

Accommodating Latecomers in a
System for Synchronous Collaboration

TR91-038

August, 1991

Goopeel Chung

The University of North Carolina at Chapel Hill
Department of Computer Science
CB#3175, Sitterson Hall
Chapel Hill, NC 27599-3175



UNC is an Equal Opportunity/Affirmative Action Institution.

Accommodating Latecomers in a System for Synchronous Collaboration

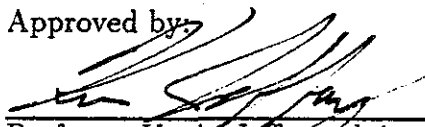
by

Goopeel Chung

A thesis submitted to the faculty of the University of
North Carolina at Chapel Hill in partial fulfillment of the
requirements for the degree of Master of Science in the
Department of Computer Science.

Chapel Hill, 1991

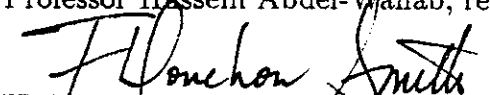
Approved by:



Professor Kevin Jeffay, advisor



Professor Hussein Abdel-Wahab, reader



Professor F. Donelson Smith, reader

©1991

Goopeel Chung

ALL RIGHTS RESERVED

GOOPEEL CHUNG. Accommodating latecomers in a system for synchronous collaboration (Under the direction of Professor Kevin Jeffay)

Abstract

This thesis deals with the problem of allowing a latecomer to join a computer-based conference that is already in progress. A conference is a synchronous collaboration session where people at remote locations are cooperating through identical copies of windows generated by applications shared by all conference participants. The shared windows are displayed by window systems residing on the collaborators' workstations according to commands issued by the shared applications. A 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. An efficient way to record the state modifications is introduced in this thesis. As a future goal, the study suggests that the recording and replaying functions be combined into the window system.

Acknowledgments

I would sincerely like to thank my thesis committee chair Professor Kevin Jeffay, who guided me through this project with many valuable comments and encouragement. I also would like to thank Professor Hussein Abdel-Wahab and Professor F. Donelson Smith, the other members of my thesis committee, for their insightful suggestions and the encouragement that they provided for this project.

Jin-Kun Lin helped me start this project by giving me valuable suggestions and encouragement. I thank him for the time he spent with me discussing important issues.

I also thank Dan Poirier for helping me begin understanding the X Window System.

I give my thanks to all Colab people for their help and encouragement. I learned much from them.

John Lusk, Manish Pandey and Amitabh Varshney gave me helpful pointers for refining my thesis with figures and LaTeX. Heather Fuller helped me with proof-reading. I thank them all.

I am grateful to all my family members. I thank grandfather, grandmother, father and mother for their love and education. Without them, I would not be what I am now. And to my special brothers, Koobong and Kooheon, I express my sincerest love.

Contents

1	Introduction	1
1.1	Motivation	1
1.2	Late Joining	3
1.3	Overview of Thesis	4
2	Architectures for accommodating late joining	6
3	Architecture	10
3.1	X Window System	10
3.1.1	Client/Server Model	10
3.1.2	Resources	12
3.1.3	An Example	14
3.2	XTV	15
3.2.1	Interception	17
3.2.2	Resource ID Translation	18
3.3	Accommodating Latecomers	19
3.3.1	Overview	19
3.3.2	Recording Modifications to Resources	20
3.3.2.1	Images in Drawables	21
3.3.2.2	Maintaining Resources	23
3.3.2.3	Dependency Relationships among Resources	23
3.3.2.4	Modifying the State of a Latecomer's Server	33
3.4	Summary	35

4	Implementation	36
4.1	Introduction	36
4.2	Data Structures	36
4.2.1	Attributes and Dependency Relationships	36
4.2.2	Information about a Tool	37
4.2.2.1	Resource Lists	40
4.3	Performance	46
4.3.1	Speed	46
4.3.2	Memory Requirements	48
4.4	Hard Problems	51
4.4.1	Images in Windows	51
4.4.2	Clients Changing others' resources	52
4.5	Status and Use	52
4.6	Summary	53
5	Conclusion	54
5.1	Summary & Conclusion	54
5.2	Future Work	56
5.2.1	Replicated Architectures Revisited	56
5.2.2	Extensions to Existing X Servers	57
	Bibliography	59

List of Figures

2.1	(a) Replicated Architecture (b) Centralized Architecture	7
3.1	Client/Server model	11
3.2	XTV architecture	16
3.3	Resource attributes and requests for modifying them	24
3.4	CreateCursor request	25
3.5	CreateWindow and ChangeWindowAttributes	26
3.6	CreateGC	26
3.7	ChangeGC and CopyGC	27
3.8	CreateColormap	27
3.9	CopyColormapAndFree	27
3.10	CreateGlyphCursor	27
3.11	CreatePixmap	27
3.12	E-R diagram of dependency relationships	28
3.13	The initial dependency graph for xpostit	29
3.14	DeleteNode algorithm	30
3.15	An example of node deletions	31
3.16	A cycle	32
3.17	Three possible cycles	32
3.18	WindowSpecial algorithm	33
4.1	Data structure for holding information about a tool	37
4.2	An instance of data structures for tools maintained inside XTV	41
4.3	WindowList	42

4.4	A node in ColormapList	43
4.5	A node in PixmapList	45
4.6	Percentage of time spent maintaining resource state information vs. number of client requests	49
4.7	Memory usage for recording resource information	50

Chapter 1

Introduction

1.1 Motivation

A group in the Department of Computer Science at the University of North Carolina-Chapel Hill is studying how computers can be used to facilitate collaboration [Smith et al. 90]. The thesis of the UNC researchers is that when people are working together, most of the effort is spent on building commonly shared conceptual, rather than physical, artifacts. This often happens in activities such as planning experiments, software design, or working on construction blueprints. People suggest, understand, negotiate, consult, modify and decide as they collaborate. Conceptual artifacts are built and maintained in each individual's mind throughout this process. It is important that these artifacts are shared by all the members involved in the collaborations. The best and most coherent conceptual model would be achieved if one exceptional individual worked on the project alone. Unfortunately, the reality is that no one individual possesses such expertise. Therefore, it is necessary to combine the resources of multiple humans in such a way that their collective efforts will bear fruit in the most efficient manner.

The focus of the UNC project is on artifacts that groups produce in the course of their interactions. For example, people might explain their idea by drawing pictures or diagrams on a whiteboard or small pieces of paper. Some artifacts such as documents or drawings may be generated with the aid of computers. These

artifacts are considered to be very important in forming the common conceptual model of the problem and its solution.

Interactions among people working together can be grouped into two categories: asynchronous and synchronous. Asynchronous collaboration involves people exchanging artifacts in an asynchronous manner (i.e., the sending and receiving of artifacts are not coupled). For example, the collaborators convey ideas through documents such as electronic mail messages that are exchanged in an asynchronous manner. Characteristics of this style of interaction are that it usually spans a long period of time and the cast of people working together varies over time.

Synchronous collaboration involves people working together at the same time such as in a meeting. They usually share a common physical space (like a room) and work on shared artifacts together. Synchronous collaborations are typically conducted as concentrated cooperative efforts to tackle problems in a rather short period of time.

With advances in computer and communication technology, it has been suggested that computers be used as a medium for facilitating collaborations [Smith et al. 90]. More specifically, computers are considered to provide a useful and efficient means of representing artifacts produced in collaborative efforts. In fact, computers have already been used to create, store and manipulate artifacts by individuals. It would be desirable if we could apply this functionality to the conceptual and physical artifacts shared in collaborations.

Computer support for asynchronous collaborations has existed for some time. Electronic mail and file transfer programs are some examples. With networked computers, augmented by audio and/or video communications, it is also possible to support synchronous collaborations when the collaborators are physically separated. All the participants of the meeting can work together from their respective offices. They share the same view of the artifacts constructed with the computers and can see any modifications made to them in real-time.

The synchronous collaborations described above are referred to as *conferences* and the people working in such collaborations are called *conferees*. The computer

systems providing the facilities to hold the synchronous collaborations are referred to as the *conference systems*.

People working in a conference can be considered to have access to a shared visual workspace through which they share a view on artifacts such as documents, images and programs. A shared visual workspace is often implemented as a shared window displayed on workstations of collaborators in the conference [Stefik et al. 87, Abdel-Wahab & Feit 91, Crowley & Forsdick 89]. This shared window is the vehicle through which the shared artifacts can be displayed and modified.

Special-purpose applications have been developed to allow multiple remote users to share the visual workspaces [Ellis et al. 89, Lantz et al. 89, Sarin & Greif 84, Stefik et al. 87]. However, in many cases, conventional single-user applications are used to provide the mechanisms for sharing windows [Patterson 90, Watabe et al. 90, Ahuja et al. 90, Lantz 86]. This is because most applications in use today are single-user applications. The ability to use familiar applications in a conference saves the conferees from the burden of learning how to use new applications. Modifying single-user applications is not only a non-trivial task, but also impractical. The real-time conference systems using single-user applications to share the visual workspaces are referred to as *shared window systems*.

1.2 Late Joining

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 behavior can be expected to occur during a conference using networked computers. For example, two people 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 provide the functionalities to accommodate dynamic joining, single-user (“collaboration-transparent”) applications are not designed to provide these functions. It is the conference system itself that should provide them.

This thesis 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.

1.3 Overview of Thesis

Shared window systems consist of application processes and window systems. Connecting all these processes in the center 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

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 thesis 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.

The next chapter describes two architectures (centralized and replicated) employed in shared window systems and the feasibility of accommodating a latecomer in each architecture.

Chapter 3 discusses how we solved the problem of accommodating a latecomer. Some background information is provided on the X Window System and XTV, on which our solution is based.

Chapter 4 is about implementation details of the system. Data structures, performance figures, some difficult implementation problems are described in this section.

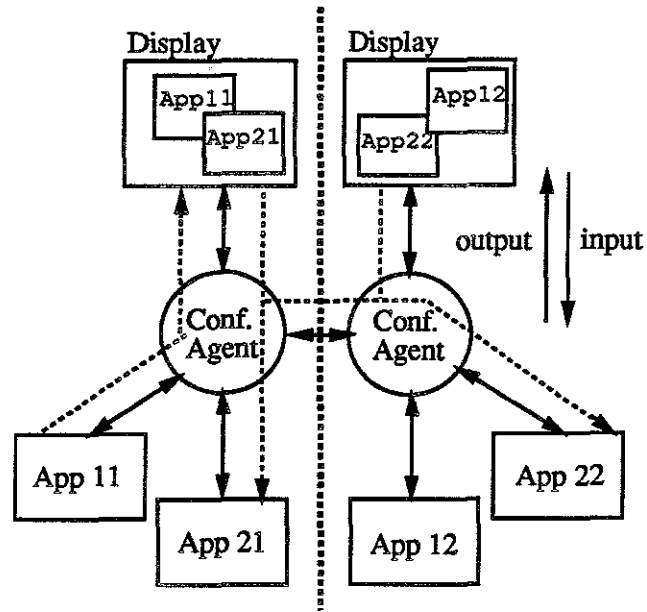
We conclude in Chapter 5 with a discussion of future works for more efficient shared window systems and contributions our system can make to them.

Chapter 2

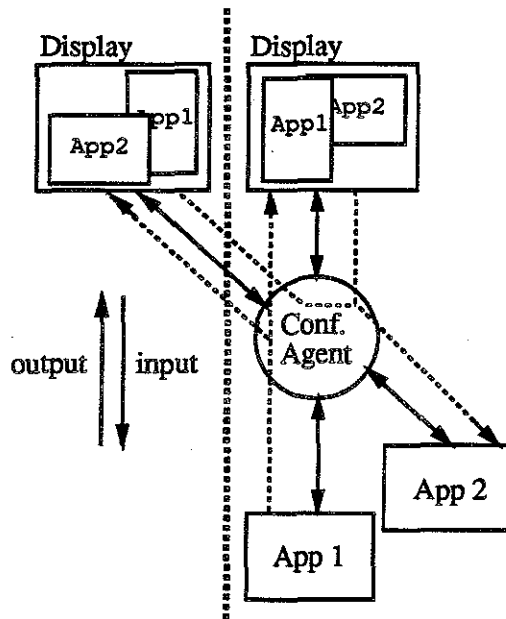
Architectures for accommodating late joining

There have been two major approaches to implementing shared window systems: centralized architectures and replicated architectures [Lauwers & Lantz 90]. Replicated architectures maintain as many copies of each application and conference agents as there are conferees, whereas centralized architectures need only one copy of each application and a single conference agent (see Figure 2.1).

In the replicated architectures, a copy of each shared application runs on every workstation in the conference. Output from each copy of the applications is conveyed to the local conference agent, and then forwarded to the window system for the local display. As far as output is concerned, there is no communication between conference agents. Input from a window system is sent to the local conference agent, which in turn distributes the input to all copies of the associated application. Because multiple window systems may provide input to an application simultaneously, each copy of the application may receive a different input sequence. Control mechanisms are used to ensure that every copy of the application sees the same input sequence. Because all copies of an application see the same input sequence, the state of each copy is synchronized. Examples of systems implemented in the replicated architectures include Vconf, Dialogo and MMConf [Ahuja et al. 90, Crowley & Forsdick 89, Lantz 86, Lauwers et al. 90].



(a)



(b)

Figure 2.1: (a) Replicated Architecture (b) Centralized Architecture. The bold dashed lines denote the machine boundaries, and the two subscripts used for applications in the replicated architecture identify the application and the replica respectively.

In the centralized architectures, input events from all window systems are sent to the single conference agent. Since there is only one copy of each shared application, the conference agent can forward the input events to their respective applications as the events arrive. Unlike the replicated architectures, the single conference agent is responsible for output to all the displays used in the conference. The conference agent has to distribute every output message of the applications to each window systems associated with each display. Mermaid, Rapport, Shadows and XTV have adopted the centralized architecture [Watabe et al. 90, Ensor et al. 88, Patterson 90, Abdel-Wahab & Feit 91].

The centralized architecture is much simpler, but the performance has the potential for degrading as the number of conferees increases. This is because a single conference agent has to take care of all input and output streams. In principle, one would expect the replicated architecture to have much better performance since the output stream is only directed to local window system. However, the replicated architecture is more complex because it is harder to maintain the identical state across all copies of an application. Moreover, the replicated architecture assumes that copies of applications are available on every site in the conference. This can be an unreasonable assumption if different machine types exist at the various sites.

To accommodate a latecomer that wishes to join an existing conference, the shared state created by the application in the conference needs to be replicated on the new participant's workstation. In a centralized architecture, the window system state on the new participant's workstation needs to be brought up-to-date so that any subsequent output from the shared applications has the desired effect. In a replicated architecture, a copy of each shared application has to be created and brought up-to-date in addition to the window system state. It would be desirable if window systems and applications provided a means to download and upload their internal states. Unfortunately, no such window system or application exists.

Therefore, given an input and output stream from a window system and application, the conference agents have to record the relevant window system state, and later "project" this state onto the window system of a latecomer's workstation.

In a replicated architecture, the conference agent would record only the shared application's state. The recording would take the form of an input sequence to the application. For centralized architecture, the conference agent would record only the local window system's state. The recording would take the form of an output sequence from the application.

Efficiently recording the state information is difficult in a replicated architecture. This is because the input stream from a window system may not provide the conference agent information as to what state the application is in unless the conference agent knows all about the application. Currently, there is no support for dynamic joining in systems using replicated architecture.

For the centralized architecture, the situation is better. There are ways to deduce the state of a window system at least in terms of the shared applications. Shadows and Mermaid both accommodate latecomers. For example, Shadows provides this feature by compressing the history of resource declarations and then sending this history to the latecomer's window system at the time the latecomer joins the conference [Patterson 90]. Other systems with a centralized architecture like XTV do not support it. XTV cites as reasons for not supporting latecomers that a great deal of state information maintained by the window system needs to be kept [Abdel-Wahab & Feit 91].

A more efficient way to keep the state of the window systems at least with respect to the shared applications will be explored in this thesis within the context of the centralized architecture. The method we introduce in this thesis is expected to be more efficient than Shadows' in terms of memory usage. Our method does not require the precise history of resource declarations to be recorded.

Chapter 3

Architecture

This chapter describes the overall architecture of the extensions to XTV that accommodates latecomers. We begin with some background. Section 3.1 describes the X Window System, the window system used by XTV. Section 3.2 describes X TerminalView (XTV) on which the feature to accommodate latecomers is based. Section 3.3 provides a detailed description of the system architecture.

3.1 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 architectures the applications will be running on [Nye 89a]. 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.

3.1.1 Client/Server Model

The X Window System is built on a familiar distributed system model: the client/server model (see Figure 3.1). A server typically is a program that offers a service such as file access. A client is an application that requests the server to perform some ac-

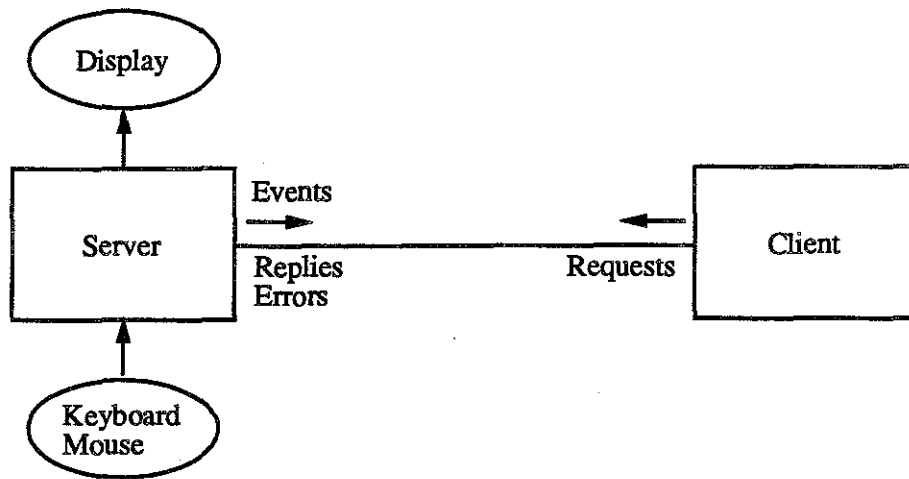


Figure 3.1: Client/Server model

tion on its behalf [Comer 90]. All communication is conducted via message passing. Servers accept request messages, perform their service, and return the result to the requester. Clients send request messages and wait for a reply.

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

Clients and servers may be on the same machine or different machines interconnected by a communications network. Message passing is via asynchronous network protocols. The protocols provide transparency on the part of both the user and the programmer of the application. That is, whether or not the application is executing on a local machine is transparent to the user, and the programmer does not have to worry about from which machine the user is interacting with the application.

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. Not all the requests received by an X server require a response. For example, requests for creating windows do not require a reply from a 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. Some request messages may also cause events to be later generated. Events are sent to the client only if the client has previously requested that the servers do so.
- 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, or asked for service that the server does not support.

3.1.2 Resources

The X application programming model presents six basic abstractions: *window*, *cursor*, *graphics context*, *pixmap*, *colormap* 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.

A Window is like a canvas on which the client may draw objects by sending pertinent request messages to the server. Once a window is created, it is in either of two states: *mapped* or *unmapped*. If a window is mapped, then the user can see any unoccluded portions of the window. Unmapped windows are never visible.

Cursors are small pointers that move on the display according to movement of the mouse. They are usually associated with one or more windows so that when

the user moves the mouse inside the windows, a different cursor is displayed as an indicator to the user where the input pointer is located.

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 color, its width or the style (e.g., solid or dashed), or when writing texts, we may want to specify the font or its color. 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.

Pixmap 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.

- A pixmap can be used in a window as the background or border pattern. The background of a window is the drawing surface on which other graphics are drawn and the border of a window is the thin outer rectangular area surrounding the window.
- A pixmap can be used as the tile, stipple or clip mask of a graphics context. Tile and stipple are alternate names for pixmaps when they are used for patterning an area of a drawable. The only difference between them is that a stipple is a pixmap with color depth one (i.e., one of two colors). A clip-mask is used to restrict graphics operations to a subset of pixels.
- 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 color of the pixel. This pixel value is an index into a list of entries called *colorcells*. A

colorcell holds three values; one for each RGB (Red, Green, Blue) value. The list of colorcells constitutes a *colormap*. Depending on which colormap is currently in use, a pixel can have a different color. Every window is associated with a colormap.

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. Another usage of fonts is as the source or mask for cursors. Fonts used for this purpose contain pre-defined patterns and shapes of cursors. The client specifies the font when creating a cursor.

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 colormap and the root window). All the resources are referred to by resource IDs (unique integers).

3.1.3 An Example

A typical session is started by the client opening a connection (a communication channel) to the server. This is done by the client sending a message to a well-known address. After verifying the connection information sent by the client, the server allows the connection by sending the client information about the server and root window. The client can now send requests to create various resources and to make changes to them.

Let us take a simple graphics application for illustration. This application draws red dots on a window with a white background at the positions where the user presses a mouse button. The window is surrounded by border tiled with a certain pattern.

The application (the client) may start by preparing the border pattern of the window on a pixmap. To do so, it creates a pixmap and a graphics context with black foreground color. The client draws the pattern on the pixmap by using requests for drawing dots on drawables. All the dot-drawing requests specify the graphics context created above. The client wants to display a different cursor whenever the user puts the mouse inside the window. It creates a cursor to display on the window using one

of the pre-defined patterns stored in a special font. When all the resources needed for the window are ready, the client creates the window specifying the pixmap for the border pattern, white color for the background and the cursor. At this time, the client also requests that the server notify it with an event whenever a mouse button is pressed on the window.

Because the client wants to use red color to draw dots on the window, it gets the pixel value for the color red by sending a request to allocate the color in the default colormap (the colormap associated with the root window). The server returns the pixel value in reply to the request. The client now sends a request message to change the foreground color attribute of the graphics context that was used to draw the border pattern of the window.

The client is now ready to interact with the user. The client makes the window visible by mapping it. The user can see that whenever the mouse moves into the window, the cursor changes to a different pattern. Mouse button press events are delivered to the client when the user pushes any of the buttons. The event message contains the information of where the user pressed the button inside the window. With this information, the client draws a point using the graphics context with red foreground color.

3.2 XTV

Most X applications have been written to interact with a single user. When such an application is used, there is only one 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 the output of X applications in real-time [Abdel-Wahab & Feit 91] (an extension of Remote Shared Workspaces system [Abdel-Wahab et al. 88]). The system also makes it

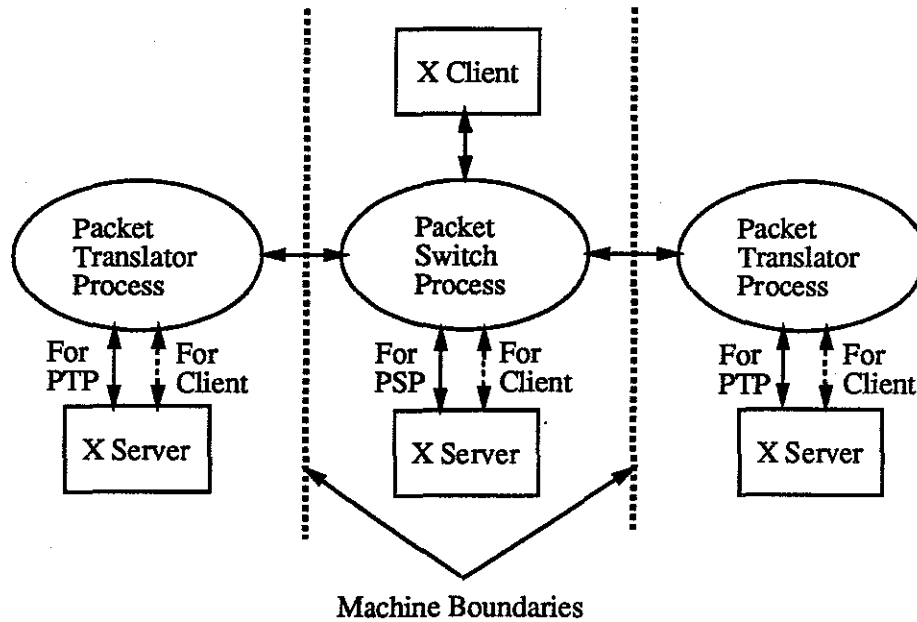


Figure 3.2: XTV architecture

possible for the users to provide input to the applications. The remote users are expected to be working at workstations running X servers and interconnected by a local area network. 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.

XTV employs the centralized architecture (see Figure 3.2) where 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 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 worksta-

tion. 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.

At the heart of this approach are two important concepts:

- Interception of traffic in the normal connection between a single client and a server.
- Translation of resource IDs in messages sent through the connection.

These concepts are explained in more detail below.

3.2.1 Interception

In the X Window System, communication with an X server is done via message passing. The server accepts messages from a port whose number is well known to all clients. The port number is defined symbolically in the X protocol as `X_TCP_PORT`. The client refers to a UNIX Shell environment variable called `DISPLAY` when trying to make a connection. `DISPLAY` takes the format *hostname:display.screen*, where *hostname* is the standard Internet host name or an Internet address (e.g., `delta.cs.unc.edu` or `128.109.136.99`). When contacting an X server, an X client makes a connection to port number `X_TCP_PORT+display` on the machine given by the *hostname* field.

The packet switch process in XTV intercepts X client message traffic by creating a port numbered `X_TCP_PORT+SessionNumber` (non-zero value). Clients that use a `DISPLAY` environment variable with a value of *hostname:SessionNumber.0* will connect to XTV rather than the X server. (*hostname* is the name of the Internet host the packet switch process is executing on.) Knowing the port number where the display server receives messages, the packet switch process can relay requests from clients to the server, while distributing the same requests to packet translator

processes participating in the conference. The packet switch and packet translator processes will therefore look like copies of the original clients from the remote servers' points of view.

3.2.2 Resource ID Translation

In X, resource IDs are assigned by the client. For each client, the server allocates a subrange of numbers representable in a 32 bit integer. The client always uses a number in this range for each resource it creates. This way, the client can ensure that the resource IDs it generates are unique.

Because the integer subrange may be different for each connection from XTV (connections from the packet switch or the packet translator processes), resource IDs need to be changed to fit into the subrange allocated by each different server. This translation is done in the packet translator process.

For example, suppose that for a shared application, the server for a packet translator process allocated 16^4 consecutive numbers starting from the hexadecimal A0000 for resource IDs, while the server for the packet switch process allocated 16^4 consecutive numbers starting from the hexadecimal B0000. Upon receiving a request message from the packet switch process, the packet translator process searches the message for resource IDs to translate. Each ID should have the form Bxxxx. For each ID so found, the packet translator process substitutes a corresponding resource ID in the subrange given by its local server (i.e., one of the forms Axxxx). Most of the time, a simple replacement of some number of significant hexadecimal digits is sufficient (the most significant hexadecimal digit in this example). The message is sent to the local server after all the translations are done.

Messages received from the local server by each packet translator process (e.g., event, reply or error messages) go through an inverse translation to the client-readable form. All resource IDs in these messages are translated back to the equivalent client resource IDs by the packet translator process and sent to the packet switch process and eventually to the client.

3.3 Accommodating Latecomers

3.3.1 Overview

Multiple remote users can use XTV to share a set of applications. One of the users becomes the *host* of the XTV conference by executing XTV (i.e., creating and connecting to a packet switch process) on her workstation. Other users join the conference by executing XTV and connecting their translator processes to the packet switch process of the host (see Figure 3.2). The host can now execute X applications which will be shared by all the participants in the conference.

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 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 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 servers.

In addition to distribution and translation of client request messages, XTV — more specifically, the packet switch process — now must 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 sends about 300,000 bytes of requests for only 4 minutes after it starts interacting with a server (see Figure 4.7 (a)). Moreover, it will take proportionally longer time for a latecomer to catch up on the conference depending on how late she joins the conference.

A quick improvement can be made to the above brute-force method. Since we are only concerned about the changes made to the server state, we can ignore requests that do not change the server state. Obviously, requests for acquiring information from the server such as query requests are in this category. This is a very minor improvement, because relatively few query requests are made by clients. What is needed is a more sophisticated approach to archiving requests.

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

3.3.2 Recording Modifications to Resources

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

- 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 color of a window background (a private resource), or it can allocate more colorcells in the default colormap (a non-private resource)), or
- change other miscellaneous environment properties such as the keyboard mapping, or 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. However, these properties are generally set only once by the client, and therefore, they can be saved in chronological order and replayed to a new server.

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 resource's creation. Only the current values will be used when creating resources on a latecomer's server.

It would be desirable to keep track of the IDs of resources created by a client, and then later get the information on attributes of these resources from the server when a latecomer joins the conference. Unfortunately the X protocol was not designed to provide this service using query requests. We can get only limited information about the attributes of a resource. Therefore, we cannot avoid duplicating some of the functions of an X server and keeping track of the attributes of all resources.

3.3.2.1 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 becomes 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 Section 4.3.2). This is because X clients generally go through two phases in their life time: 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. For example the client may highlight a window used as a radio button, or display the output of the computations requested by the user. Since the interaction phase is generally much longer, the majority of messages sent from clients to server are requests for graphics operations.

3.3.2.2 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.

The Figure 3.3 shows the attributes the client sets and modifies for each resource category. The attributes are accompanied by the requests that set or modify them. (For a description of the function of each attribute see [Nye 89a, Nye 89b].)

3.3.2.3 Dependency Relationships among Resources

A problem arises if we naively apply the method in the previous section for all resources. For example, in idraw [Linton et al. 89], a MacDraw-like X application, 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 in order 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 a dependency relation. A resource A depends on a resource B if the resource A has as one of its attribute values the resource B . For example, the

Resource	Attribute	Request
Window	depth, window ID, class, visual, parent ID, x, y, width, height, border width, background pixmap, background pixel, border pixmap, border pixel, bit-gravity, win-gravity, backing-store, backing-planes, backing-pixel, override-redirect, save-under, event-mask, do-not-propagate-mask, colormap, cursor, state, properties associated with the window, grabbing associated with the window, relationship with siblings	CreateWindow, ChangeWindowAttributes, ConfigureWindow, CirculateWindow, ReparentWindow, MapWindow, UnmapWindow, MapSubwindows, UnmapSubwindows, DestroyWindow, DestroySubwindows, ChangeProperty, DeleteProperty, RotateProperty, GrabKey, UngrabKey, GrabButton, UngrabButton
Pixmap	depth, pixmap ID, drawable, width, height, state	CreatePixmap, FreePixmap
Graphics Context	graphics context ID, drawable, state, function, plane-mask, foreground, background, line-width, line-style, cap-style, join-style, fill-style, fill-rule, tile, stipple, tile-stipple-x-origin, tile-stipple-y-origin, font, subwindow-mode, graphics-exposures, clip-x-origin, clip-y-origin, clip-mask, dash-offset, dashes, arc-mode, dashes, clip rectangles	CreateGC, CopyGC, ChangeGC, SetDashes, SetClipRectangles
Colormap	alloc, colormap ID, window, visual, state, colors in the colormap	CreateColormap, CopyColormapAndFree, FreeColormap, InstallColormap, UninstallColormap, AllocColor, AllocNamedColor, AllocColorCells, AllocColorPlanes, FreeColors, StoreColors, StoreNamedColors
Font	font ID, name, state, font path	OpenFont, CloseFont, SetFontPath
Cursor	cursor ID, source, mask, hotspot, fore-red, fore-green, fore-blue, back-red, back-green, back-blue, state	CreateCursor, CreateGlyphCursor, RecolorCursor, FreeCursor,

Figure 3.3: Resource attributes and requests for modifying them

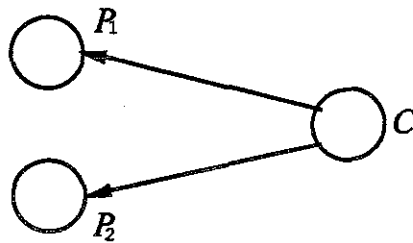


Figure 3.4: CreateCursor request

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 using a directed graph, called a dependency graph, such as in Figure 3.4. 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, Figures 3.5 through 3.11 show the dependency graphs that result from common requests¹.

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.

¹Details on the exact semantics of these requests can be found in [Nye 89a, Nye 89b].

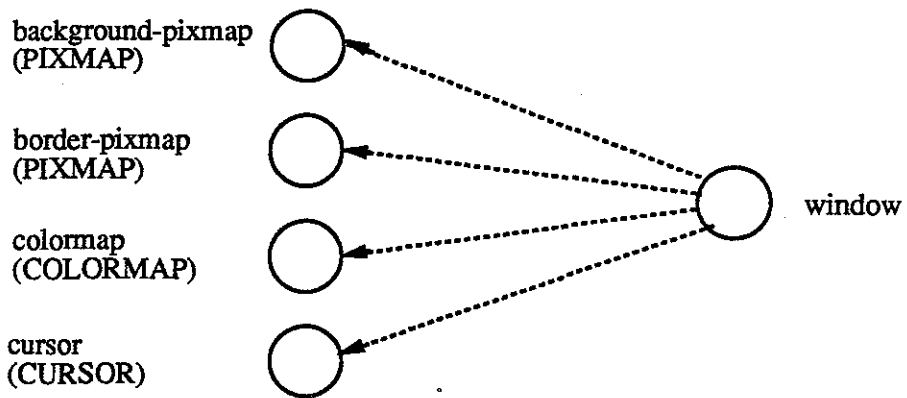


Figure 3.5: CreateWindow and ChangeWindowAttributes

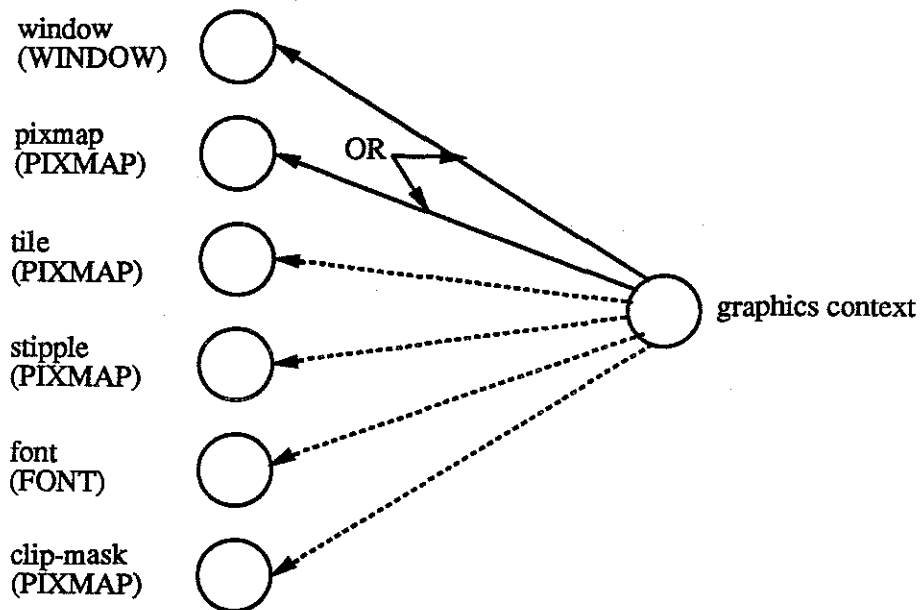


Figure 3.6: CreateGC (create a graphics context)

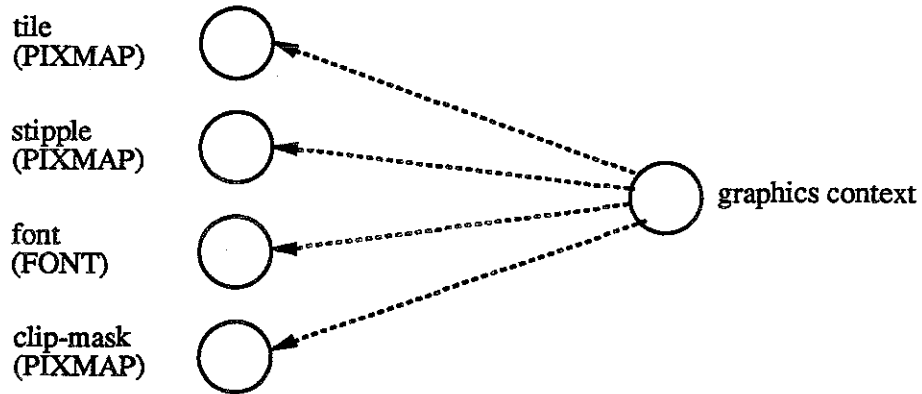


Figure 3.7: ChangeGC (change attributes of a graphics context) and CopyGC (copy from another graphics context)



Figure 3.8: CreateColormap



Figure 3.9: CopyColormapAndFree

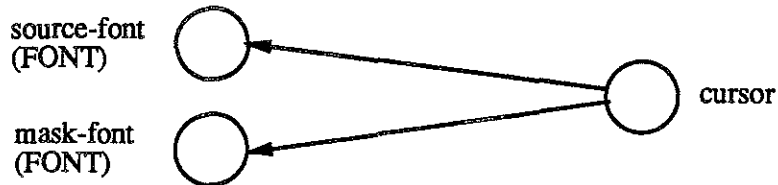


Figure 3.10: CreateGlyphCursor

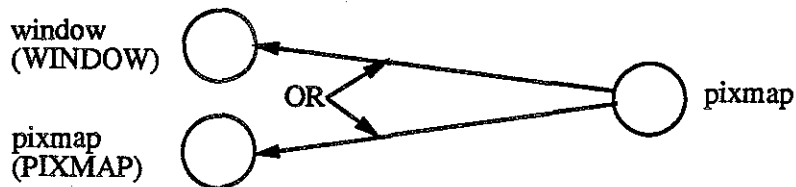


Figure 3.11: CreatePixmap

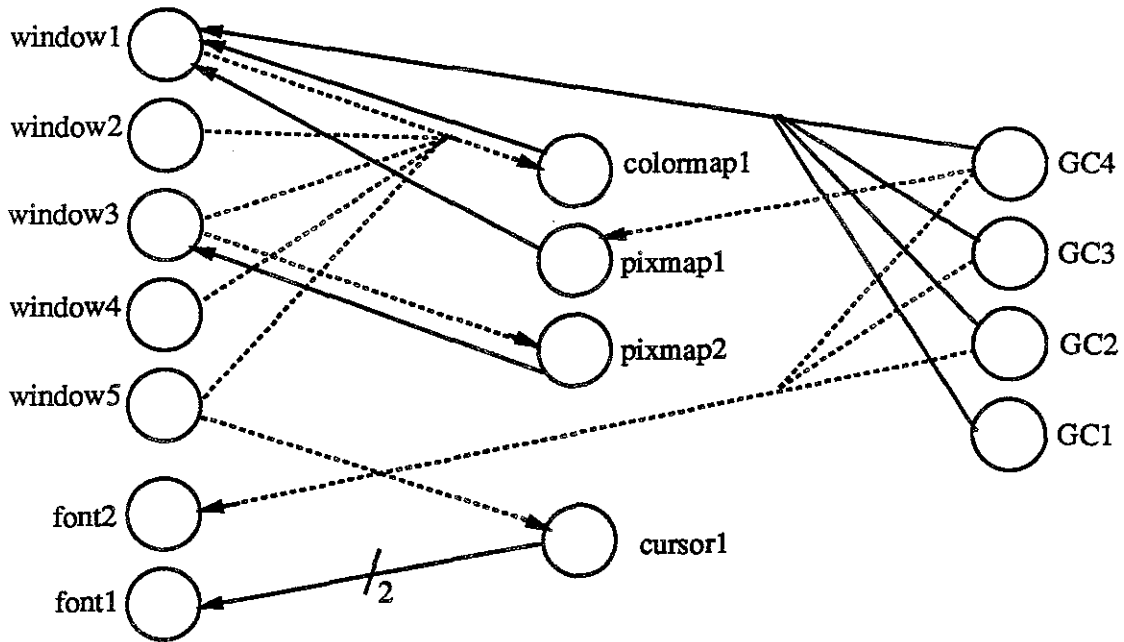


Figure 3.13: The initial dependency graph for xpostit

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). For example, consider the graph in Figure 3.15 (a). White nodes represent non-freed nodes and black nodes represent freed nodes. If a request arrives from the client to free node *B*, we first mark *B* as freed (color it black) and apply DeleteNode algorithm (Figure 3.15 (b)). There are no optional edges, so the first For-loop does not apply. Since the node *C* still depends on node *B*, the algorithm does nothing on the graph. If next a request to free node *C* arrives, we color node *C* black (Figure 3.15 (c)), and apply DeleteNode to *C*. Since no node depends on *C*, it will be deleted and DeleteNode recursively calls on *B*. With the dependency on

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 3.14: DeleteNode algorithm

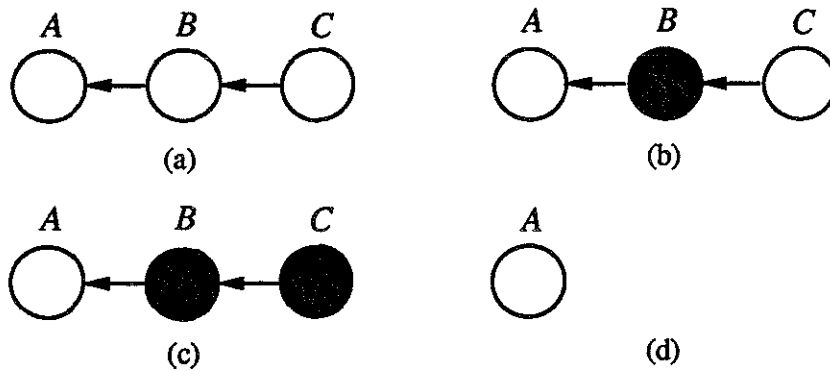


Figure 3.15: An example of node deletions

B now gone, node B will be deleted and `DeleteNode` recursively called on node A . Since A has not been freed, the algorithm returns without taking any action on it (Figure 3.15 (d)).

The `DeleteNode` algorithm works only with acyclic graphs. However, an examination of our E-R diagram (Figure 3.12) 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 3.16. 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 `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 must contain a window node (see Figure 3.12). In fact, there are only 3 types of cycles as shown in Figure 3.17. 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 3.18 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

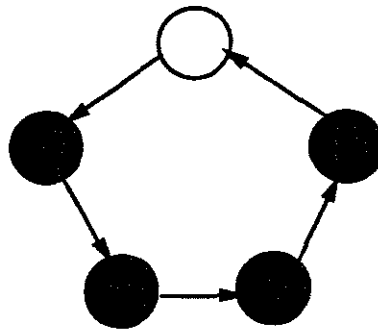


Figure 3.16: A cycle

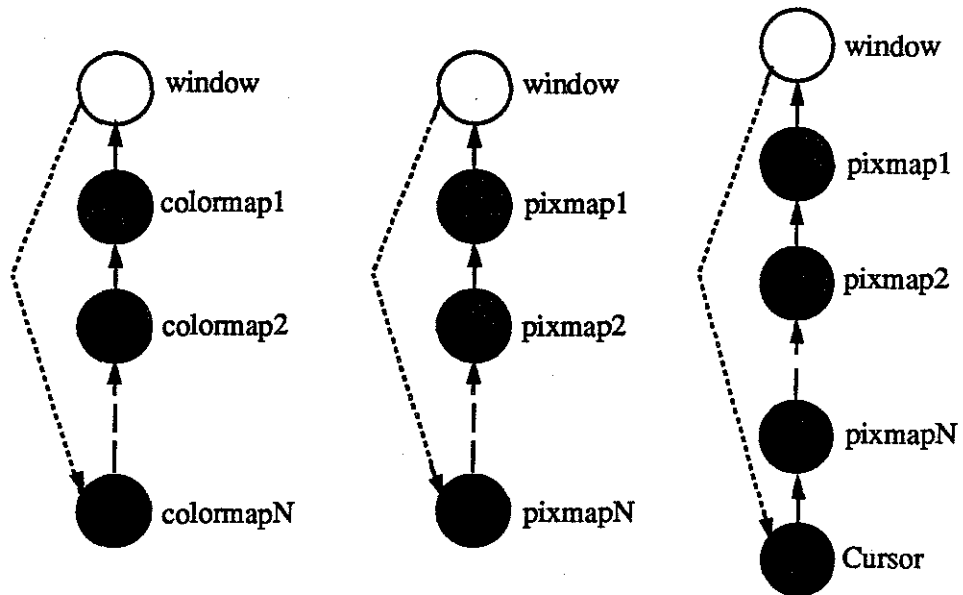


Figure 3.17: Three possible cycles

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 black;
If node has no incoming edges then
    Delete node;
End If
```

Figure 3.18: WindowSpecial algorithm

delete a dependency relation.

3.3.2.4 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. One possible method is to create requests to perform the following (in order);

1. Create windows with no resource-related attributes set.
2. Open fonts.
3. Create colormaps and do all of the color related operations such as *AllocColor* and *AllocColorCells*.
4. Create pixmaps and put the image of each pixmap acquired through *GetImage* request.
5. Create graphics contexts setting all attributes (essential and optional).
6. Set the remaining attributes of windows. Use *ChangeWindowAttributes* requests to do so.
7. Map any windows that are currently mapped on remote servers.
8. Generate other window related requests such as grabbing requests and property changing requests.
9. Set other environment parameters that are not necessarily related to the resources using requests such as *SetKeyboardMapping* and *ChangePointerControl*.

After creating all these messages, 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.

3.4 Summary

In this chapter, we have identified that in order to bring a latecomer's server up-to-date we need to record the modifications to the server state implied by the sequence of request messages from each shared application, and then later apply the recorded modifications to the latecomer's server.

X applications modify the server state by creating private X resources and changing attributes of resources (private or non-private). Therefore, the modifications can be recorded by keeping attributes of each resource up-to-date in a data structure. A data structure is created when an application creates a resource or first modifies a non-private resource. When a resource is freed, the data structure containing information about the resource is deleted.

Along with the attributes of resources, dependency relationships among resources should be maintained. This is to prevent information on freed resources from being deleted when other non-freed resources have the freed resources as attribute values.

When a latecomer joins a conference in progress, requests to create private resources with current attribute values and to modify attributes of non-private resources are sent to the latecomer's server. These requests are generated while making sure that no resource is created before all the resources it depends on are created on the latecomer's server.

Chapter 4

Implementation

4.1 Introduction

In this chapter we describe our implementation of the system described in the previous chapter. Our system for accommodating latecomers contains approximately 5,500 lines of code out of about 20,000 lines of code for the current release of XTV. The system is written in C programming language. Familiarity with C is assumed throughout.

We briefly describe the data structures we use for maintaining client state information and then discuss the performance of the system. We conclude with a brief discussion of some of the problems we encountered in the course of the implementation.

4.2 Data Structures

4.2.1 Attributes and Dependency Relationships

The packet switch process in XTV maintains a data structure for each resource that contains the attributes of the resource and a record of the other resources that the resource depends on. These data structures form the nodes in the dependency graph for a tool (an application program used in a conference). If a resource *A* depends

```

typedef struct ToolState{
    CursorObj      *CursorList;      /* to cursor list head */
    FontObj        *FontList;        /* to font list head */
    GCObj          *GCList;          /* to GC list head */
    PixmapObj      *PixmapList;      /* to pixmap list head */
    ColormapObj    *ColormapList;    /* to colormap list head */
    InstalledColormapNode *ICList;    /* to installed colormap list */
    WindowObj      *WindowList;      /* to window tree root */
    WindowObj      *Destroyed;       /* to destroyed window list */
    char           *StartToolPacket; /* initial connection message */
    char           *InternalAtom;     /* InternAtom requests buffer */
    CARD32         InternalAtomSize; /* size of the buffer */
    char           *SaveAlls;         /* miscellaneous requests buffer */
    CARD32         SaveAllsSize;     /* size of the buffer */
    struct PixmapAndGC PAG[65];      /* drawable depth and GC */
    CARD32         RecentResourceID; /* recently used ID */
} ToolState;

```

Figure 4.1: Data structure for holding information about a tool

on resource B , then resource A 's data structure will contain a pointer to resource B 's data structure. The pointers to resource data structures are the edges in the dependency graph. In addition, in resource B 's data structure we maintain a count of the number of resources that depend on B . If resource A is destroyed and its corresponding node in the dependency graph can be deleted, then the packet switch process follows the pointer from A to B and decrements the counter value in B . If B 's dependency counter is 0 and B has been freed by the client, then B 's data structure can be deleted as well.

4.2.2 Information about a Tool

A list of all resources created or used by a tool is maintained on a per-tool and per-resource type basis. Figure 4.1 shows this data structure for a generic tool.

Lists for five of the six resource types (cursor, font, graphics context, pixmap and colormap) are implemented as a doubly-linked list. When a tool creates an instance of one of these resources, a new instance of the appropriate resource data structure is appended to the front of the resource list in the tool structure. The data structures for window resources created by the tool are organized as a tree. `WindowList` is the pointer to the root of the window tree structure. These resource lists will be discussed in more detail below.

The initial connection message for the tool is contained in a buffer pointed to by

StartToolPacket. The connection message contains information needed to start the tool that the latecomer's packet translator process will need when connecting to its X server. It also contains information to help the packet translator process to translate resource IDs in subsequent request messages. This message will be the first one sent to the packet translator process on a latecomer's machine.

InternalAtom points to a buffer containing information about *atoms* introduced by the tool, and **InternalAtomSize** is the size of the buffer. An atom is a four byte integer uniquely identifying a group of templates for storing and retrieving information. A group of templates usually has a string name of arbitrary length, and an atom is used when referring to the group so that arbitrary length string names need not be sent across the connection. Each template in a group is associated with a different window and is called a *property* of the window. A property is used by X clients to communicate arbitrary data with each other. The buffer pointed to by **InternalAtom** contains the string names and the atoms used by the X client to refer to the properties.

SaveAlls points to a buffer containing miscellaneous messages that the tool sent to the server that are not related to resources. These messages are queued in chronological order and **SaveAllsSize** keeps track of the current size of the buffer. The messages include request messages such as *ChangeKeyboardMapping* and *ChangePointerControl*.

Our approach uses *PutImage* requests to bring the image attributes of pixmaps up-to-date. Every *PutImage* request must specify a graphics context which has been created for use with drawables of the same depth as the destination drawable of the *PutImage* request. A graphics context can have quite different effects on a drawable depending on how each of its attributes is set. If we use the graphics contexts created by the tool, we may have to change some of their attributes and change them back to the previous values. Even worse, graphics context for drawables of a certain depth may not exist. Therefore, we create graphics contexts as needed and destroy them after their use. **PAG[65]** is an array where element *i* contains

(1) the number of drawables¹ (windows and pixmaps) whose depth is i and (2) a graphics context ID that can be used to draw on drawables of depth i . The number of drawables is recorded to reflect the current number of drawables of depth i for the tool. When a latecomer joins the conference, the packet switch process creates a graphics context for each depth of the drawables the tool created, and put its ID in `PAG[]` array. The variable `RecentResourceID` has the most recent resource ID value that was created by the tool. The packet switch process uses this number to create its own graphics contexts to put in the `PAG[]` array. It uses as many numbers as are needed that are larger than the current `RecentResourceID` value. Because we use the subrange of IDs allocated for the tool, the packet switch process must create request messages to free the graphics contexts created above after the image attributes of all pixmaps have been updated.

`ICList` is a doubly-linked list of currently installed colormaps. This list is maintained in such a way that the internal data structure for a newly installed colormap is placed at the front of the list, and the data structures for colormaps that have been “uninstalled” are deleted. The packet switch process creates *InstallColormap* requests for colormaps starting from the back of the list so that the most recently installed colormap gets installed last. This handling of colormaps is necessary because some high performance workstations maintain more than one installed colormaps simultaneously. The maximum number of installed colormaps can differ depending on the X server. The installed colormaps are recorded in an ordered queue inside the server so that the most recently installed colormap is placed at the front of the list. If a new colormap is installed when the queue is full, the colormap at the end is deleted (uninstalled) from the queue.

`Destroyed` is also a doubly-linked list of window data structures that have been destroyed by the tool, but cannot be deleted because other resources depend on them. Information on these windows is stored separately to simplify the handling of the window tree data structures. (This is because then the tree contains only data

¹Image attributes of some windows need to be updated using *PutImage*. See Section 4.4.1 for details.

structures for undestroyed windows.) The windows in `Destroyed`s are stripped of all optional attributes, and created as the children of the root window on a latecomer's server. These windows are destroyed as soon as other resources depending on them have been created on the latecomer's server.

XTV maintains a list of data structures like the one shown in Figure 4.1 for all the shared applications. Figure 4.2 shows an instance of such a list. The first tool is `xpostit` (whose dependency graph is shown in Figure 3.13), and the n 'th tool is the example X application introduced in Section 3.1.3. Note that only resource lists are shown for simplicity. Directed edges are used to represent the dependency pointers. Optional pointers are in dashed edges and essential pointers in solid edges. Bold dashed lines represent double links in each list.

4.2.2.1 Resource Lists

GCList, CursorList and FontList These are simple doubly-linked lists. For fonts, however, we store additional information. Font data is stored in UNIX files that can be located in different directories in the UNIX file system. The particular directory for a font is specified by *SetFontPath* request. The packet switch process's data structure for the first font resource to use a font in this directory will contain the *SetFontPath* request. This request will be sent to the new packet translator process before any *OpenFont* request expecting this directory is sent.

WindowList Windows are maintained in a tree structure with the server's root window at the root of the tree. Because a window can have a variable number of children, each window structure has a single pointer to a doubly-linked list of its children (other windows). In addition, each window then has forward and backward pointers to its siblings. Figure 4.3 shows a window tree structure and the internal representation of the tree.

ColormapList Data structures for colormap resources are stored in `ColormapList`. Each colormap structure contains a list of all the requests to perform an operation

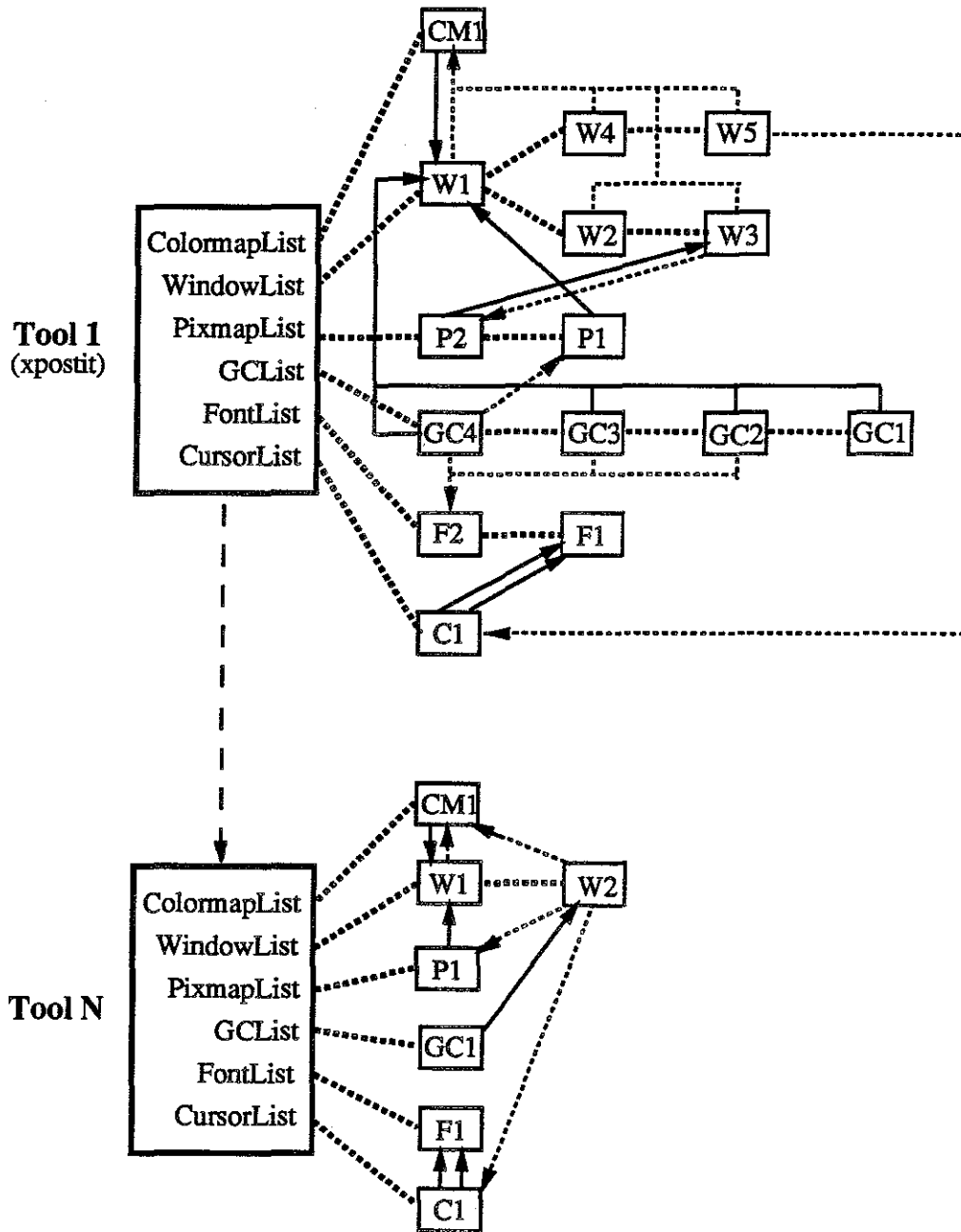
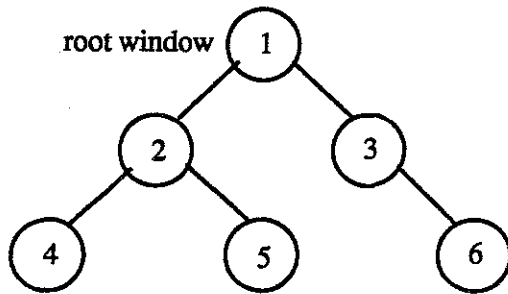
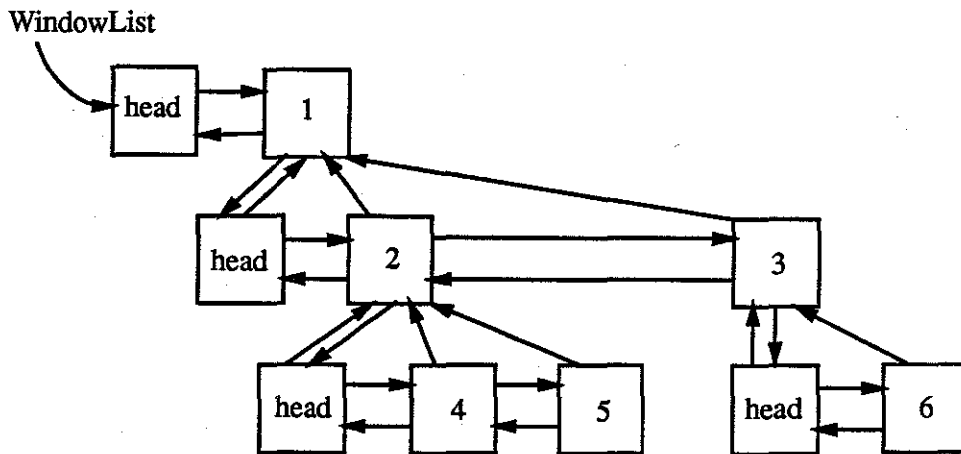


Figure 4.2: An instance of data structures for tools maintained inside XTV



(a) window tree structure



(b) internal representation

Figure 4.3: WindowList

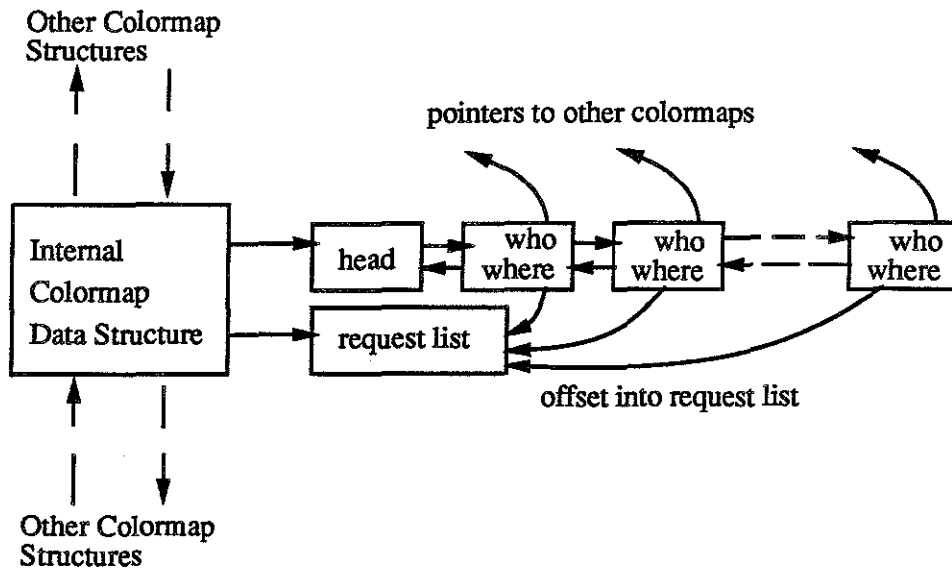


Figure 4.4: A node in ColormapList

on the colormap. These requests include *CreateColormap*, *CopyColormapAndFree*, *AllocColor*, and *AllocNamedColor*. In addition, each structure maintains a doubly-linked list of other colormaps that depend on this colormap (generated by *CopyColormapAndFree* requests). An offset from the beginning of the request list is kept to the place where the request *CopyColormapAndFree* should be sent. Figure 4.4 illustrates the interconnections of these data structures.

When the packet switch process generates requests related to a specific colormap (taken from the colormap's request list), it checks the next node in the list of dependent colormaps so that it can generate *CopyColormapAndFree* requests at the right place. After generating all requests for a colormap, the packet switch process advances to the next colormap on *ColormapList*.

PixmapList The list of pixmap data structures has a different organization because of the way pixmaps can be used. A pixmap's contents typically does not change over time. However, it is possible for a client to use a pixmap for different drawings. For example, the client may first draw a pattern on the pixmap to use as the source of a cursor, and then later erase the pixmap's contents and draw a different pattern on it to use as the background of a window. According to the X

Window System documentation, the effect of drawing into a pixmap after it was assigned to an attribute of a window or graphics context is not defined. The server may or may not save the pixmap contents in a private store. Cursors, however, are an exception. The shape of the cursor does get saved by the server on the cursor's creation. Therefore, if a pixmap is used in a cursor, and later used by others for a different drawing, the contents used for the cursor should be recorded in addition to the current contents.

Although in general the reuse of pixmaps is undefined, we have chosen to deal with this problem. We record the drawings of a pixmap used for windows and graphics contexts as well as cursors. We distinguish between two different types of pixmap dependencies: dependencies on the contents of a pixmap, and dependencies on the pixmap itself. For the case of dependencies on the contents of a pixmap, a doubly-linked list of image structures is maintained in each pixmap structure (see Figure 4.5). Each of the pixmap image structures stores a different pixmap image, and the most recent image is contained in the image structure at the front of the doubly-linked list. Each image structure also has a linked list of information on what other resources depend on the image. Pointers from other resources that represent dependency relations are to the pixmap image structures rather than to the main pixmap structure.

When generating requests for updating a latecomer's server, the following method is used to update resources that depend on the images in the pixmaps. Instead of traversing the doubly-linked list in each pixmap structure, we traverse the elements in *CursorList* in the order of their creations (from the back of the list to the front). For each cursor that uses pixmaps as the source and the mask, we follow the pointers in the cursor structure that represent the dependencies on the pixmap images. For each pixmap image structure we arrive at in this manner, we check if the current image structure and any previous ones (i.e., pixmap image structures behind the current one) need to be "processed". By processing we mean first generating *PutImage* request using the pixmap image available in a pixmap image structure, then generating requests such as *ChangeWindowAttributes* and *ChangeGC* to up-

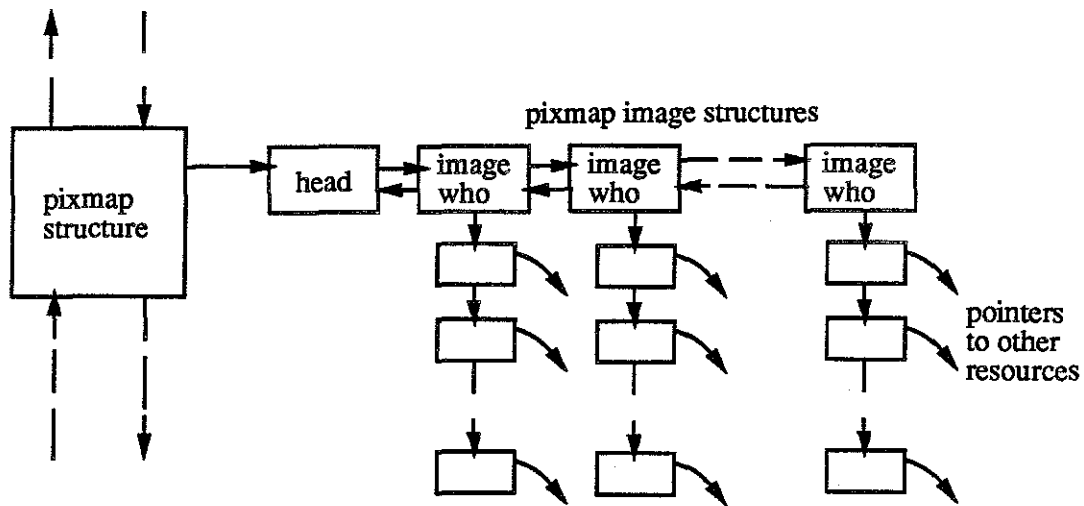


Figure 4.5: A node in PixmapList

date resources that depend on the image in the image structure, and finally marking the pixmap image structure as processed. Note that except for the current pixmap image structure, these “unprocessed” structures should have only windows or graphics contexts depending on them. This is because the “outmost loop” in this whole process traverses through `CursorList` in chronological order, and hence the first previous image structure involving a cursor must have been taken care of before along with other image structures that are behind it. The two pixmaps used for the current cursor should be good for use after processing those image structures traversed from both ends of the two pointers.

The `CreateCursor` request is now generated. The pixmap image structures are marked as “unprocessed” after all requests have been generated for updating a latecomer’s server.

The images stored in pixmap image structures are acquired from the server as late as possible, that is, when the packet switch process cannot avoid getting the images. The packet switch process must get the image in a pixmap when:

- a graphics request arrives for a pixmap and there is a dependency on the current image in the pixmap,
- a request to free a pixmap arrives and there is a dependency on the current

image in the pixmap, or

- we need the current image in a pixmap for generating requests to update a latecomer's server.

This scheme minimizes the number of *GetImage* requests because there are cases where dependency relationships on pixmap images are changed. If an image were acquired everytime a dependency on a pixmap image is created, we may have to delete the image data when the dependency is deleted and a new dependency is created on a different contents of the pixmap.

As an aside, when getting the image of the pixmap from the server, the packet switch process uses the connection to the server that it opened on behalf of the tool in question rather than the connection it uses for its own communication with the server (see Figure 3.2). This is because the X Window System guarantees in-order execution of requests only for each client. If we use the connection established for the packet switch process, a race condition may exist between a client's request that modifies or frees the pixmap, and packet switch process's *GetImage* request. As a result of this race condition, we may either get the wrong image or an error message (because the desired pixmap no longer exists). If we use the client's connection to the server to send the *GetImage* request, then the race condition is eliminated.

4.3 Performance

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.

4.3.1 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 4.6 shows the percentage of the time spent in `ArchivePacket` to the time spent in `HandleIncomingClientPacket`. Figure 4.6 (a) gives this percentage for `idraw` — a client that generates a large number of graphics requests. Figure 4.6 (b) gives the percentage for `xterm` — a client which emulates a terminal in the X Window System and also generates a large number of graphics requests (especially text-writing 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. Both graphs in Figure 4.6 show the overhead for a conference with one and four conferees. The observation interval for `idraw` was approximately 3 minutes 30 seconds for one conferee and 5 minutes for four conferees. The interval for `xterm` was approximately 4 minutes 20 seconds for one conferee and 10 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 case of `idraw` (Figure 4.6 (a)), the client first sends some query requests to the server to get information on 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. Since idraw creates many more pixmaps than xterm, it forces the packet switch process to spend more time in resource recording. Note also that xterm begins the interaction phase much earlier than idraw because it does not have as many resources to create.

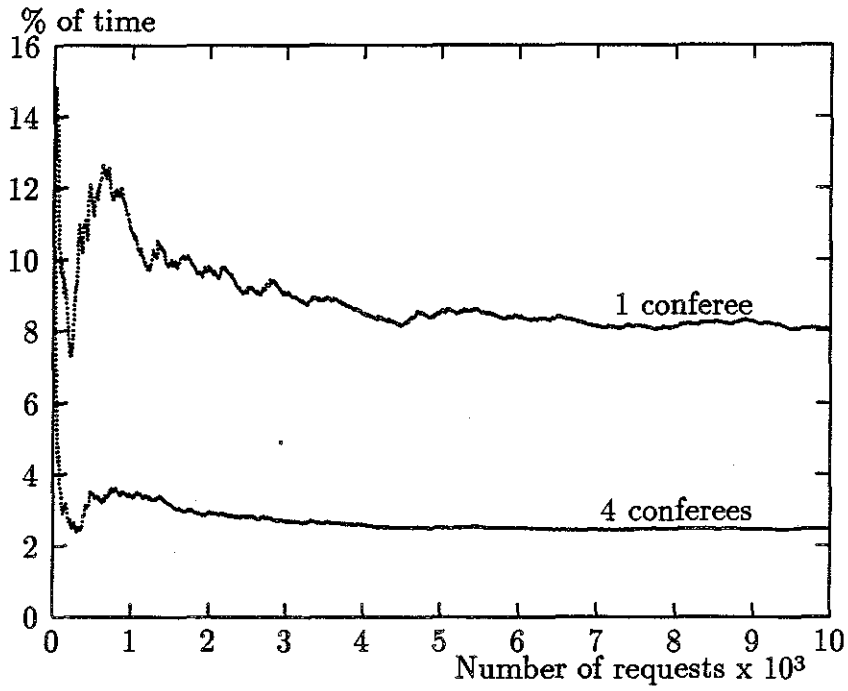
If we assume that in practice the number of conferees is more than three, then in the limit, resource recording overhead accounts for approximately two percent of the overall message processing time in the packet switch process: i.e., it adds an insignificant overhead to performance as observed by the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

4.3.2 Memory Requirements

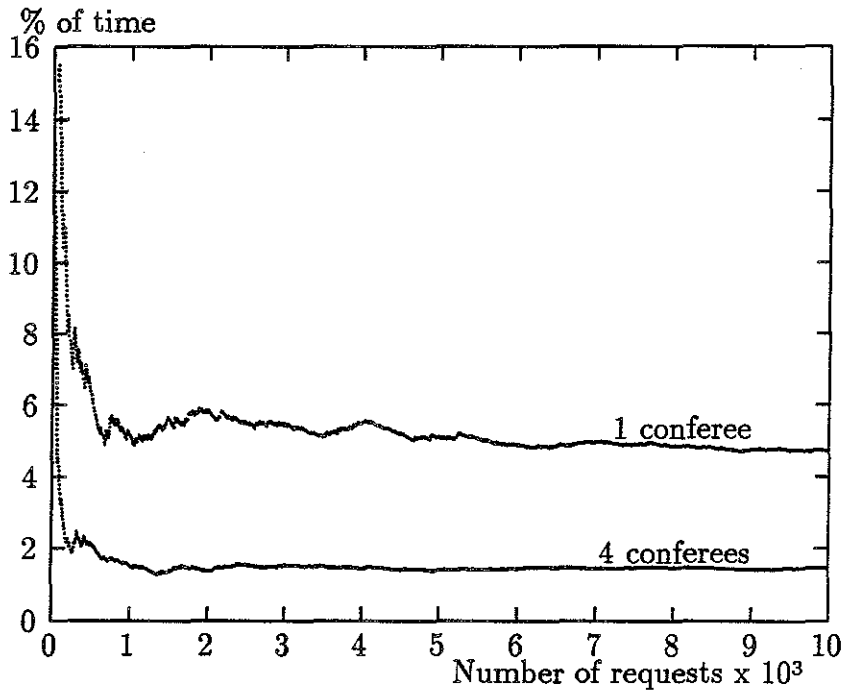
Figure 4.7 illustrates the memory requirements for maintaining the state of a client's resources using our methodology. The dotted lines represent the total number of bytes of requests sent by the client, while the solid lines represent the number of bytes used for recording state information. For both xterm and 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 savings 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 4.7 (a) are worth noting. Idraw creates and later destroys a large number of resources everytime the user pulls down a menu. This behavior accounts for the small spikes in the memory requirement curve. The large spike near request 9,000 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

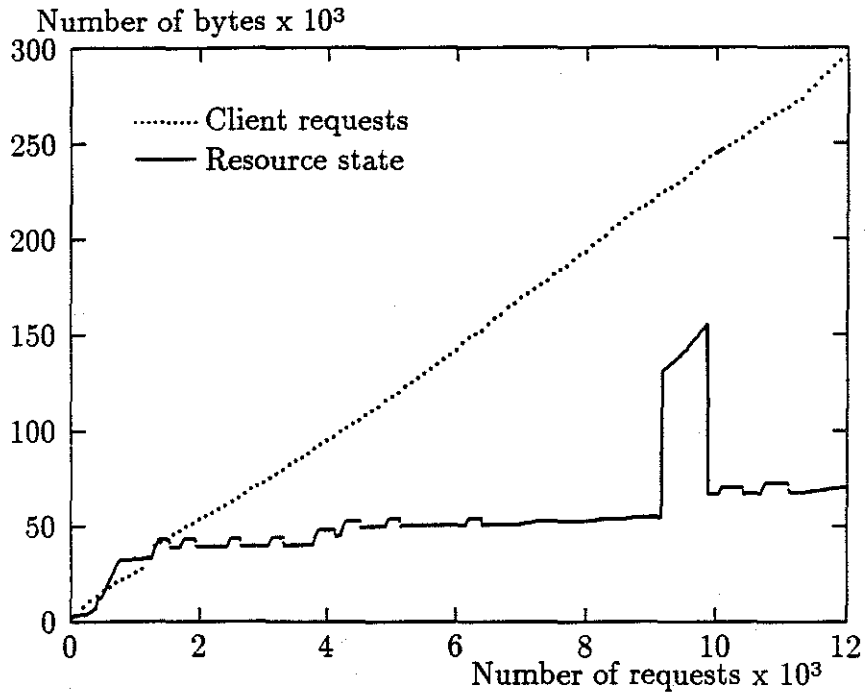


(a) idraw

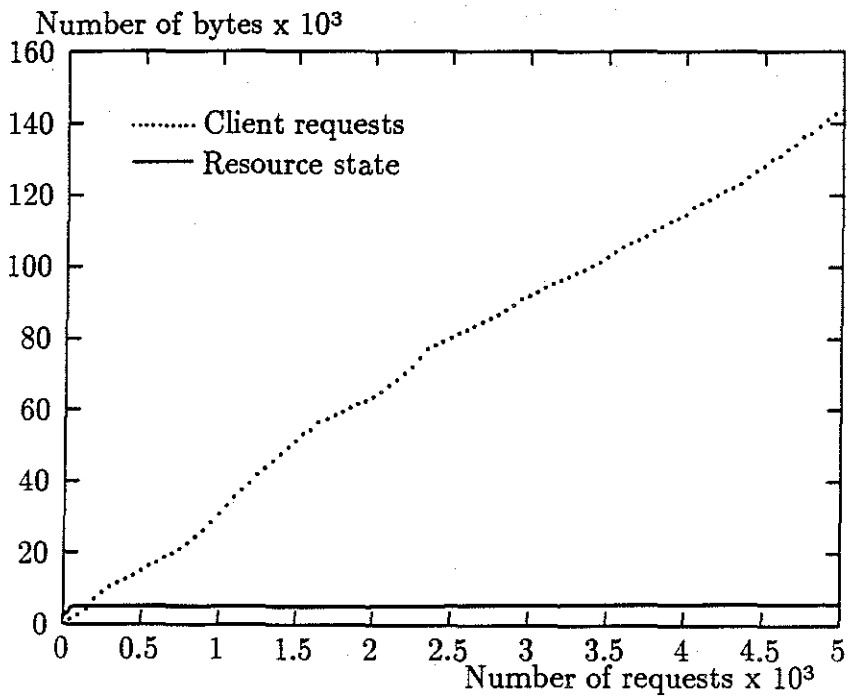


(b) xterm

Figure 4.6: Percentage of time spent maintaining resource state information vs. number of client requests



(a) idraw



(b) xterm

Figure 4.7: Memory usage for recording resource information

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.

4.4 Hard Problems

This section describes two of the more difficult and unsolved problems encountered in the process of accommodating latecomers. These problems are difficult because X does not directly support the use of clients by multiple users. It is largely because of this that we chose not to solve these problems.

4.4.1 Images in Windows

Images in windows do not have to be recorded because the client is expected to refresh the contents of its windows when expose events are sent from the server. There are, however, some exceptions. In rare cases, where it is extremely hard for the client to refresh the window's image, the client can request the server to do the refresh. The client requests this service by setting an attribute called *backing-store* on its windows. The server then stores the contents of the windows and refreshes each window when it would have normally generated an expose event. The implication is that there may exist clients that depend solely on the server's backing store feature and hence will not handle expose events. This is a problem since we have based our scheme for recording the state of window resources on the fact that the server would generate, and the client would handle, an expose event².

In cases where we cannot depend on the client refreshing the contents of a window, there is a way to get the contents of a window from the server using the *GetImage* request. This approach, however, is limited since the *GetImage* request

²Note that the backing store attribute is not widely used for most windows since the X Window System definition states that it is possible for the server to stop providing the backing store feature at any time and that all clients are expected to be ready to refresh the contents of their windows.

requires that the window in question be viewable (i.e., the window and all of its ancestors are mapped) and fully contained within the root window (i.e., fully visible if there were no descendant or overlapping windows). If a conferee places a tool's window off the console display, then there is no way we can get the complete contents of the tool's window.

4.4.2 Clients Changing others' resources

In the X Window System, a client may modify the attributes of resources created by another client. If this occurs, then the changes made by clients not in a conference on the resources maintained by the packet switch process will not be reflected on a latecomer's server. This is a difficult problem to solve because there are not sufficient X query requests to get such information in the X Window System.

4.5 Status and Use

Our method for accommodating latecomers has been successfully integrated into XTV and works well for clients such as idraw, xterm, xclock, xcalc and bitmap. By providing this feature to XTV, XTV has gained flexibility in how it may hold and maintain a conference.

The feature to accommodate latecomers in XTV is enabled by default when XTV is first invoked on the machine of the host of the conference. The feature can be disabled using a flag at the invocation. If the feature is enabled, an additional function is called inside a routine that receives and examines packets from the clients. The additional function is responsible for maintaining the changes to the server state for each client on the basis of each packet received. Consider a conference in which some number of clients (tools) have been used. When a new conferee wants to join the conference, she executes XTV with proper arguments in order to connect to the packet switch process of the conference. It is possible for any number of new participants to join the conference simultaneously. As soon as it receives the connection, the packet switch process sends all the requests generated from the state

information held in data structures for each tool. The new packet translator process handles these requests just like any other requests received from the packet switch process, except that it acknowledges each request. When the packet switch process completes the sending of packets it generated, the new packet translator process works just like any other normal packet translator processes. The new conferee is now indistinguishable from the original conferees. She may provide input to the remote clients and see the effect of other conferee's actions.

XTV is being used in the Department of Computer Science at University of North Carolina-Chapel Hill for research into cooperative work among students and faculty. Students and faculty members use personal workstations (some augmented with audio/video links).

4.6 Summary

The dependency relations among the resources are represented using pointers and counters in data structures created for holding information about the resources. The data structures for resources are grouped into doubly-linked lists according to their classes. We have demonstrated that our method can efficiently (in terms of time and space) capture the necessary state of resources and create this state in a latecomer's server.

While many problems are solved, there are some problems which are difficult. One is acquiring the image of a window solely depending on server's backing store feature, and another is clients modifying attributes of shared resources. These problems are due to the design of the X Window System assuming only single-user applications.

Chapter 5

Conclusion

5.1 Summary & 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 system. In this thesis, the problem of accommodating a latecomer to an XTV conference already in progress is studied. XTV is an X window-based shared window system employing a centralized architecture for the distribution of the output of applications used in the conference. The objective of our study is to ensure that a latecomer (a new participant) is equivalent to an original participant with respect to conference input and output. Specifically:

- The new participant can see the output of the applications used in the conference through shared windows displayed on her workstation.
- The new participant can provide input to the shared applications.
- Any subsequent operations by the shared applications have the desired effects on the X display server on the new participant's workstation (as well as on other workstations in the conference).

These goals are 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. The collective environment created on a server by the applications used in a conference is called the server state. Each X application modifies the server state by creating private resources (e.g., windows) or by making changes to attributes of resources (private or non-private). 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.

This approach is complicated by dependency relations that exist among resources created or used by applications in the conference. A dependency relation is formed when a resource A has another resource B as one of its attribute values. X servers record the attributes associated with resource A in their internal data structures, and hence, clients may later free resource B without affecting A 's attributes. The problem is that the freed resource B must be created on a latecomer's server before any resource depending on it (A in this example) can be created. Therefore, when the request to free resource B arrives, XTV should not delete the information it has stored on resource B . The dependency relations are used to prevent XTV's internal data structure for a freed resource from being deleted when there exist other resources that depend on the freed resource. The dependency relations are identified by analyzing request messages sent from clients. Internally, dependencies are represented as a directed graph where nodes represent resources and edges represent dependencies. The problem of recording the server state for a client thus is reduced to a graph maintenance problem. Algorithms to record the server state using minimum size graphs are also introduced in this thesis.

When a latecomer joins the conference, the recorded state is projected onto the new participant's server by sending request messages to this server to create the private resources with current attributes and to modify (change the attributes of) non-private resources. These request messages are generated from the resource and attributes value lists maintained inside XTV in such a manner that dependency

relations are not violated. (I.e., no resource is created on the new server before all the resources it uses have been created.) After all the generated request messages are sent to the new server, the latecomer can share the applications used in the conference.

The method presented in this thesis has been demonstrated to accommodate a latecomer in a practical and efficient manner. Saving all the request messages sent by the shared applications can consume a lot of memory, and hence the practical use of such a scheme is limited. 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 Section 4.3, the execution time overhead of recording the server state is quite tolerable for conferences of reasonable size. For example, with 4 participants the recording process increases XTV's overhead by 2% for an interactive drawing program — an application that extensively modifies the state of the server — while consuming only a fifth of memory required to record all request messages. Moreover, as the conference progresses, the memory required to store the state of client resources grows at a near zero rate.

5.2 Future Work

5.2.1 Replicated Architectures Revisited

As the method described in this thesis was developed on top of XTV, output messages from each application (destined for the X server) are used to record the modifications made to the server state by the application. While useful, this approach is not well suited for conferencing systems that employ a replicated architecture. In a replicated architecture, the messages coming from the servers (destined for the application) have to be examined in order to infer the state of each shared application. In a replicated architecture, it is the state of the shared applications that will be projected onto a new copy of the same application when a latecomer joins

the conference. As discussed in Chapter 2, it is difficult to efficiently maintain the desired state in this case.

If the definition of a replicated architecture can be expanded, then the method introduced in this thesis can be used in a replicated architecture. The resulting hybrid architecture can be described as follows. The conference is maintained in the “pure” replicated architecture until a latecomer joins the conference. While the conference progresses, a conference agent records the modifications made to the server state using the method described in this thesis. All latecomers contact this agent process, who in turn sends requests to update the latecomers’ servers. Latecomers and the conference agent form a sub-architecture that is centralized. In this manner, latecomers share the copies of applications used by the conference agent instead of having local copies of the shared applications executing on this workstation.

5.2.2 Extensions to Existing X Servers

Although the approach suggested in this thesis provides an efficient method for accommodating latecomers, it duplicates an identical set of information and functions contained in an X server (albeit the server maintains this information for all X application that it communicates with). It would therefore be desirable to combine the capability to accommodate a latecomer into the window system itself; specifically the ability to obtain the dependency graph for an application. 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. The server, in response, would send the appropriate information. This information would be identical to that maintained for each application in our method (and possibly more accurate due to the problems mentioned in Section 4.4.2). There should also be another special request message to impose the changes (e.g., “*PutClientsServerState*”) on a second server. This request would

be sent by the conference agent to a latecomer's server.

In order to provide these services, the server should maintain its state on application-by-application basis. The collective sets of information maintained by our method for each application form the collective server state. If this is implausible, a method should exist to identify the contribution to the server state modifications by each application.

Ultimately, the collaboration-aware window systems should be developed. These combine all the functionalities of the conference agents, thereby making possible the direct communications between servers and clients.

Bibliography

- [Abdel-Wahab et al. 88] Abdel-Wahab, H. M., Guan, Sheng-Uei, Nievergelt, J., [1988] Shared Workspaces for Group Collaboration: An Experiment using Internet and UNIX Interprocess Communications, *IEEE Communications Magazine*, Vol. 26, No. 11, 10-16 (November 1988).
- [Abdel-Wahab & Feit 91] Abdel-Wahab, H. M., Feit, M. A., [1991] 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).
- [Ahuja et al. 90] Ahuja, S. R., Ensor, J. R., Lucco, S. E., A comparison of application sharing mechanisms in real-time desktop conferencing systems, *Proceedings, IEEE Conference on Office Information Systems*, 238-248 (April 1990).
- [Comer 90] Comer, D., [1990] *Internetworking with TCP/IP: Principles, Protocols and Architecture*, Second Edition, Prentice Hall, (1990).
- [Crowley & Forsdick 89] Crowley, T., Forsdick, H., [1989] MMConf: The Diamond Multimedia Conferencing System, *Proceedings, Groupware Technology Workshop*, IFIP Working Group 8.4, (August 1989).
- [Ellis et al. 89] Ellis, C., Gibbs, S. J., Rein, G. L., Design and use of a group editor, *Proceedings, Working Conference on Engineering for Human-Computer Interaction*, IFIP Working Group 2.7, (August 1989).
- [Ensor et al. 88] Ensor, J. R., Ahuja, S. R., Horn, D. N., Lucco, S. E., The Rapport Multimedia Conferencing System - A Software Overview, *Proceedings, IEEE Conference on Computer Workstations*, Santa Clara, 52-58 (March 1988).
- [Lantz 86] Lantz, K. A., [1986] An Experiment in Integrated Multimedia Conferencing, *Proceedings, CSCW 86 Conference on Computer-Supported Cooperative Work*, MCC Software Technology Program, 267-275 (December 1986).
- [Lantz et al. 89] Lantz, K. A., Lauwers, J. C., Arons, B., Binding, C., Chen, P., Donahue, J., Joseph, T. A., Koo, R., Romanow, A., Schmandt, C.,

Yamamoto, W., Collaboration technology research at Olivetti Research Center, *Proceedings, Groupware Technology Workshop*, IFIP Working Group 8.4, (August 1989).

- [Lauwers et al. 90] Lauwers, J. C., Joseph, T., Lantz, K. A., Romanow, A., Replicated Architectures for Shared Window Systems: A Critique, *Proceedings, IEEE Conference on Office Information Systems*, 249-260 (April 1990).
- [Lauwers & Lantz 90] Lauwers, J. C., Lantz, K. A., [1990] 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).
- [Linton et al. 89] Linton, M. A., Vlissides, J. M., Calder, P. R., [1989] Composing User Interfaces with InterViews, *IEEE Computer* Vol. 22, No. 2, 8-22 (February 1989).
- [Nye 89a] Nye, A., [1989] *X Protocol Reference Manual for Version 11*, Volume 0, O'Reilly & associates, Inc., Sebastopol, CA (1989).
- [Nye 89b] Nye, A., [1989] *Xlib Programming Manual for Version 11*, Volume 1, O'Reilly & associates, Inc., Sebastopol, CA (1989).
- [Patterson 90] Patterson, J. F., The Good, the Bad, and the Ugly of Window Sharing in X, *Proceedings, 4th Annual X Technical Conference*, (January 1990).
- [Sarin & Greif 84] Sarin, S. K., and Greif, I., Software for interactive on-line conference, *Proceedings, 2nd Conference on Office Information Systems*, 46-58 (June 1984).
- [Smith et al. 90] Smith, J. B., Smith, F. D., Calingaert, P., Hayes, J. R., Holland, D., Jeffay, K., Lansman, M., UNC Collaboratory Project: Overview, *Technical Report*, Chapel Hill, North Carolina, (November 1990).
- [Stefik et al. 87] Stefik, M., Foster, G., Bobrow, D. G., Kahn, K., Lanning, S., Suchman, L., Beyond the chalkboard: Computer support for collaboration and problem solving in meetings, *Communications of the ACM*, 30(1):32-47, (January 1987).
- [Ullman 88] Ullman, J. D., [1988] *Principles of Database and Knowledge-base Systems*, Volume 1, Computer Science Press, 34-42 (1988).
- [Watabe et al. 90] Watabe, K., Sakata S., Maeno K., Fukuoka H., Ohmori T., Distributed Multiparty Desktop Conferencing System: MERMAID, *Proceedings, CSCW 90 Conference on Computer-Supported Cooperative Work*, (October 1990).