about a selected participant is obtained by pressing on the `Participant_Info` button; a participant is selected by clicking into her entry in the list. The chairman has the power to dismiss any participant, for whatever reason, by pressing on the `Drop_Participant` button.
2. **Tools area:** the scrollbar window of this area contains a list of the active tools in the session. Tools are added to the session using the `New_Tool` button, and are deleted using the `Terminate_Tool` button. Two buttons, `Get_Floor` and `Drop_Floor`, are provided to control the floor (i.e. the ability to provide keyboard and mouse input) of a selected tool. The chairman may grab the floor of a selected tool using the `Snatch_Floor` button. Detailed information about the floor status of a selected tool is obtained using the `Floor_Info` button.
3. **Message area:** the scrollbar window of this area displays all messages produced by XTV, including those messages produced by the participant and floor information buttons. To terminate a session, the chairman uses the `Exit XTV` button, which cleans up all active tools and the associated windows.

A participant may join an ongoing session at any time by simply knowing the session number and where the chairman is located. A participant panel such as that shown in *Figure 4* appears in the participant's workstation. It is similar to the chairman's panel of *Figure 3* with the following differences:

1. There are no `Drop_Participant` or `Snatch_Floor` buttons in the participant's panel. Because of the disruptive nature of these two buttons, they are available only to the chairman.
2. Unlike the chairman, a participant is provided with a `Join_Tool` button in the tools area of the panel. In the current implementation of XTV, the chairman is assumed to be an active user of all created tools, while a participant may dynamically join or drop a tool at any time. Thus the set of tools listed on a participant's panel is a subset of the set of tools listed on the chairman's panel.
3. The semantics of the `Exit XTV` button is different from that of the chairman. Here it means that a participant is leaving the session; the chairman and other participants may continue the session unaffected by her departure.

ACCOMMODATING LATECOMERS

Overview

Consider the case of a user who is late for the conference that is in progress; that is, all the shared applications have been in use for quite some time. Although the latecomer can join the conference by connecting to the packet switch process, the set of applications cannot be shared with the latecomer. This is because the X server on the latecomer's workstation does not have any of the resources created by the shared applications, and hence request messages from the applications will make no sense to the new server. In other words, the latecomer's server is not in a state to receive requests from the shared applications. The problem is to change the state of the new server so that a late arriving participant can share the applications that have been in use. Currently, there is no direct method to capture the state of one server and impose it on a new server. The specification of the X protocol does not provide a way to do it. Therefore, to change the state of the new server, we must depend on the requests that have been sent by the clients; that is, we can get the new server into the appropriate state by applying the changes implied by the sequence of requests that have been sent to the original server.

In addition to distribution and translation of client request messages, XTV – more specifically, the packet switch process – must now maintain a record of the

changes made to the server state by each client. A very simple solution to this problem is to keep a history log of all requests that came from the clients, and later replay the history to a new server (i.e. send each request to the new server) when a latecomer arrives. However, this can be very inefficient, since storing all the requests consumes a large amount of memory space. For example, a client like idraw[15], a MacDraw-like X application, sends about 300,000 bytes of requests for only 4 minutes after it starts interacting with a server (see *Figure 12*). Moreover, it will take proportionally longer time for a latecomer to catch up on the conference, depending upon how late she joins the conference.

The following section describes how we can maintain the changes made to the server state more efficiently.

Recording modifications to resources

Our approach is to catalogue changes a client can make to the server state. A client may change the server state as follows:

- create private resources (e.g. a client can create a set of windows for its use),
- change attributes of resources (note that the client can change the attributes of resources it did not create itself. For example, a client can change the colour of a window background (a private resource), or it can allocate more colourcells in the default colourmap (a non-private resource)), or
- change other miscellaneous environment properties such as the list of machines allowed to connect to the server.

Modifications of the server state can be recorded by maintaining a list of the resources (private and non-private) that are handled by the client and ensuring that the attributes of these resources are kept up-to-date. When a latecomer joins the conference, the recorded modifications can be applied to the new server of the latecomer. Private resources will be created with their attributes assigned current values. Resources that the client did not create but has modified must have their appropriate attributes updated. The miscellaneous environment properties can be also handled on a per-property basis.

This approach of concentrating on the modifications made to resources guarantees that a minimal set of information is kept about the changes made by the client to the server state. For example, many resources are created and then later destroyed by the client. Consider pop-up menus. Pop-up menus are implemented as temporary windows. They are created and then destroyed after the user has selected an item in the menu. When the window is deleted, we can delete the data structures holding the information about these menu windows. Also, by keeping the up-to-date values for attributes of resources, we can save a lot of memory space that may otherwise be wasted for saving requests to change attributes of resources. This is because it is only the current attribute values of resources that count, and not the history of the attribute values since the resource's creation. Only the current values will be used when creating resources on a latecomer's server.

Images in drawables

Some further optimizations on this scheme are possible. The image attribute of a drawable (windows and pixmaps) need not be recorded. This attribute is changed by graphics requests sent by the client. One would expect that the graphics requests need to be recorded and replayed to a new server in chronological order. However, it is the case that it is possible to ignore all graphics requests. This is because X servers do not guarantee that the contents of a window will be preserved when portions of the window become obscured. When portions of a window become visible, an X server will send *expose* event messages to the client that created the window. Each expose event specifies a rectangular region inside a window that has become visible. On receiving these event messages, a client is expected to send the appropriate graphics requests to draw an up-to-date image on its window. Given that the client will refresh the contents of the whole window when expose events are generated, graphics requests for windows need not be recorded. This is because the first time a window is displayed for the latecomer, the X server on the latecomer's workstation will send expose events for the client.

The image in a pixmap cannot be seen unless it is copied into a window. Hence, there is no concept of an expose event for pixmaps. There is, however, a way to make the contents of a pixmap up-to-date other than recording all of the graphics requests for the pixmap. The client can obtain the image in a pixmap through a request message called *GetImage*. When a new participant joins the conference, the packet switch process will send this request to its local server to get the contents of pixmaps used by the shared applications. The packet switch can then send a request called *PutImage* to the new server. The *PutImage* request will put the acquired image into the appropriate pixmap on the latecomer's server.

Ignoring the graphics requests greatly reduces the memory requirements of XTV (see below). This is because X clients generally go through two phases in their lifetime: a set-up phase and an interaction phase. In the set-up phase, the majority of a client's resources are typically created. In the interaction phase, the majority of requests are for graphics operations to draw objects on the client windows. Since the interaction phase is generally much longer, the majority of messages sent from clients to server are requests for graphics operations.

Maintaining resources

Other than the image attributes of drawables, we have to keep the attributes of resources up-to-date. Whenever a new resource is created by a client, data structures within XTV are created to record the attributes of the resource. When a client changes the attributes of a resource, XTV will modify the data structure corresponding to the resource. When a client sends a request to free (destroy) the resource, XTV deletes its data structures for the resource. When a client first modifies attributes of a resource that it did not create, similar data structures are also created to keep the contents of changes.

Dependency relationships among resources

A problem arises if we naively apply the method in the previous section for all resources. For example, in idraw, a cursor is created using separate pixmaps for its source and mask. The server will record the shape of the cursor in its internal data structures. Unless these pixmaps will be explicitly referred to later on, idraw can free these resources. If idraw does free those two pixmaps after the creation of the cursor, the data structures in XTV for these pixmaps would be deleted. But this should not be done because for XTV to later create the cursor on a new participant's local server, it has to create the very two pixmaps it no longer has any information on. The problem is that we need some method to prevent information on resources that are explicitly freed by a client from being thrown away when there are other resources that require this information.

Creating dependency relationships To represent the relationships among resources, we define the dependency relation $\triangleright$. A resource A depends on a resource B, written as $A \triangleright B$, if the resource A has as one of its attribute values the resource B. For example, the *CreateCursor* request creates a dependency between a cursor C and two pixmaps P_1 (a source pixmap) and P_2 (a mask pixmap). We can represent dependency relation $\triangleright$ using a directed graph, called a dependency graph, such as in *Figure 5*. A node represents a resource and an edge represents the dependency relationship with the interpretation that the node on the tail of the edge depends on the node on the head of the edge. XTV constructs such dependency relations as requests are encountered. For example, *Figure 6* shows the dependency graphs that result from the create graphics context request*.

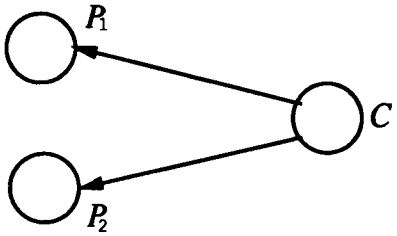

Figure 5 CreateCursor request

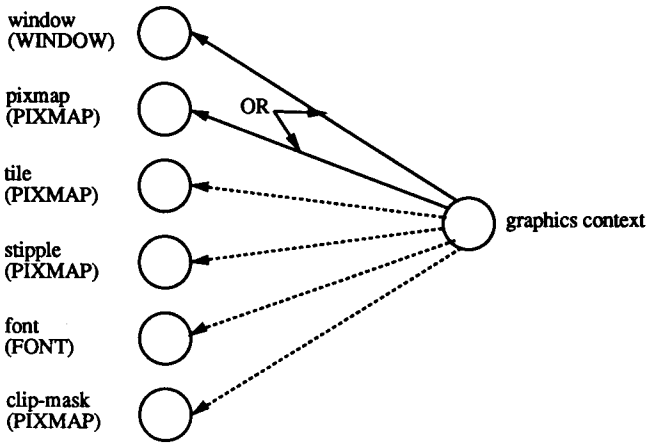

Figure 6 CreateGC (create a graphics context)

Dashed edges represent optional attributes whose values need not be, and typically are not, set at the time of resource creation. These attributes can be set after the resource is created using requests such as *ChangeWindowAttributes* or *ChangeGC*. Adjacent edges annotated with the label OR cannot both appear in any actual instance of the graph.

The dependency relations among the six resource types can be summarized with an Entity-Relationship-like diagram[19], as shown in *Figure 7*. Note that some edges have been combined for simplicity into a single edge with a label representing the number of original edges.

For any shared application, XTV will create an instance of the E-R diagram in *Figure 7*.

Deleting dependency relationships When a request to free a resource R is encountered, XTV must check to see if any other resource depends on R. Only when no other

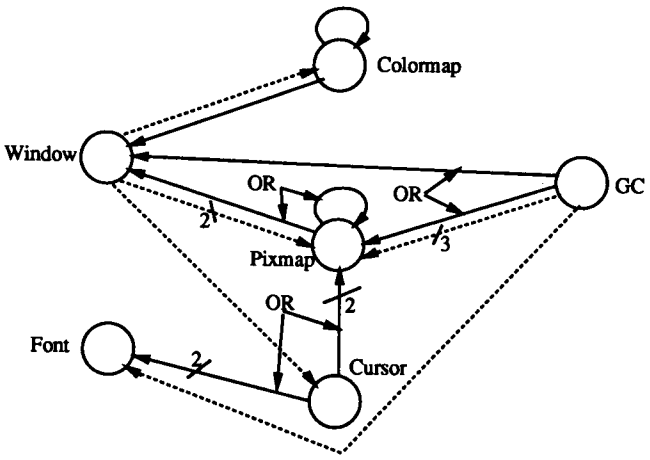

Figure 7 E-R diagram of dependency relationships

*Details of the exact semantics of all requests can be found elsewhere[16–18].

resource depends on R can R's information be deleted from XTV's internal data structures. If there is another resource that depends on R, R's data structure should be marked as freed, but not actually deleted. The data structures of freed resources will be deleted later when all their dependency relations have been removed. Therefore, when a resource A removes a dependency on another resource B, XTV has to make sure that information on B is deleted if B is marked as freed and it has no other dependencies. *Figure 8* shows the recursive depth-first traversal algorithm that is applied to the node corresponding to the resource that has just been freed by the client. Note that the algorithm is applied after the associated node is first marked as freed.

The algorithm first deletes all the optional edges leaving the current node. This can be done unconditionally because if a resource has optional attributes, then any other resource that uses this resource cannot depend on these attributes being correctly set. Next, all essential edges are deleted if the node has no edge coming into it (i.e. no resource depends on the resource corresponding to this node).

The DeleteNode algorithm works only with acyclic graphs. However, an examination of our E-R diagram (*Figure 7*) reveals that dependency graphs may contain cycles. The following example illustrates how the algorithm can fail when the graph contains a cycle. Consider the graph shown in *Figure 9*. White nodes represent non-freed nodes and black nodes represent freed nodes. If the client sends a request to free the remaining white node, all the nodes in the graph can be deleted. However, if we apply the DeleteNode algorithm, it will terminate without taking any action because each node has an incoming edge. To enhance the algorithm to deal with cycles, we first observe that all cycles in *Figure 7* must contain a window node. We introduce a second procedure that is used to delete window nodes when a *DestroyWindow* or *DestroySubwindows* request is encountered. The algorithm shown in *Figure 10* called WindowSpecial works by breaking the cycle and then applying DeleteNode to a node that the window was dependent upon. The recursive nature of DeleteNode will ensure all free nodes in the cycle are deleted. Note that we do not mark the window node as free before we apply the WindowSpecial algorithm.

In summary, the problem of maintaining the server state for a client reduces to a graph maintenance problem requiring operations to create a new node, change some attributes of a node, add dependency relations to the graph, delete a node and delete a dependency relation.

```
DeleteNode (node)

IF node is marked as freed THEN
  FOR each optional edge leaving node
    Delete the edge;
    Apply DeleteNode to the node at the head of the
    edge;
  END FOR
  IF node has no incoming edges THEN
    FOR each essential edge leaving node
      Delete the edge;
      Apply DeleteNode to the node at the head of the
      edge;
    END FOR
    Remove node
  END IF
END IF
```

Figure 8 DeleteNode algorithm

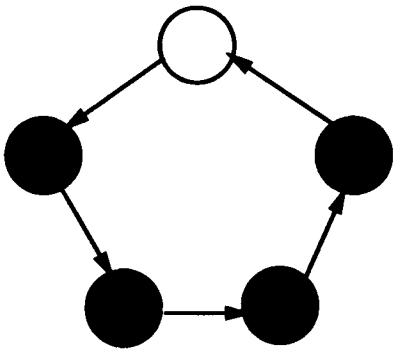

Figure 9 A cycle

```
WindowSpecial (node)

FOR each edge going out of node
  /*These will all be optional edges.*/
  Delete the edge;
  Apply DeleteNode to the node at the head of the
  edge
END FOR
Mark node as freed;
IF node has no incoming edges THEN
    Remove node;
END IF
```

Figure 10 WindowSpecial algorithm

Modifying the state of a latecomer's server

When a new participant joins a conference that is already in progress, we must set up the proper environment for each shared application in the conference on the server of the latecomer. The goal is to ensure that (1) future output requests from a shared application have the same effect on the latecomer's display as on the displays of participants that were already in the conference, and that (2) any messages from the new participant's server will be delivered to the shared application without errors. In terms of X, the packet switch process must generate a series of requests to the latecomer's server in such a manner that all of the resources that have been created for each shared application on the servers of current participants are also created on the latecomer's server. Furthermore, all attributes of each resource must be set correctly.

All of the resources should be created without violating the dependency relations among them. The dependency relations may be violated because of the

existence of cycles. The key to the solution of this problem lies in the characteristics of the optional edges. The attributes associated with the optional edges do not have to be set at the time of the resource creation. (Recall they can be set at a later time using requests like *ChangeWindowAttributes* or *ChangeGC*.) Therefore, we ignore optional edges going out of window nodes for now. Without these edges, we have a directed acyclic graph. We now can generate requests to create each resource without violating the dependency relations by traversing the graph in topological order.

After creating all messages resulting from traversing the dependency graph, each message is sent in order to the packet translator process on the latecomer's machine.

Note that a shared application can generate requests while the new participant's server is being brought up-to-date. These requests are appended to a separate message queue and will be sent after all the messages relating to joining the conference have been sent to the latecomer's packet translator process. Included in this message queue will be the graphics requests required to make the displayed image current (see Chung[16] for implementation details).

PERFORMANCE

Our system for accommodating latecomers contains approximately 5500 lines of code out of about 20,000 lines of code for the first release of XTV.

In this section, we discuss the performance of our method for recording the requests sent by each client that change the state of the server.

Speed

Whenever a request arrives from the client, a function *HandleIncomingClientPacket* is invoked from the packet switch process. This function is responsible for distributing the request message to all packet translator processes and recording the modifications to the server state. A function *ArchivePacket* is invoked from within *HandleIncomingClientPacket* to record the modifications. *Figure 11* shows the percentage of the time spent in *ArchivePacket* to the time spent in *HandleIncomingClientPacket* for idraw – a client that generates a large number of graphics requests. A UNIX system call *getrusage* was used to measure the elapsed times. *getrusage* system call returns information on how much time the process used since it started. The call was made at the beginning (*A*) and the end (*D*) of *HandleIncomingClientPacket*, and at the beginning (*B*) and the end (*C*) of *ArchivePacket*. The difference between times measured at *A* and *D*, and that between times measured at *B* and *C* were calculated to find how long each function took to execute. The graph in *Figure 11* shows the overhead for a conference with one and four

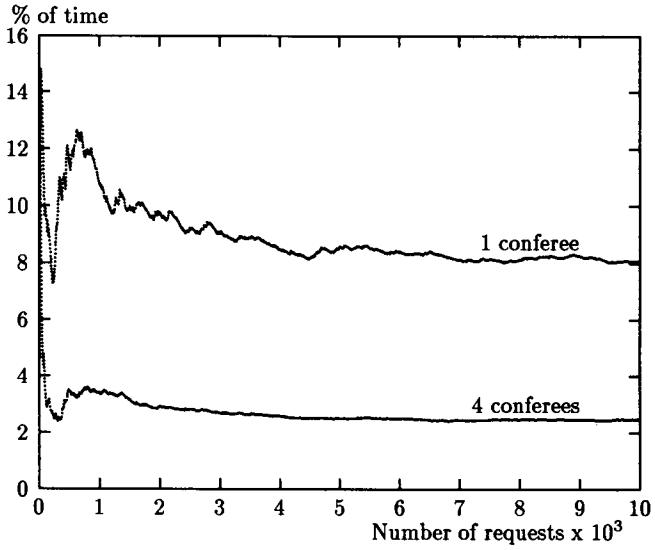

Figure 11 Percentage of time spent maintaining resource state information *versus* number of client requests in idraw

conferees. The observation interval for idraw was approximately 3 minutes 30 seconds for one conferee and 5 minutes for four conferees. Because the time required to record resource state information is independent of the number of conferees, the proportional cost of the recording function decreases as the number of conferees increases. Note that the cost of recording is the highest at the initial stage (the set-up phase) of client execution. This is because most of the resources are created at this time and the resource recording function has to do time-consuming operations such as initializing data structures, allocating memory space for new resources, and searching for resources to make dependency relations. In the case of idraw (*Figure 11*), the client first sends some query requests to the server to get information on the server. These are ignored by the recording routine, thereby creating temporary dips in the overhead curves. As the client progresses to the interaction phase (where most of the requests are graphics-oriented), the overhead percentage approaches a constant. In this phase, the majority of time is consumed checking to see if graphics requests are for pixmaps. Therefore, the more pixmap resources the client creates, the greater overhead.

If we assume that in practice the number of conferees is more than three, then in the limit, resource recording overhead accounts for approximately 2% of the overall message processing time in the packet switch process; i.e. it adds an insignificant overhead to performance as observed by the users.

Memory requirements

Figure 12 illustrates the memory requirements for maintaining the state of a client's resources using our methodology. The dotted lines represent the total number of bytes of requests sent by the client, while

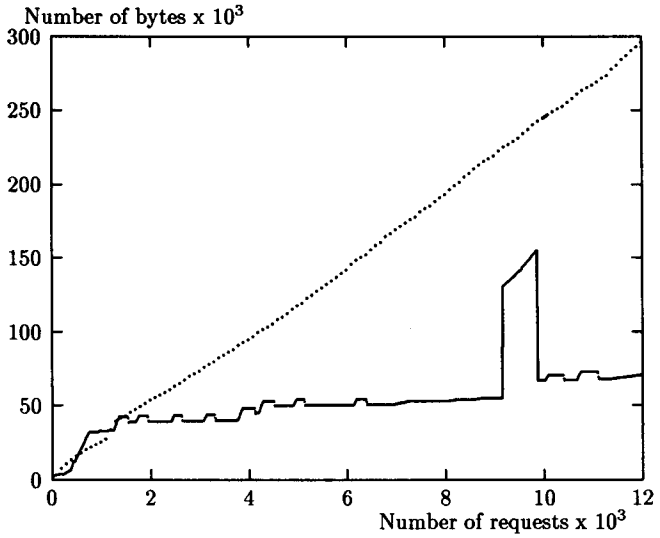

Figure 12 Memory usage for recording resource information in idraw. .....: Client requests; ——: Resource state

the solid lines represent the number of bytes used for recording state information. For idraw, after the set-up phase, the memory required to store the state of client resources grows at a near zero rate, while the memory required to store all client requests grows at a super-linear rate.

As the client progresses, we realize a dramatic saving of memory (over the approach of saving every request), considering the small cost (in terms of time) of processing each request.

Some additional facts about *Figure 12* are worth noting. Idraw creates and later destroys a large number of resources every time the user pulls down a menu. This behaviour accounts for the small spikes in the memory requirement curve. The large spike near request 9000 is due to a latecomer joining the conference. At that time the packet switch process creates a set of messages to send to the latecomer's server containing the resource information it has recorded. The creation of these messages accounts for the spike in memory use. Memory usage continues to increase after all messages have been created because the packet switch process appends requests coming from the client while the latecomer's server is being updated. The updating process is completed slightly before the 10,000th request is received from the client. At this time the memory used for messages is freed. The slight increase in the memory usage after the latecomer's server has been updated is due to the image contents of pixmaps that were acquired while the packet switch process was creating messages for the server. These pixmap images are kept for future use.

CONCLUSION

We believe that the ability to accommodate latecomers to a computer-based conference is important as it adds versatility and flexibility to an otherwise rigid conferencing system. In this paper, an efficient solution to the problem of accommodating a latecomer in XTV, a collaborative X window-based conference system developed at Old Dominion University and the University of North Carolina at Chapel Hill, and which is available in the public domain, is presented.

This goal is attained by first recording the environments that applications used in the conference create over time on X servers of original participants in the conference, and then creating these environments on the latecomer's X server when she joins the conference. We have modified XTV to record the current state of a server by maintaining a list of the resources and their current attribute values used by each application in the conference. These lists are updated based on the contents of request messages sent from conference applications to the X server.

The technique presented in this paper has been demonstrated to accommodate a latecomer in a practical and efficient manner. By keeping the memory requirements for recording state information to a minimum, the ability to accommodate a latecomer does not unduly burden the conference system. Moreover, as demonstrated in our performance analysis, the execution time overhead of recording the server state is quite tolerable for conferences of reasonable size.

We are now in the process of applying the techniques presented in this paper to allow two or more XTV conferences to merge into one XTV conference. This may be useful in situations where a subgroup of participants wants to work on a subproject, and at some point in time they want to share their work with the rest of the group.

Finally, it would be desirable to combine the capability to accommodate a latecomer into the X window system itself. This will relieve the conference agent from the burden of maintaining the modifications made to the server state for each shared application, and thereby eliminate the duplication of effort. Instead, the conference agent should be able to query the modifications made by an application by sending a special request message (e.g. '*GetClientsServerState*') to the local server. There should also be another special request message to impose the changes (e.g. '*PutClientsServerState*') on a second server.

REFERENCES

1 Ellis, C, Gibbs, S J and Rein, G L 'Design and use of a group editor', *Proc. Working Conf. on Eng. for Human-Computer Interaction*, IFIP Working Group 2.7 (August 1989)

2 Lantz, K A, Lauwers, J C, Arons, B, Binding, C, Chen, P, Donahue, J, Joseph, T A, Koo, R, Romanow, A, Schmandt, C and Yamamoto, W 'Collaboration technology research at Olivetti Research Center', *Proc. Groupware Tech. Workshop*, IFIP Working Group 8.4 (August 1989)

3 Sarin, S K and Greif, I 'Software for interactive on-line conference', *Proc. 2nd Conf. on Office Infor. Syst.*, (June 1984) pp 46–58

4 Stefik, M, Foster, G, Bobrow, D G, Kahn, K, Lanning, S and Suchman, L 'Beyond the chalkboard: Computer support for collaboration and problem solving in meetings', *Commun. ACM,* Vol 30 No 1 (January 1987) pp 32–47

5 Patterson, J F 'The Good, the Bad, and the Ugly of Window Sharing in X', *Proc. 4th Ann. X Tech. Conf.,* (January 1990)

6 Watabe, K, Sakata, S, Maeno, K, Fukuoka, H and Ohmori, T 'Distributed multiparty desktop conferencing system: MERMAID', *Proc. CSCW Conf. on Computer-Supported Cooperative Work* (October 1990)

7 Ahuja, S R, Ensor, J R and Lucco, S E 'A comparison of application sharing mechanisms in real-time desktop conferencing systems', *Proc. IEEE Conf. on Office Infor. Syst.* (April 1990) pp 238–248

8 Lantz, K A 'An experiment in integrated multimedia conferencing', *Proc. CSCW Conf. on Computer-Supported Cooperative Work,* MCC Software Technology Program (December 1986) pp 267–275

9 Crowley, T and Forsdick, H 'MMConf: The Diamond multimedia conferencing system', *Proc. Groupware Tech. Workshop,* IFIP Working Group 8.4 (August 1989)

10 Ensor, J R, Ahuja, S R, Horn, D N and Lucco, S E 'The Rapport multimedia conferencing system – a software overview', *Proc. IEEE Conf. on Computer Workstations,* Santa Clara, (March 1988) pp 52–58

11 Lauwers, J C, Joseph, T, Lantz, K A and Romanow, A 'Replicated architectures for shared window systems: a critique', *Proc. IEEE Conf. on Office Infor. Syst.* (April 1990) pp 249–260

12 Lauwers, J C and Lantz, K A 'Collaboration awareness in support of collaboration transparency: Requirements for the next generation of shared window systems', *Proc. Conf. on Human Factors in Computer Syst.,* ACM (April 1990)

13 Scheifler, R W and Gettys, J 'The X window system', *ACM Trans. Comput. Graphics* (May 1986) pp 79–109

14 Abdel-Wahab, H M and Feit, M A 'XTV: A framework for sharing X window clients in remote synchronous collaboration', *Proc. IEEE Conf. on Commun. Software: Commun. for Distributed Applic. & Syst.,* Chapel Hill, NC | (April | 1991) pp | 159–167

15 Linton, M A, Vlissides, J M and Calder, P R 'Composing user interfaces with inter-views', *IEEE Computer,* Vol 22 No 2 (February 1989) pp 8–22

16 Chung, G *Accommodating latecomers in a system for synchronous collaboration,* MS Thesis, University of North Carolina at Chapel Hill, NC (August 1991)

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

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

19 Chen, P 'The entity relationship model – Toward a unified view of data', *ACM Trans. Database Syst.,* Vol 1 No 1 (March 1976) pp 9–36