Protocol Enhancement and Compression for X-Based Application Sharing

Diplomarbeit
von
Martin Mauve
aus
Hirschhorn

vorgelegt am
Lehrstuhl für Praktische Informatik IV
Prof. Dr. Effelsberg
Fakultät für Mathematik und Informatik
Universität Mannheim

März 1997

Betreuer: Dr. Rainer Storn

Ehrenwörtliche Erklärung

Hiermit versichere ich, die vorliegende Diplomarbeit ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus den Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Mannheim, den 24. März 1997 Martin Mauve

Acknowledgments

This thesis was written at the International Computer Science Institute (ICSI) in Berkeley. The ICSI provided me with an friendly and helpful environment where research is professionally organized. Numerous discussions with staff and visitors of the ICSI had a significant impact on this thesis.

My advisor, Rainer Storn, showed me how to work on a master’s thesis and supported me in many other ways. Andres Albanese, the leader of ICSI’s networking group, helped me to find the right people to discuss my work with.

Siemens made this thesis possible by letting me use the source code of their application-sharing software. Wladimir Minenko answered numerous questions about this software.

Martin Mauve,
Mannheim
1997

Table of Contents

List of Figures

List of Tables

Abbreviations

AS	Application Sharing
CCITT	Comite Consultatif Internationale de Telegraphie et Telephonie
client	X Window System client
CSCW	Computer-Supported Cooperative Work
CSLIP	Compressed Serial Line Internet Protocol
DDX	Device Dependent X (layer)
DIX	Device Independent X (layer)
FHBX	Fast Higher Bandwidth X
HBX	Higher Bandwidth X
IP	Internet Protocol
ISDN	Integrated Services Digital Network
LAN	Local Area Network
LBX	Low Bandwidth X
LZW	Lempel-Ziv-Welch (compression algorithm)
master	master pseudo server
NCD	Network Computing Devices
OS	Operating System (layer)
PPMC’	Prediction by Partial Match, method C’
PPP	Point-to-Point Protocol
RGB	Red, Green, Blue
SDC	Structured Data Compression
server	X Window System server
slave	slave pseudo server
STC	Siemens Telecooperation Center
TCP	Transmission Control Protocol
X	X Window System

1 Introduction

With a rapidly growing demand for expert knowledge and the tendency to work in globally distributed teams, computer solutions for cooperation and information exchange become more and more important. This area of research is known as Computer-Supported Cooperative Work (CSCW) [17] and should lead to the development of adequate computer tools to make cooperation between humans easier and more efficient.

One part of CSCW deals with synchronous distributed cooperation tools, e.g. tools that are used simultaneously by different persons in geographically different places. The following scenario gives an impression of how those tools could support human interaction and cooperation:

> *The system administrator Reak just encountered a problem while installing new software. Knowing that it might take him days to solve the problem, he decides to get some expert advice. With a mouse click, he opens his conference manager. After consulting his electronic phone book, he establishes a connection to a company specialized in answering questions about the product he was attempting to install. Immediately a video pops up on Reak's screen and an expert asks him to describe his problem. Reak explains the problem and sends the window he was using for the installation to the screen of the expert. In order to discover the problem, Reak and the expert take turns entering commands into the window they now share. Finally they decide that they need to modify some system files to get the software running on Reak's computer. Reak sends his editor to the screen of the expert, who in turn suggests some changes to the system files. Together they edit the files and try once more to install the software. The software is now working. Reak takes back the installation and editor windows from the expert's screen and disconnects using his conference manager.*
>
> *Several hours later, Reak finishes his work on a document that proposes a new network structure for his company. He wants to get some input on his proposal from fellow administrators in different branches. Again using the conference manager, he establishes a connection to them. He sends the window of his desktop publishing tool to the screen of the conference participants. Explaining his proposal, he points to parts of the document with a pointer which is shared between all participants (telepointer). After Reak explains his view, the conference participants take turns in modifying the document, viewing and discussing these modifications.*

Examining the tools the person in the scenario was using, one can distinguish between tools that were explicitly developed for cooperative use and those that were developed for a single user. The conference manager, audio, video and the telepointer belong to the first group, while the window for the installation, the editor and the desktop publishing tool belong to the second. Those groups are known as cooperation aware and cooperation unaware tools, respectively [25].

While it is conceivable that in the distant future the majority of tools are developed as cooperation aware tools, most applications today and in the near future will belong to the cooperation unaware group. In order to make those applications available to cooperative work, a special software is necessary. This software makes it possible that unmodified single user applications are displayed on more than one screen simultaneously and receive input from more than one user. To give the user the feeling of a true cooperative environment, any cooperation unaware application should be sharable at any time with any user. A software that makes this possible is called application-sharing software. Consequently application sharing (AS) is defined as “... a technology which allows two or more users located at geographically different places to synchronously work with a single-user application, i.e. online and at the same time” [25].

The first system for which application-sharing solutions were developed is the X Window System Version X11 [22]. It is the first window system that is based on a network protocol between application and window system instead of function calls. It also has a number of other attributes interesting for application sharing, such as hardware transparency, freely available source code and a huge number of cooperation unaware applications using X Window.

Virtually all of the application-sharing tools developed for X Window use the X protocol to display the output of a single user application on more than one screen and to receive response from more than one user. However, this protocol was developed for usage over a local area network with low latency and high bandwidth. Returning to the scenario, it would be a common case that the expert is not located in the same area as the one who seeks advice. In fact CSCW should especially support cooperation between users that are separated by long distances, maybe even working on different continents. Another common usage would be application sharing over low-bandwidth links. In the scenario one of the administrators might currently be at home, connected only via an ISDN line.

A standard X Window application like Netscape needs more than 450 synchronous protocol requests and it transfers well over 450 kbyte of data just at start-up of the application. With either a high-latency link (200+ ms round-trip delay) or a low-bandwidth connection (< 64 kbit), this leads to a delay of 60–90 seconds! The number of synchronous requests and the amount of data sent make the usage of the basic X protocol unsatisfying for those scenarios.

While there have been some efforts to optimize and compress the X protocol for high-latency and low-bandwidth links, up to now there exists no integration of these efforts into application-sharing tools.

Objectives of this thesis

This thesis is about protocol enhancement and compression for X-based application sharing. The main objectives are as follows:

- give an overview over the existing efforts to enhance and compress the X protocol,
- show that those mechanisms can be integrated into existing application-sharing products by providing a prototype integration,
- measure the compression rate and the reduction of synchronicity achieved with the integration and
- identify areas of future work.

Chapter Overview

The second chapter gives an introduction to the X Window system. It explains the basic concepts of the X Window architecture, the X Window protocol, what an X server is and how the sample X server distributed with X works. Chapter Three starts with an introduction to the different approaches for application sharing under X Window. It then describes in more detail the application-sharing tool (XpleXer) that was chosen for the integration. The final part of Chapter Three shows the problems of application sharing over high-latency and low-bandwidth links. Chapter Four summarizes existing efforts to enhance and compress the X Window protocol for high-latency and low-bandwidth connections. It describes why one tool (Low Bandwidth X) was preferred for the integration and under which circumstances a different tool should be chosen. The fifth chapter deals with the prototype integration of XpleXer and Low Bandwidth X. It proposes an architecture for the prototype and shows which Low Bandwidth X features could be integrated and how they where changed to fit into the new environment. This chapter also evaluates the compression rate and the reduction of synchronous protocol requests achieved by the integration. The sixth chapter is dedicated to areas of future work and research that were identified while working on this thesis. It shows that there are numerous ways to further increase the compression rate and to enhance the elimination of synchronicity. A summary of this thesis can be found in Chapter Seven.

2 The X Window System: An Overview

This chapter gives an overview of the X Window System. It will explain only the concepts and terminology necessary to understand the remaining chapters of this thesis. For a detailed explanation of the X Window System, the X protocol and the X server, see [27], [26] and [20].

2.1 X Window System Basics

2.1.1 Architecture

The X Window System Version X11 (X) is a network oriented, hardware independent, graphical window system. Its source code is freely available. The current release of X is R6; most application-sharing products, mentioned in this thesis, were written for R5. Figure 1 shows the basic architecture for X: The user receives output from the applications through one or more bitmapped screens. Input is entered with various input devices such as mouse and keyboard. Together screens and input devices are called a display. One display is managed by a piece of software known as the X server (server). The server gets requests from application programs — the X clients (clients) — and acts accordingly. A typical request from a client could be: draw a square on the screen. The server would receive this request, translate it into pixels on the screen and draw those pixels in the desired color. Answers to information requests, error reports and events (like mouse movement) are sent from the server to the client.

In the X Window System version X11, the communication between client and server is based on a protocol rather than on function calls. This allows the user to run multiple clients on different computers, using only one display for in- and output. The protocol used for the communication between client and server is called X protocol.

To shield application programmers from the X protocol, the XLib was developed. XLib is the C application programming interface to X. Using XLib, the application programmer can develop software for X in a way similar to developing software for a non-network-based window system. Knowledge about the X protocol is not needed.

As a window system, X defines a window as a rectangular area of the screen. The X server maintains a data abstraction for every window. This data abstraction has different attributes, like width and height, which are called window attributes. Windows are the basic building blocks for every graphical user interface under X. Buttons, menus, text fields and icons are all examples of windows.

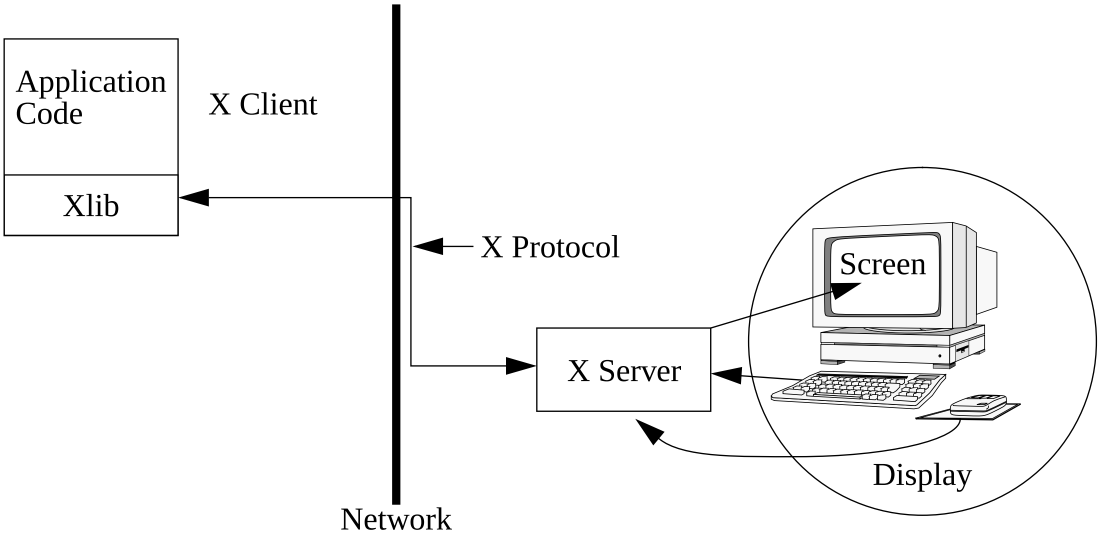


Figure 1 X Window System architecture

To coordinate and manage multiple clients and their windows, which compete for resources like screen space, a special software is needed. This software is called a window manager. Most window managers have a user interface that allows the user to resize, iconify and move application windows. For X, a window manager is realized as a client with certain responsibilities. Although X provides a sample window manager, its usage is not mandatory. Instead it is up to the user to choose a window manager he or she likes.

2.1.2 Interclient Communication

Since X clients frequently need to exchange information, X has to provide ways for interclient communication. One such method is the use of properties. A property is an arbitrary information associated with a window. In order to access this information, a client needs to know the property name, represented by a string and the window with which the property is associated. The same property name for different windows points to different information.

To save bandwidth, numerical aliases — called atoms — are used to refer to property names. Some of those atom-to-property-name mappings are predefined. Other property names don't have a standard mapping to atoms. For those properties, the client has to ask the server about the mapping: It sends the property name as a string to the server, which replies with the atom. This mapping is guaranteed to not change during the lifetime of the server. If another client asks for the mapping of the same property name, it will get the same atom as a response. The action of asking the server for an atom is known as “intern an atom”.

To set or read a property, a client specifies a pair of an atom and a window. This kind of interclient communication is often used between clients and the window manager.

2.1.3 Color Handling

X is designed to support bitmapped graphics screens. The color of each pixel on the screen is defined by the state of 1 to 32 bits, depending on the hardware of the screen. Black-and-white screens, for example, need just one bit for each pixel to represent the two possible states. Color screens have more bits per pixel. Those bits are arranged in planes, where each plane has one bit for every pixel on the screen. Figure 2 shows an eight-plane screen. The pixel in the lower right corner of the screen is described by eight bits, one in each plane. Those bits together are called the pixel value. The pixel value does not directly specify a color. Instead it describes an index to a color table where the actual RGB (Red, Green, Blue) values (usually 16 bit for each primary color) for the electron beams are located. This has the following advantage: The number of colors that can be displayed simultaneously is only 2^8. But, if the colormap is writable, the number of colors from which to choose these colors is 2^{48} (16 bit per primary color). The color map looks slightly different for grey scale and high-end graphics screens [27].

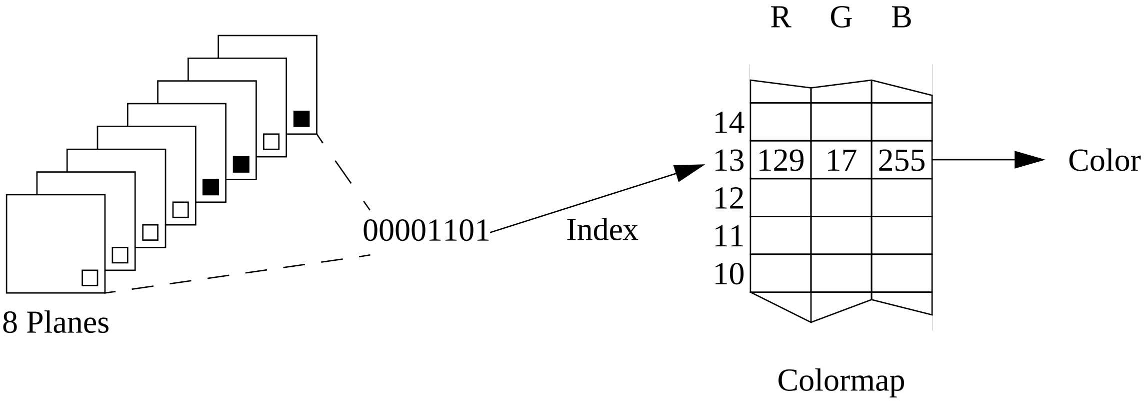


Figure 2 X Window color handling

Usually all clients of one server share one common colormap. But although most screens have just one hardware colormap, every window can potentially define its own private colormap. In this case, the window manager is responsible for loading the colormap for the right window into the hardware colormap whenever necessary. However, applications usually avoid defining their own colormaps since all windows that do not use the private colormap will be displayed in the wrong colors when the private colormap is loaded.

Each member of a colormap is called a colorcell. In the example, the colorcell for pixel value 13 has the RGB values of 129, 17 and 255.

In order to use a color, a client has to ask the server for permission. There are two different things the client can ask for:

- allocation of a shared color: The client specifies the RGB values it wants to use. The server replies with an index to a colorcell, which contains the closest RGB values physically possible on that screen. The client can use this color, but it is not allowed to change the content of the colorcell. The advantage of this method is that the colorcell can be used by more than one client at the same time.
- allocation of private colors: The client asks for a number of private colorcells. The server answers with the index of those cells. The client can then use and change the content of those colorcells whenever it wants. This prevents other clients from using the same colorcell, but it is very useful if the client needs to do color manipulation.

To use the advantage of a shared color, two or more clients have to specify exactly the same RGB values. Without further assistance it is very unlikely that two clients will ever choose the same values out of 2^{48} possibilities. To make sharing and specification easier, a server side colorname database is provided. To use it the client asks the server for information about a color described by a string. The server answers with the RGB values that matches the requested color.

2.1.4 Extensions

The X Window System is extensible. This means that there exists a standard way to enhance X by additional functionality. This additional functionality is called an extension to X. Adding an extension is done by providing new Xlib functions, developing new protocol messages and modifying the server. There exists a number of extensions to X, for example the shape extension which lets the programmer use non-rectangular windows. A server is never required to support any extensions, therefore a client must ask the server if a certain extension is present before using it. Several of the approaches for X protocol compression described in this thesis make use of the extension mechanism.

2.2 The X Protocol

> *The X protocol is the true definition of the X Window System, and any code in any language that implements it is a true implementation of X. It is designed to communicate all the information necessary to operate a window system over a single asynchronous bidirectional stream of 8-bit bytes.* [26]

The X protocol defines four different types of messages:

- request: With a request, the client asks the server to do something (like drawing a line) or to send back information (like an atom for a property name). As an example, Figure 3 shows the InternAtom request. Each request type is identified by a major request opcode included in the request. Those opcodes range from 0 to 127 for the basic X protocol. Another field common to all requests is the request length. It specifies the length of the request in multiples of 4 bytes. Padding is added to make sure that word and long word datatypes have the right alignment and that the message length is indeed a multiple of 4.

Position	Size	Value	Description
0	1	16	opcode: intern atom
1	1	BOOL	only if exists
2	2	2+(n+p)/4	request length
4	2	n	length of property name
6	2		padding
8	n	STRING	property name
8+n	p		padding

Figure 3 The InternAtom request

- replies: Replies are sent from the server to the client as an answer to certain requests. Only a small number of requests need a reply. Figure 4 shows the InternAtom reply. The type of the reply is identified by the request which caused the reply. To distinguish replies from events and errors, the server includes a 1 as the opcode in the message. The message for all replies, events and errors contain the number of the last request processed by the server — this number is called a sequence number. The sequence number makes it possible for the client to match server messages with the request that caused them. This is especially useful for debugging. The reply length is given in multiples of 4 in excess of 32 bytes.

Position	Size	Value	Description
0	1	1	opcode: reply
1	1		padding
2	2	unsigned integer	sequence number
4	4	0	reply length
8	4	ATOM	atom
12	20		padding

Figure 4 The InternAtom reply

- events: Whenever the server needs to notify the client about a change of state (like mouse movement or side effects of previous requests), it sends the client an event. Figure 5 shows the Expose event. The Expose event is generated by the server whenever an area of a window needs to be repainted by the client. Events are identified by their opcode which ranges from 2 to 34 and have a fixed length of 32 bytes.

Position	Size	Value	Description
0	1	12	opcode: expose event
1	1		padding
2	2	unsigned integer	sequence number
4	4	WINDOW	window
8	2	unsigned integer	x
10	2	unsigned integer	y
12	2	unsigned integer	width
14	2	unsigned integer	height
16	2	unsigned integer	count
18	14		padding

Figure 5 The Expose event

- errors: If the server encounters an error caused by a client request, it notifies that client by sending an error message. An error which could be generated by the InternAtom request is the Alloc error. Receiving an Alloc error tells the application that the server was not able to allocate enough memory to execute the request. The alloc error is shown in Figure 6. The type of error can be determined by the error code included in the message. The major opcode shows which request type caused the error. All errors use the opcode 0 and have a length of 32 bytes.

Position	Size	Value	Description
0	1	0	opcode: error
1	1	11	error code: alloc
2	2	unsigned integer	sequence number
4	4	0	unused
8	2	unsigned integer	minor opcode
10	1	byte	major opcode
11	21		padding

Figure 6 The Alloc error

As mentioned in the previous section, the X protocol is extensible. The major request opcodes 128 through 255 are reserved for extensions. Each extension gets one major opcode and has to include a minor opcode in each of its requests to distinguish between different requests from that extension. For extension events, X provides the opcodes 64–127 and for errors the error codes 128–255.

A request which needs a reply is called a round-trip request, all other requests are called one-way requests. Round-trip requests are synchronous, e.g., the client issuing the round-trip request waits until it receives the answer (or an error) to the request. In high-latency networks, a large number of round trips is devastating for the performance of the client. Even a well-designed client using the basic X protocol uses enough round trips to be seriously delayed over those networks.

2.3 The X Server

2.3.1 Structure of the Sample X Server

The source code of the sample server provided with the distribution of X has been developed to be easily portable to new hardware platforms. The main building blocks of the sample server are shown in Figure 7 [20].

The Device Independent X (DIX) layer contains the code that is portable across platforms. Dependencies from input and output devices are encapsulated by the Device Dependent X (DDX) layer. The OS layer contains operating system specific functions. Functions for managing fonts are combined in the Font Library.

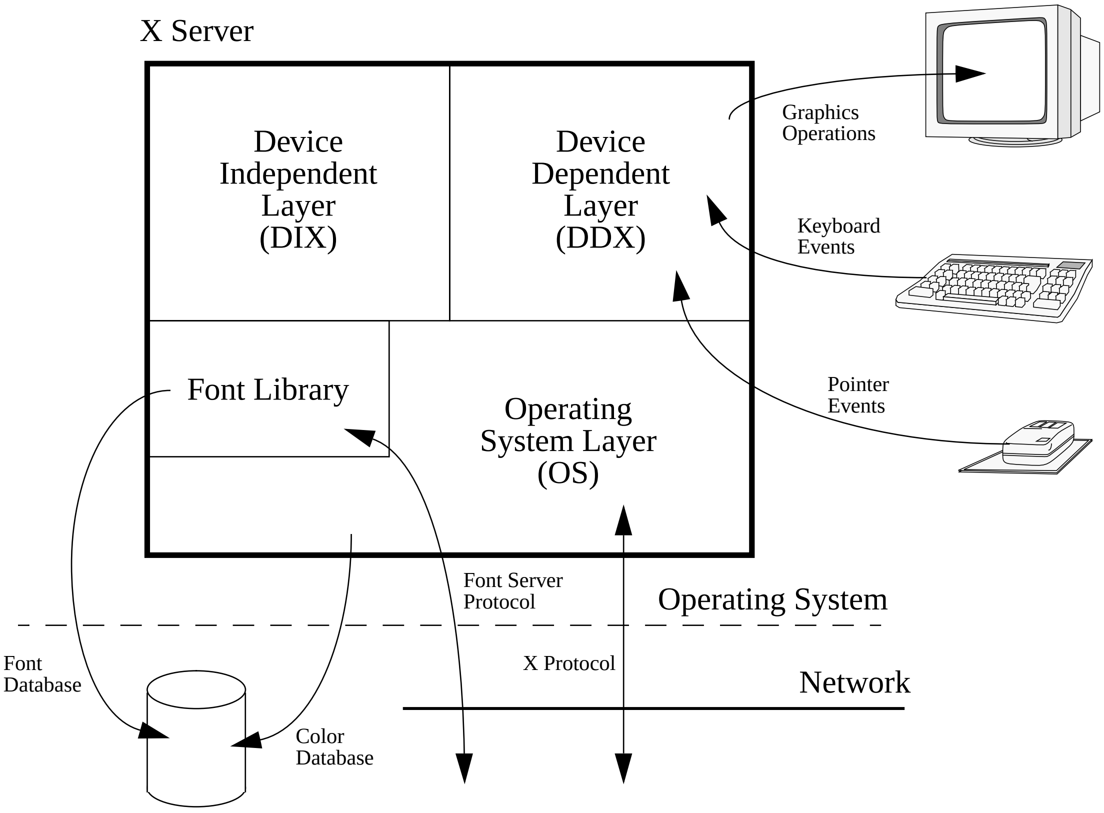


Figure 7 Structure of the sample X server

The DIX and OS layers are of special interest in the context of this thesis.

The main OS layer function is to encapsulate the communication with the clients. It manages client connections, reads client requests from the network, forwards them to the DIX layer and writes errors, events and replies from the other layers back to the client. It provides input and output buffers and tells the DIX layer when new requests arrive.

The two main tasks of the DIX layer are request dispatching and event delivery. Incoming requests from the OS layer are sent (dispatched) to the appropriate request functions. Those functions can call DDX functions (for output to the screen) or do the requested work in the DIX layer (like interning an atom). Input events are generated in the DDX layer and are forwarded to the DIX layer. Other events are generated in the DIX layer itself. In both cases the DIX layer processes those events and determines which events should be sent to which client. Then it hands the events over to the OS layer for transmission. With those two tasks, the DIX layer is the control center of the server. It directs which parts become active and how the data flows through the system.

2.3.2 Flow of Control

An example for the flow of control between the three key functions of the server is shown in Figure 8. The server is started at t_1 and begins with the initialization of the data structures in the DIX layer function `main()` before the function `Dispatch()` is called. As there are no clients yet, `Dispatch()` tells the OS layer to `WaitForSomething()` to happen. At t_2, the first client connects and starts sending requests to the server. Now `Dispatch()` distributes those requests to appropriate functions and calls `WaitForSomething()` to wait for new requests from the OS layer, or input from input devices. At t_3 the last client disconnects. This causes `Dispatch()` to return and `main()` decides to reset the server. A new cycle, called server generation begins. The first client in this generation connects in t_4, the last client disconnects at t_5 causing `Dispatch()` to return. This time `main()` decides to exit, ending the X Server process.

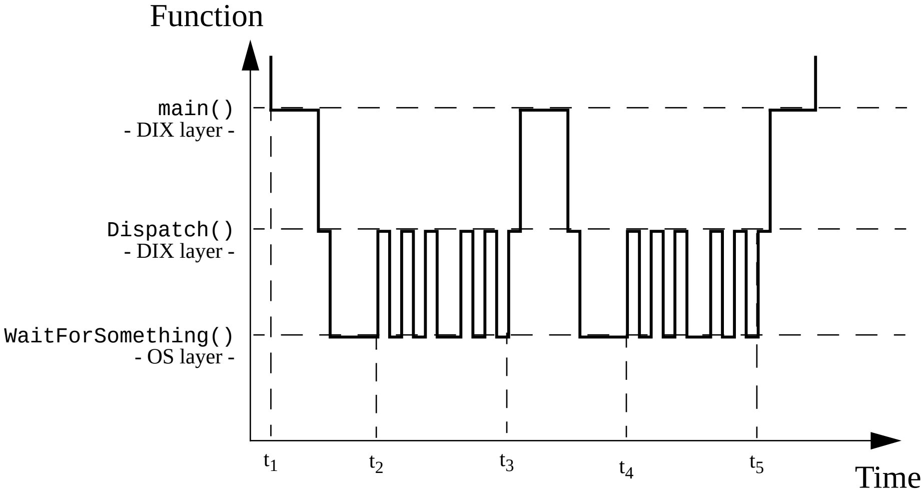


Figure 8 Example for the flow of control in the X server

The `main()` function is the starting point for the flow of control in the server. Figure 9 [20] shows that after processing command line arguments and creating server structures, the server begins with dispatching client requests. When the last client disconnects, the server structures are reset and the server is either restarted or terminated.

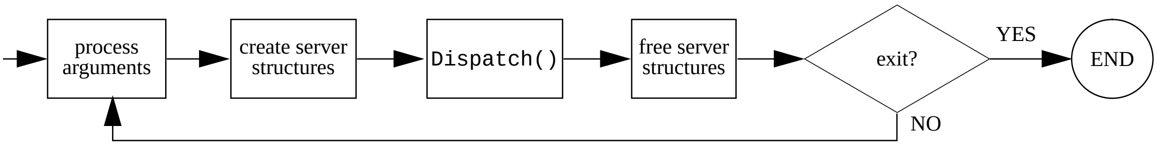


Figure 9 Flow of control in main()

The dispatch loop is the heart of the server. As shown in Figure 10 [20], it starts with the decision to return or to continue dispatching. Usually the dispatch loop ends when the last client disconnects. If there is input from the input devices, the DDX function `ProcessInputEvents()` is called to get the events. Certain input events, for example pointer movement, are required to be delivered timely. Those events are called critical output (output from the server to the client). Critical output is flushed to the client in `FlushIfCriticalOutputPending()`. After handling device input, the dispatch loop uses the OS layer function `WaitForSomething()` to sleep until new requests arrive or the next input event occurs. If some clients have requests, those requests are dispatched for one client at a time. After dispatching the requests from one client, all output to all clients is flushed.

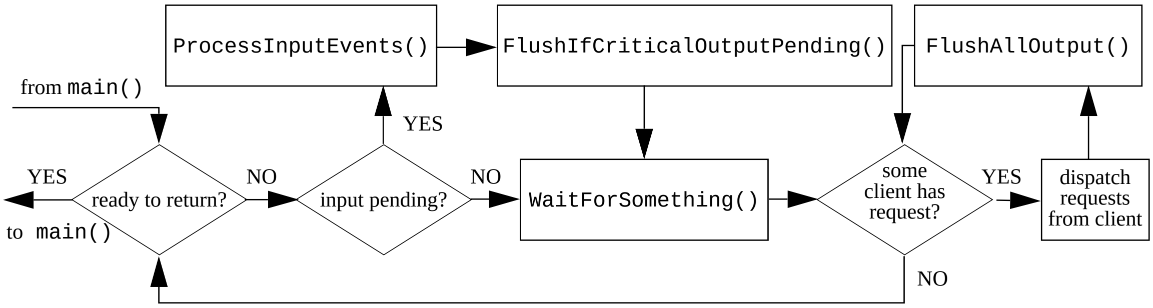


Figure 10 Flow of control in Dispatch()

Figure 11 [20] shows how requests from a single client are dispatched. The loop continues until all available requests are processed or until some other parts of the server tell `Dispatch()` to stop processing requests for that client. This is necessary to prevent a hyperactive client from dominating the server. Before reading the request from the OS layer with `ReadRequestFromClient(),` the DDX layer gets a chance to process input events and, if applicable, critical output is flushed. This is done to keep the pointer device responsive.

`ReadRequestFromClient()` can return either with or without a new request. If no request is returned and an error occurred while trying to read the requests, the responsible client is closed. If no error occurred, the loop knows that there are no more requests for that client and resumes with dispatching other clients.

After a request is successfully read, it is dispatched to the appropriate function identified by the major opcode in the request. If this function returns an error, it is examined. In the case of a fatal error, the client is closed. If the error was not fatal, it is reported to the client and `Dispatch()` continues dispatching requests for the next client.

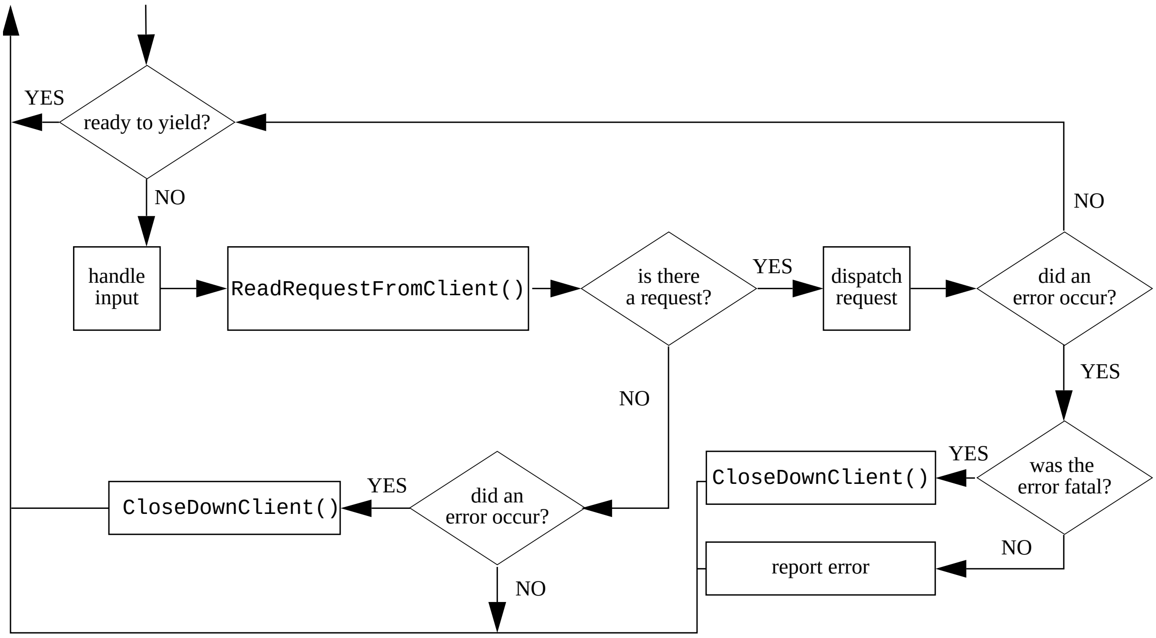


Figure 11 Dispatching requests from a client

Extensions are integrated into the flow of control by providing a dispatch function. All requests belonging to the extension are dispatched to that function. Examining the minor opcode of the request, this function can distinguish between different request types of the extension. It then forwards the request to whichever extension function is responsible for handling this type of request.

3 X-Based Application Sharing

3.1 Approaches and Architectures for X-Based Application Sharing

Application sharing is realized by intercepting the communication between applications and the window system. Doing this, the application-sharing software is able to multiplex the output of an unmodified single user application to multiple users and to forward their actions to the application. For X, the locations where the interception can take place are the Xlib, the server, or the X protocol. All of those locations have been used to develop application-sharing software. The following sections describe the different approaches and architectures of application-sharing products for X. It will be shown that virtually all of them use the X protocol for communication over potentially high-latency or low-bandwidth connections. A detailed discussion about the advantages and disadvantages of the approaches can be found in [25] and [19].

3.1.1 Server Modification

Figure 12 shows the architecture of an application-sharing product that is based on a server modification. In this architecture, a special server is not only able to display an application on its screen, but can also share it with any other unmodified server across the network. For the communication, the basic X protocol is used since the unmodified servers have no information about application sharing. For them the server with the application-sharing modification looks like an ordinary client from which they receive requests and to which they deliver replies, errors and events. It is the modified servers job to forward requests to the other servers and to shield the client from the knowledge that more than one user interacts with its windows.

In this architecture, the X protocol is used over the connections from the client to the modified server and from the modified server to the other servers. As indicated by the dashed arrows in Figure 12, any of those connections might be of high-latency or low-bandwidth: the one between client and modified server, because it might be necessary to run the application on a remote computer; the connection between the modified server and an unmodified server because the users of those two servers might not be located in the same local area network (LAN).

An example for this type of architecture is SharedX from Hewlett Packard [16].

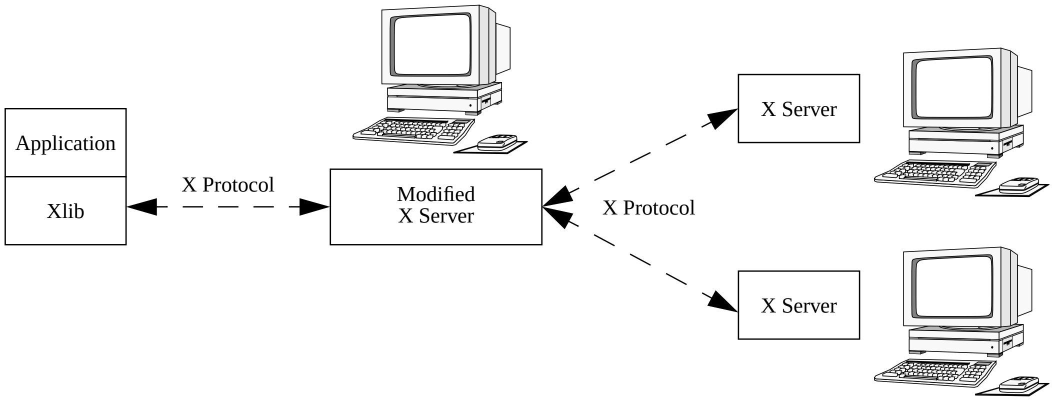


Figure 12 Server modification

3.1.2 Xlib Modification

In this approach, the application-sharing functionality is integrated into the Xlib. This might look like a violation of the rule that the single user application must not be changed in any way. However, almost all UNIX-based operating systems support shared libraries. Those libraries are not statically linked to a program; instead they are loaded at run time. If the client is compiled and linked to use Xlib as a shared library, it is possible to replace the old Xlib with a new one which supports application sharing. This does not require a new linking of the client.

Figure 13 shows the basic architecture for this approach. The Xlib function calls are mapped to requests for all participating servers. Replies, errors and events are collected from all servers and handed to the application as if they were generated by one server.

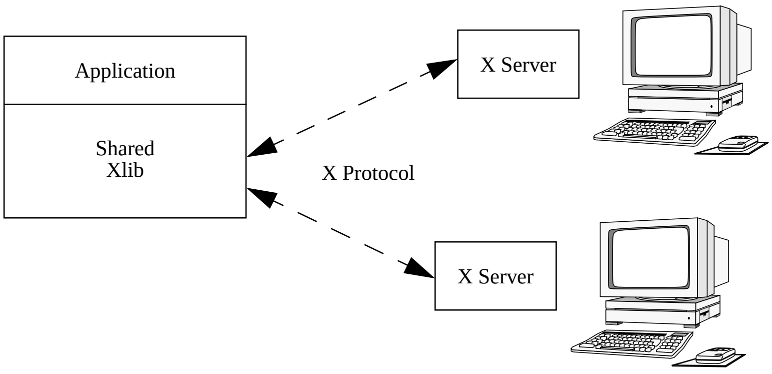


Figure 13 Xlib modification

The shared Xlib uses the standard X protocol to communicate with the X Servers. Some of them might be reachable over a high-latency or low-bandwidth connection only.

An implementation of this architecture is shXlib from DEC in Karlsruhe [2].

3.1.3 Centralized Pseudo Server

In this architecture, the communication between application and window system is intercepted at the X protocol level. As shown in Figure 14, a special server without a display — called pseudo server — is inserted between client and real servers. To the client, the pseudo server looks like an ordinary server; for the real servers it pretends to be an ordinary client.

A client that has to be shared is started in such a way that it connects to the pseudo server instead of to the users real server. The pseudo server then multiplexes the requests from the shared client to multiple real servers. It collects the replies, events and errors from those servers, forwarding them to the client as if they were issued by only one server. The pseudo server can be thought of as a service: Whenever someone wants to share a cooperation unaware application, it is started using this service.

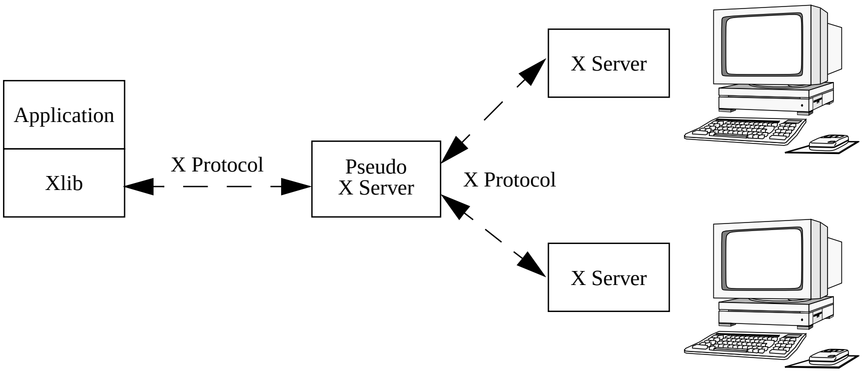


Figure 14 Centralized pseudo server

Just as in the modified server architecture, all connections might be high-latency or low-bandwidth connections. In addition to that, it is not unlikely that someone might want to use the service, even if he or she is not located in the same LAN as the computer that provides this service. In this case, the connection from the client to the pseudo server and from the pseudo server to the user's real server could be of high-latency or low-bandwidth even if client and real server run on the same computer.

Xmux, developed by Greg McFarlane, OTC Ltd., is an application-sharing tool that is based on a centralized pseudo server architecture. A comparison of Xmux with other application-sharing programs can be found in [4].

3.1.4 Distributed Pseudo Server

To prevent the pseudo server from becoming the bottleneck of the system, the distributed pseudo server architecture was developed. One of the most time consuming tasks in the centralized approach is the generation of requests for the different real servers when a client request arrives at the pseudo server. The generation of requests is not a simple copying of the incoming requests to the real server. Rather it is a mapping of the information the client sends in the request to the information a real server needs to understand the request. Examples of information that have to be mapped are sequence numbers, atom numbers, color information, window coordinates and various identifiers.

In the distributed pseudo server architecture shown in Figure 15, this work is done by a pseudo client running on the same workstation as the real server. Whenever necessary, the pseudo server forwards rough data from the client requests to the pseudo client. From this information, the pseudo client creates a request for the local server, performing all necessary mapping. Along with the data for requests, the pseudo server sends information about which user is currently allowed to interact with the application and which users are just watching. According to that information, the pseudo clients report the events, errors and replies from the real servers back to the pseudo server.

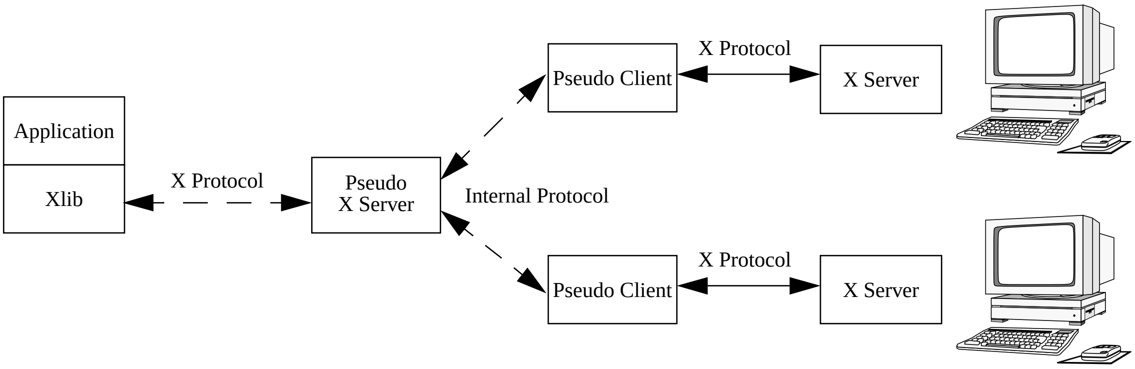


Figure 15 Distributed pseudo server

For the data exchange between pseudo server and pseudo client, an internal protocol could be used. But, since the X protocol is sufficient to transfer most of this information, the internal protocol is usually realized by using the X protocol with an extension. For the same reasons as in the centralized approach, the connections from client to pseudo server and from the pseudo server to the pseudo clients can be high-latency or low-bandwidth connections.

Examples for distributed pseudo server architectures are Xy [19] and Xwedge [18].

3.1.5 Distributable Pseudo Server

The distributable pseudo server architecture was developed to make it possible for users without the pseudo client software to participate in the sharing of an application, while retaining the distributed architecture for those users who have the necessary software installed. As shown in Figure 16, the distributable architecture is a combination of the centralized and the distributed approach.

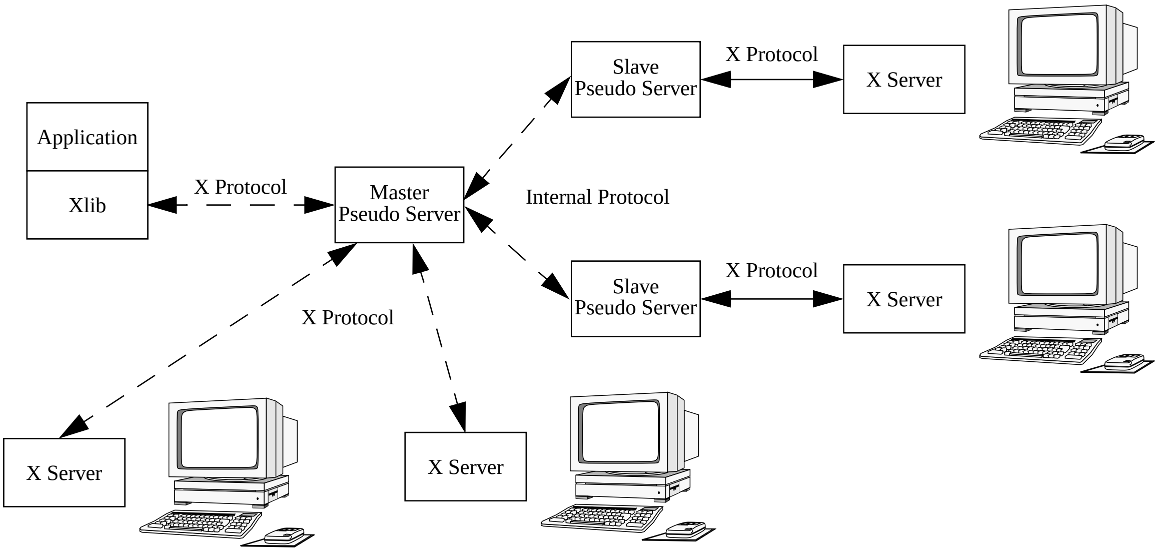


Figure 16 Distributable pseudo server

In this architecture, a pseudo server is running on some workstations in addition to the real X server. When a client has to be shared, it is started in such a way that it connects to one of those pseudo servers. This pseudo server becomes the master pseudo server (master) for that client. The master notifies all other pseudo servers which run on the same workstation as a real server participating in the sharing of the application. Those pseudo servers become slave pseudo servers (slaves) for the client.

The master is responsible for distributing client requests. For X servers that run on the same workstation as a slave, it simply forwards the unmodified client request to the slave. The slave then does the required mapping and sends the request to the real server, just like a pseudo client in the distributed approach. For servers that do not have a slave running on their workstation, the master does the slaves' job, mapping and sending the request to the real server. Replies, events and errors are sent back from the slaves and the real servers to the master. The master forwards them to the client as if they were generated by only one server. For the communication between master and real servers, the X protocol is used since those servers know nothing about application sharing. Master and slaves communicate by using a slightly modified X protocol for the forwarded requests, events, errors and replies. Additional information is transferred by using an extension to the X protocol. For the same reasons as in the centralized approach, the connections from the client to the master, from the master to the real servers and from the master to the slaves can be high-latency or low-bandwidth connections.

The only application-sharing product that supports this advanced architecture is the XpleXer [25], which is the basis for the prototype integration described in this thesis.

Table 1 gives an overview over the different application-sharing architectures described in Sections 3.1.1–3.1.5.

Architecture	Communication between application and window system is intercepted at:	Transparent to the X Window System?	Suitable for sessions with many participants?	Every participant needs special software?	Uses X protocol over potentially high-bandwidth or low-latency connections?
Server modification	X Server	No	No	No	Yes
Xlib modification	XLib	No	No	No	Yes
Centralized pseudo server	X Protocol	Yes	No	No	Yes
Distributed pseudo server	X Protocol	Yes	Yes	Yes	Yes
Distributable pseudo server	X Protocol	Yes	Yes	No	Yes

Table 1 Application-sharing architectures — overview

3.2 The XpleXer — An Application-Sharing Tool for the X Window System

The XpleXer is an application-sharing software that was developed at the Siemens Telecooperation Center (STC) in Saarbrücken. It is the central component of the heterogeneous multi-media collaboration system GroupX which was developed within the scope of the BERKOM-2 project. GroupX is sold by Siemens Private Networks Department.

3.2.1 Sharing Applications Using the XpleXer

Whenever the user wants a client to be sharable, it is started in such a way that it connects to a pseudo server instead of the user's real server (see Figure 16). The pseudo server forwards all requests from the client to the real server as well as all replies, errors and events from the real server to the client. The real server of the user who started the client is called the primary server and, as described above, the pseudo server to which the client is connected is called the master. For all practical purposes, the master looks to the client as if it were the primary server. Because of that, the master does not need to do any mapping for this server — it can just forward all messages. This makes the processing for a sharable application, which is not actually being shared, very fast.

When the user wants to share one of his sharable clients with another user, he or she tells the master to add a new server to the client. In order to do this, the master tries to contact a pseudo server on the workstation of the new server. If there is one, it becomes the slave; if there is none, the master connects directly to the new server. As soon as the master has contact to either the slave or the new server, it starts replicating the client's output which is already shown on the primary server's display. To do this, it queries the primary server for information. However, some information necessary for the replication cannot be retrieved in that way. Therefore the master has to keep some information in a private database. Using both types of information, the master generates requests for the replication. If a slave is present the requests are sent without any mapping to the slave. The slave then does all the necessary transformations and hands the modified requests to the new server. If no slave is present, the master has to do the mapping and the requests are sent directly to the new server. After the replication is completed, the master multiplexes all new requests from the client to both the primary and the new server, using a slave if possible. The action of sharing a client while it is running is called spin-out sharing. If a slave is used to do the required mapping, it is also known as cooperative sharing [25].

As soon as a client is shared between more than one user, it is the master's job to shield the client from the fact that more than one server sends replies, errors and events. Replies and errors are generated in response to requests. Since the client thinks it is interacting only with the primary server, only the replies and errors from the primary server are sent back to the client. Replies and errors from other servers are either discarded or handled internally in the pseudo servers.

Input events are processed differently: An input token is assigned to any one of the users participating in the sharing of the application. The master forwards only input events from the token holder to the client. All input events from other users are ignored. The XpleXer provides mechanisms to assign the input token any time to any participant. The user interface that is controlling the XpleXer is responsible for implementing suitable policies for the assignment of the input token using the provided mechanisms.

3.2.2 Architecture of the XpleXer Pseudo Server

As shown in Figure 17 the XpleXer pseudo server can be divided into three layers. Because of the functional similarity with a real server, the OS and the DIX layers were taken from the sample server of X11 Release 5 and modified to fit into the new environment.

The OS layer was enhanced to accept connections to a master XpleXer from a special pseudo-client port. The DIX layer's only remaining job is the scheduling of the pseudo server. All incoming requests are handed to the new Application-Sharing (AS) layer. No requests are processed by the DIX layer itself.

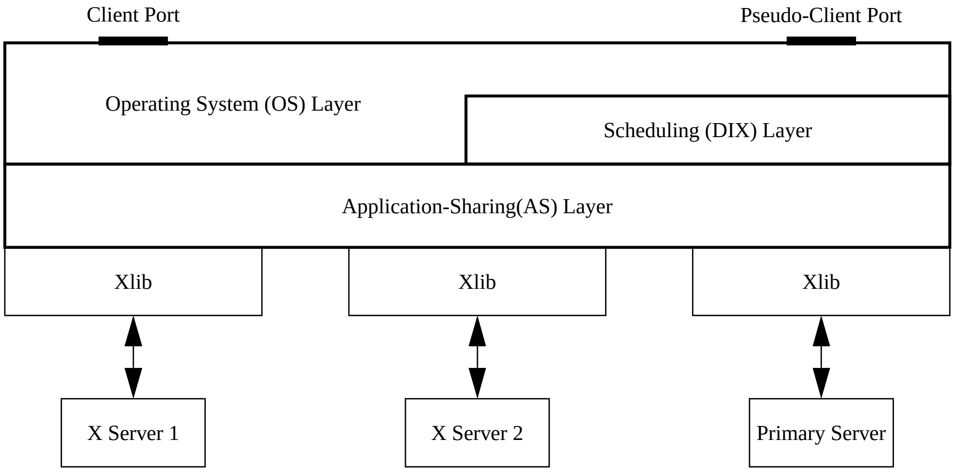


Figure 17 Architecture of the XpleXer pseudo server

The AS layer has a request processing function for each type of request. In those functions, the request mapping is done when necessary and the modified request is sent to the real server by using Xlib function calls. If a client is being shared, a received request is passed to the same processing function multiple times, each time with a reference to a different destination server. The AS layer is also responsible for event handling and spin-out sharing.

3.2.3 Flow of Control in the XpleXer Pseudo Server

The flow of control in the dispatch loop is very similar to the one in the sample server described in Section 2.3.2, Figure 10. As can be seen in Figure 18, the main difference is the handling of events. The pseudo server doesn't generate events. Instead the events generated by the real servers are mapped and forwarded to the client. The OS layer function `WaitForSomething()` directly forwards any incoming events from the real servers to the AS layer for processing.

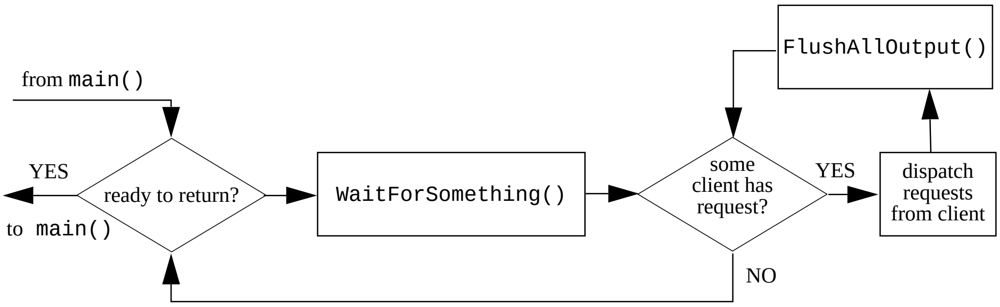


Figure 18 Flow of control in the XpleXer's Dispatch() function

Figure 19 shows how requests from a single client are dispatched: After a request is successfully read from the client, it is first dispatched to the primary server. If the client is shared, the request is additionally dispatched to each of the servers participating in the sharing. As explained above, only errors from the primary server are sent to the client or can cause the client to be closed down.

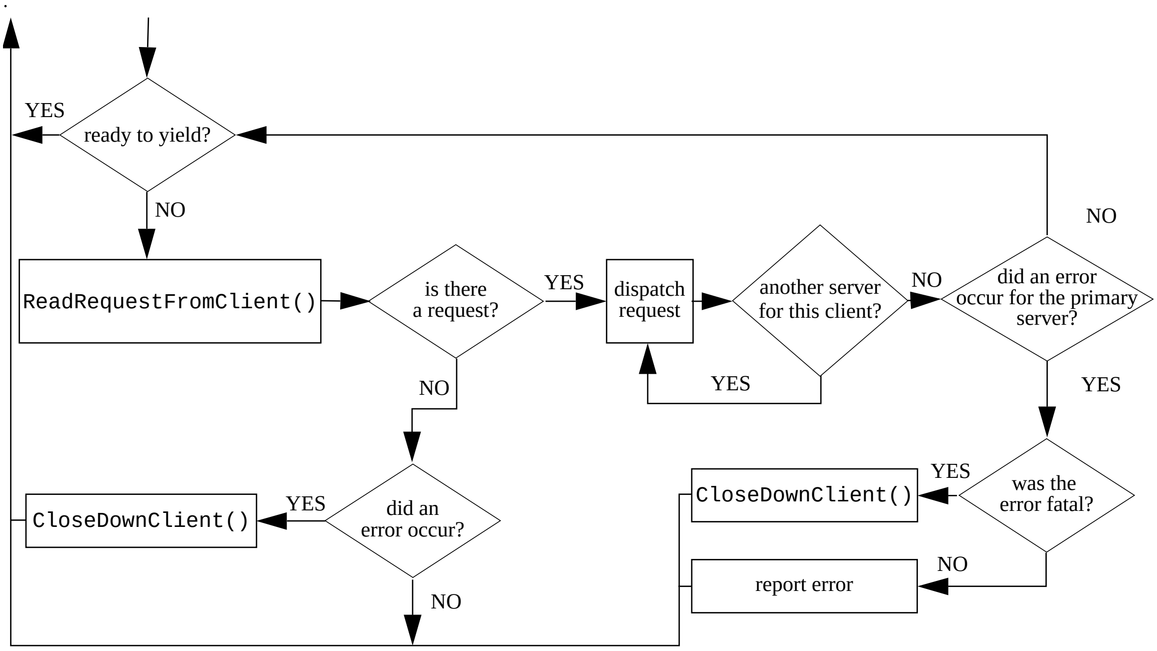


Figure 19   Dispatching requests from a client in the XpleXer

3.3 X-Based Application Sharing Over Low-Bandwidth and High-Latency Connections

In the standard X Window environment, client and server usually run on the same workstation or in the same LAN. The only common exception is the usage of a modem or an ISDN line to access an application remotely from “home”.

For an application-sharing environment, this scenario changes completely. As mentioned above, there are two main reasons why the X protocol is used over non-LAN links for application sharing: The application-sharing service might be a remote service and two users might not be located in the same LAN. The second reason is particularly strong, since the purpose of application sharing is “to allow two or more users located at geographically different places to synchronously work with a single-user application” [25].

In addition to the inherent usage of non-LAN links, application sharing increases the amount of bandwidth needed, since protocol data is exchanged with not only one, but with several servers. Unless a reliable multicast protocol is used for the transmission, this leads to an enormous increase in the required bandwidth.

To get an impression about the data volume and the number of round trips a usual client sends, the start-up of some applications has been monitored. The data from Table 2 was generated by applications that were started shareable using the XpleXer with a Sun Sparc 5 station (8 bit frame buffer) as the primary server.

Application	Request Data (Bytes)	Reply, Error and Event Data (Bytes)	Round Trips
Netscape	347136	140608	484
Frame Maker	55668	38660	167
xtetris	5216	13760	13
emacs	9908	37064	125
xterm	2348	10532	11

Table 2 Data volume and round-trip requests at application start-up

A high-latency link could be a connection between Berkeley/USA and Berlin/Germany, which has an average round-trip delay of about 200 ms for a high speed ATM link, and about 500 ms for an internet connection. This would lead to an additional start-up (or spin-out sharing) delay of 96.8 to 242.0 seconds for Netscape and still 25 to 62.5 seconds for emacs.

Typical low-bandwidth connections are ISDN (64 kbps) or a telephone with modem (14.4 kbps). For those connections, a start-up (or spin-out sharing) of Netscape would be 43.4 to 192.9 seconds slower than usual, if the full bandwidth is used and no protocol overhead exists. Since application sharing is not of much use without audio communication, and protocol overhead does exist, the usable bandwidth is usually less and therefore the delay is higher.

These numbers show that the basic X protocol has to be enhanced in order to be suitable for application sharing via high-latency or low-bandwidth connections.

4 Compressing and Enhancing the X Protocol

Since the appearance of the first X protocol compression approach — XRemote [7] — in 1992, a lot of work has been done to further compress and enhance the X protocol. Up to now, there exist two main directions for this work: The first was originated by Xremote and is still very active with its successor Low Bandwidth X [14] [28]. The second is the development of Higher Bandwidth X [10], Fast Higher Bandwidth X [12] and dxpc. The main difference between those directions is that XRemote and Low Bandwidth X use sequential data compression, while Higher Bandwidth X, Fast Higher Bandwidth X and dxpc relay on prediction-based compression. This chapter gives an overview of the X protocol enhancement and compression approaches mentioned above.

4.1 XRemote

XRemote is a protocol developed by Network Computing Devices (NCD) for using X over a serial line. This includes multiplexing of several client connections on a single serial line and compressing the X protocol. XRemote was developed before compressed serial line IP (CSLIP) [21] and point-to-point protocol (PPP) [24] were widely available, therefore it defines its own network and data link layer. Since it is now common to use CSLIP or PPP over serial lines, this section will focus on the multiplexing and compression aspects of XRemote.

4.1.1 XRemote Architecture

In order for XRemote to be transparent to client and server, the architecture shown in Figure 20 was developed. The remote clients connect to a pseudo server running in the same LAN as the clients. This pseudo server multiplexes all data streams from the clients into one stream and compresses it. The compressed stream is sent over the serial line to a pseudo client. There the data is decompressed and split into separate streams which are sent to the X server. In the reverse direction, data is multiplexed and compressed in the pseudo client, and the pseudo server decompresses and splits the data stream.

Neither clients nor server are aware that the X protocol is intercepted, multiplexed and compressed. For clients, the pseudo server looks like the real server, and for the real server, the pseudo client looks like a real client. In this architecture, XRemote is completely transparent to the whole X environment of the user.

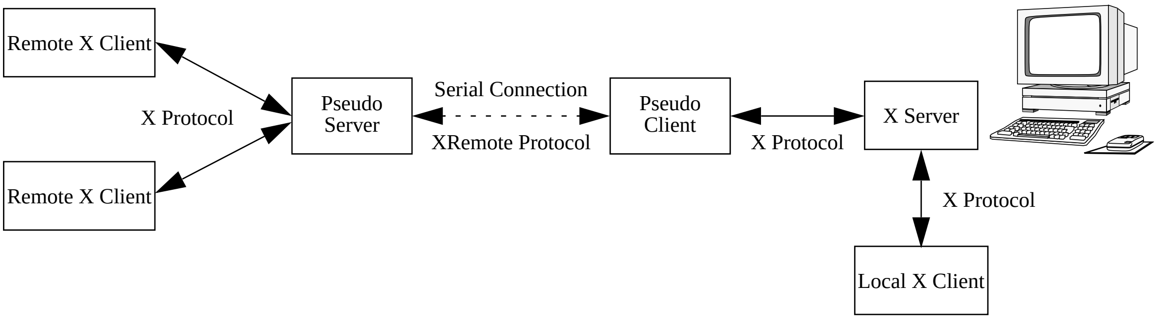


Figure 20 XRemote architecture

4.1.2 XRemote Layers

The functionality of XRemote can be divided into the layers shown in Figure 21. The X protocol data arrives at the XRemote layer. The first processing step is the multiplexing of several client streams of data in one stream. To accomplish this, three protocol primitives are used in the XRemote Layer. The first is NewClient, which signals that a new client has connected and tells the peer XRemote Layer that all following data in the stream belongs to the new client connection. When data from a different client connection is sent, this must be indicated by ChangeClient, which identifies the client connection to which all following data belongs. When a client disconnects, a CloseClient message is sent to inform the peer XRemote layer about the removal of a client stream.

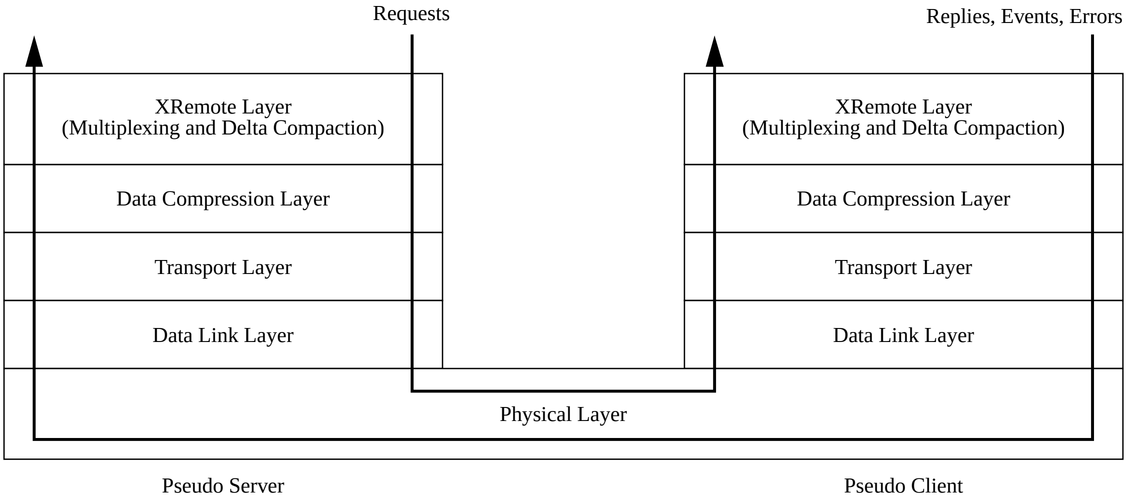


Figure 21 XRemote layers

After multiplexing into one stream, the X protocol messages are run through a delta compactor. In this step, a cache of the last 16 messages of length 64 bytes or less is kept. The current message is compared to the messages in the cache. This comparison of messages is done by sequentially comparing the bytes of two messages. If the difference between a message in the cache and the current message can be expressed in fewer bytes than the current message, then the difference is sent instead of the current message. This message is then called a delta message. The result of the processing in the XRemote layer is a stream of X protocol, NewClient, ChangeClient, CloseClient and delta messages. This stream is handed to the data compression layer as a stream of bytes.

In the data compression layer, the stream of bytes is broken into fixed-sized chunks. This is necessary to guarantee that the compressed data fits into one transport layer packet. The chunks are then compressed by using LZW[1] sequential data compression.

The compressed data is sent to the peer using the transport layer service. The peer decompresses the data, restores delta compressed information and splits up the stream sending X protocol messages to the clients or the server. The construction of pseudo client and pseudo server is symmetrical in respect to what the pseudo server does for requests, the pseudo client does for replies, error, events and vice versa.

4.1.3 Efficiency of XRemote

The compression performance of the XRemote protocol has been evaluated in [9]. The tests were done using several X protocol traces of real users. The overall results are as follows:

> *On average, client data compressed to 44% of its original size while server data compressed to about 28% of its original size. On average, server data accounted for 19% of each trace with client data accounting for the remaining 81% of the transferred bytes. In an average trace, compression would have reduced the amount of transferred data to 41% of its original size, giving a compression ration of 2.4:1. [9]*

It was found that delta compression is very efficient for input events like pointer movement and keyboard input. For all other data, the delta compression reduces the LZW compressed data by less than 7%.

Especially unsatisfying was the bad compression performance for images (3:1 where fax protocols regularly achieve a 10:1 compression) and for graphic requests (worse than 2:1).

[1] The basic algorithm was developed by Lempel and Ziv [30] and is know to the compression community as LZ78 coder. It was popularized by Welch, hence the term LZW.

4.2 Low Bandwidth X

Low Bandwidth X (LBX) is the successor of XRemote. In addition to improved protocol compression LBX offers elimination of round trips. This makes the usage of LBX as useful for high-latency links as for low-bandwidth connections. LBX was designed to improve the compression rate and reduce the number of round trips by taking advantage of various X protocol characteristics.

The version of LBX that was available for the integration is distributed with X11R6.0 and is a work-in-progress version. The final LBX standard is not completely compatible to this version. Version 1.0 of the final standard is part of X11R6.3 and is available since January 1997. This section discusses the work-in-progress version of LBX.

4.2.1 LBX Architecture

The main difference between the XRemote and the LBX architecture is the absence of a pseudo client. As shown in Figure 22, a special LBX X server is necessary to use LBX. The advantage of this architecture is that more information is available in the X server than in the XRemote pseudo client. This is true because the pseudo client has information about only remote clients, whereas the server has information about remote and local clients. Using this additional information, LBX is able to increase the compression rate. The disadvantage of this architecture is that LBX is not fully transparent to the X environment: A new X server has to be installed to use LBX.

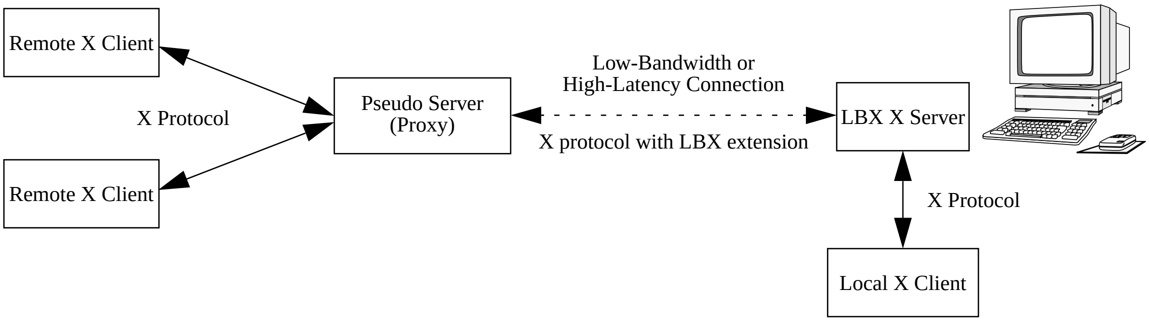


Figure 22 LBX architecture

In LBX terminology, the pseudo server is called a proxy. The LBX X server and the proxy communicate using the X protocol with an LBX extension.

4.2.2 LBX Layers

The functionality of LBX can be divided into the layers shown in Figure 23. LBX is basically XRemote with a preprocessing layer. In the preprocessing layer, all incoming X protocol messages are examined and optimized for compression or reduction of round trips. If a message can be optimized, it is handed as an LBX extension message in the optimized form to the XRemote layer. If no optimization is possible, the original X protocol message is forwarded to the XRemote Layer. The XRemote layer multiplexes the message streams of multiple clients in one stream and performs delta compaction on that stream. Finally the message stream is LZW compressed in the data compression layer and transmitted using an existing transport service like CSLIP or PPP.

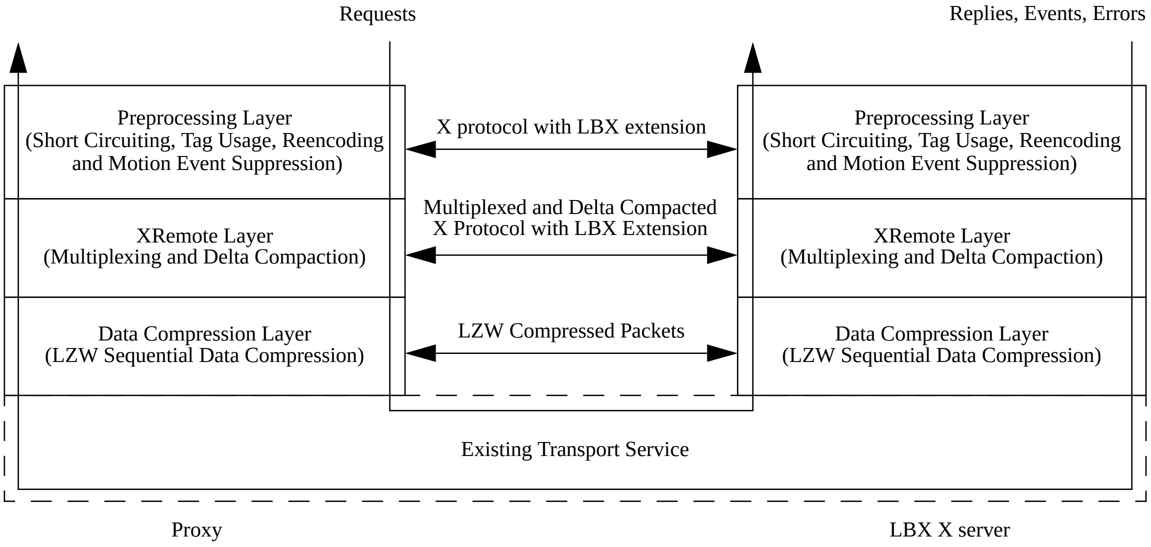

Requests
Replies, Events, Errors
Preprocessing Layer
(Short Circuiting, Tag Usage, Reencoding
and Motion Event Suppression)
X protocol with LBX extension
Preprocessing Layer
(Short Circuiting, Tag Usage, Reencoding
and Motion Event Suppression)
XRemote Layer
(Multiplexing and Delta Compaction)
Multiplexed and Delta Compacted
X Protocol with LBX Extension
XRemote Layer
(Multiplexing and Delta Compaction)
Data Compression Layer
(LZW Sequential Data Compression)
LZW Compressed Packets
Data Compression Layer
(LZW Sequential Data Compression)
Existing Transport Service
Proxy
LBX X server

Figure 23 LBX layers

The Proxy and LBX X server provided with the work-in-progress version of LBX are realized by reusing the sample X server described in Chapter 2.

The XRemote and the data compression layers of LBX are integrated into the OS layer of the proxy and the LBX X server. Whenever messages are sent over the connection between proxy and LBX X server, they are automatically processed by the XRemote and the data compression layers.

For the proxy, the preprocessing layer replaces the DDX layer of the sample server. All requests from clients are dispatched to optimization functions located in the preprocessing layer. If possible, the optimization functions replace the requests with optimized LBX extension requests. In the reverse direction, optimized replies, errors and events from the LBX server are converted back to X protocol messages in the preprocessing layer.

On the LBX X server side, the preprocessing layer is realized as an extension. The LBX extension requests that were generated by the proxy are dispatched to LBX extension functions, which take appropriate actions to execute the request. Requests that were not optimized are dispatched and executed as usual. In addition to request handling, replies, events and errors are optimized by the LBX extension.

The following sections discuss the functionality of the preprocessing layer.

4.2.3 Short Circuiting

Short circuiting is the replacement of round-trip requests with one-way requests. The main idea is that whenever a remote client gets constant information from the server through the proxy, this information is stored in the proxy. If a remote client later asks for the same information, the proxy can directly answer the request without asking the server first. To keep the server updated about sequence numbers and client states, one-way requests are sent to the server if the proxy short circuits requests. The effect of short circuiting is that round-trip requests over a potentially high-latency connection (remote client → server) are replaced by round-trip requests within one LAN (remote client → proxy).

There are five X protocol requests for which LBX supports short circuiting: InternAtom, GetAtomName, LookupColor, AllocColor and AllocNamedColor.

In order to intern an atom, a client uses the InternAtom request to send the server a property name. The server replies with the atom that refers to the property name. This mapping is constant for the lifetime of the server. The first time the proxy sees an InternAtom request and its reply for a property name, it stores the mapping. If a remote client later asks for the atom of the same property name, the proxy looks at the stored information and answers the request directly. To update the sequence number for the client, the proxy sends an LbxModifySequence request to the server. If several InternAtom requests are short circuited in a row, only one LbxModifySequence request needs to be sent.

With the GetAtomName request, the client asks for the property name of a given atom. As this requires the same information as the InternAtom request, it is handled in the same way, using the same stored information.

The LookupColor request is used to get the RGB values of a color described by a string. The mapping between color name and RGB values is constant for a server. Therefore the

LookupColor request is handled similarly to the InternAtom request. The only difference is that colorname and RGB values are stored instead of property-name and atom values.

To allocate a read-only colorcell with given RGB values in a given colormap, a client issues an AllocColor request. The server replies with an index and the contents of a colorcell that contains the closest RGB values physically possible on the screen. For each read-only colorcell the server counts the number of clients that have allocated it. If this number drops to zero, the server knows that no client has allocated the colorcell and frees it.

In order to short circuit AllocColor requests, the proxy stores the mapping from RGB values to the index and the contents of a colorcell. This is done the first time a color is allocated by a remote client in a colormap. If thereafter a remote client wants to allocate the same color in the same colormap, the proxy responds with the stored data. In order to update the counter for the colorcell and the sequence number for the client, an LbxIncrementPixel request is sent to the server. When the last remote client deallocates a read-only color, the mapping for that color is removed from the proxy.

The AllocNamedColor request is a combination of the LookupColor and AllocColor request. It asks the server to map a colorname string to a RGB value and allocate this value in a colormap. If the proxy has both pieces of information the request is short circuited as an AllocColor request. If not, the request is sent on to the server and the reply information is stored for colorname to RGB value mapping and for RGB value to colorcell index and content mapping.

Short circuiting causes a very interesting problem: X guarantees that replies, errors and events that were generated by an earlier request are delivered to the client before replies, errors and events generated by a later request are delivered. As shown in Figure 24, this can’t be guaranteed if short circuiting is used. In this example the first request issued by the remote client is the one-way drawing request PolyLine. This request is optimized to an LbxPolyLine request and sent to the server. At the server, an error occurs while executing the request and a BadAlloc error is sent back to the proxy, which forwards the error to the client. In the meantime the client has issued an AllocColor request, which was short circuited by the proxy. The result is that the client receives the response to the AllocColor request before it gets the error from the PolyLine request.

Usually the Xlib can compensate this and shield the application from the problem. For all applications tested in this thesis, there have been no problems except for a few warnings from the Xlib. To avoid any problems with out-of-order event and error delivery the proxy can be asked to do short circuiting only if it can guarantee that no events and errors from earlier messages can arrive. However, this reduces the number of round trips that can be short circuited seriously.

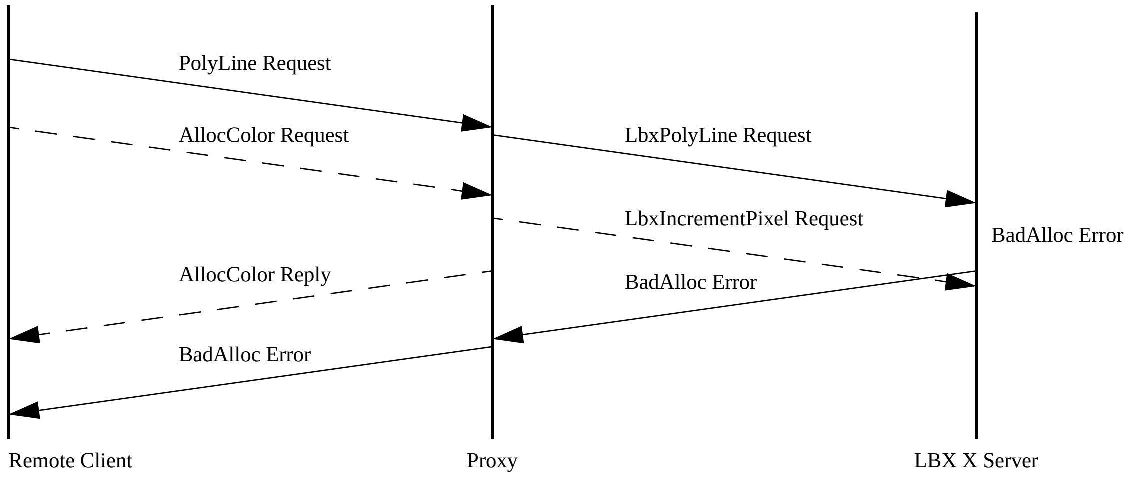


Figure 24 Problem with Short Circuiting

4.2.4 Tag Usage

Tags are used to support large data items that rarely change and are queried by many clients, sometimes more than once by a single client. The first time those data items are sent as a reply from the server to the proxy, a tag is included in the message. The proxy caches the data item with the tag. The next time the server wants to reply with the same data item it sends the tag instead. The proxy then takes the data item that belongs to the tag from the cache and sends it to the client.

The usage of tags is different from short circuiting because the data items are not necessarily constant. Therefore the proxy has to ask the server about the data item before it can reply to the client. The server replies with a tag only if the data item has not changed. If it has, the server sends the new data item and a new tag. Consequently this method is only used for requests where the queried data item might change on rare occasions. It reduces the required bandwidth but does not reduce the number of round trips.

In order to manage tag usage and caching, additional LBX messages have been defined. The LbxInvalidateTag request is sent from the proxy to the server if a tag has been removed from the cache. This happens when the proxy runs out of cache memory. An LbxInvalidateTag event is sent by the server to tell the proxy that the data associated with a tag has changed and can therefore be removed from the cache. If the proxy has lost the data associated with a tag it can get it back from the server with an LbxQueryTag request.

Four X protocol requests and their replies are optimized using tags in this way: The request to connect to the server, which returns information about the display; the GetKeyboardMapping and GetModifierMapping requests, which return information about the mapping between keyboard key numbers and how they should be interpreted by the client; and the QueryFont request, which returns logical information about a certain font.

A second usage of tags is the support of property data which is used for interclient communication. A client usually sends property data by issuing a ChangeProperty request for the desired property. The server stores the property data and if another client asks for it with a GetProperty request, the server sends back that information as a reply.

If both of the clients that want to communicate with each other use the proxy, it is not necessary to send the property data over the connection between proxy and server. When the proxy receives a ChangeProperty request from a client, it stores the property data. Using the LbxChangeProperty request, the proxy then asks the server for a tag that refers to the property data without actually sending it. When a client asks the proxy for property data that it has stored, the proxy sends an LbxGetProperty to the server. If the property is still valid, the server replies with the tag of the property data. If it has been changed by a client not connected to the proxy, the server sends the new property data to the proxy. If the data is needed by the server because a client not connected to the proxy asks for it, an LbxQueryTag event is sent to the proxy. The proxy then sends the actual data of the property to the server.

Tag usage is the **only** part of LBX that requires an architecture with a modified X server. Everything else could be implemented using an architecture similar to that of XRemote. However, in order to invalidate tags or request property data from the proxy, LBX needs information about all clients that are connected to the server.

4.2.5 Reencoding

X protocol messages that are not efficiently coded are reencoded in the preprocessing layer. There are four types of reencoding: Image compression, drawing request reencoding, font metric reencoding and event squishing.

The two X protocol requests PutImage and GetImage are used to transfer images between client and server. Both requests transfer images as uncompressed bitmaps. LBX provides mechanisms to integrate arbitrary image compression techniques for those requests. In the current version CCITT[1] G4 [5], compression is used for two-color images and PackBit [8] compression for multi-color images. The image included in a PutImage request is

[1] Comite Consultatif Internationale de Telegraphie et Telephonie

compressed in the preprocessing layer and sent to the server using an LbxPutImage request. For GetImage, the image in the reply is compressed and sent from the server to the proxy with an LbxGetImage reply. Since further LZW compression of the already-compressed image data is not efficient, the LbxPutImage and LbxGetImage requests are passed through the compression layer without further processing.

Drawing requests use 4 byte identifiers for the identification of the destination (resource ID) and the style (graphic context ID) of the drawing request. These identifiers are extremely repetitive. Therefore a small cache of 15 entries for both identifiers is kept in proxy and server. If a cache hit occurs, the 4 bit index of the cache entry is sent. For a miss, all 4 bits are set to 1 and the identifier is sent. If, for example a hit occurs for both graphic context ID and resource ID, only one byte has to be sent instead of 8. For the determination of coordinates, lengths, widths and angles, 2 byte fields are used in the X protocol. However in many cases 1 byte is sufficient to hold the information. LBX reduces all of those 2 byte fields to 1 byte where possible. The drawing requests that are reencoded in this manner are: CopyArea, CopyPlane, PolyPoint, PolyLine, PolySegment, PolyRectangle, PolyArc, FillPoly, PolyFillRectangle, PolyFillArc, PolyText and ImageText. The LBX extension requests generated to replace these requests are: LbxCopyArea, LbxCopyPlane, LbxPolyLine, LbxPolySegment, LbxPolyRectangle, LbxPolyArc, LbxFillPoly, LbxPolyFillRectangle, LbxPolyFillArc, LbxPolyText and LbxImageText.

The X protocol request QueryFont asks for the description of a certain font, which is called a font metric. The QueryFont reply describes each character of the font with the following information fields: left side bearing, right side bearing, character width, ascent to top, descent to bottom and attributes. For each of those information fields, two bytes are reserved in the QueryFont reply. However, in most cases one byte is sufficient. Therefore LBX reduces the information fields to one byte whenever possible before sending the reply.

All X protocol events are padded to 32 bytes to make reading and processing easier. As shown in [28], up to 26 bytes of an event may be unused. LBX squishes events to the size of the used data before transmitting it. There are no new LBX extension events necessary to do this: The original X protocol events are used without sending the padding.

4.2.6 Motion Event Suppression

In order to keep the client informed about the position of the pointer device, the server sends MotionNotify events to the client. With a high speed connection, a large number of MotionNotify events lead to a smooth animation on the screen. However, for a low-bandwidth connection, it might consume a big amount of the available bandwidth. In order to prevent MotionNotify events from using too much bandwidth, the number of MotionNotify events is reduced by LBX. To do this, the server keeps a counter of how many MotionNotify events it is allowed to send. For each sent MotionNotify event, this counter is decreased by one. If it reaches zero, no more MotionNotify events are generated. When the proxy receives MotionNotify events and enough bandwidth is available (e.g., transmission buffers are empty), it sends an LbxAllowMotion request to the server. As soon as the server receives this request, its counter is increased by the number contained in the request.

4.2.7 Performance of LBX

Since there exists virtually no information about the performance of LBX, it is within the scope of this thesis to evaluate the efficiency of LBX. To do this, the LBX proxy was modified to collect information about compression rates and elimination of round trips. The modified proxy generates the following data:

- Amount of data sent in requests from the client to the proxy. This data is called uncompressed client data.
- Amount of compressed data sent in requests from the proxy to the server. This data is called compressed client data.
- Amount of data sent in replies, errors and events from the proxy to the client. This data is called uncompressed server data.
- Amount of compressed data sent in replies, errors and events from the server to the proxy. This data is called compressed server data.
- Amount of total data exchanged between client and proxy, which is the sum of the uncompressed server and client data. This data is called the uncompressed total data.
- Amount of total compressed data exchanged between proxy and server, which is the sum of the compressed server and client data. This data is called the compressed total data.
- The number of round-trip requests without short circuiting.
- The number of round-trip requests that are InternAtom, GetAtomName, LookupColor, AllocColor and AllocNamedColor requests. Those round-trip requests are called eliminatable round trips.
- The number of round trips that were actually eliminated by LBX.

Five standard X applications have been selected for evaluation with the modified proxy: xterm, emacs, xtetris, FrameMaker and Netscape. Each of these applications has been monitored under three different conditions:

- At application start-up with a proxy that was newly started and therefore had an empty cache.
- At application start-up where the same application was already running over the proxy, which leads to a nearly optimally filled cache.
- During the usage of the application, e.g., browsing the web with Netscape, editing a document with FrameMaker, playing xtetris, writing a paragraph with emacs, and issuing shell commands using xterm.

All tests have been done on a Sparc5 station with an 8 bit frame buffer, running Solaris 2.5. The X server was the X11R6.0 X server and the window manager was twm.

Table 3 shows the results from application start-up with an empty cache. The compression rate for the total data lies between 2.3:1 and 3.9:1 with an average of about 2.8:1. This is marginally better than XRemote. The existence of eliminated round trips for emacs and Netscape is very interesting. Since the proxy had no information prior to the start of the application, this shows that those programs make inefficient use of the X protocol: They requested the same constant information multiple times.

	xterm	emacs	xtetris	FrameMaker	Netscape
Uncompressed client data (bytes)	2348	9904	4992	55704	345644
Compressed client data (bytes)	1184	5036	2329	19463	178733
Compression rate	2.0:1	2.0:1	2.1:1	2.9:1	1.9:1
Uncompressed server data (bytes)	10532	37064	13504	39036	142908
Compressed server data (bytes)	3876	7160	4869	13643	38260
Compression rate	2.7:1	5.2:1	2.8:1	2.9:1	3.7:1
Uncompressed total data (bytes)	12880	46968	18496	94740	488552
Compressed total data (bytes)	5060	12196	7198	33106	216993
Compression rate	2.5:1	3.9:1	2.6:1	2.9:1	2.3:1
Round trips without short circuiting	11	125	13	167	484
Eliminatable round trips	4	49	8	31	371
Round trips eliminated	0	13	0	0	97
Percentage of round trips eliminated	0	10.4	0	0	20.0

Table 3 Application start-up with empty cache

As can be seen in Table 4 a filled cache leads to a large increase in the compression rate. The main reasons are the filled tag cache, the filled delta compactor and the filled identifier cache. In this scenario, the compression rate lies between 2.4:1 and 9.2:1 with an average of about 5.3:1, which is more than two times the compression rate of XRemote. However, the start-up of Netscape and FrameMaker shows that the advantage of caching decreases with the amount of data sent. For applications that exchange only a small amount of data with the server (xterm, emacs and xtetris), the information sent back from the server dominates the total amount of data. This information can be dramatically reduced by using tags. For applications that exchange a large amount of data with the server (FrameMaker and Netscape), the client data dominates the total amount of data. This information is unaffected by tag usage.

Short circuiting is effective in this scenario, ranging from a 18% elimination for FrameMaker to a 74.6% elimination for Netscape with an average of 40.4% elimination. For simple X applications like xterm, emacs and xtetris, the short circuiting of InternAtom and LookupColor requests are most important. For graphics-oriented applications like Netscape, the AllocColor request is the one that is most often short circuited. Since FrameMaker allocates only private colorcells, only a few round trips can be eliminated.

	xterm	emacs	xtetris	FrameMaker	Netscape
Uncompressed client data (bytes)	2348	9872	4992	55772	340952
Compressed client data (bytes)	946	3580	1964	18961	176080
Compression rate	2.5:1	2.8:1	2.5:1	2.9:1	1.9:1
Uncompressed server data (bytes)	10532	36968	13440	39420	140348
Compressed server data (bytes)	450	3156	2435	8221	27244
Compression rate	23.4:1	11.7:1	5.5:1	4.8:1	5.2:1
Uncompressed total data (bytes)	12880	46840	18432	95192	481300
Compressed total data (bytes)	1396	6736	4399	27182	203324
Compression rate	9.2:1	7.0:1	4.2:1	3.5:1	2.4:1
Round trips without short circuiting	11	125	13	167	484
Eliminatable round trips	4	49	8	31	371
Round trips eliminated	3	45	6	30	361
Percentage of round trips eliminated	27.3	36.0	46.2	18.0	74.6

Table 4 Application start-up with filled cache

The efficiency of LBX during application usage is shown in Table 5. Here the compression rates for the total data range from 2.7:1 to 6.5:1 with an average of about 4.3:1. The image compression is the main factor for the good compression result of Netscape. The compression rate for xtetris was achieved because of graphics reencoding and delta compaction of extremely repetitive requests.

With the exception of FrameMaker, the number of round trips is small compared to the number of round trips at application start-up. The number of eliminatable round trips is negligible. The only application that uses a lot of round trips during usage is FrameMaker. A closer examination of those round trips revealed that about 80% of those round trips are GetInputFocus requests, sometimes more than 20 in a row. Asked about this inefficient usage of the X protocol, the developers said that this might be an overcompensation of a problem with an earlier version of FrameMaker. In general, the data from Table 8 suggests that round trips do not play an important role during the usage of an application.

	xterm	emacs	xtetris	FrameMaker	Netscape
Uncompressed client data (bytes)	9508	12192	129008	1305352	7312664
Compressed client data (bytes)	4261	3345	21068	469477	1111403
Compression rate	2.2:1	3.6:1	6.1:1	2.8:1	6.6:1
Uncompressed server data (bytes)	5120	12864	18816	94904	70504
Compressed server data (bytes)	1105	3297	5298	27825	18508
Compression rate	4.6:1	3.9:1	3.6:1	3.4:1	3.8:1
Uncompressed total data (bytes)	14628	25056	147824	1400256	7383168
Compressed total data (bytes)	5366	6642	26366	497302	1129911
Compression rate	2.7:1	3.8:1	5.6:1	2.8:1	6.5:1
Round trips without short circuiting	3	2	2	764	105
Eliminatable round trips	0	0	0	6	13
Round trips eliminated by short circuiting	0	0	0	0	5
Percentage of round trips eliminated	0	0	0	0	4.8

Table 5 Application usage

As a summary, it can be said that LBX improves the compression of XRemote by a factor between 1.5 and 2, and if the proxy’s cache is filled, LBX eliminates about 40.4% of the round trips at application start-up.

4.3 Higher Bandwidth X, Fast Higher Bandwidth X and dxpc

Higher Bandwidth X (HBX) is an approach to compress the X protocol by using structured data compression (SDC). HBX and SDC were developed 1994 by John Danskin at Princeton University. Fast Higher Bandwidth X (FHBX) can be considered a successor to HBX. It was developed to speed up the compression process of HBX by using hash based prediction. dxpc is an X protocol compressor that was greatly influenced by HBX and FHBX. It is freely distributed in the contrib section of X and was developed in 1995 by Brian Pane. The current version of dxpc is 3.3.0.

The architectures of HBX, FHBX and dxpc are similar to that of XRemote and thus fully transparent to the X environment of the user. For HBX, FHBX and dxpc, the compression and decompression is basically done in one step. A division of the functionality in different layers is therefore not necessary.

The following sections will explain the basic concepts of HBX, FHBX and dxpc.

4.3.1 Structured Data Compression

> *SDC uses arithmetic coding and statistical modeling to achieve high compression rates. The beauty of this technique is the separation of modeling and coding. In this method, the sender and receiver each have a predictive model which generates a table of probabilities for the next input symbol. An arithmetic coder on the sender side accepts the table and the symbol s_i which actually occurred in the input, and communicates the symbol to the decoder in just $-log_2(p_i)$ bits, where p_i was the estimate probability of symbol s_i appearing at this position in the input. The arithmetic decoder in the receiver decodes the coded bits using its identical table of probabilities as an inverse map. Bits are shared between adjacent messages, which means that individual messages can be expressed in fractional bits. For example, when $p_i=.5$ then 1 bit is necessary as expected, but when $p_i=.75$ then only ≈.41 bits are necessary to update the receiver.[11]*

The fundamental idea of arithmetic coding [3] can be described as follows: Given is a set S of input symbols s_i with probabilities p_i, where the sum of all p_i is 1. Each s_i occupies a certain interval $I(s_i)$ between 0 and 1 according to its probability. The intervals are non-overlapping and completely cover the Interval [0,1). A string $S_{1...n}$ consisting of the input symbols $S_1 ... S_n \in S$ will be encoded by finding its interval $I(S_{1...n})$, which lies in [0,1). This interval can be found by recursively applying $I(S_n)$ to $I(S_{1...n-1})$. The result of an interval $I_1=[L_1,H_1)$ applied to an interval $I_2=[L_2,H_2)$ is an interval

$$I_3=[\ L_2+L_1(H_2-L_2)\ ,\ L_2+H_1(H_2-L_2)\ )$$

As an example, Figure 25 shows how to encode the string S_{1-3}=ABA for the input symbol set s_1=A s_2=B s_3=C with the probabilities $p_1=0.5$, $p_2=0.25$, $p_3=0.25$ and the intervals $I(s_1)=[0,0.5)$, $I(s_2)=[0.5,0.75)$, $I(s_3)=[0.75, 1)$.

To find the interval $I(S_{1...3})$, we have to apply $I(S_3)$ (=I(s1) for A) to $I(S_{1...2})$. To get $I(S_{1...2})$, we have to apply $I(S_2)$ (=$I(s_2)$ for B) to $I(S_1)$ (=$I(s_1)$ for A).

With $I(S_2)$=[0.5,0.75) and $I(S_1)$=[0,0.5) we get $I(S_{1...2})$=[0.25, 0.375). Applying $I(S_3)$ to $I(S_{1...2})$ we get $I(S_{1...3})$=[0.25, 0.3125).

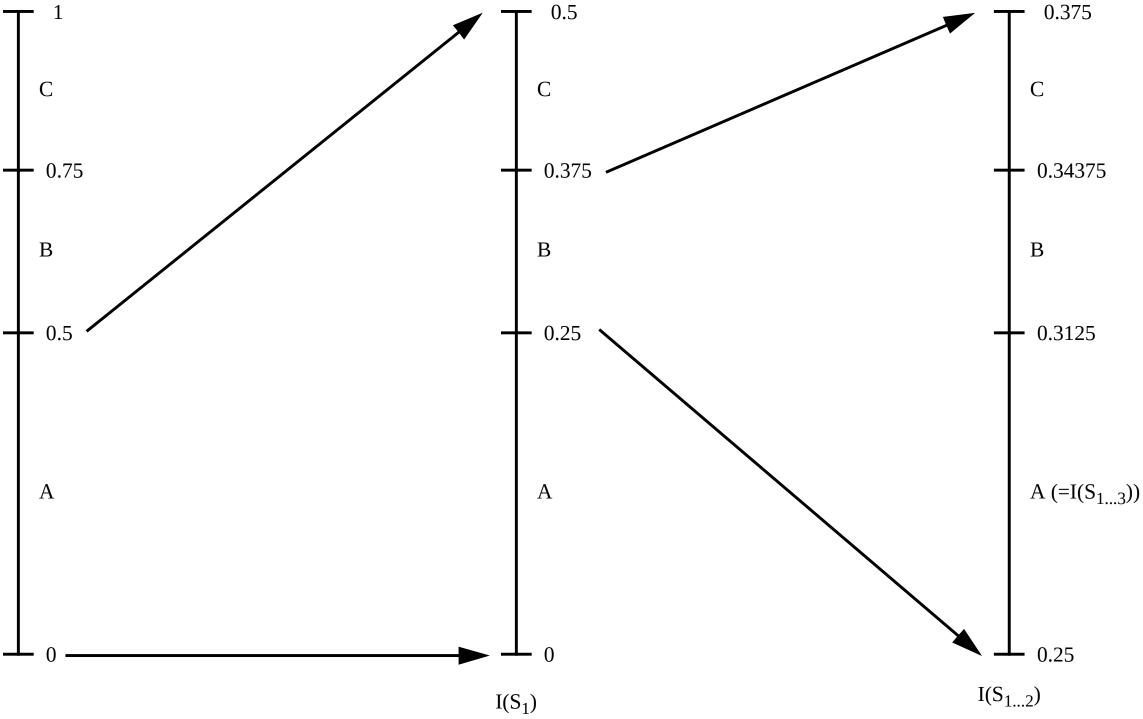


Figure 25 Example for arithmetic coding

If the length of the input string is known, any number from the resulting interval is sent by the encoder to update the decoder. In our example the transmission of 0.25 would be sufficient to transmit the string ABA.

The decoder recursively looks up the intervals in which the transmitted number lies, reconstructing the input string. For our example, 0.25 lies in I(s1)=[0,0.5) which yields an A as the first input symbol. Relative to I(S1), 0.25 lies within the interval I(s2), which yields B as the second input symbol. Finally 0.25 lies within the interval I(s1) relative to $I(S_{1...2})$, which yields A as the third input symbol.

Given the exact probability distribution of the input symbols, it can be shown that arithmetic coding is optimal. The key for a good compression is therefore the availability of a good approximation for the probability distribution of the input symbols.

SDC uses predictive models to get a good approximation for the probability distribution of the input symbols. Those predictive models are adaptive, which means that the predicted probability distribution for the next input token depends on previously processed input tokens. The usage of these predictive models is called statistical modeling in the context of SDC.

A simple predictive model for text compression could be based on relative frequencies of input characters. If, for example, at a certain point of time the character ‘e’ has appeared 200 times more often than the character ‘j’ the predicted probability for ‘e’ would be 200 times higher than for ‘j’. A predictive model like this is called a 0 order context model. A first order context model for text compression would pay special attention to the character directly in front of the next input character. If, for example, the character in front of the next input character was a ‘q’, the probability that the next input character is a ‘u’ is greater than 0.99. For this example, ‘q’ would be the context for the following input character. For each possible context, there would be a particular predictive model with relative frequencies. The context is used to index a particular predictive model (for example, the model for characters following a ‘q’) from a set of predictive models. The order describes the number of values used to index the model. English text is most efficiently compressed at 3rd or 4th order [3].

4.3.2 Predictive Models in Higher Bandwidth X

One of the most interesting parts of HBX are the predictive models used to approximate the probability distribution for X protocol message fields.

- Text: For text compression, Moffatt’s Prediction by Partial Match, method C’ (PPMC’) [3] is used. This is basically a 3rd order context model.
- Coordinates in drawing requests: Since the shape of objects is important and objects are frequently moved on the screen, absolute coordinates are transferred into relative ΔX and ΔY coordinates. The relative coordinates are then predicted by a third order context model.
- Small Bi-Level Images: Small bi-level images are usually glyphs, which are used for special characters or buttons. These small images are cached and predicted like text.
- Large Bi-Level Images: The color of a pixel in a large image is predicted by examining other pixels that are close to the predicted pixel. The state of those pixels are the context of the predicted pixel. The predictive model indexed by the context contains the probabilities for black or white.
- Color Images: Not supported.
- Incrementing fields: Those fields are typically sequence numbers or time stamps. The Δ of those fields is predicted using a 0 order context model.

- Default model: For all other fields, a simple 0 order context model is used. This is particularly interesting for window IDs, graphic context IDs and length fields.

4.3.3 Performance of Higher Bandwidth X

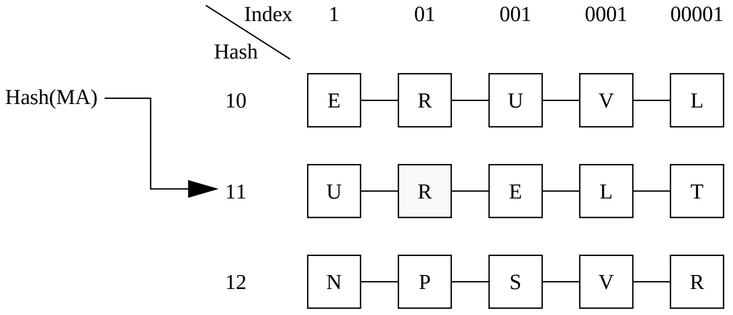
HBX achieved a 7.5:1 compression over a test suite described in [11]. The main drawback of HBX is that arithmetic coding and statistical modeling is much more CPU intensive than LZW compression. HBX is therefore about 10 times slower than XRemote or LBX. In addition the lack of colored image compression could be devastating for the compression performance. For programs like Netscape, the transfer of color images could easily be responsible for over 80% of the data exchanged between client and server.

4.3.4 Fast Higher Bandwidth X

The goal of FHBX is to achieve a similar compression rate as HBX with a CPU performance similar to XRemote. In order to reach this goal the arithmetic coding and statistical modeling of HBX is replaced with hash based prediction [29].

Hash based prediction has been developed for text compression. In this technique, the last few processed characters are used as a context for the next character by generating a hash code. The hash code is used to select a move to front list. As shown in Figure 26, the selected move to front list is indexed by using an efficient integer coding scheme like the Huffman code. In order to identify the next input character for the decoder, the encoder searches the character in the move to front list and sends its index to the decoder. The decoder selects the same move to front list by using the context of the transmitted character and gets the character using the transmitted index.

After the transmission of a character, it is moved closer to the front of the move to front list that was selected by the context. This ensures that the characters with a higher probability for a certain context are located at the beginning of the move to front list where the index can be expressed with fewer bits.



In this example, R is encoded with the context of MA. The hash of MA is a pointer to a move to front list where R can be found in position 2. The index is encoded a 01 and sent to the decoder.

Figure 26 Example for hash based prediction

Hash based prediction can be used as an approximation to SDC: For SDC, a context is used to select a particular predictive model; for hash based prediction, a context is used to select a move to front list. SDC uses arithmetic coding to encode an input symbol optimally. Hash based prediction uses the index of a move to front list to encode an input symbol nearly optimally.

Tests for the X protocol messages PutImage and MotionNotify have shown that FHBX has a compression rate of about 10%-20% less than HBX with a CPU performance similar to XRemote.

4.3.5 dxpc

The only documentation available for dxpc is the README file provided with the Version 3.3.0 distribution. From this file it can be learned that dxpc basically uses caching and HBX/FHBX techniques to compress the X protocol.

Testing dxpc with the same applications as LBX resulted in the compression rates for the total data shown in Table 6. A comparison to those of LBX shows that dxpc has, in most cases, significantly higher compression rates. One noticeable exception is the usage of Netscape. This is not surprising, as it is stated in the README file that “the implementation of color image compression in dxpc is fairly unsophisticated”.

	xterm	emacs	xtetris	FrameMaker	Netscape
Application start-up with empty cache (LBX values)	3.0:1 (2.5:1)	4.7:1 (3.9:1)	3.4:1 (2.6:1)	3.9:1 (2.9:1)	2.7:1 (2.3:1)
Application start-up with full cache (LBX values)	9.0:1 (9.2:1)	6.8:1 (7.0:1)	8.2:1 (4.2:1)	4.8:1 (3.5:1)	2.8:1 (2.4:1)
Application usage (LBX values)	5.0:1 (2.7:1)	6.4:1 (3.8:1)	7.3:1 (5.6:1)	6.6:1 (2.8:1)	4.7:1 (6.5:1)

Table 6 Compression rates for dxpc compared to LBX compression rates

4.4 Which Technique to Choose for the Prototype Integration

The choice of which technique to integrate into the XpleXer had to be made between HBX/FHBX/dxpc and LBX. The advantages of FHBX/HBX/dxpc are a higher compression rate and an architecture that is easy to integrate. LBX however offers the unique feature of round-trip elimination.

Since the objective of this thesis is the integration of synchronicity-reducing techniques as well as compression into an application-sharing product, the choice was made to integrate

LBX. For scenarios where low-bandwidth is the primary issue, HBX/FHBX/dxpc should be preferred. As will be explained in Chapter 6: Future Work, a combination of both techniques could be an optimal solution for a product-level integration.

5 Integrating Protocol Enhancement and Compression into Application Sharing

This chapter describes the integration of LBX into the XpleXer. Since LBX was still under development at the time the integration was done, the result of the integration is still in a prototype stage. The objectives of the prototype integration were to discover potential problems of an integration, to solve those problems and to show what effect the integration has on the used bandwidth and the number of round-trip requests.

The functionality of this prototype is limited in two ways:

- The LBX version integrated in the XpleXer is the work-in-progress version distributed with X11R6.0 and not the final version of LBX.
- Only the centralized part of the XpleXer is supported (e.g., a scenario where no slave pseudo servers exist). Supporting the distributable architecture would be necessary for a product-level integration but does not yield further general insight about how to integrate protocol enhancement and compression into application sharing. A discussion about supporting the distributable architecture can be found in Chapter 6: Future Work.

5.1 Architecture of the Prototype

The XpleXer has a centralized application-sharing architecture if it is running without any slaves. Figure 27 shows a scenario where the centralized XpleXer is used for application sharing between two LANs that are connected by a low-bandwidth or high-latency network. In this scenario, two users have each started one client sharable by connecting it to the XpleXer instead of connecting it to their own server. This connection between client and XpleXer is a low-bandwidth or high-latency connection for the client in LAN 2. If both clients are shared between the users in LAN 1 and LAN 2, then the XpleXer is connected to one primary server and one additional server for each client. The connection between the XpleXer and the primary server is a non-LAN connection for the client in LAN 2, while the connection between the XpleXer and the additional server is a non-LAN connection for the client in LAN 1.

This scenario shows that there are two types of low-bandwidth or high-latency connections: the connections between client and XpleXer as well as the connections between XpleXer and server. To support both types of connections, LBX has to be used between client and XpleXer as well as between XpleXer and server.

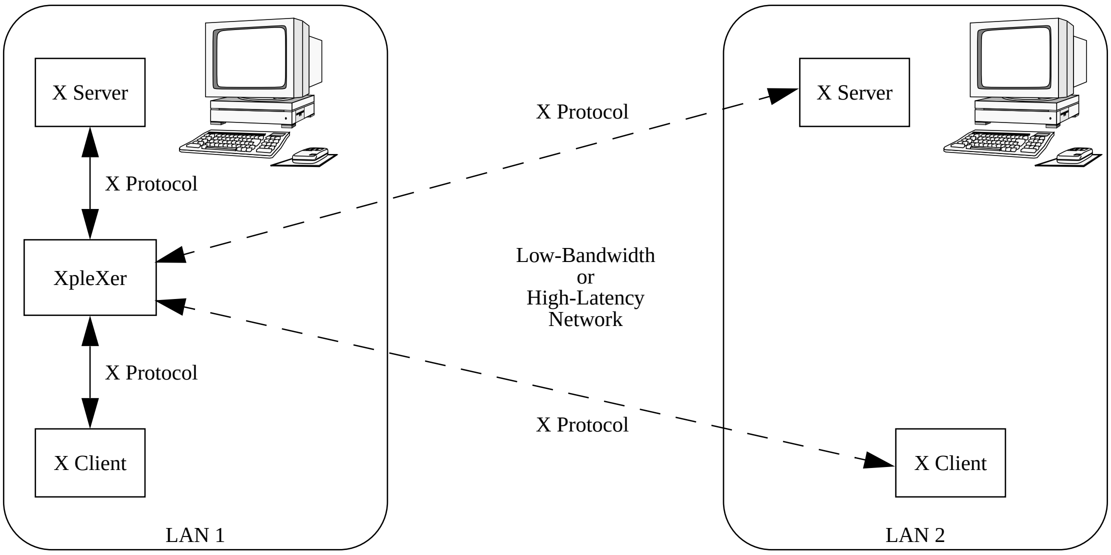


Figure 27 Architecture of the centralized XpleXer without LBX

The integration of LBX into the centralized architecture is shown in Figure 28. All connections that cross LAN boundaries use LBX to enhance and compress the X protocol. The integration of LBX in the connection between XpleXer and server is a trivial task: For this connection, LBX can be used without any modifications because LBX was specifically designed to enhance a connection between client (the XpleXer looks to the LBX proxy like a normal client) and server. The only restriction is that the user has to have an LBX capable server.

The interesting part is the integration of LBX into the connection between client and XpleXer. This part is drawn in bold lines in Figure 28. For this integration, the functionality of the LBX part of the LBX server must be added to the XpleXer. The XpleXer must be able to understand and decode the data sent by the LBX proxy.

A closer examination of the architecture shows that the integration of LBX into the connection between client and XpleXer results in an architecture identical to that of XRemote: The LBX proxy connects to the XpleXer, which performs the tasks of a pseudo client in addition to its usual application-sharing functionality. The proxy no longer has a direct connection to the server. This has the following consequences:

- The LBX tag mechanism cannot be integrated into the connection between client and XpleXer.

- A successful integration of LBX into the XpleXer is a transformation of the LBX architecture into a transparent architecture identical to the one of XRemote.

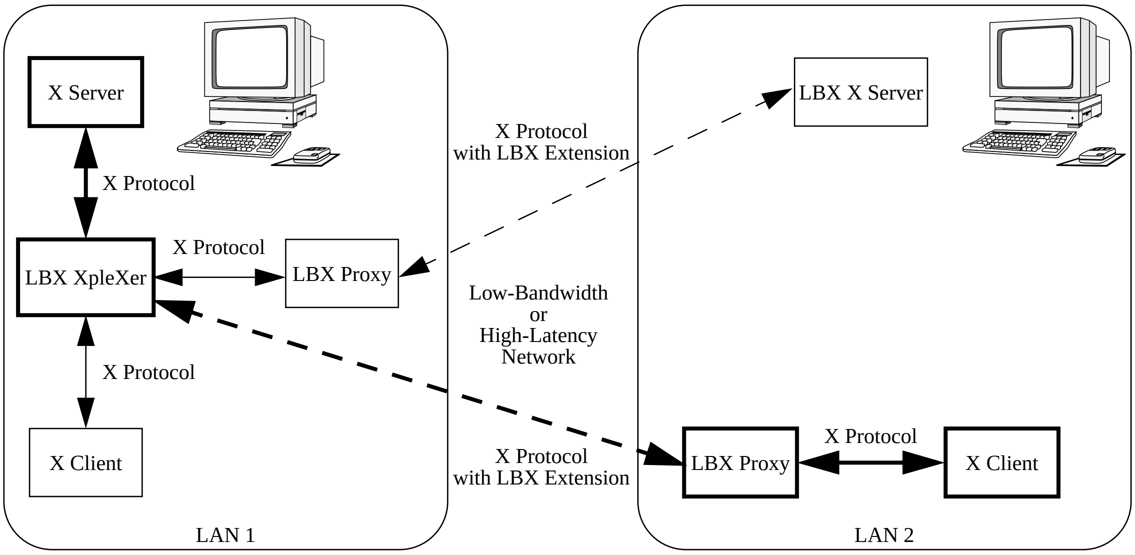


Figure 28 Architecture of the centralized XpleXer with LBX

5.2 Integrating the Low Bandwidth X Features into the XpleXer

One challenging aspect of the prototype integration is the volume of the software involved. The volume of the original XpleXer source code is approximately 65k lines of code and that of the LBX source code approximately 35k lines of code. The resulting prototype integration has a volume of nearly 90k lines of code.

An additional problem is that the XpleXer was developed reusing source code from X11R5, while LBX was developed for X11R6. The combination of the large volume and the version differences made merging the two software pieces particularly troublesome and time consuming. The following sections will explain how the different LBX mechanisms were integrated into the XpleXer.

5.2.1 XRemote Layer and Data Compression Layer

As mentioned in the previous chapter, the XRemote layer and the data compression layer are integrated in the OS layer of the LBX server. For the prototype integration, the LBX part of the LBX server's OS layer was extracted and inserted into the OS layer of the XpleXer. Even though no major changes were necessary, the difference between the X11R6 and the X11R5 code caused several minor conflicts that had to be solved.

Succinctly, it can be said that the integration of the XRemote layer and the data compression layer was straightforward but time consuming.

The original implementation of the data compression layer in LBX has one drawback: Monitoring the IP packets sent by LBX X server and proxy, it can be seen that the usage of LBX increases the number of IP packets by a factor of two. As an example, Table 7 shows the number of IP packets sent at the start-up of Netscape.

	Netscape start-up
IP packets for client data without LBX	656
IP packets for client data with LBX	1459
IP packets for client data with LBX and buffering of LZW chunks	1202
IP packets for server data without LBX	599
IP packets for server data with LBX	1278
IP packets for server data with LBX and buffering of LZW chunks	839

Table 7 Number of IP packets sent at the start-up of Netscape

With TCP/IP header compression (combined TCP/IP overhead per IP packet about 5 byte), the increase of IP packets should have no significant impact on the compression performance of LBX. Nevertheless, a reduction of the number of IP packets generated by LBX is desirable for scenarios where no header compression is used, or where the delay for the network access is high.

A closer examination reveals two reasons why the number of IP packets is increased when using LBX:

- The Xlib buffers client requests as long as possible and sends several of them in one TCP packet. Usually this TCP packet is mapped to one IP packet. The decision when to buffer requests and when to send them is made using local information of the client. The proxy, however, sends the requests to the LBX server without special request buffering, causing the number of IP packets to increase.
- The LZW compressor splits the data stream into small chunks, compresses them and immediately sends them to the decompressor, one chunk at a time. Because of this, the LZW compression of large requests and replies can cause multiple IP packets to be sent.

While the proxy does not have enough information about the client to do request buffering, the LZW compressor can be tuned to produce output in a more efficient way: For the prototype integration, a buffer was inserted after the LZW compressor. This buffer is only flushed if enough data is assembled, or if no further data needs to be compressed. As can be seen in Table 7, the result of buffering LZW chunks is a significant reduction of IP packets, compared to the original LBX. The remaining difference between LBX with buffering of LZW chunks and no LBX is caused by the lack of request buffering in the proxy.

5.2.2 Short Circuiting

The integration of short circuiting for InternAtom requests required some extra work because of the handling of atoms in the XpleXer. The XpleXer keeps a list of all interned atoms for each client. When the client is shared using spin-out sharing, those atoms are interned by the XpleXer from the new server. The mapping between atoms of the primary server and atoms of the new server is stored in the XpleXer. If the client later wants to set property data, the data is set in the primary server as well as the new server using the stored mapping.

Short circuiting, as it is used by LBX, does not give the XpleXer enough information to keep the list of interned atoms for each client: When an InternAtom request is short circuited, only an LbxModifySequence request is sent to the XpleXer. There is no way for the XpleXer to figure out the property name in the short circuited request. Therefore the LBX extension protocol had to be modified for the integration. Figure 29 shows the definition of a new LbxInternAtom request which replaces the LbxModifySequence request.

The LbxInternAtom request contains all the information of the InternAtom request from the X protocol. With this information, the XpleXer is able to keep the list about which client has interned which atoms.

Position	Size	Value	Description
0	1	extension opcode	major opcode: LBX request code
1	1	35	minor opcode: LBXInternAtom request
2	2	2+(n+p)/4	request length
4	1	BOOL	only if exists
5	1		padding
6	8	n	length of property name
8	n	STRING	property name
8+n	p		padding

Figure 29 The LbxInternAtom request

Another problem had to be solved for the integration of short circuiting AllocColor and AllocNamedColor requests. As described in the previous chapter, the LBX proxy sends an LbxIncrementPixel when an AllocColor or an AllocNamedColor request is short circuited, so that the server can increment the counter for the respective colorcell.

Since the real colormap is located in the primary server and not in the XpleXer, the handling of LbxIncrementPixel requests had to be changed for the integration. In order to increment the counter for a colorcell, the XpleXer has to send an appropriate AllocColor request to the real server whenever it receives an LbxIncrementPixel request. For the server, this request looks like an ordinary AllocColor request. It is therefore executed by incrementing the counter for the colorcell that contains the RGB values specified in the request. The reply to this request is discarded by the XpleXer because the original request was short circuited and the client already received the reply from the proxy.

The original LbxIncrementPixel request just contains an index to a colorcell. The XpleXer cannot generate an appropriate AllocColor request from this information, since the index can be interpreted only with the real colormap. In order to give the XpleXer all the necessary information, the LbxIncrementPixel request had to be changed. As shown in Figure 30, the new version of the LbxIncrementPixel request contains all information that the XpleXer needs to generate the AllocColor request for the server.

Position	Size	Value	Description
0	1	extension opcode	major opcode: LBX request code
1	1	8	minor opcode: LBXIncrementPixel request
2	2	5	request length
4	4	integer	colormap id
8	2	short integer	red value
10	2	short integer	green value
12	2	short integer	blue value
14	2		padding
16	4	integer	amount

Figure 30 The new LbxIncrementPixel request

Short circuiting of LookupColor and GetAtomName was integrated into the XpleXer without major modifications of the original LBX.

5.2.3 Profiling and Cache Prefill for Short Circuiting

As described in Chapter 4, LBX eliminates a significant number of round-trip requests at application start-up if the proxy's cache is filled. This improves the performance for applications that are started after the proxy has been used for some time. Short circuiting would be even more useful if applications that are started early in the lifetime of the proxy could also take advantage of the short circuiting mechanism. This would especially apply to application-sharing scenarios, where usually only a very small number of clients is involved in each session.

Given the fact that most users use the same applications repeatedly, it makes sense to preserve parts of the proxy's short circuiting cache not only while the proxy is running, but also thereafter for the next start of the proxy. In the context of the prototype integration, the action of preserving parts of the proxy's cache is called profiling, since the resulting data is essentially a profile of certain eliminatable round-trip requests.

However, since the peer server or XpleXer for an LBX proxy might have been restarted while the proxy was not running, only the information that is provided by the client can be reused. For example, if the proxy has a property-name-to-atom mapping in its cache, only the property name is saved in the profile. At start-up, the proxy therefore has to request the missing information from the server or XpleXer. Asking the server or XpleXer about the missing information and storing the combined information in the proxy’s short circuiting cache is called cache prefill.

The effect of profiling and cache prefill is that round trips which are exchanged between proxy and server/XpleXer are shifted from the start-up of the applications to the start-up of the proxy. For the user, this should be an improvement, since the proxy is usually started long before it is actually being used. If this is not satisfying, a special LbxCachePrefill request could be used to prefill the proxy’s cache, using only one round-trip request.

The prototype integration supports profiling and cache prefill for property-name-to-atom mapping and colorname-to-RGB-value mapping. The profile for property names and colornames is written to special files in the user’s home directory whenever the proxy’s last client disconnects. At the start-up of the proxy, those files are read and the appropriate InternAtom and LookupColor requests are issued by the proxy in order to prefill the cache.

Profiling and cache prefill is not supported for the mapping of RGB values to the index and the contents of a colorcell. The reason for this is that the cache prefill for the mapping would require the allocation of shared colorcells that might never be used. Since colorcells are a scarce resource in the server, this should not be done.

5.2.4 Tag Usage

As mentioned above, the LBX tag mechanism cannot be integrated into the Xplexer: The XpleXer is not aware of clients that connect directly to the primary server and is therefore not able to invalidate tags or request the transmission of property data.

However, a product-level integration could replace the LBX tag mechanisms with a caching strategy that does not require knowledge about the actions of clients that are directly connected to the primary server. To realize this strategy, the proxy and the XpleXer would keep identical caches for the last replies to GetKeyboardMapping, GetModifierMapping, QueryFont and connect requests. If one of those requests arrives at the XpleXer, it would be forwarded to the primary server. The reply of the primary server would be compared with the content of the cache in the XpleXer. If the content of the cache and the reply are identical, the XpleXer would send only a short reply which tells the proxy to send the content of proxy’s cache as a reply to the client. If the content of the cache and the reply are different, the whole reply would be sent to the proxy, and the caches in XpleXer and proxy would be updated.

5.2.5 Reencoding

The decoding of reencoded messages had to be integrated into the XpleXer in a way that made sure that a request is decoded only once, even if the client is shared by multiple servers. Usually the XpleXer dispatches incoming requests once for every server. For reencoded messages, this would have led to an unacceptable increase in decoding overhead. In order to decode a reencoded request only once, it is now dispatched to an LBX decoding function only for the primary server. This function replaces the encoded request with the decoded equivalent of the request. The decoded request is then dispatched once for every server.

All other parts of message reencoding were integrated into the XpleXer without major modifications.

5.2.6 Motion Event Suppression

Motion event suppression was integrated into the XpleXer in order to reduce the number of MotionNotify events that are generated by the input token holder. All MotionNotify events from other servers are discarded by the XpleXer before they reach the motion event suppression of LBX. For the MotionNotify events of the input token holder, the unmodified LBX motion event suppression was used.

5.3 Performance of the Prototype

For the performance evaluation of the prototype integration, the same tests as for the original LBX were used. In order to make the comparison between the performance of the original LBX and the prototype integration easier, additional lines with the values from the original LBX have been included in Tables 7–9. The following results apply to the integration of LBX into the connection between client and XpleXer. For the connection between XpleXer and server, the original LBX, enhanced by cache prefill and buffering of LZW chunks, is used. Consequently, that part of the integration has compression rates equal to those of the original LBX, and round-trip elimination rates equal to those described in this section.

The results for a newly started proxy with a prefilled short circuiting cache is shown in Table 8. As can be seen, the compression rates for the client data is identical to that of LBX. The compression rates for the server data are slightly lower than for the original LBX because tag usage was not integrated into the prototype.

For a newly started proxy, the tag usage only reduces server data for the reply to a connect request — the proxy knows the reply to the connect request, since it is itself connected to the XpleXer — and for replies to requests which are issued more than once during the start-up of the application. Accordingly, the impact of the missing tag usage on the total compression performance is relatively low.

For this scenario, the prefill of the short circuiting cache leads to a large increase in the eliminated round-trip requests. The average percentage of eliminated round trips is 25.8% for the integration, opposed to only 6.1% for the original LBX.

	xterm	emacs	xtetris	FrameMaker	Netscape
Uncompressed client data (bytes)	2348	9908	5216	55668	347136
Compressed client data (bytes)	1169	4995	2389	19363	180156
Compression rate	2.0:1	2.0:1	2.2:1	2.9:1	1.9:1
(Original LBX)	(2.0:1)	(2.0:1)	(2.1:1)	(2.9:1)	(1.9:1)
Uncompressed server data (bytes)	10532	37064	13760	38660	140608
Compressed server data (bytes)	4284	9184	5059	13834	40626
Compression rate	2.5:1	4.0:1	2.7:1	2.8:1	3.5:1
(Original LBX)	(2.7:1)	(5.2:1)	(2.8:1)	(2.9:1)	(3.7:1)
Uncompressed total data (bytes)	12880	46972	18976	94328	487744
Compressed total data (bytes)	5453	14179	7448	33197	220782
Compression rate	2.4:1	3.3:1	2.5:1	2.8:1	2.2:1
(Original LBX)	(2.5:1)	(3.9:1)	(2.6:1)	(2.9:1)	(2.3:1)
Round trips without short circuiting	11	125	13	167	484
Eliminatable round trips	4	49	8	31	371
Round trips eliminated	3	39	4	26	117
Percentage of round trips eliminated	27.3	31.2	30.8	15.6	24.2
(Original LBX)	(0)	(10.4)	(0)	(0)	(20.0)

Table 8 Application start-up with prefilled short circuiting cache

Table 9 shows the second scenario with an optimally filled cache. As in the first scenario, the compression rates for the client data are nearly identical. For the server data, the original LBX achieves significantly better compression results than the prototype integration. The reason is obvious: With an optimally filled cache, the usage of tags is a big advantage, especially for ‘small’ applications like xterm or emacs. However, for applications that transfer a large amount of data at start-up, like FrameMaker or Netscape, the overall compression rates of the original LBX and the prototype integration are still fairly similar.

As can be seen in Table 9, for this scenario the elimination of round trips is not affected by the integration into the XpleXer.

	xterm	emacs	xtetris	FrameMaker	Netscape
Uncompressed client data (bytes)	2484	9872	4992	55772	341324
Compressed client data (bytes)	945	3963	2106	19650	177595
Compression rate	2.6:1	2.5:1	2.4:1	2.8:1	1.9:1
(Original LBX)	(2.5:1)	(2.8:1)	(2.5:1)	(2.9:1)	(1.9:1)
Uncompressed server data (bytes)	10532	36968	13600	39332	142244
Compressed server data (bytes)	3112	7114	3872	13810	40657
Compression rate	3.4:1	5.2:1	3.5:1	2.8:1	3.5
(Original LBX)	(23.4:1)	(11.7:1)	(5.5:1)	(4.8:1)	(5.2:1)
Uncompressed total data (bytes)	13016	46840	18592	95104	483568
Compressed total data (bytes)	4057	11077	5978	33460	218252
Compression rate	3.2:1	4.2:1	3.1:1	2.8:1	2.2:1
(Original LBX)	(9.2:1)	(7.0:1)	(4.2:1)	(3.5:1)	(2.4:1)
Round trips without short circuiting	11	125	13	167	484
Eliminatable round trips	4	49	8	31	371
Round trips eliminated	3	45	6	30	361
Percentage of round trips eliminated	27.3	36.0	46.2	18.0	74.6
(Original LBX)	(27.3)	(36.0)	(46.2)	(18.0)	(74.6)

Table 9 Application start-up with filled cache

From the data in Table 10 can be learned that, for application usage, the integration has basically the same compression rates and round-trip elimination percentages as the original LBX.

The evaluation shows that the performance of the original LBX and the prototype integration are similar for most cases. The original LBX has a significantly higher compression rate only for application start-up with an optimally filled cache. For a product-level integration, the lack of tag usage could be compensated by the integration of the caching strategy described in 5.2.3. Due to cache prefill, the prototype integration has a significantly higher round-trip elimination rate as the original LBX for a newly started proxy.

	xterm	emacs	xtetris	FrameMaker	Netscape
Uncompressed client data (bytes)	7004	13888	120744	1364448	6980512
Compressed client data (bytes)	3237	3905	19909	485509	1042329
Compression rate	2.2:1	3.6:1	6.0:1	2.8:1	6.7:1
(Original LBX)	(2.2:1)	(3.6:1)	(6.1:1)	(2.8:1)	(6.6:1)
Uncompressed server data (bytes)	4832	14528	17920	92568	66592
Compressed server data (bytes)	1240	4259	4915	25954	18094
Compression rate	3.9:1	3.4:1	3.6:1	3.6:1	3.7:1
(Original LBX)	(4.6:1)	(3.9:1)	(3.6:1)	(3.4:1)	(3.8:1)
Uncompressed total data (bytes)	11836	28416	138664	1457016	7047104
Compressed total data (bytes)	4477	8164	24824	511463	1060423
Compression rate	2.6:1	3.5:1	5.6:1	2.8	6.6:1
(Original LBX)	(2.7:1)	(3.8:1)	(5.6:1)	(2.8:1)	(6.5:1)
Round trips without short circuiting	4	2	2	781	77
Eliminatable round trips	0	0	0	6	10
Round trips eliminated	0	0	0	4	7
Percentage of round trips eliminated	0	0	0	0	9.1
(Original LBX)	(0)	(0)	(0)	(0)	(4.8)

Table 10 Application usage

6 Future Work

This chapter identifies future work that should be done in order to transform the prototype into a product-level integration. The first section discusses the integration of LBX into the distributable architecture of the XpleXer. Sections 6.2 through 6.4 present approaches to improve the X protocol enhancement and compression techniques described in Chapter 4. The last two sections cover miscellaneous topics that must be take into account in order to make X protocol enhancement and compression more efficient for an integration into application sharing.

6.1 Supporting the Distributable Architecture

Support for the distributed part of the XpleXer must be added to the prototype for a full integration of LBX into the distributable architecture of the XpleXer. Figure 31 shows a scenario where the distributed XpleXer is used to share two clients. For client 1, XpleXer 1 is the master and XpleXer 2 is the slave. For client 2, XpleXer 2 is the master and XpleXer 1 is the slave. The only connections that cross LAN boundaries are the connections between the two XpleXers. The communication between master and slave is based on the X protocol with an extension for additional information.

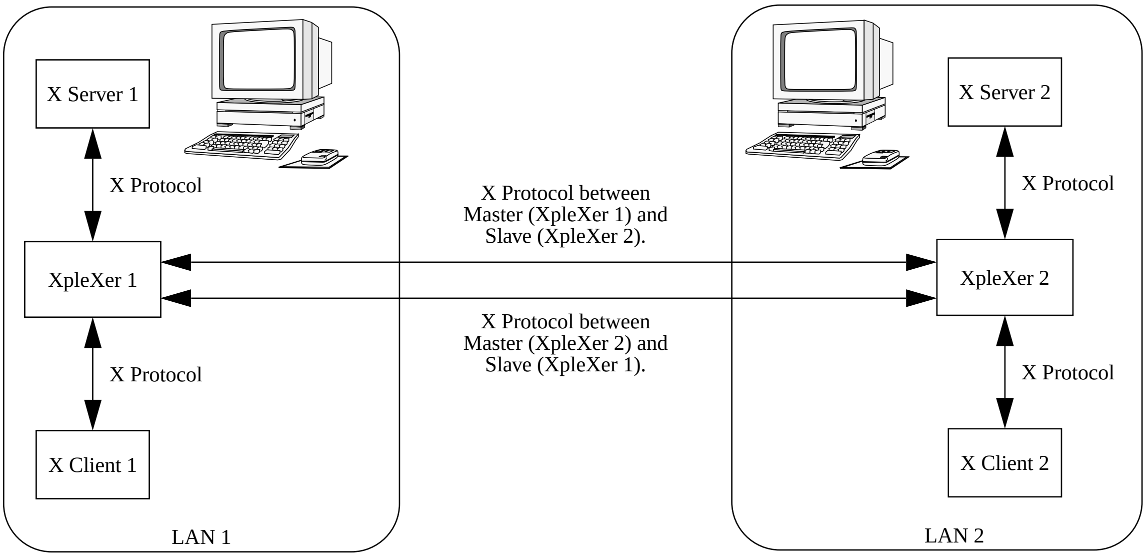


Figure 31 Architecture of the distributed XpleXer without LBX

The architecture for an integration of LBX into the distributed part of the XpleXer is shown in Figure 32. For most parts of this integration, the code of the prototype integration can be reused. However, there is one additional problem that must be solved: While the communication between master and slave is based on the X protocol, there exists one major difference between the protocol that is used between client and server and the protocol that is used between master and slave. For some information requests, the client is interested only in the reply from the primary server. In those cases, it is not necessary for the slave to send the replies from the additional server to the master. Instead, the information contained in those replies is stored in the slave for the mapping of future requests. To make the LBX proxy work with the slave, it needs to be modified to accept this behavior. This is the major task for the integration of LBX into the distributed architecture.

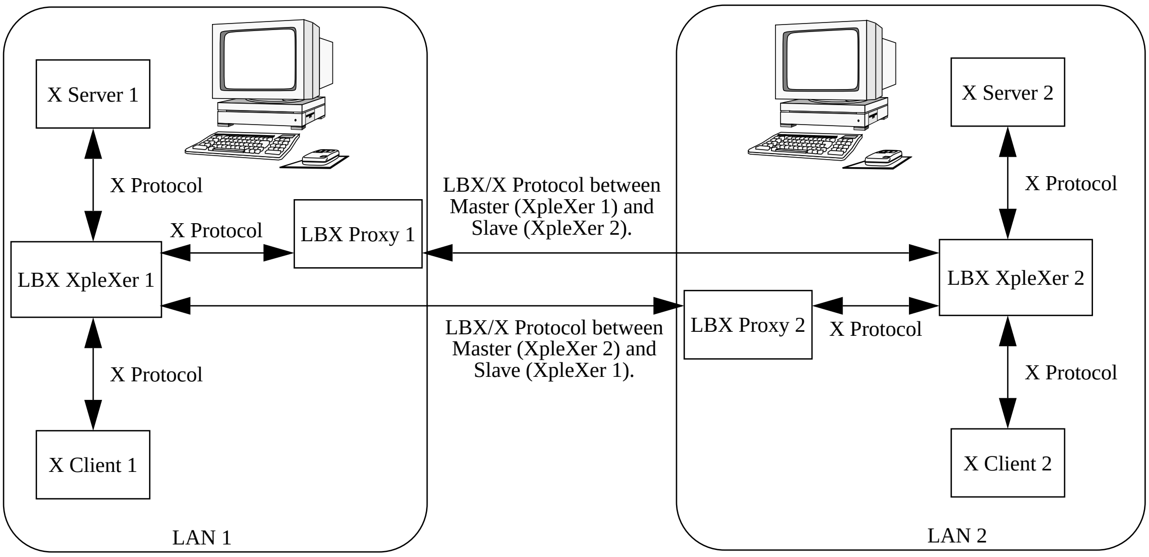


Figure 32 Architecture of the distributed XpleXer with LBX

The combined integration of LBX into the centralized and the distributed part would provide full support for the distributable architecture of the XpleXer.

6.2 The New Low Bandwidth X

The final LBX standard for protocol version 1.0 [6] was published in January 1997, after the prototype integration was completed. This section gives an overview of the additional features of the final standard.

A lot of work has been done to enable short circuiting of AllocColor and AllocNamedColor requests for the case where the short circuiting cache of the proxy is empty. In order to make this possible, a mechanism called colormap grabbing has been developed. Colormap grabbing allows the proxy to gain control over a colormap, when a client connected to that proxy starts issuing AllocColor or AllocNamedColor requests. In order to grab a colormap, the proxy issues an LbxGrabCmap request. The server replies to this request with the description of the grabbed colormap. Thereafter the proxy controls the colormap: It is allowed to allocate colors in the colormap and to reply to AllocColor or AllocNamedColor requests without contacting the server first. The server is updated with regard to the status of the colormap by the one-way request LbxAllocColor that contains the same data as the AllocColor request. The LbxAllocColor request is sent by the proxy as a substitution for short circuited AllocColor and AllocNamedColor requests. If other clients try to allocate colors in a grabbed colormap, the server sends an LbxReleaseCmapEvent to the proxy. Upon receiving this event, the proxy passes the control of the colormap back to the server by sending an LbxReleaseCmap. Other clients are allowed to deallocate colors in a grabbed colormap. If the counter for a specific colorcell reaches zero, the server informs the proxy by sending an LbxFreeCellsEvent.

Another convenient feature that has been added to the LBX standard is a way to intern more than one atom with a single round-trip request. This is done by using the LbxInternAtoms request. With the presence of this request, the cache prefill for property name to atom mapping can be done in a single round-trip request.

With colormap grabbing and cache prefill, the full power of short circuiting can be used right after the start of the proxy. The time necessary for the prefill of the atom cache can be dramatically reduced by using LbxInternAtoms. An additional LbxLookupColors request would be very desirable to reduce the whole cache prefill to two round trips.

Arbitrary algorithms can be used for the stream compression of the new LBX. The implementation provided by the X Consortium supports the DEFLATE compressed data format [13], which is a combination of the LZW algorithm and Huffman coding. The usage of the DEFLATE compressed data format should not significantly change the compression performance of LBX because of its similarity to the LZW compression that is used by the work-in-progress version of LBX.

The remaining parts of the new LBX standard are similar to those of the work-in-progress version described in Chapter 4. The main problem with an integration of the new LBX standard into application sharing would therefore be the integration of colormap grabbing. The algorithms for colormap grabbing require the knowledge about actions of clients that are directly connected to the server. As explained in Chapter 5, the same reason prevented the integration of tag usage into the prototype. A proposal of how this problem could be solved is presented in section 6.3.

6.3 A Transparent Architecture for Low Bandwidth X

One of the main disadvantages of LBX is that the architecture is not transparent to the X window environment of the user. With the architecture proposed by LBX, everyone who wants to use LBX must install a new X server. This is hardly optimal, taking into account the huge installed base of X servers and the large number of operating systems and hardware platforms to which the X server code has been ported. For application sharing, a transparent architecture would be even more important, since only the transparent parts of LBX can be integrated into application sharing. A very interesting question is therefore:

"Can the LBX architecture be converted into a transparent architecture, retaining support for all compression and short circuiting mechanisms?"

To answer this question one has to reconsider why a non-transparent architecture is proposed for LBX. The reason is that the X server has information about all connected clients. A pseudo client, as it is used by XRemote or HBX/FHBX, does not have information about clients that are directly connected to the server. This information is essential for tag usage and colormap grabbing.

If it were possible to give the pseudo client information about all clients of a server, all LBX mechanisms could be used with a transparent architecture. How the pseudo client could get this information is shown in Figure 33. The key idea is that no client would be allowed to connect directly to the server. Even local clients would use the LBX pseudo client to communicate with the real server. The LBX pseudo client would forward most of the requests unchanged from local clients to the server. Only requests that affect tag usage or colormap grabbing would be examined closer and appropriate actions — like invalidating tags, requesting property data or handling requests for grabbed colormaps — would be taken.

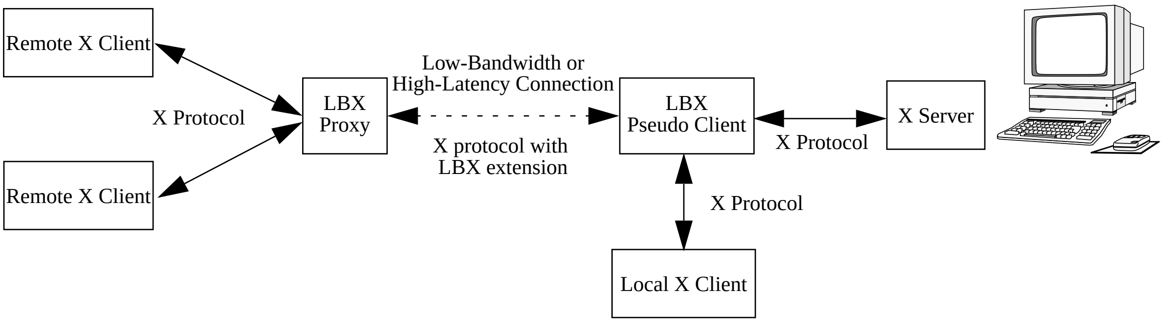


Figure 33 Transparent architecture for LBX

For an integration into application sharing, the XpleXer could do the job of the pseudo client in addition to its usual application-sharing functionality. With the guarantee that all clients of the primary server connect through the XpleXer, all LBX mechanisms could be integrated into application sharing.

6.4 Combining Different X Protocol Enhancement and Compression Techniques

As shown in Chapter 4, both the HBX/FHBX/dxpc and the LBX approach have different advantages. HBX/FHBX/dxpc usually achieves a higher compression rate, while LBX supports short circuiting and motion event suppression as well as superior tag usage and image compression. A combination of the different X protocol enhancement and compression techniques is therefore desirable.

Figure 34 proposes the functional layers for a combination of LBX and HBX/FHBX/dxpc. The LBX preprocessing layer of the combination would be responsible for short circuiting, tag usage, reencoding of images and motion event suppression. A HBX/FHBX/dxpc layer would replace the delta compaction and compression layers of LBX. The data received from the preprocessing layer would be compressed in the HBX/FHBX/dxpc layer by using prediction based compression.

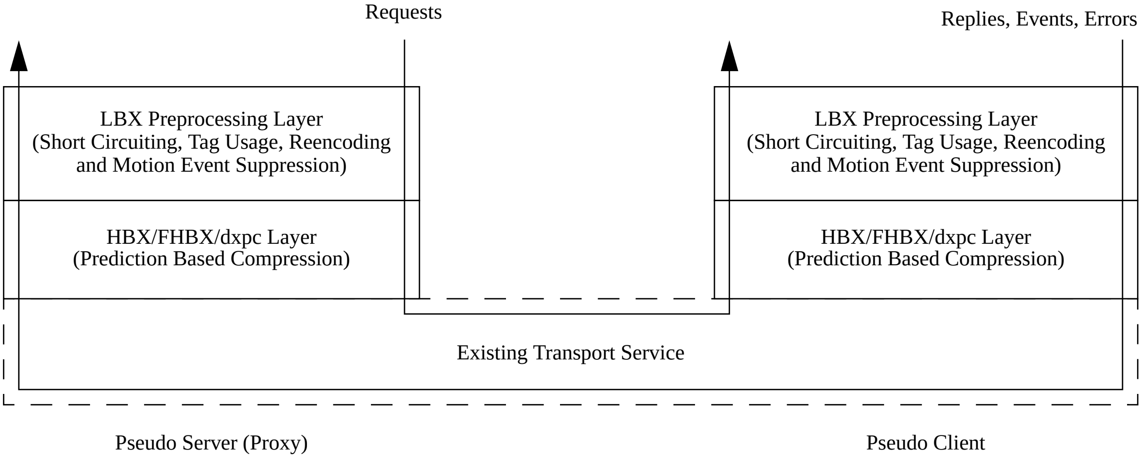

Requests
Replies, Events, Errors
LBX Preprocessing Layer
(Short Circuiting, Tag Usage, Reencoding
and Motion Event Suppression)
HBX/FHBX/dxpc Layer
(Prediction Based Compression)
LBX Preprocessing Layer
(Short Circuiting, Tag Usage, Reencoding
and Motion Event Suppression)
HBX/FHBX/dxpc Layer
(Prediction Based Compression)
Existing Transport Service
Pseudo Server (Proxy)
Pseudo Client

Figure 34 Functional layers for a combination of LBX and HBX/FHBX/dxpc

Such a combination of different X protocol enhancement and compression techniques should be able to achieve an average compression rate higher than the one of HBX/FHBX/dxpc, and synchronicity reduction rates equal to those of LBX.

6.5 Transport Layer Support

This thesis focuses on the optimization of the X protocol being used over existing transport services. However, the used transport services do have a significant impact on synchronicity and bandwidth of X-based application sharing. Forward error correction and multicast are two examples, where transport layer services could enhance the communication for application sharing.

6.5.1 Forward Error Correction

With the usage of TCP, the average round-trip delay depends not only on the latency, but also on the loss rate of the underlying network. When a client issues a round-trip request, it waits until it receives the reply from the server. If packet loss occurs either for the request or the reply, the round-trip delay is increased by the time TCP needs for a time-out and retransmission of the lost packet. One-way requests are less sensitive to the loss rate of the network since the client does not have to wait for a reply to the request. Because of this difference between round-trip and one-way requests, the X protocol can be considered to have two priorities: a higher priority for data that belongs to a round-trip request and a lower one for the remaining data. If applied to the higher priority data, forward error correction could reduce the round-trip delay for lossy networks by decreasing the number of retransmissions. It should be interesting to investigate how existing forward error correction schemes like PET [1] or AdFEC [23] could be used to enhance the X protocol performance.

6.5.2 Multicast

A reliable multicast protocol could tremendously reduce the required bandwidth for the data sent from the master to the slaves: Application-sharing sessions with many participants would basically require a similar amount of bandwidth as sessions with two participants. Only the data sent from the slaves to the master would require unicast. This is not a major limitation since usually only the connection to the slave of the input token holder is active. All other slaves filter the replies, errors and events from their servers and do not send them to the master. A combined integration of multicast and X protocol enhancement and compression techniques into the XpleXer should result in an optimal reduction of the used bandwidth.

6.6 Graceful Quality Degradation

For very low-bandwidth connections like modem or cellular modem connections, the user might be interested in trading quality for speed.

Since the transfer of images is one of the most bandwidth-consuming activities, it would be interesting to investigate how graceful degradation of image quality could be included in X protocol enhancement and compression. In this context, graceful degradation would mean a reduction in quality while retaining the essential information for the user. The algorithms for graceful degradation would have to work in real time: intercepting the image sent by the client, performing graceful degradation of the image and then sending the degraded image over the low-bandwidth connection to the server. The degradation of images could be performed by reducing the number of colors, by reducing the resolution, or by employing existing lossy image compression techniques. A similar problem is currently been investigated for world wide web access over low-bandwidth connections [15].

Another example where graceful quality degradation might be acceptable for the user is the refreshing of windows. Usually the application is asked to repaint the content of a window, as soon as parts of it that have been obscured become visible. Over a very low-bandwidth connection it might make sense to let the user decide when to repaint which window. Typically a user would request the refreshing of a window after finishing window manipulation (resizing, moving, raising and lowering windows).

Other parts of X might qualify for graceful quality degradation also. Overall, it should be very interesting to examine the impact of graceful quality degradation on the used bandwidth and the perceived quality.

7 Conclusion

The topic of this thesis is the integration of protocol enhancement and compression into X-based application sharing.

Application sharing is “... a technology which allows two or more users located at geographically different places to synchronously work with a single-user application, i.e. online and at the same time” [25]. To make this technology available to the network-based X Window System, several different software products have been developed. As shown in this thesis, virtually all of them use a protocol similar to the X Window System protocol to display the output of a single user application on more than one screen and to receive response from more than one user. However, this protocol was designed to be run over a fast LAN. Used over a high-latency or a low-bandwidth connection, it leads to serious delays and loss of interactivity. In these environments, the start-up of a standard application like Netscape can easily take more than 90 seconds. For low-bandwidth or high-latency connections, the X protocol must be enhanced in order for X-based application sharing to become a universal cooperation technology.

In this thesis, existing approaches to enhance the X protocol for low-bandwidth or high-latency connections were reviewed, and an integration of one of those approaches into the XpleXer, an application-sharing product developed by Siemens, was presented.

The X protocol enhancement and compression approaches that were summarized in this thesis are: XRemote, Low Bandwidth X (LBX), Higher Bandwidth X, Fast Higher Bandwidth X and dxpc. For the integration, LBX was the compression method of choice since it is the only approach that supports high-latency connections by eliminating synchronous X protocol requests. LBX uses stream and delta compression as well as several request-specific algorithms to reduce the amount of data that must be sent over a low-bandwidth connection. Higher Bandwidth X, Fast Higher Bandwidth X and dxpc use prediction-based compression to achieve better compression results as LBX. However, none of them supports high-latency connections.

In order to prove that LBX can be integrated into application sharing, a prototype integration into the XpleXer has been done. Several modifications had been necessary to fit LBX into the new environment, including changes to the LBX protocol. In order to make LBX more efficient, cache prefill and LZW packet buffering were added to the original LBX. The overall compression rate of the prototype integration is about 3.3:1, and at the start-up of an application, up to 74% of the synchronous X protocol requests can be eliminated. With these results, application sharing over low-bandwidth or high-latency connections is considerably improved.

While working on the prototype integration, several areas of future work were identified. These areas include further improvements for X protocol enhancement and compression as well as advanced, product-level integration of those techniques into application sharing. The prototype integration only scratches the surface of what is possible. A future product-level integration should be able to achieve compression rates of at least 6:1 with a very stable rate of eliminated synchronous X protocol requests.

References

[1] A. Albanese, M. Luby, "PET - Priority Encoding Transmission," in W. Effelsberg, O. Spaniol, A. Danthine, D. Ferrari (Eds.), High - Speed Networking for Multimedia Applications, pp. 247-262, Kluwer Academic Publishers, 1996.

[2] M. Altenhofen, B. Neidecker-Lutz, P. Tallett, "Upgrading a window system for tutoring functions," Internal Report, DEC Karlsruhe, 1990.

[3] T. C. Bell, J. G. Cleary, I. H. Witten, "Text Compression," Prentice Hall, Englewood Cliffs, NJ, 1990.

[4] J. E. Baldeschweiler, T. Gutekunst, B. Plattner, "A survey of X Protocol Multiplexors," ACM SIGCOMM Computer Communication Review, pp. 16-24, ftp://ftp.tik.ee.ethz.ch/pub/projects/cio/papers/Baldesch93.ps, April 1993.

[5] CCITT, "Recommendation T.6, Facsimile coding schemes and coding control functions four Group 4 facsimile apparatus," vol. VII - Fascicle VII.3, 48-57.

[6] D. Converse, J. Fulton, C. Kantarjiev, D. Lemke, R. Mor, K. Packard, R. Tice, D. Tonogai, "Low Bandwidth X Extension, Protocol Version 1.0," X Consortium Standard, X11R6.3 documentation, ftp://ftp.x.org/pub/R6.3/xc/doc/specs/Xext/lbx.mif, January 1997.

[7] D. Cornelius, "XRemote: a serial line protocol for X," 6th Annual X Technical Conference, Boston, MA, 1992.

[8] S. Carlsen, "TIFF Revision 6.0, Section 9: PackBits Compression," http://www.igd.fhg.de/www/projects/icib/it/defacto/company/aldus/read.html#SEC_9, June 1992.

[9] J. Danskin, P. Hanrahan, "Compression Performance of the XRemote Protocol," 1994 Data Compression Conference. Full paper in Technical Report CS-TR-441-94, Department of Computer Science, Princeton University, Princeton, NJ, http://www.cs.dartmouth.edu/~jmd/decs/DECSpage.html, January 1994.

[10] J. Danskin, "Higher Bandwidth X," ACM Multimedia 94, Second ACM International Conference on Multimedia, 15-20 October 1994, San Francisco, CA, pp 89-96, http://www.cs.dartmouth.edu/~jmd/decs/DECSpage.html.

[11] J. Danskin, "Compressing The X Graphics Protocol," PhD Thesis, Department of Computer Science, Princeton University, Princeton, NJ. Available as Princeton Technical Report CS-TR-465-94, http://www.cs.dartmouth.edu/~jmd/decs/DECSpage.html, November 1994.

[12] J. Danskin, D. Gessel, Q. Zhang, "Fast Higher Bandwidth X," International Multimedia Networking Conference, Aizu Japan, http://www.cs.dartmouth.edu/~jmd/decs/DECSpage.html, Sept 27-29, 1995.

[13] P. Deutsch, "DEFLATE Compressed Data Format Specification version 1.3," RFC 1951, May 1996.

[14] J. Fulton, C. Kantarjiev, "An Update on Low Bandwidth X (LBX)," Proceedings of the 7th Annual X Technical Conference, O’Reilly and Associates, January 1993.

[15] A. Fox, E. Brewer, "Reducing WWW Latency and Bandwidth Requirements by Real–Time Distillation," Fifth International World Wide Web Conference, Paris France, http://www5conf.inria.fr/fich_html/papers/P48/Overview.html, May 6-10, 1996.

[16] D. Garfinkel, B. Welti, T. Yip, "HP SharedX: A Tool for Real-Time Collaboration," Hewlett-Packard Journal April 1994, pp.23-36.

[17] I. Greif, "Computer-Supported Cooperative Work: A Book of Readings," *Morgan Kaufmann Pub. Co.*, Palo Alto, CA, May 1988.

[18] T. Gutekunst, "A Distributed and Policy-Free General-Purpose Shared Window System," IEEE/ACM Transactions on Networking, ftp://ftp.tik.ee.ethz.ch/pub/projects/cio/papers/Gutekunst95.ps, February 1995, pp. 51-62.

[19] G. Hoffmann, "Xy: Unterstützung von Telekooperation und Mobilität im X Window System," Study report, Computer Science Department, Technical University Berlin, 1993.

[20] E. Israel, E. Fortune, "The X Window System Server," Digital Press, Burlington, MA, 1992.

[21] V. Jacobson, "Compressing TCP/IP Headers for Low-Speed Serial Links," RFC 1144, February 1990.

[22] O. Jones, "Introduction to the X Window System," Prentice-Hall International, Inc., London, 1991.

[23] B. Lamparter, O. Böhrer, W. Effelsberg, V. Turau, "Adaptable Forward Error Correction for Multimedia Data Streams", Technical Report TR-93-009, Department of Computer Science, University of Mannheim, December, 1993.

[24] G. McGregor, "PPP Internet Protocol Control Protocol (IPCP)," RFC 1332, May 1992.

[25] W. Minenko, "Advanced Design of Efficient Application Sharing Systems under X Window," Ph.D. Thesis, Department of Computer Science, University Ulm, January 1996.

[26] A. Nye, "Volume 0: X Protocol Reference Manual," The X Window System Series, O'Reilly & Associates, Inc., 1990.

[27] A. Nye, "Volume 1: Xlib Programming Manual," The X Window System Series, O'Reilly & Associates, Inc., 1990.

[28] K. Packard, "Design and Implementation of LBX: An Experiment Based Standard," Proceedings of the 8th Annual X Technical Conference, O'Reilly and Associates, January 1994.

[29] T. Raita, J. Teuhola, "Predictive Text Compression by Hashing," Proceedings of the 10th Annual ACM SIGIR conference on Research and Development in Information Retrieval, New Orleans, 3-5 June 1987, pp.223-233.

[30] J. Ziv, A. Lempel, "A Universal Algorithm for Sequential Data Compression," IEEE Transactions on Information Theory, May 1977, pp. 337-343.