Fermi National Accelerator Laboratory

FERMILAB-Pub-92/17
(NHM-91/003)

HEPnet
NHM TECHNICAL NOTES
Analysis of X Protocol and the Underlying Networking Interface

F. Abar

Fermi National Accelerator Laboratory
P.O. Box 500, Batavia, Illinois 60510

January 1992

Analysis of X Protocol and the Underlying Networking Interface

Farhad A. Abar
Fermi National Accelerator Laboratory
Batavia, IL

Abstract

The X Window System is a network transparent portable window system that is being adopted as a standard by nearly every workstation manufacturer. X Window System provides the environment for distributed graphical applications where the application (X client) and its display (X server) are separate process entities possibly running in different machines. The communication between the application and the display is standardized through a well-defined protocol known as X Protocol over a reliable byte stream network interface.

The physical separation between the server and client can introduce significant performance degradation due to communication delays. These delays are characterized as *latency* delay which results from trip delays between the server and client and *transmission* delay which results from volume of data communicated through networks between server and clients.

This paper investigates and categorizes network load and delay associated with X Window System as implemented under 4.3BSD operating system.

A client-server distributed application is developed to emulate network activities between the X server and client. Based on this application a collection of measurements are conducted to estimate the delays associated with a series of common X requests. The measurements are first done in a LAN environment with server and client running in separate Ethernet subnets. Subnets connectivity is established by routers and the organization backbone Ethernet. The measurement procedure is extended to examine server and client network characteristics in a WAN environment which includes the ESnet national backbone.

Based on the insight from the experimental data an analytical model is constructed to help predict X Window System performance in more general terms.

Table of Contents

This Page Intentionally Left Blank

List of Figures

This Page Intentionally Left Blank

List of Tables

1. Objective

The X protocol as a tool for the distribution of graphical displays is fairly complex. Its impact on today's Local Area Networks (LAN) and Wide Area Networks (WAN) with very limited bandwidth is not clearly understood. By contrast, the X architecture as a client-server model is based on few premises that this paper aims to explain.

The main focus of this paper will be to;

1. illustrate the impact of X as a layer on top of the Internet protocol suites,

2. examine the underlying networking activities present during X applications,

3. provide an initial assessment on the network loads and delays associated with a set of more frequent X activities over a LAN and a WAN configurations, and

4. create an analytical model to predict X Window System performance under various network configurations.

2. X Protocol Introduction

Distributed computing paradigms as computing models have gained considerable popularity with the emergence of powerful and cost effective desktop computers [LIDINSKY]. According to this model of computing the applications are not restricted in using local resources and they should be able to, transparently to the user, access resources located throughout the network. The computing task is divided between the local resource and the remote resource, where the remote resource takes on the responsibility of a *server*, accepting and servicing requests from one or more of processing entities *(clients)* scattered throughout the network.

One key advantage of this distributed model is that each part of the software can be developed independently, hardware and operating system wise, of the others. In a heterogenous computing environment this makes available the services of the special purpose hardware and software.

The network of near future is modeled and constructed to take advantage of distributed computing and cooperating processing paradigm. Open Network Computing Network File System (ONC/NFS) originated by Sun Microsystem, Open Software Foundation Distribute Computing Environment (OSF/DCE) and MIT's X Window System Protocol are examples of this emerging technology.

Lidinsky pointed out that some of key properties that a network must have in order to adequately service client-server systems are:

- Low response time (i.e., network delays)
- Able to handle bursty traffic efficiently or at least cost effectively
- Simple network protocol
- Low intrinsic bit error rate (this allows simple protocols)
- Privacy and security

The X Window System, or X, is a network-transparent window system. With X, multiple applications can run simultaneously on a bitmap display[1]. This has been done before (e.g., kernel based systems such as Windows 3.0 and MACOS) but not in an open environment with high emphasis on portability to many different brands of hardware, from PCs to supercomputers. Network transparency means that application programs can run on machines scattered throughout the network. X allows applications to be device-independent, which eliminates the need for rewriting and recompiling applications in order for them to work with new display hardware.

The X Window system provides a hierarchy of resizable windows and supports high performance device-independent graphics. The most distinguishing aspect of X as

[1] In bitmapped graphics (also referred as raster graphics), each dot on the screen (called a pixel) corresponds to one or more bits in memory. Programs modify the display simply by writing to display memory.

compared to other approaches such as MACOS is that it is based on an asynchronous network protocol rather than on procedure or system calls.

In the X protocol a display is defined as a workstation consisting of a keyboard, a pointing device such as a mouse, and one or more screens. Multiple screens can work together, with mouse movement allowed to cross physical screen boundaries. As long as multiple screens are controlled by a single user with a single keyboard and pointing device, they comprise only a single display.

The bitmapped screens are controlled via the X protocol. For better screen utilization a screen can be divided up into smaller areas called windows. A window is a rectangular area that works in several ways like a miniature screen. Each window on a screen running X can be involved in a different activity. Figure 1 provides the graphical representation of displays, screens, and windows in X protocol.

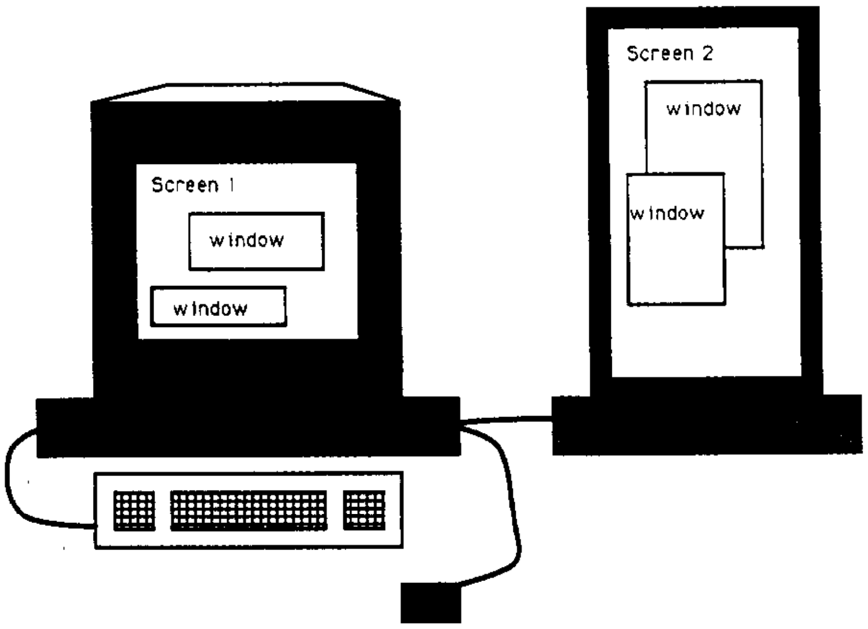

Figure 1: Displays, Screens, and Windows Definitions in X Protocol

The way a kernel-based window system operates is inherent in the window system itself. By contrast, the X Window System concentrates control in a window manager. The window manager largely determines the look and feel of X on a particular system. The window manager is just another X application except that by convention it is given special authority to control the layout of windows on the screen.

X applications can be written solely with *Xlib*. *Xlib* is the C library which includes a low level procedural interface to the X protocol. They could also take advantage of higher level subroutine libraries known as toolkits. Toolkits implement a set of user interface features such as menus or command buttons (referred to generically as toolkit widgets) and allow applications to manipulate these features.

To give the reader an idea of how efficient the X protocol is, "redrawing a normal 80x24 character terminal window using the X protocol would take about 2 seconds of network time at 9600 baud" [NYE]. This calculation takes into account that the X protocol requires some information in addition to the character codes that a hard-wired terminal requires. To show the efficiency of the of the X protocol, a hard-wired terminal would take 80% as much time to refresh an entire screen of the same dimensions at the same serial speed [NYE].

For the cases where a graphics image is being redrawn (as opposed to characters) performance is often limited more by the time required to draw graphics than by the overhead in the protocol.

2.1. X Architecture, Client-Server Model

Most window systems are kernel-based; that is, they are closely tied to the operating system itself and can only run on a discrete system, such as a single workstation. The X Window System is not part of any operating system, but is instead comprised entirely of "user-level" programs. The architecture of the X Window System is based on a client-server model. The system is divided into two distinct parts: display **servers** that provide display capabilities and keep track of user input, and **clients**: application programs that perform specific tasks.[2] The client programs make requests that are communicated to the hardware display by the server. For instance, client software could be running on a powerful remote system, and all the user input and displayed output occur on the PC or workstation server.

X is a network-oriented windowing system. An application need not be running on the same system that actually supports the display. While many applications will execute locally, other applications may execute on other machines, sending requests across the network to a particular display and receiving keyboards and pointer events from the system controlling the display.

X is one component in an overall distributed systems architecture. Distributed applications based on this architectural model are separated into two autonomous and independent yet co-operative softwares modeled as server and client. In this model of computing the computing task is broken up (server and client) and distributed to different systems in a heterogenous computing environment. The server and the client then communicate with each other (send messages) with an agreed set of protocols by means of an underlying network connection.

The following is the description of some of the terminology used in X literature and Figure 2 provides a graphical representation of X server, X clients, and X Window Manager.

[2]X defines "client" and "server" opposite to the way other distributed systems such as ONC/NFS define these terms

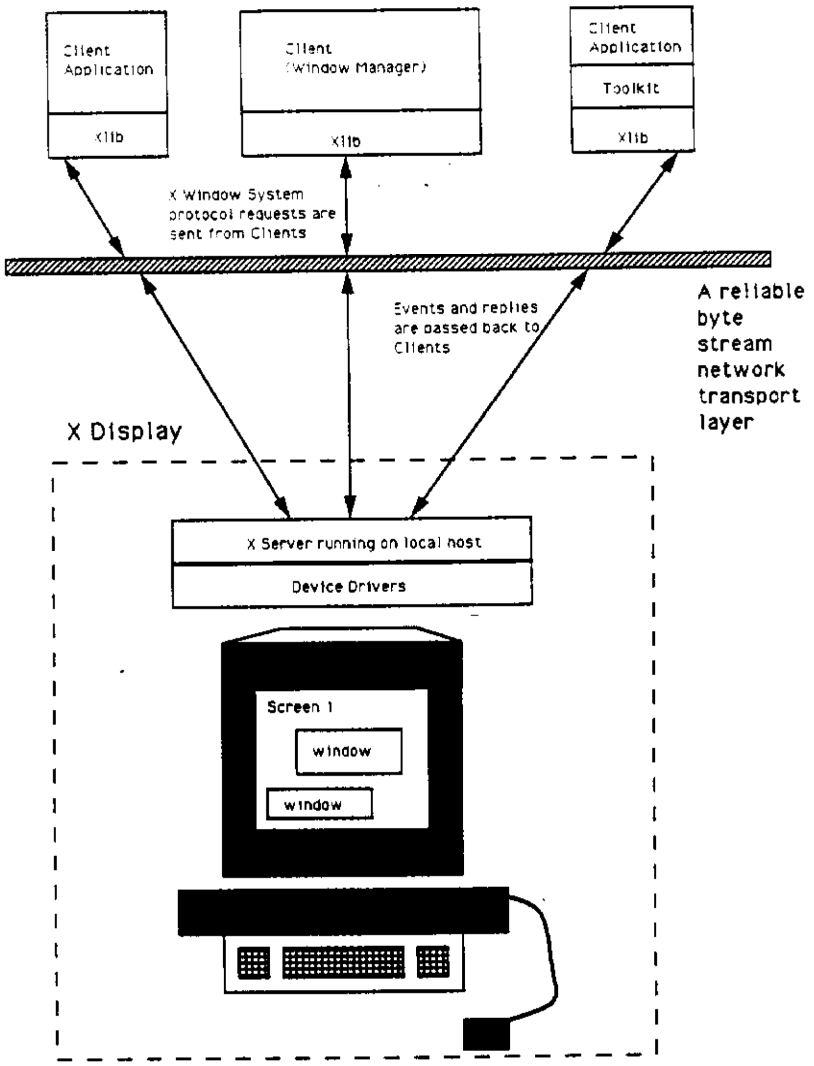
Client Application
Xlib
Client (Window Manager)
Xlib
Client Application
Toolkit
Xlib
X Window System protocol requests are sent from Clients
Events and replies are passed back to Clients
A reliable byte stream network transport layer
X Display
X Server running on local host
Device Drivers
Screen 1
window
window

Figure 2: Server, Client, and Window Manager in X Protocol

In X the program that controls each display (the physical device) is known as a **server.** The terminology is different from other well-known servers such as file or print servers where the server is remotely accessed across the network. In X protocol server is the local process[3] that interact the client which may be running locally or remotely.

The X display server is a program that keeps track of all input coming from input devices such as the keyboard and mouse, and input from other clients that are running. As the display server receives information from a client, it updates the appropriate window on the display. The display server may run on the same computer or on an entirely different machine than a client.

Servers are available for PCs, workstations, and even for special terminals (X terminals) with an integral Ethernet network interface, which may have the server downloaded from another machine or stored in ROM.

[3]A "local" process is a process running on the machine into which the user is physically interacting. A "remote" process on the other hand runs on a machine which is connected via some communication paths or network to the machine where the user is physically interacting.

The server is typically made up of a device-independent layer and a device dependent layer. The device-independent layer includes code that is valid for all machines. The device-dependent part must be customized for each hardware configuration.

The server acts as an intermediary between user program (called **X clients** or **X applications**) running on either the local or remote systems and the resources of the local system.

The server performs the following tasks:

- Allow access to the display by multiple clients.
- Interprets network messages from clients.
- Passes user input to the clients by sending network messages.
- Does two-dimensional drawing-graphics (that are performed by the display server rather than by the client).

The server is designed in such away as to never trust clients to provide viable or executable data. In situation where the sever has to wait for a response from a client, it must be possible to continue servicing other clients. Therefore, the server is designed so that it can interact with the client in non-blocking I/O. This way a bad client or a network failure could never cause the entire display to hang.

Clients are applications that communicate with the server by means of calls to a low-level library (Xlib) implementing X protocol. Xlib provides functions for connecting to a particular display server, creating windows, drawing graphics, responding to events, and so on. Xlib calls are translated to protocol requests that are passed either to the local server or to another server across the network. *xterm*, an X based terminal emulator, *xcalc*, a calculator utility, and *xclock* a clock utility are examples of X clients.

In practice, each user is sitting at a server and can start applications locally to display on the local server or can start applications on remote hosts for display on the local server, if the remote hosts have permission to connect to the local server.

In the process of designing the X protocol, much thought went into the division of responsibility between the server and the client, since this determines what information has to be passed back and forth through requests, replies, and events. Scheifler and Gettys provide an excellent source of information on design of the X protocol [SCHEIFLER].

The decisions ultimately reached were based on portability of client programs, ease of client programming, and performance.

2.2. X Protocol Definition

X protocol is what defines the X Window System. It is designed to allow many different types of machines to cooperate within a network and to communicate all the information necessary to operate a window system over a single asynchronous bidirectional stream of 8-bit bytes. This was one of the major innovations in the X design.

The X protocol can be implemented using a wide variety of languages and operating systems. *Xlib* is the C language implementation of X protocol. There is also a Lisp interface. A program in any programming language that can generate and receive X protocol requests can communicate with a server and be used with the X Window System. At present, *Xlib* is the most popular programming interface used with X.

The protocol basis and portability of the X window System is especially important today, when it is common to have several makes of machines in a single network.

What the X protocol specifies is the X packet structure and the information within the packets that gets transferred between the server and client in both directions. The same protocol is also used when the server and client are running on the same machine. However, in this case information is transferred via some internal channel instead of the external network.

The X protocol specifies four types of messages that can be transferred over the network. ***requests*** are sent from the client to the server, while ***replies***, ***events***, and ***errors*** are sent from the server to the client.

Padding bytes are required because each network packet generated by X is always a multiple of 4 bytes long, and all 16- and 32-bit quantities are placed in the packet such that they are on 16- or 32-bit boundaries. This is done to make implementation of the protocol easier on architectures that require data to be aligned on 16- or 32-bit boundaries. Length of data in the X protocol are always specified in units of 4 bytes.

Any events caused by executing a request from a given client must be sent to the client before any reply or error is sent.

The following subsections provides the definitions of these four types of messages along with their format, size, and an example of each message type according to MIT X Consortium Standard.

2.3. X Request

A request is generated by the client and sent to the server. A protocol request can carry a wide variety of information, such as specification for drawing a line or an inquiry about the current size of a window. A protocol request can be any multiple of 4 bytes in length.

Format

Every request consists of four bytes of a header followed by zero or more additional bytes of data. The header contains an 8-bit major opcode, a 16-bit length field expressed in units of four bytes, and a data byte. The length field defines the total length of the request, including the header. Unused bytes in a request are not required to be zero. Major opcodes 128 through 255 are reserved for extensions. Every request on a given connection is implicitly assigned a sequence number by the server , starting with 1. This sequence number is used in replies, error, and events.

Sample Request

The AllocColor request specifies which colormap the client wants to use, and the red, green, and blue values for the desired color.

# of Bytes	Type	Values	Description
1		84	opcode
1			unused
2		4	request length
4	COLORMAP		colormap ID
2	unsigned int		red
2	unsigned int		green
2	unsigned int		blue
2			padding

Maximum-request-length specifies the maximum length of a request, in 4-byte units, accepted by the server. This limit might depend on the amount of available memory on the server. The server simply discards the requests longer than this limit. Since the field length is a 16-bit value and is in units of 4 bytes, the maximum request size is 262,140 bytes.

Maximum-request-length will always be at least 4096 (e.g., requests of length up to and including 16,384 bytes will be accepted by all servers).

Appendix C provides a listing of all the X protocol requests with brief description.

2.4. X Reply

A reply is sent from the server to the client in response to certain requests. Not all requests are answered by replies - only the ones that ask for information. Request that specify drawing, for example, do not generate replies, but requests that inquire about the current size of a window do. Replies can be any multiple of 4 bytes in length, with a minimum of 32 bytes. There are no requests that sometimes have replies and other times do not. Replies are always immediate. If the client is a little slow at reading data from the network, the server can get an error from the underlying reliable protocol entity (e.g., TCP) indicating that the network was unable to transmit all the data.

Format

Every reply consists of 32 bytes followed by zero or more additional bytes of data. Replies must be at least 32 byte long. The additional data is stored after the 32 bytes. Every reply contains at least one byte of reply opcode, one data byte, a 16-bit sequence number of the corresponding request, and a 32-bit length field in units of four bytes. The length field specifies the length of additional data after the first 32 bytes.

Sample Reply

Upon receiving an AllocColor request from a client, the server forwards the following reply to its client:

# of Bytes	Type	Values	Description
1		1	reply opcode
1			unused
2	unsigned int		sequence number
4		0	reply length
2	unsigned int		red
2	unsigned int		green
2	unsigned int		blue
2			padding
4	unsigned int		pixel value
12			padding

As one can see the reply length is set to zero since there is no additional data beyond the minimum required 32 bytes.

A protocol request that requires a reply is called a round-trip request. Round-trip requests have to be minimized in client programs because they lower performance when there are network delays.

Appendix D provides the format and a listing of all the requests that have replies.

2.5. X Event

An event is sent from the server to the client and contains information about a device action or about a side effect of a previous request. All events are stored in a 32-byte long structure to simplify queueing and handling them.

The X server is capable of sending many types of events to the client, only some of which most clients need. X provides a mechanism whereby the client can express an interest in certain events but not others. Not only does this prevent wasting of network time on unneeded events, but it also speeds and simplifies clients by avoiding the testing and throwing away of these unnecessary events.

An event is sent from the server to the client and contains information about a device action (keyboard entry) or about a side effect of a previous request (mapping a new window on the screen).

Format

Events are 32 bytes long. Every event contains an 8-bit type code. The most significant bit in this code is set if the event was generated from a **SendEvent** request. Event codes 64 through 127 are reserved for extensions. Every core event (with the exception of **KeymapNotify**) also contains the least-significant 16 bits of the sequence number of the last request issued by the client that was processed by the server.

Sample Event

From the client's point of view, the only true indication that a window is visible is when the server generates an **Expose** event for it. The following is the Expose event, as sent from the server:

# of Bytes	Type	Values	Description
1		1	event code
1			unused
2	unsigned int		sequence number
4	WINDOW		window
2	unsigned int		x
2	unsigned int		y
2	unsigned int		width
2	unsigned int		height
2	unsigned int		count
14			padding

Appendix E provides a listing of all the X11 protocol event types accompanied by a brief description.

2.6. X Error

An error is like an event, but it is handled differently by clients. Unlike events, which are queued by the client library to be read later, errors are dispatched immediately upon arrival to an error-handling routine by the client-side programming library. Error messages are the same size as events (32 bytes).

Format

Error reports are 32 bytes long. Every error includes an 8-bit error code. Error codes 128 through 255 are reserved for extensions. Every error also includes the major and minor opcodes of the failed request and the least-significant 16 bits of the sequence number of the request.

Sample Error

As an example, lets say the client sends a request to draw a line to the server but gets the window and GC arguments reversed. The following is what the server will return as a **BadWindow** error report;

# of Bytes	Type	Values	Description
1		0	error (always zero for error)
1		3	code(BadWindow
2	unsigned int		sequence number
4	unsigned int		bad resource id
2	unsigned int		minor opcode
1	unsigned int		major opcode
21			padding

Error are basically treated just like events all the way to the routine in the client library that receives them. It is at this point that they are sent to the error-handling routine instead of being queued.

Appendix F provides a listing of all the Error reports with a brief description.

2.7. A Sample X Session

The following section describes what happens over the network during a simple application that creates a window, allocates a color, waits for events, draws into the window, and quits. This example uses three of the four types of X network messages as they would occur in an application. The fourth is the error. The sample X server client session is illustrated in Figure 3.

Here are the network events that will take place during a successful X client session:

1. Client opens connection to the server and sends information describing itself.

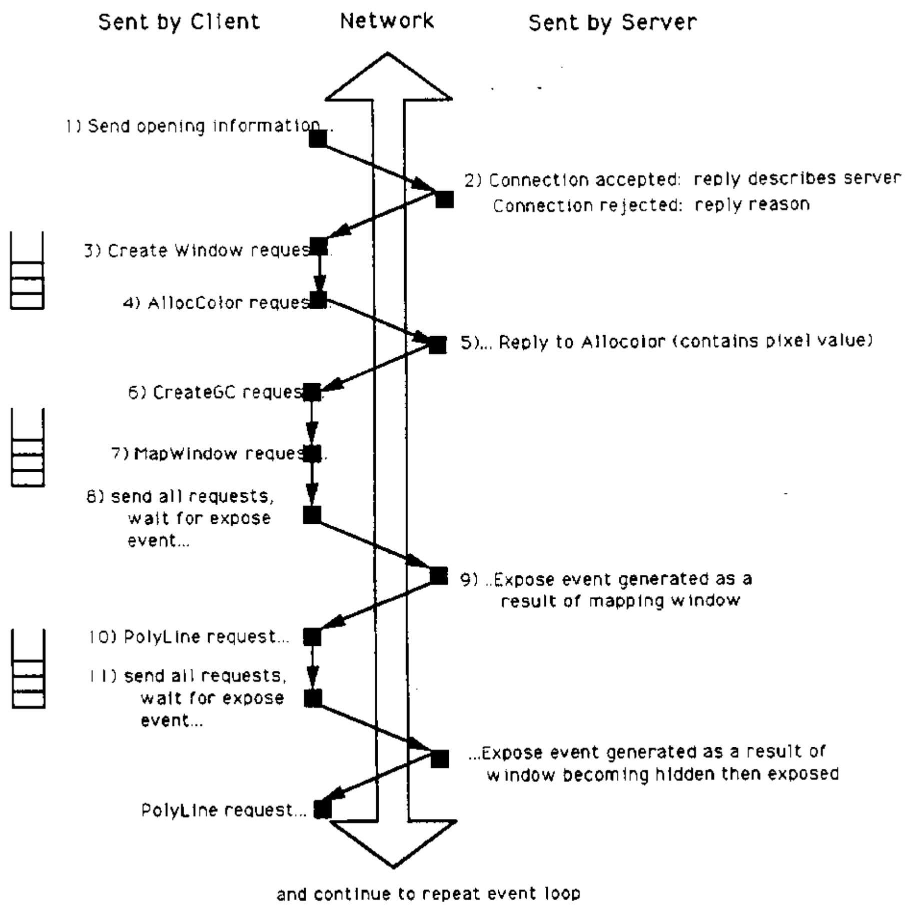
Sent by Client
Network
Sent by Server
1) Send opening information...
2) Connection accepted: reply describes server
Connection rejected: reply reason
3) Create Window request...
4) AllocColor request...
5)... Reply to Allocolor (contains pixel value)
6) CreateGC request...
7) MapWindow request...
8) send all requests, wait for expose event...
9) ..Expose event generated as a result of mapping window
10) PolyLine request...
11) send all requests, wait for expose event...
...Expose event generated as a result of window becoming hidden then exposed
PolyLine request...
and continue to repeat event loop

Figure 3: A Sample X Client-Server Session

2. Server sends back to client data describing the server or refusing the connection request.

3. Client makes a request to create a window. This request has no reply. It is queued up by the client's *Xlib* for subsequent transmission.

4. client makes a request to allocate a color. Since this request requires a reply from the server, the *Xlib* sends all of the pending requests along with it.

5. Server sends back a reply describing the allocated color.

6. Client makes a request to create a graphics context, for using in later drawing requests.

7. Client makes a request to map (display on the screen) the created window.

8. Client makes a request identifying the types of events it requires. In this case, **Expose** and **ButtonPress** events. It then waits for an **Expose**

event before continuing. This sends the accumulated requests to the server.

9. Server sends to client an **Expose** event indicating that the window has been displayed.

10. Client makes a request to draw a graphic, using graphic context.

11. Loop back to wait for an **Expose** event.

The first byte of data in the connection phase identifies the byte order employed on the client's machine. The value 102 (ASCII uppercase B) means values are transmitted most significant byte first, and the value 154 (ASCII lowercase l) means values are transmitted least significant byte first. All 16- and 32-bit quantities, except those involving image data, are transferred in both directions using this byte order specified by the client. X servers are required to swap the bytes of data from machines with different native byte, in all cases except in image processing. The first byte in the packet that opens the connection between the client and the server, sent from the client library, tells the server which byte order is native on the host running the client.

Image data is always sent to the server and received from the server using the server's byte order because image data is likely to be voluminous and byte swapping is expensive. The client is told the server's byte order in the information returned after connecting to the server.

3. X Protocol Communication

The X protocol is designed to communicate all the information necessary to operate a window system over an asynchronous bi-directional stream of 8-bit bytes. From the user's point of view and from the application programmer's point of view the network is transparent since both local and network connections can be operated in the same way using the protocol.

Below the X protocol, any lower layer of network can be used, as long as it is bidirectional and delivers bytes in sequence and unduplicated between a server and a client process. When the client and server are on the same machine, the connection is based on local interprocess communication (IPC) channels such as pipes, shared memory, etc.

Normally, clients implement the X protocol using a programming library that interfaces to a single underlying network protocol, typically TCP/IP or DECnet.

The sample implementation provided by MIT [MIT] of the C language client programming library called *Xlib* uses sockets on systems based on Berkeley UNIX.

Servers are usually designed to understand more than one underlying network protocol so that they can communicate with clients on more than one type of network at once. For example, the DECwindows server accepts connections from clients using TCP/IP or from clients using DECnet. Currently, TCP/IP and DECnet are the two most popular network protocols commonly supported in the X servers.

Figure 4 illustrates the relationship between the X protocol and the Internet suite of protocol. As it is shown in Figure 4, graphics are performed as X applications on the remote host make the appropriate function calls to their *Xlib*. The *Xlib* then translates the function calls to X protocol messages that are understood by X server on the local host. The X messages are then delivered to TCP/IP communication entities within the system kernel through the socket system call interface. From this point on it is the responsibility of the TCP/IP communication layers (TCP, IP, and hardware interface) on both sides to reliably deliver X messages from remote host to X server running on the local host.

Figure 5 illustrates X message encapsulation steps as messages (requests, replies, events, and errors) traverse through TCP/IP network layers down to the Ethernet hardware interface.

As it is shown in Figure 5 it is common for a TCP segment to contain several X messages. This is because X server and *Xlib* buffer X messages instead of sending them immediately to one another, so that they can continue running instead of waiting to gain access to the network. This is possible for several reasons:

- Most requests are drawing requests that do not require immediate action.

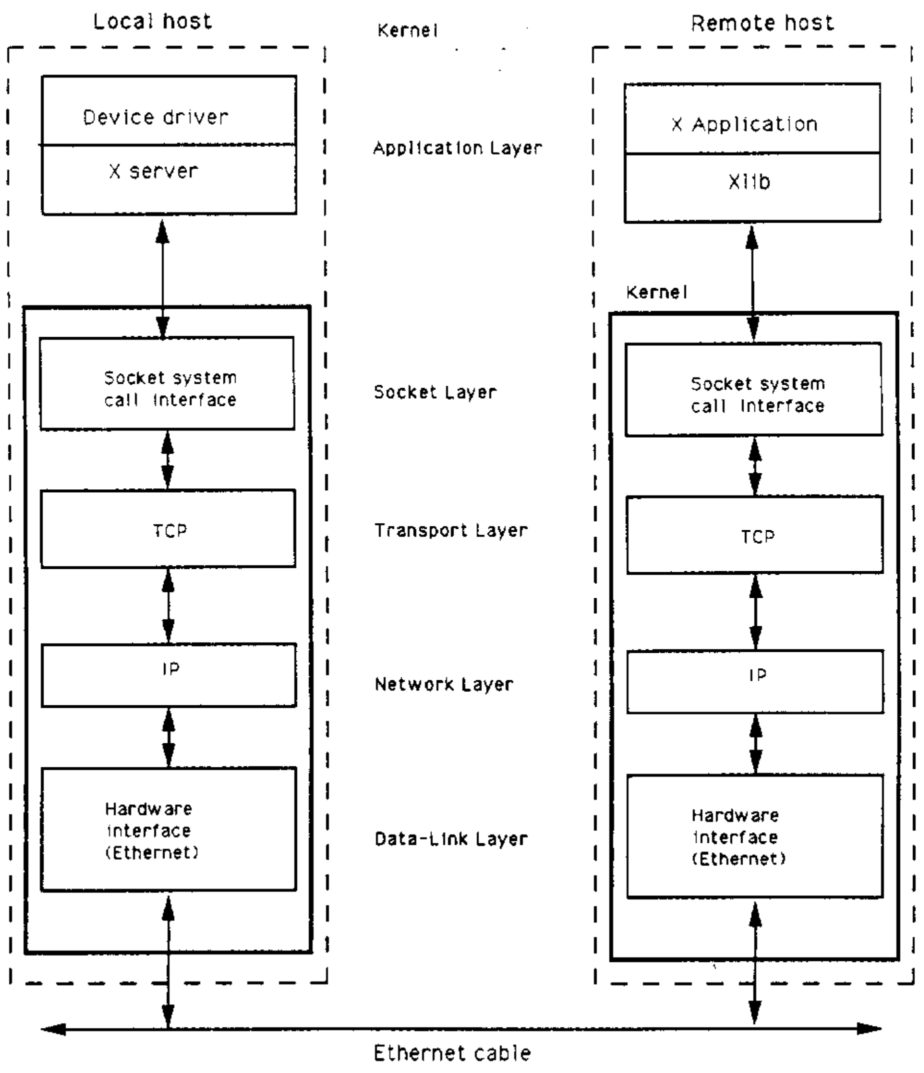

Figure 4: X under TCP/IP Suite of Protocol

- The network stream is reliable; therefore, no confirmation message from the server is necessary.

This X message buffering technique is illustrated in Figure 6.

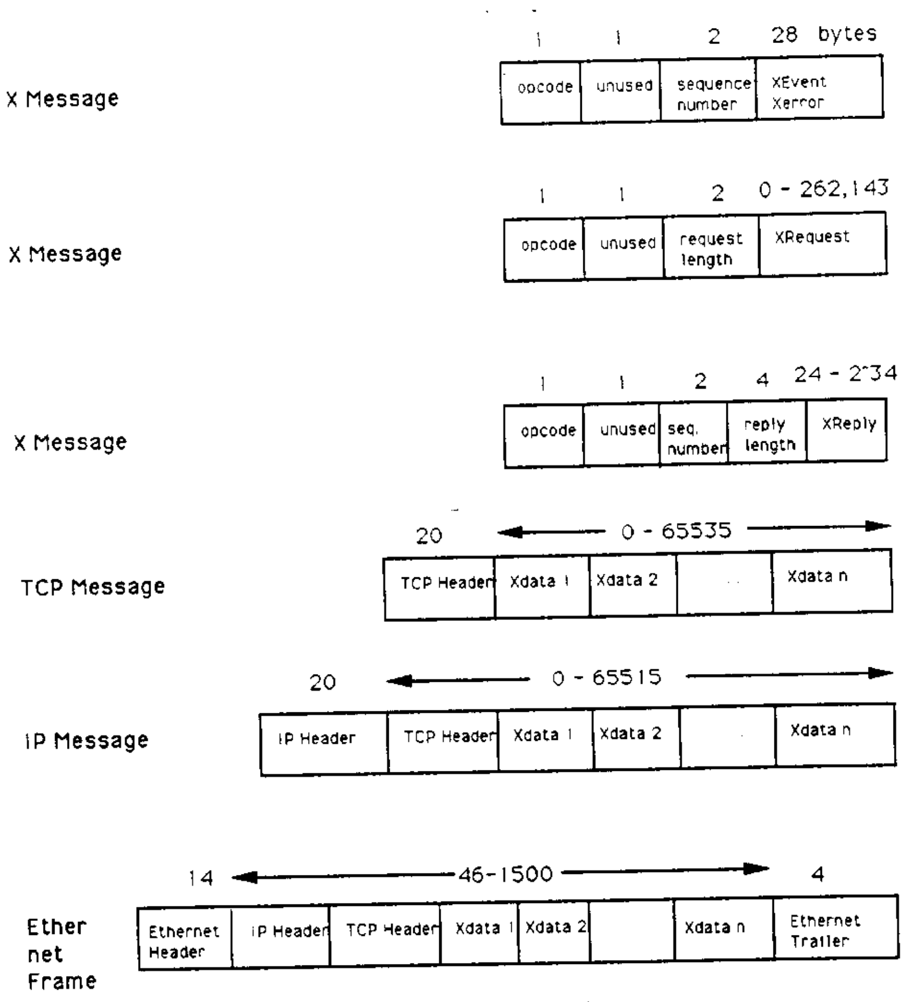

Figure 5: X Protocol Encapsulation under TCP/IP and Ethernet Protocol

3.1. X Protocol Network Calls under 4.3BSD

The following is an overview of network related calls for X protocol under UNIX 4.3BSD operating system. The code segments were obtained from X11R4 source available form MIT project Athena [MIT]. Stevens [STEVENS] provides an in-depth explanations of UNIX network calls.

The information presented here is used later to develop a distributed client-server application which emulated the X client and server to help measure delays involved.

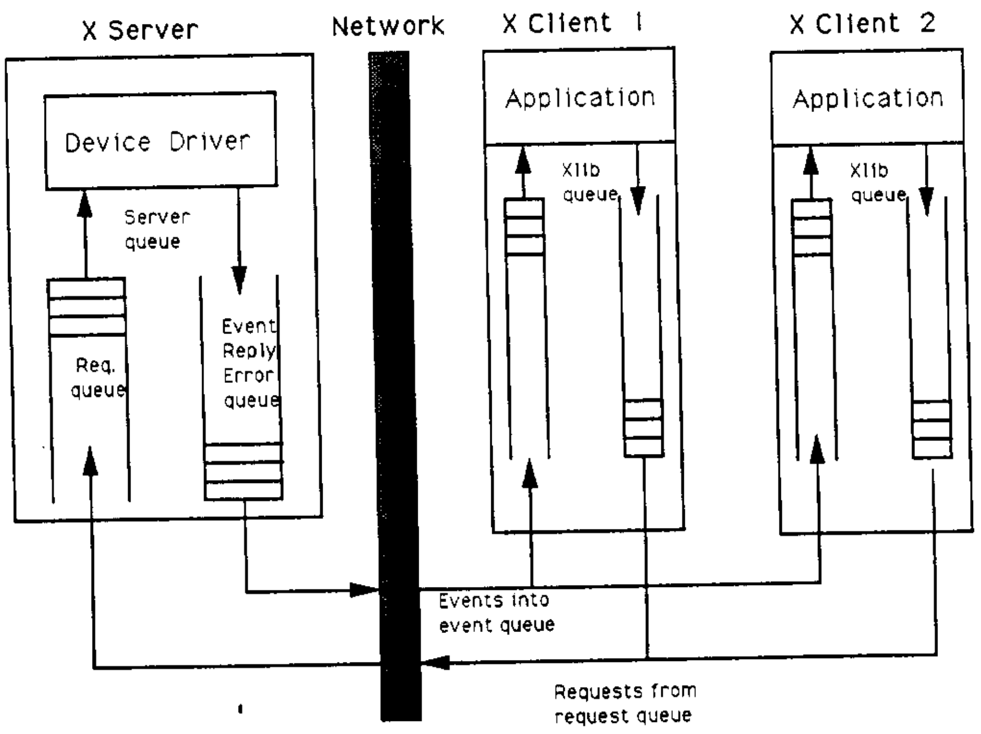

Figure 6: X Server and Clients Queues

Figure 7 is a graphic representation of the X server and client network interaction. The functions in bold format such as **socket()** are system calls specific to 4.3BSD while other functions in regular text (i.e., *CreatewellknownSockets*) are programmer's defined functions in the X server code and *Xlib*.

X Server Networking Interaction

Upon startup, the server initiates network connectivity by issuing the function calls *CreatewellknownSockets* and *Open_TCP_Connection* and subsequently the following system calls as depicted in Box-1 of Figure 7

1. **socket()** system call: A process starts network I/O by specifying the type of communication protocol with the help of *socket* system call. The *socket* system call is analogous to the *open* system call for files and returns a socket descriptor.

2. **bind()** system call: The *bind* system call assigns a name to an unnamed socket descriptor. Basically, X server register its address with the system. It tells the system "this is my address and any X messages received for this address are to be given to me."

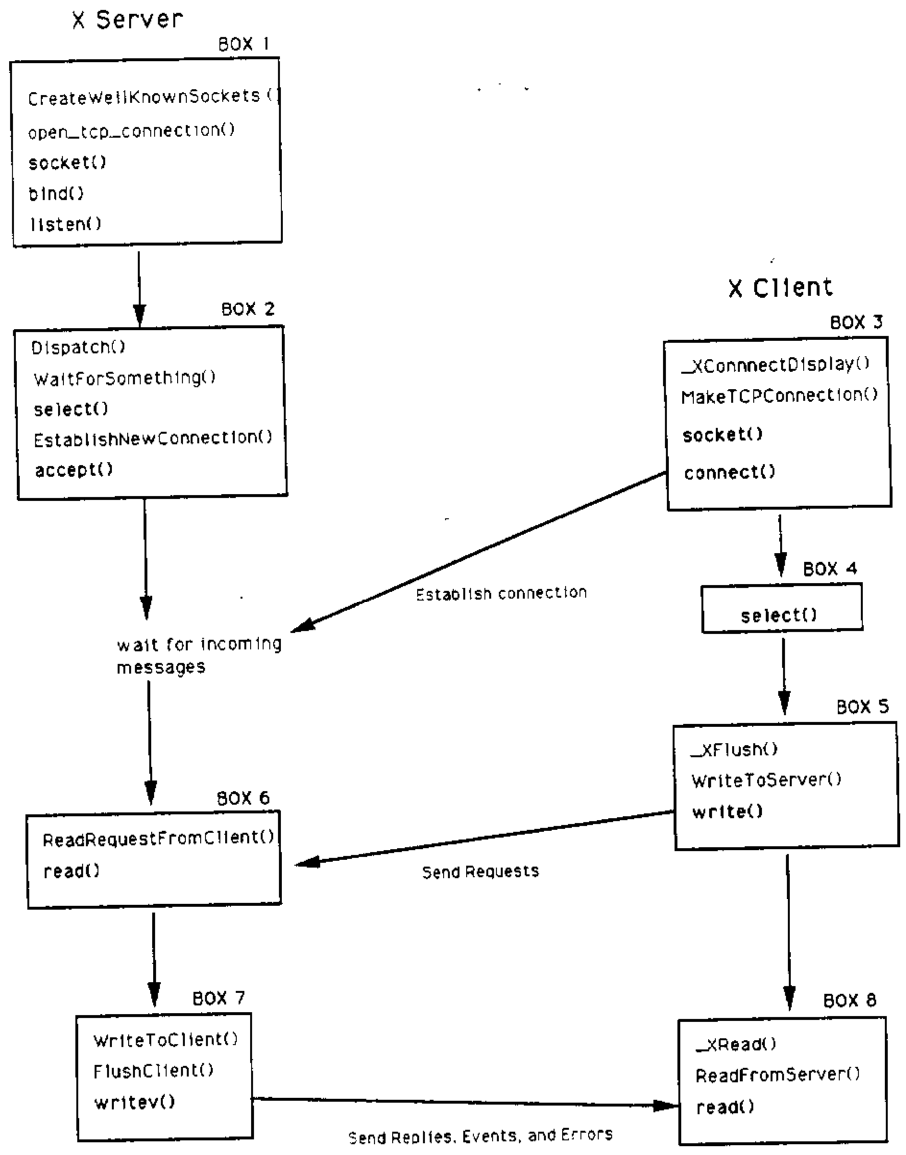

Figure 7: X Server and Client Calls under 4.3BSD

3. **listen()** system call: This system call is used by X server to indicate its willingness to accept connections from X clients.

For TCP connection, displays on a given host are numbered starting from 0, and the server for display N listens and accepts connections on port 6000+N.

Once network initialization process is completed the X server then proceeds to issues network related system calls in Box-2 of Figure 7.

1. **select()** system call: This system calls allows the X server to instruct the kernel to wait for any one of multiple events to occur and to wake up the X server only when one of these events occurs, such as an incoming X client request for connection.

2. **accept()** system call: This system call takes the first connection request on the queue and creates another socket with the same properties as the X server socket.

To reduce the number of small packets and thereby decreasing in the traffic load on a WAN there is a scheme in 4.3BSD to allow only a single small packet to be outstanding on a given TCP connection at any time. In this scheme the process's TCP buffers the small packets until the previous small packet is acknowledged. This is known as TCP coalescence. This scheme could significantly increase network delays for X client-server applications. The *TCP_NODELAY* option is used here by the **setsockopt** system call to defeat this buffering algorithm which allows the X server's TCP to send small packets as soon as possible.

The *fcntl()* system call with the *FNDELAY* parameter designates the new socket as **nonblocking**. This means that any I/O request that can not be done on a non-blocking manner is rejected. This feature does not allow the X server to be hung indefinitely by a bad or slow client.

X messages are then delivered to the server by the TCP entity. Upon their arrival the server proceeds by reading X messages through **read()** system call and processing clients' requests (Box-6).

The X server then returns the appropriate responses back to the X client by delivering its replies to the TCP entity through **writev** system call.

X Client Networking Interaction

The X client attempts to connect to server, given the display name. The display names may be of the following format:

[hostname] : displaynumber [.screennumber]

where the hostname is the Internet address of the X server, and the *displaynumber* and *screennumber* are the desired display and screen controlled by the X server, most often specified as zeros.

131.225.85.1:0.0 is a typical example of a display name.

The absence of hostname is interpreted by the *Xlib* to make the most efficient local connection to the X server on the same machine. This is usually:

- shared memory
- local stream
- UNIX domain socket
- TCP to local host

Within *Xlib* library, the *MakeTCPConnection()* routine sets up the socket data, create a socket and attempts to make a connection to the specified X server (Box-3, Figure 7).

As was the case with the server side, the **setsockopt()** system call is used to turn off the TCP coalescence, and **select()** system call is used for I/O multiplexing (Box-4, Figure 7).

Functions such as *WriteToServer*, *ReadFromServer*, *ReadRequestFromClient*, and *WriteToClient* are used by the X server and *Xlib* which then use the **read())** and **writev()** system calls to exchange messages with its peer entity (X server) (Box 5, 6, 7, and 8, Figure 7).

3.2. X Client-Server Emulation: *emulX*

To obtain the network performance characteristics between two processes utilizing TCP/IP network connection (such as X-server and X-clients), a client-server distributed application (*emulX*) was created which closely emulates the underlying networking interactions between the an X server and an X client as described in the previous section.

It should be noted that *emulX* application objective is to mimic the networking steps that are taken by the X server and client processes under 4.3BSD as were shown previously in Figure 7. The *emulX* client and server simply exchange messages of specific sizes with each other. The transfer mechanism is provided by the reliable byte stream services of the TCP protocol as is the case with the X protocol implementation under 4.3BSD.

The goal of such network interface emulation is to provide lower bounds[4] on delays associated with X protocol messages as they go through only the TCP/IP network layers in a typical X client-server session.

emulX is programmed to act as a server or client. The program acting as as client would accept for arguments the address of the remote host for connection, the size of X request , the corresponding reply size, and finally the number and frequency of transmissions. It then tries to establish a connection over the designated TCP port to the remote host where its peer entity resides. Once connected, and the size of the request and reply messages communicated the client starts up its timer and sends its request to the server. The client then listens on the port and awaits the arrival of the reply message from the server. Upon the receipt of the last byte of message from the server it then stops the timer, and outputs the TCP delay involved in milliseconds.

The *emulX* program emulating an X server is started on a remote host and accepts connection from *emulX* clients on the network. A reply message associated with the incoming requests is immediately transmitted back to its client.

It should be noted that the TCP delay reported is not just the delay between the physical interfaces of the local and remote hosts but also includes the delays associated with Network layers (IP), Transport layers (TCP) and the socket layers.

Figure 8 provides the schematics for this program functionality.

emulX program is used later on to estimate the lower bounds on delays experienced by X clients and X server applications.

Appendix A provides the complete source code for the *emulX* application.

[4]The delays include only the network delays and not X message processing delays

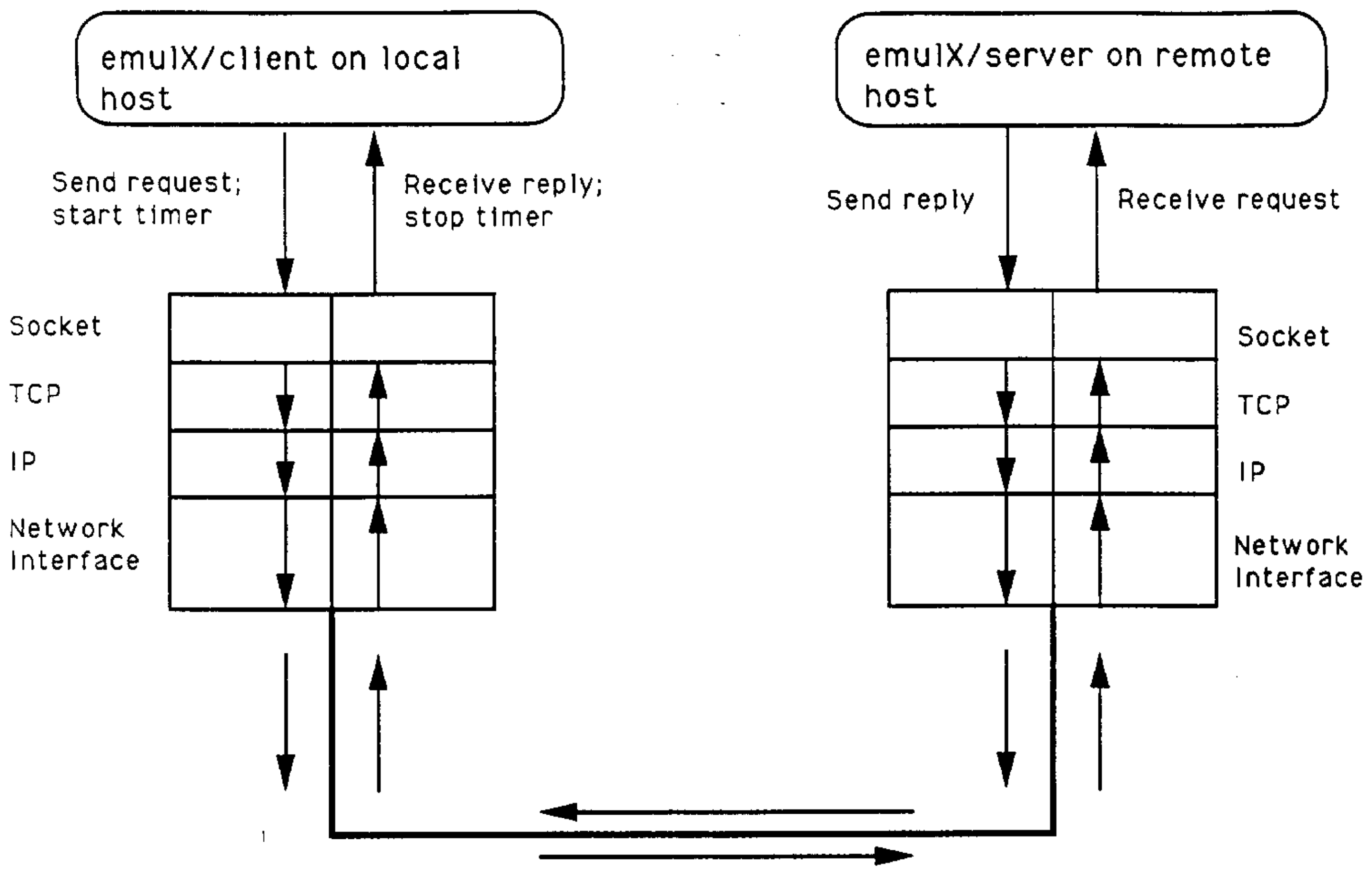
emulX/client on local host
Send request; start timer
Receive reply; stop timer
emulX/server on remote host
Send reply
Receive request
Socket
TCP
IP
Network Interface
Socket
TCP
IP
Network Interface

Figure 8: *emulX* Application Network Interaction

4. X Protocol Network Load Measurements

4.1. Objective

The objective of this section is to identify the network load associated with the X Window System application. Because of great diversities between various X applications, an attempt was made to identify some basic X application's *primitives* which are frequently found within X applications.

4.2. Procedure

The software tools used in identifying the network load imposed by the X applications over a network stream were an X application based on *x11perf* [X11PERF], a network monitoring tool, *tcpdump* [TCPDUMP], and an X protocol monitoring tool, *xscope* [XSCOPE].

The hardware tools for this experiment were two Sun SPARCstation 1+ workstations, one for running the X application and another for monitoring the network, and one Silicon Graphics workstation for running the X server. Figure 9 provides the schematics for this experiment.

x11perf

x11perf is an X11 server performance test program. The **x11perf** application consists of 222 separate tests that stress nearly every aspect of X-server functionality. The **x11perf** command that was executed for the purpose of generating *graphical primitives* such as drawing dots was:

```
% x11perf  -reps 1  -repeat 1  -sync -dot
```

where,

reps 1	fix the repetition count to one. By default x11perf automatically calibrates the number of repetitions of each test, so that each should take approximately the same length of time to run across different servers. We need to by pass the calibration and enforce the program to only execute the desired request only once.
repeat 1	run the test once.
sync	runs the test in synchronous mode. This mode was chosen to prevent the XLIB from queuing the requests. As explained earlier, there is a buffering mechanism in the *Xlib* in which the X requests are not immediately sent to the X server in order to minimize network loads. This option, however, instructs the *Xlib* to by pass the buffering scheme and send the request immediately. We need this mode of operation in order to correctly capture network loads generated by individual X requests.

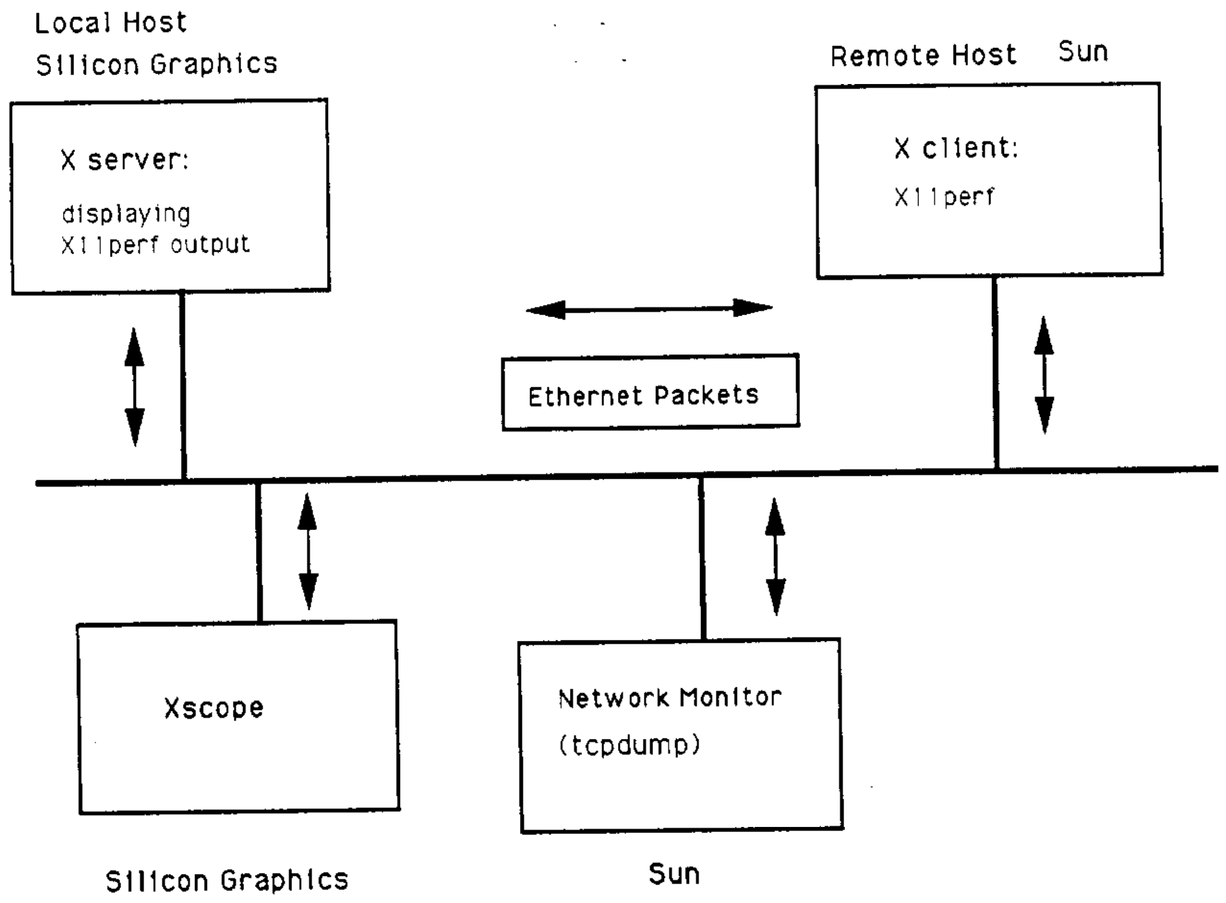

Figure 9: X Network Load Test Bed

dot Generate the request for drawing dots on the screen.

The *x11perf* program was used to setup the connection to an X server, create a window, and perform the desired graphical requests in an synchronous mode. This mode was used to make sure that the Xlib does not provide any queuing of the requests. A separate run for each of the *graphical primitives* under investigation was conducted.

The current program is mostly the responsibility of Joel McCormick. It is based upon the x11perf developed by Phil Karlton, Susan Angebranndt, and Chris Kent, who wanted to assess performance differences between various servers.

tcpdump

tcpdump is a network protocol monitoring tool that records traffic on the Ethernet, the basis of protocol type and source/destination address specification. The **tcpdump** command that was typically executed was:

```
% tcpdump tcp port 6000 and host cdsun and cdsgi
```

where,

tcp port	Port 6000 is typically used in TCP/IP for X data. *tcp port 6000* specifies to **tcpdump** to record all TCP data that was exchanged on port 6000.
host	Record all Ethernet traffic between nodes *cdsun* and *cdsgi*. *cdsun* This was the name of the host where X client *x11perf* was running.
cdsgi	This was the name of the host where X server was running and processing the *x11perf* requests.

The *tcpdump* was run concurrently with the *x11perf* on a different platform monitoring the network medium and reporting all the X protocol related ethernet packets communicated between the two hosts running the X server and the X client.

The program is loosely based on SMI's "etherfind" although none of the etherfind code remains. It was originally written by Van Jacobson, Lawrence Berkeley Laboratory, as part of an ongoing research project to investigate and improve TCP and Internet gateway performance.

xscope

xscope is a program to monitor the connections between the X11 window server and a client program. *xscope* runs as a separate process. By adjusting the host and/or display number that a X11 client attaches to, the client is attached to *xscope* instead of X11. *xscope* attaches to X11 as if it were the client. All bytes from the client are sent to *xscope* which passes them on to X11; All bytes from X11 are sent to *xscope* which sends them on to the client. *xscope* is transparent to the client and X11.

In addition to passing characters back and forth, *xscope* will print information about this traffic on stdout, giving performance and debugging information for an X11 client and server.

The **xscope** command that was typically executed was:

```
% xscope -hcdsgi3 -i1
```

where,

-hcdsgi	Specifies the name of the host where the desired X server is running.
-i1	Tell the *xscope* to listen to port 6001 (6000+1) for the incoming X messages from the host running the *x11perf* application. It should be noted that the display for the *x11perf* was deliberately chosen to be the hostname where the *xscope* is running with port 6001.

xscope was used in this experiment to capture and identify all X protocol messages communicated between the X server and X client, *x11perf*.

4.3. Examples of Common X Client Requests

The following is the listing of some of the common X Client Requests which were generated by the *x11perf* application and a summary of their network activities captured by the network monitoring tool.

The network traffic summary for each request includes:

- Total Ethernet packets sent by the X client to the X server for the given request.
- Total Ethernet packets sent by the X server as a reply to the X client.
- Total X request data [bytes] encapsulated in the Ethernet frames from X client.
- Total X reply data [bytes] encapsulated in the Ethernet frames from X server.
- Sum of X data [bytes] exchanged between the X server and client.
- Total number of bytes exchanged between the local and remote hosts.
- Total network overhead [bytes] (TCP, and IP headers plus Ethernet header and trailer overhead) involved for each X request.

Appendix B provides the synopsis and the raw data collected for these requests.

Opening the Display

Objective: Connects the client to the server controlling the hardware display through TCP, or UNIX or, DECnet streams.

X Client Call: *XOpenDisplay*

Network Traffic:

```
Total Ethernet packets sent by X client: 5
Total Ethernet packets sent by X server: 3
Total X client data generated [bytes]: 56
Total X server data generated [bytes]: 216
Total X traffic [bytes]: 272
Total network traffic [bytes]: 760
Total network overhead [bytes]: 488
```

Creating Windows

Objective: Creates an unmapped *InputOutput* subwindow of the specified parent window.

X Client Call: *XCreateSimpleWindow*

Network Traffic:

```
Total Ethernet packets sent by X client: 2
Total Ethernet packets sent by X server: 1
Total X client data generated [bytes]: 44
Total X server data generated [bytes]: 32
Total X traffic [bytes]: 76
Total network traffic [bytes]: 192
Total network overhead [bytes]: 116
```

Changing Window Attributes

Objective: Changes any or all of the window attributes that can be changed.

X Client Call: *XChangeWindowAttributes*

Network Traffic:

```
Total Ethernet packets sent by X client: 2
Total Ethernet packets sent by X server: 1
Total X client data generated [bytes]: 28
Total X server data generated [bytes]: 32
Total X traffic [bytes]: 60
Total network traffic [bytes]: 240
Total network overhead [bytes]: 180
```

Mapping a Window

Objective: Maps a window, making it eligible for display.

X Client Call: *XMapWindow*

Network Traffic:

```
Total Ethernet packets sent by X client: 2
Total Ethernet packets sent by X server: 2
Total X client data generated [bytes]: 12
Total X server data generated [bytes]: 32
Total X traffic [bytes]: 44
Total network traffic [bytes]: 288
Total network overhead [bytes]: 244
```

Copying Window to Window

Objective: To examine the network load involved in **copying a 100x100 square pixels from window to window.**

X Client Call: *XCopyArea*, combines (copies) the specified rectangle of *src* with the specified rectangle of *dest*. Both *src* and *dest* must have the same root and depth.

Network Traffic:

```
Total Ethernet packets sent by X client: 2
Total Ethernet packets sent by X server: 1
Total X client data generated [bytes]: 32
Total X server data generated [bytes]: 32
Total X traffic [bytes]: 64
Total network traffic [bytes]: 244
Total network overhead [bytes]: 180
```

Moving Window

Objective: To examine the network load involved in **moving a child window.**

X Client Call: *XMoveWindow*, changes the position of the origin of the specified window relative to its parent.

Network Traffic:

```
Total Ethernet packets sent by X client: 2
Total Ethernet packets sent by X server: 1
Total X client data generated [bytes]: 24
Total X server data generated [bytes]: 32
Total X traffic [bytes]: 56
Total network traffic [bytes]: 236
Total network overhead [bytes]: 180
```

Resizing Window

Objective: To examine the network load involved in **changing a window's size.**

X Client Call: *XResizeWindow* , changes the inside dimension of the window. The border is resized to match but its border width is not changed.

Network Traffic:

```
Total Ethernet packets sent by X client: 2
Total Ethernet packets sent by X server: 1
Total X client data generated [bytes]: 24
Total X server data generated [bytes]: 32
Total X traffic [bytes]: 56
Total network traffic [bytes]: 236
Total network overhead [bytes]: 180
```

Moving a Pointer

Objective: Move the pointer suddenly from one point on the screen to another.

X Client Call: *XWarpPointer*

Network Traffic:

```
Total Ethernet packets sent by X client: 2
Total Ethernet packets sent by X server: 2
Total X client data generated [bytes]: 28
Total X server data generated [bytes]: 32
Total X traffic [bytes]: 60
Total network traffic [bytes]: 304
Total network overhead [bytes]: 244
```

Creating Graphic Resources

Objective: Creates a new graphics resource in the server.

X Client Call: *XCreateGC*

Network Traffic:

```
Total Ethernet packets sent by X client: 2
Total Ethernet packets sent by X server: 1
Total X client data generated [bytes]: 28
Total X server data generated [bytes]: 32
Total X traffic [bytes]: 60
Total network traffic [bytes]: 240
Total network overhead [bytes]: 180
```

Destroy Sub-windows

Objective: Destroys all descendants of the specified window (recursively).

X Client Call: *XDestroySubwindows*

Network Traffic:

```
Total Ethernet packets sent by X client: 2
Total Ethernet packets sent by X server: 1
Total X client data generated [bytes]: 12
Total X server data generated [bytes]: 32
Total X traffic [bytes]: 44
Total network traffic [bytes]: 224
Total network overhead [bytes]: 180
```

Clearing Windows

Objective: Clears a window, but does not cause exposure events.

X Client Call: *XClearWindow*

Network Traffic:

```
Total Ethernet packets sent by X client: 2
Total Ethernet packets sent by X server: 2
Total X client data generated [bytes]: 20
Total X server data generated [bytes]: 32
Total X traffic [bytes]: 52
Total network traffic [bytes]: 296
Total network overhead [bytes]: 244
```

Getting Window Attributes

Objective: Returns the *XWindowAttributes* structure containing the current window attributes.

X Client Call: *XGetWindowAttributes*

Network Traffic:

```
Total Ethernet packets sent by X client: 4
Total Ethernet packets sent by X server: 3
Total X client data generated [bytes]: 20
Total X server data generated [bytes]: 108
Total X traffic [bytes]: 128
Total network traffic [bytes]: 542
Total network overhead [bytes]: 414
```

Freeing Graphic Context

Objective: Frees all memory associated with a graphics context, and removes the GC from the server and display hardware.

X Client Call: *XFreeGC*

Network Traffic:

```
Total Ethernet packets sent by X client: 2
Total Ethernet packets sent by X server: 1
Total X client data generated [bytes]: 12
Total X server data generated [bytes]: 32
Total X traffic [bytes]: 44
Total network traffic [bytes]: 224
Total network overhead [bytes]: 180
```

Destroying Window

Objective: The window and all inferiors (recursively) are destroyed, and a *DestroyNotify* event is generated for each window.

X Client Call: *XDestroyWindow*.

Network Traffic:

```
Total Ethernet packets sent by X client: 2
Total Ethernet packets sent by X server: 2
Total X client data generated [bytes]: 12
Total X server data generated [bytes]: 32
Total X traffic [bytes]: 44
Total network traffic [bytes]: 288
Total network overhead [bytes]: 244
```

Appendix B provides the synopsis and the raw data collected for each of these X requests.

4.4. Examples of X Graphical Primitives

The following is the listing of all the *graphical requests* under study which were generated by the *x11perf* application and a summary of their network activities captured by the network monitoring tool.

The network traffic summary is as explained earlier.

Appendix B provides the synopsis and the raw data collected for these *graphical primitives.*

Drawing Points

Objective: To examine the network load involved in **drawing 1000 points.**

X Client Call: *XDrawPoints*, draws one or more points into the specified drawable.

Network Traffic:

```
Total Ethernet packets sent by X client: 5
Total Ethernet packets sent by X server: 2
Total X client data generated [bytes]: 4016
Total X server data generated [bytes]: 32
Total X traffic [bytes]: 4048
Total network traffic [bytes]: 4468
Total network overhead [bytes]: 420
```

Drawing Lines

Objective: To examine the network load involved in **drawing 1000 10-pixel thin lines.**

X Client Call: *XDrawLines*, draws a series of lines joined end-to-end.

Network Traffic:

```
Total Ethernet packets sent by X client: 5
Total Ethernet packets sent by X server: 2
Total X client data generated [bytes]: 4020
Total X server data generated [bytes]: 32
Total X traffic [bytes]: 4052
Total network traffic [bytes]: 4472
Total network overhead [bytes]: 420
```

Drawing Text

Objective: To examine the network load involved in **writing an 80 character string** in (6x13) font.

X Client Call: *XLoadQueryFont*, loads a font and fill information structure.

Network Traffic:

```
Total Ethernet packets sent by X client: 4
Total Ethernet packets sent by X server: 3
Total X client data generated [bytes]: 28
Total X server data generated [bytes]: 1796
Total X traffic [bytes]: 1824
Total network traffic [bytes]: 2244
Total network overhead [bytes]: 420
```

X Client Call: *XChangeGC()*

Network Traffic:

```
Total Ethernet packets sent by X client: 2
Total Ethernet packets sent by X server: 2
Total X client data generated [bytes]: 20
Total X server data generated [bytes]: 32
Total X traffic [bytes]: 52
Total network traffic [bytes]: 232
Total network overhead [bytes]: 180
```

X Client Call: *XDrawString()*

Network Traffic:

```
Total Ethernet packets sent by X client: 2
Total Ethernet packets sent by X server: 1
Total X client data generated [bytes]: 104
Total X server data generated [bytes]: 32
Total X traffic [bytes]: 136
Total network traffic [bytes]: 316
Total network overhead [bytes]: 180
```

X client Call: *XFreeFont()*

Network Traffic:

```
Total Ethernet packets sent by X client: 2
Total Ethernet packets sent by X server: 2
Total X client data generated [bytes]: 12
Total X server data generated [bytes]: 32
Total X traffic [bytes]: 44
Total network traffic [bytes]: 288
Total network overhead [bytes]: 244
```

<u>**Drawing Image**</u>

Objective: To examine the network load involved in **drawing a 100x100 pixels image on a window.**

X Client Call: *XPutImage* , Draws a section of an rectangle in a window or pixmap.

Network Traffic:

```
Total Ethernet packets sent by X client: 10
Total Ethernet packets sent by X server: 3
Total X client data generated [bytes]: 10028
Total X server data generated [bytes]: 32
Total X traffic [bytes]: 10060
Total network traffic [bytes]: 10709
Total network overhead [bytes]: 649
```

Getting Image

Objective: To examine the network load involved in **getting a 100x100 pixels image.**

X Client Call: *XGetImage*, dumps the contents of the specified rectangle, a drawable, into a client-side *XImage* structure, in the format specified.

Network Traffic:

```
Total Ethernet packets sent by X client: 6
Total Ethernet packets sent by X server: 8
Total X client data generated [bytes]: 24
Total X server data generated [bytes]: 10064
Total X traffic [bytes]: 10088
Total network traffic [bytes]: 10926
Total network overhead [bytes]: 838
```

5. X Protocol Network Delay Measurements

5.1. Objective

The objective of this section is to estimate the X Protocol network delay experienced between the X server and client processes communicating through TCP streams. The measured delays are then tabulated for the *graphical primitives* under study.

5.2. Procedure

Two remote hosts and a local hosts were chosen to represent typical communication paths over a LAN and a WAN configuration.

In the LAN configuration discussed in the next section the remote and local hosts were located in two separate subnetworks of Ethernet LAN. The connectivity between the subnets was established by the help of two routers and a backbone Ethernet between them.

In the WAN configuration discussed in section 5.4 the local host remain the same as the LAN configuration, however, the remote host was located in a remote Ethernet subnet. The connectivity between the remote and local subnets was established through the *ESnet* Wide Area Network.

The *emulX* application was started as a server on the remote hosts and was queried by the *emulX* client running on the local host.

The measurements on network delays were done for the request and reply sizes of the *graphical primitives* of interest. 60 sample points were logged in for each *graphical primitives* at prime working hours of 10 a.m to 11 a.m and 2 p.m. to 3 p.m. on weekdays for a period of two weeks.

The average network delays were then tabulated and summarized for the two network configurations.

The system configuration of the local host where the client is running had to be changed to increase its clock resolution from the default setting of 1/100 HZ (10 msec) to 1/1800 HZ (0.5 msec) which was the highest clock resolution available. The high resolution was necessary especially in the LAN configuration where typical communication delays are much lower than default system clock resolution of 10 msecs.

5.3. LAN Configuration: X Protocol Network Delay Estimates

The client and the server were invoked on two hosts located in two separate subnetworks connected by two CISCO routers. The entire LAN is comprised of over 50 subnetworks connected by backbone Ethernet.

The *traceroute* [TRACEROUTE] program was used to identify network devices between the two hosts. Figure 10 provides the schematics for this LAN configuration.

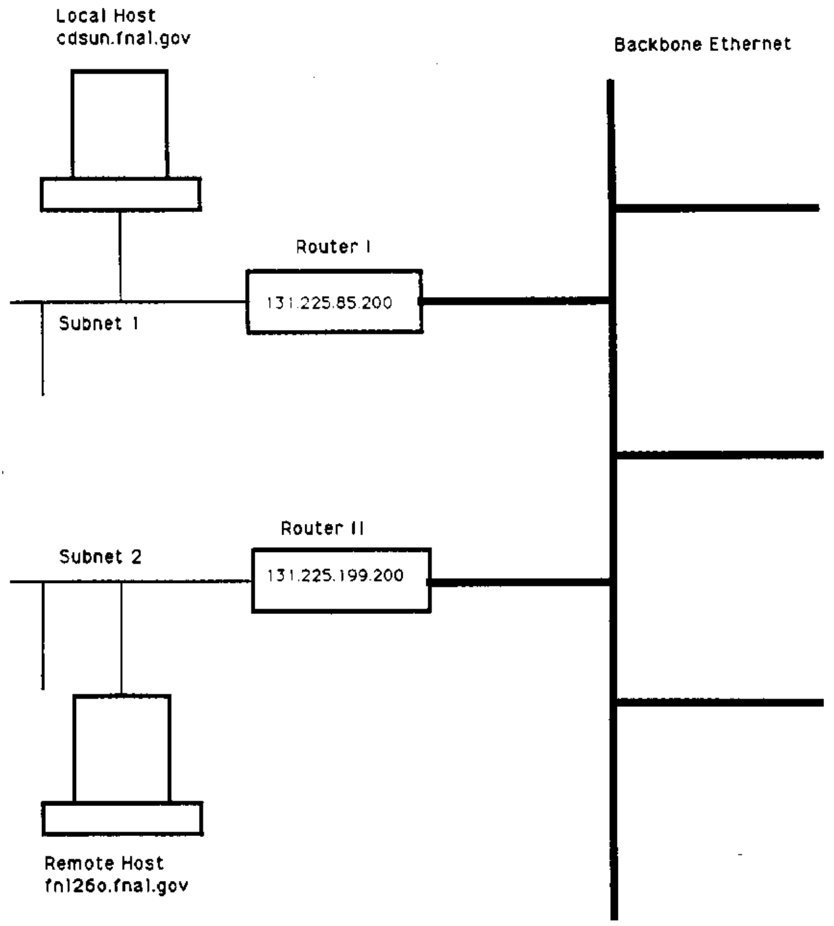

Figure 10: Schematic of the LAN Configuration

Table 1 provides the summary of the network loads and delays associated with the X client requests under study.

Average Network Loads and Delays in a LAN		
X Client Requests	**Network-Load[bytes]**	**Average Delay[ms] and Standard Deviation**
Opening Display	760	7.1, 1.9
Create a Window	192	6.2, 2.1
Change Window Attributes	240	6.0, 1.6
Map Window	288	6.2, 1.8
Move Pointer	304	6.9, 1.8
Create Graphic Context	240	6.9, 2.6
Destroy Subwindow	224	5.8, 1.8
Clear Window	296	6.3, 2.4
Get Window Attributes	542	6.4, 2.0
Free Graphic Context	224	6.9, 2.3
Destroy Windows	288	5.7, 1.6
Drawing 1000 points	4468	18.1, 2.8
Drawing 1000 10-pixel thin lines	4472	19.2, 2.6
Query and Load a Font	2244	14.6, 2.1
Writing an 80 character 6x13 font	316	9.2, 2.5
Free Font	288	6.8, 2.1
Copying a 100x100 square pixels from window to window	244	6.3, 1.8
Drawing a 100x100 pixels image	10709	69.2, 15.5
Getting a 100x100 pixels image	10926	71.2, 41.8
Moving a window	236	6.1, 2.1
Resizing a window	236	6.2, 1.8

Table 1: Average Network Load and Delay for X Client Requests in a LAN

5.4. WAN Configuration: X Protocol Network Delay Estimates

The client and the server were invoked on two hosts located in two separate sub-networks. The subnets were connected through their Ethernet backbone and the *ESnet* Wide Area Network.

The *traceroute* program was used to identify network devices between the two hosts. Figure 11 provides the schematics for this WAN configuration.

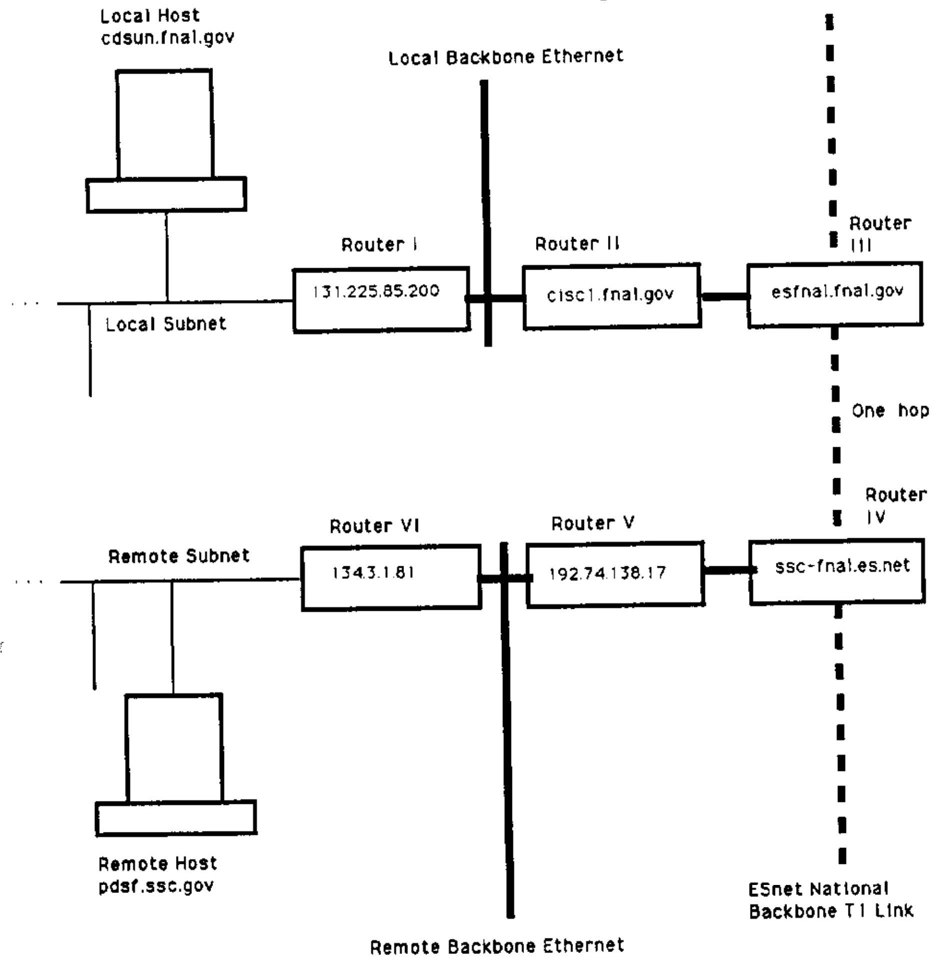

Figure 11: Schematic of the WAN Configuration

Table 2 provides the summary of the network loads and delays associated with the X client requests under study.

Average Network Loads and Delays in a WAN		
X Client Requests	**Network-Load[bytes]**	**Average Delay[ms] and Standard Deviation**
Opening Display	760	43.5, 7.3
Create a Window	192	43.8, 6.5
Change Window Attributes	240	40.3, 5.7
Map Window	288	43.3, 5.9
Move Pointer	304	52.6, 5.7
Create Graphic Context	240	38.2, 4.6
Destroy Subwindow	224	38.9, 7.4
Clear Window	296	42.2, 4.9
Get Window Attributes	542	45.3, 7.2
Free Graphic Context	224	43.3, 7.7
Destroy Windows	288	48.3, 6.4
Drawing 1000 points	4468	84.6, 10.1
Drawing 1000 10-pixel thin lines	4472	78.9, 10.7
Query and Load a Font	2244	35.6, 8.8
Writing an 80 character 6x13 font	316	30.7, 5.1
Free Font	288	30.9, 4.5
Copying a 100x100 square pixels from window to window	244	43.3, 6.0
Drawing a 100x100 pixels image	10709	297.3, 42.2
Getting a 100x100 pixels image	10926	276.3, 51.1
Moving a window	236	41.3, 6.2
Resizing a window	236	39.3, 5.6

Table 2: Average Network Load and Delay for X Client Requests in a WAN

6. Network Traffic Profile: LAN Configuration

6.1. Objective

The objective of this section is to briefly describe the Simple Network Management Protocol (SNMP) [SNMP] which was instrumental in gathering network statistics for the LAN configuration. The statistical observations are used later on in the queuing analysis of the LAN configuration.

6.2. Network Monitoring Tool: SNMP

SNMP (Simple Network Management Protocol) is a lightweight network management protocol for TCP/IP networks which was created to help monitor network performance, configure network devices, and detect network faults. SNMP evolved from the SGMP (Simple Gateway Monitoring Protocol) protocol in 1988 and has been deployed in networks since early 1989 [ROSE].

SNMP allows a network management station to communicate with the managed network entities. Each of these network entities, such as hosts, routers, printers, etc., must run agent software implementing the SNMP protocol which will update network management data about managed objects (e.g., packet counts, types, sizes, etc.) on that entity and retrieve this data when requested to do so by the network management station. The managed objects are collectively referred to as the Management Information Base (MIB).

The network management station monitors the network by communicating with the SNMP agents on the network entities (hosts, routers, etc.), collecting data about the managed objects from them, and providing applications and reports which allow the network administrator to monitor faults and analyze the data. Figure 12 illustrates a network configuration in which a network management station monitors and manages other network entities through SNMP protocol.

The network statistics for the two routers (network entities) with the LAN configuration tested were obtained by querying their SNMP agents for information on their *Interface group*. The *Interface group* is a group of managed objects that contain generic information about the interface (physical) layer of the network entity. The following is the list of managed objects that were queried from the SNMP agents:

- **ifIndex**: interface number
- **ifInOctets** and **ifOutOctets**: number of bytes received/sent (counter).
- **ifInUcastPkts** and **ifOutUcastPkts**: number of unicast (not broadcast/multicast) packets accepted/sent (counter).

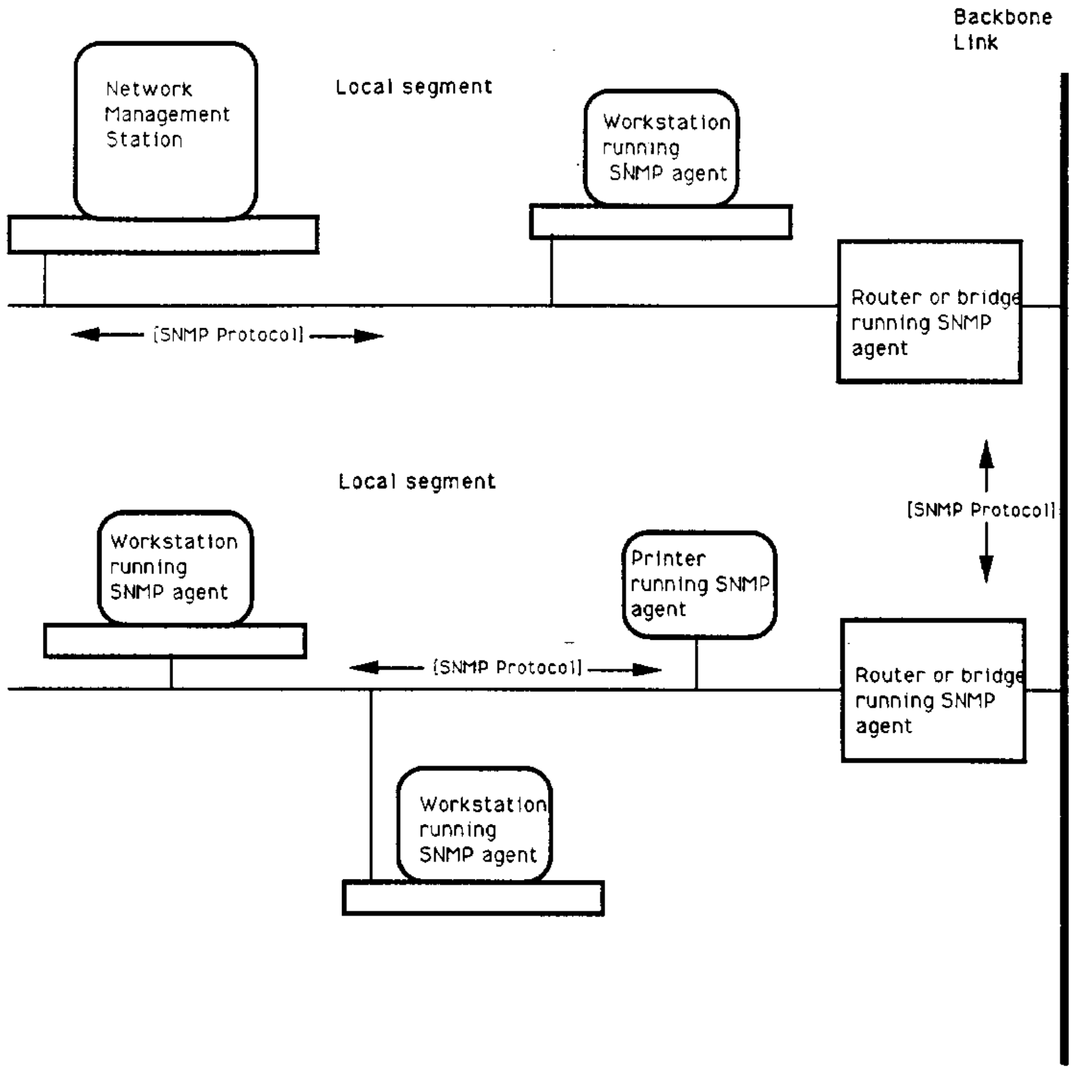
Backbone Link
Network Management Station
Local segment
Workstation running SNMP agent
Router or bridge running SNMP agent
[SNMP Protocol]
Local segment
Workstation running SNMP agent
Printer running SNMP agent
[SNMP Protocol]
[SNMP Protocol]
Router or bridge running SNMP agent
Workstation running SNMP agent

Figure 12: Schematic of a Network Configuration Managed by SNMP Protocol

- **ifInNUcastPkts** and **ifOutNUcastPkts**: number of non-unicast accepted/sent (counter).
- **ifInDiscards** and **ifOutDiscards**: number of incoming and outgoing packets discarded due to resource limitations (counter).
- **ifInErrors** and **ifOutErrors**: packets discarded due to format error.
- **ifInUnknownProtos**: number of packets with unknown protocol received (counter).

Figure 13 illustrates the Network data obtained by querying the routers' SNMP agents.

As is shown, packets arrive at the interface (physical) layer from the layer below (transmission medium). The good packets (packets with no error or known protocol) are delivered to the above layer (IP) with the counters representing their type (e.g., *ifInUcastPkts*) updated. The bad packets (packets with error, unknown protocol, etc.) are discarded and their corresponding counters updated.

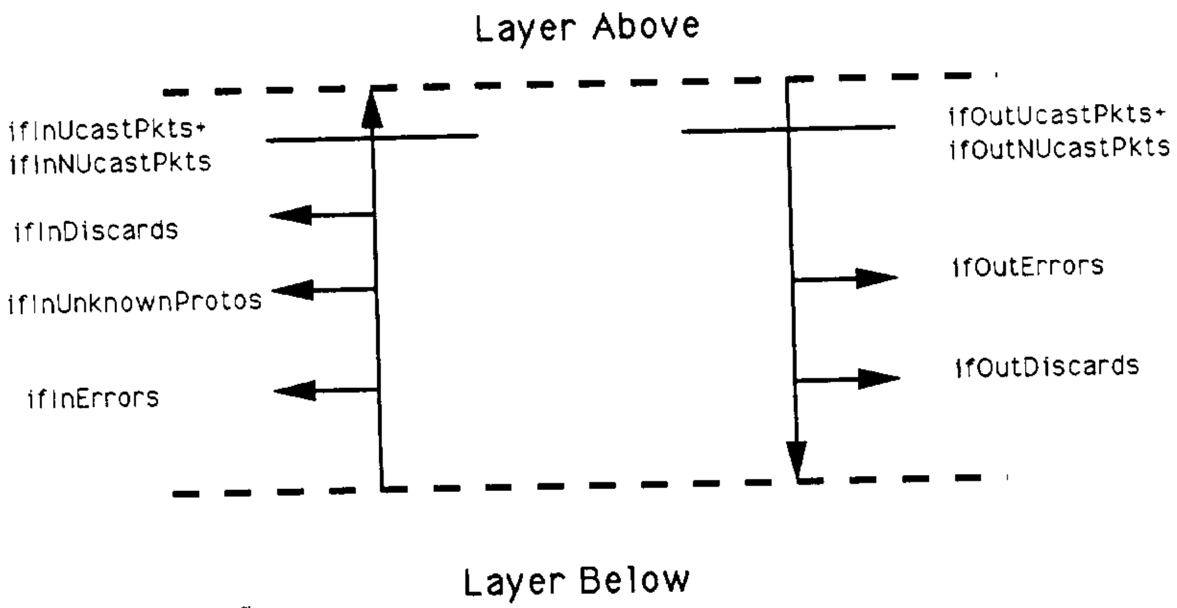

Figure 13: SNMP Network Statistics for the Interface Layer

As packets arrive from the above layer to the interface layer for delivery the outgoing packet statistics are also updated.

The network management station monitors network entities by accessing their statistics through SNMP protocol.

6.3. Procedure

With the help of a Network Management Station (NMS) the routers involved in the LAN configuration were queried for a period of about 4 weeks on week days to determine the average packet rate and size distribution. The querying periods were chosen to be during the high traffic hours of 10:00 a.m. to 11:00 a.m. and 2:00 p.m. to 3:00 p.m. During those hours SNMP messages were sent to the SNMP agents of the routers with their responses logged in at five minutes interval. Five hundred eleven of such reports were collected to obtain the following observation on the characteristics of the routers involved in our LAN configuration.

The responses coming from the SNMP agents such as *ifInOctets* are counter values and the actual data is the difference between the report at time t+300 and t. This difference was then divided by the elapsed time (300 seconds) to obtain an average rate.

The average packet rate was determined by subtracting all of the incoming packet counter reports at time t from all of the incoming packet counter reports at time t+300 for the two network interfaces of the router and then dividing the difference by the interval time of 300 seconds.

The average bit rate was determined from the difference of incoming bytes counters at two consecutive time interval multiplied by 8 and dividing it by the elapsed time.

The average packet size was determined by dividing the total received bytes by the total number of packets received during the same time interval.

6.4. Network Statistics on Router I

The following is a summary of the traffic profile observed on router-I with the Internet address of 131.225.85.200 at high traffic hours for a period of 4 weeks.

Input Traffic Profile

Packet Rate: The minimum input packet rate is the minimum of all the average packet rates observed during the high traffic hours. The average input packet rate during high traffic hours was determined by averaging over all the average packet rates during querying interval (300 seconds). The maximum input packet rate is the maximum of all the average packet rates observed during the high traffic hours.

Bit Rate:The minimum, average, and maximum bit rate are determined similar to the packet rate calculations.

Average-Packet-Size Distribution: The minimum input packet size is the minimum of all the average packet sizes observed during the high traffic hours. The average input packet size during high traffic hours was determined by averaging over all the average packet sizes during querying interval (300 seconds). The maximum input packet size is the maximum of all the average packet sizes observed during the high traffic hours.

Table 3 summarizes input traffic profile obtained from router I.

Figure 14 is a graphical representation of average input packet size distribution on router-I during high traffic hours.

Input Rate Statistics					
	Min	Max	Median	Average	Std. Dev.
Packet Rate (packet/sec)	102.3	319.1	161.6	166.6	31.2
Bit Rate (Kbit/sec)	86.3	560.4	139.8	148.8	42.7
Packet Distribution (bytes/packet)	90.8	168.6	105.5	109.7	12.9

Table 3: Router-I Input Traffic Profile Statistics

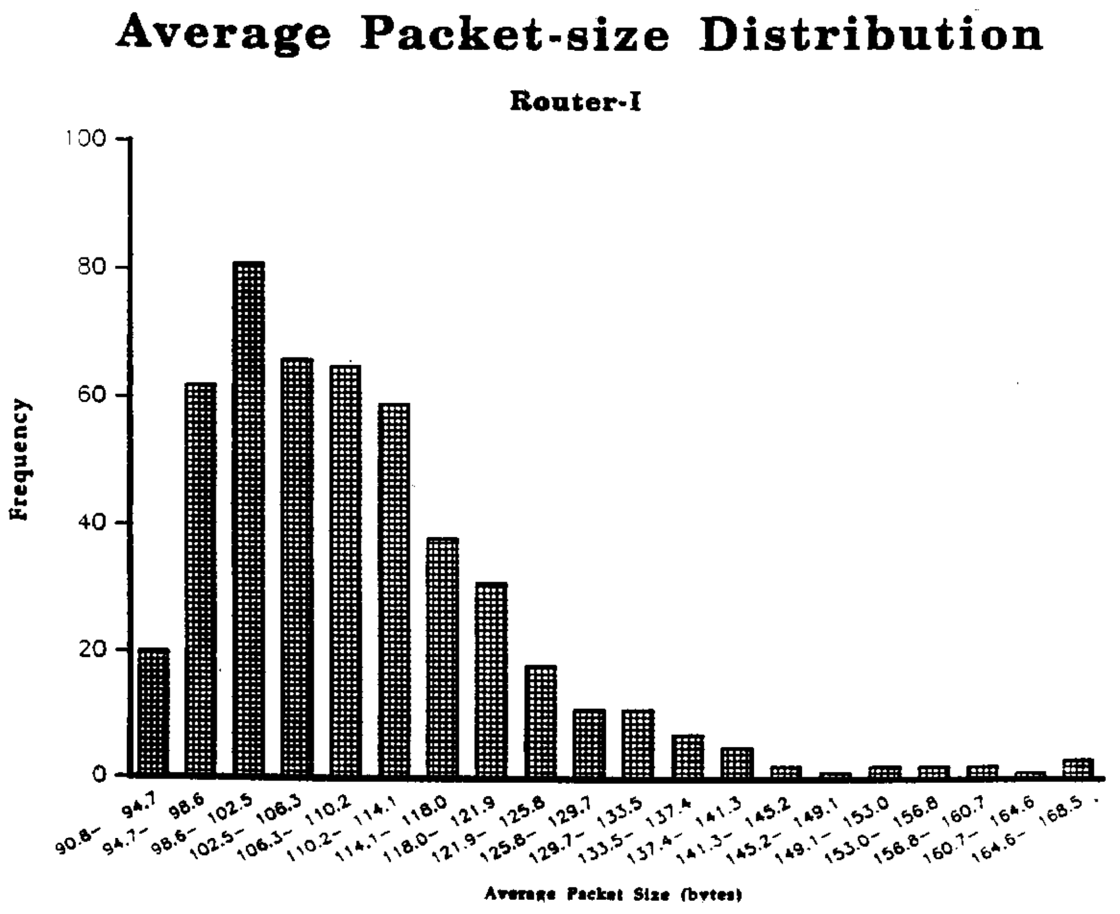

Figure 14: Average Input Packet-size Distribution on Router-I

<u>Output Traffic Profile</u>

Packet Rate: The following packet rate values were determined similar to the the input packet rate calculations described previously.

Bit Rate: The bit rate values were determined similar to the the input bit rate calculations described previously.

Table 4 summarizes output traffic profile obtained from router I.

Output Rate Statistics					
	Min	**Max**	**Median**	**Average**	**Std. Dev.**
Packet Rate (packet/sec)	98.6	315.5	158.3	163.6	36.8
Bit Rate (Kbit/sec)	84.0	555.7	136.7	146.4	46.2

Table 4: Router-I Output Traffic Profile Statistics

6.5. Network Statistics on Router II

The following is a summary of the traffic profile observed on router-II with the Internet address of 131.225.199.200 at high traffic hours for a period of 4 weeks.

<u>Input Traffic Profile</u>

Packet/Bit Rate: The description of the minimum, average, and maximum packet/bit rates are similar to the router-I values as described earlier.

Average-Packet-Size Distribution: The description of the minimum, average, and maximum packet sizes are similar to the router-I values as described earlier.

Table 5 summarizes input traffic profile obtained from router II.

Input Rate Statistics					
	Min	**Max**	**Median**	**Average**	**Std. Dev.**
Packet Rate (packet/sec)	86.2	295.7	104.9	111.4	25.1
Bit Rate (Kbit/sec)	84.7	573.9	102.9	123.1	76.7
Packet Distribution (bytes/packet)	104.2	272.2	123.4	128.2	22.3

Table 5: Router-II Input Traffic Profile Statistics

Figure 15 is a graphical representation of average input packet size distribution on router-II during high traffic hours.

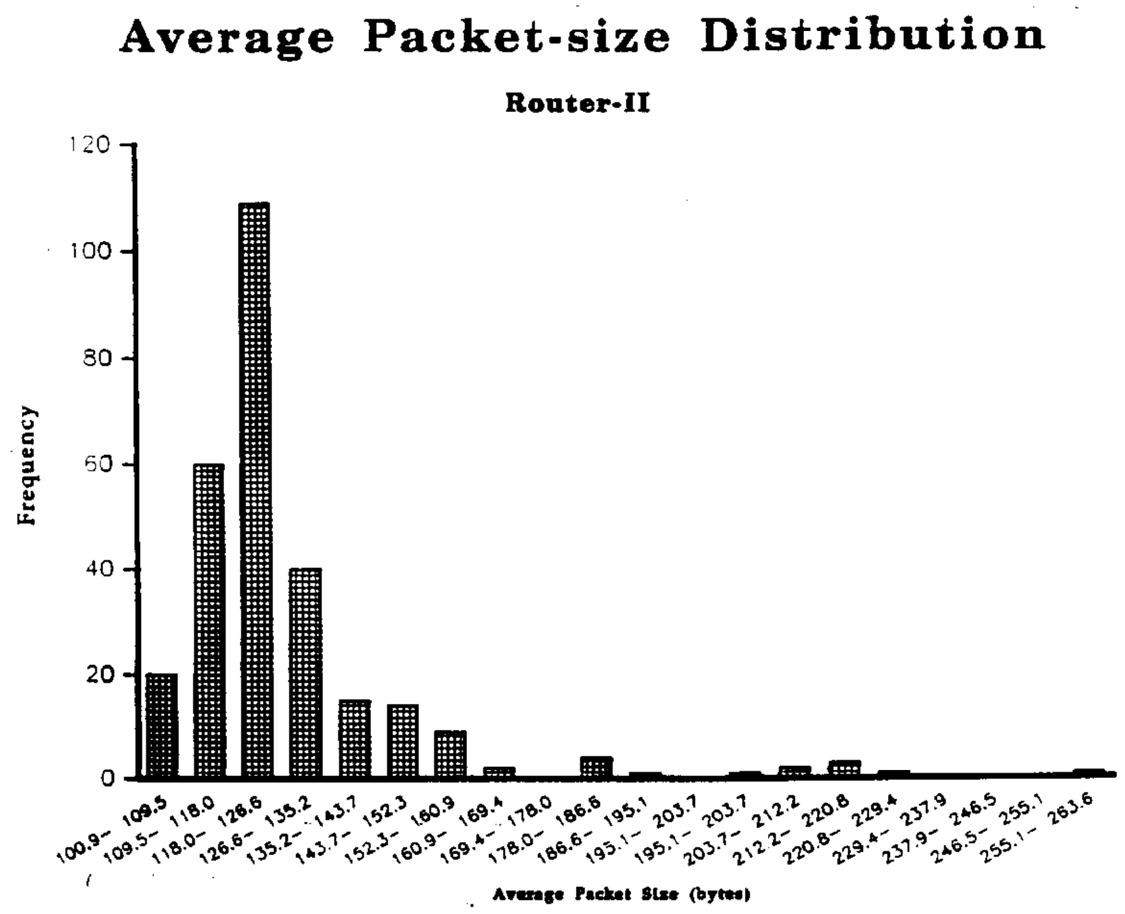

Figure 15: Average Input Packet-size Distribution on Router-II

Output Traffic Profile

Table 6 summarizes output traffic profile obtained from router II.

Output Rate Statistics					
	Min	**Max**	**Median**	**Average**	**Std. Dev.**
Packet Rate (packet/sec)	82.2	285.5	99.8	105.7	24.7
Bit Rate (Kbit/sec)	81.5	575.8	100.5	121.2	77.5

Table 6: Router-II Output Traffic Profile Statistics

The input statistics (e.g., packet rate) on both routers are slightly greater than their corresponding output statistics (e.g., the average input packet rate on router I is 166.6 packet/sec while the average output packet rate is 163.6 packet/sec with greater standard deviation). This phenomenon is due to the fact that some of the input packets are not forwarded due to some errors or more likely the input packets were addressed to the router itself. In general, as might be expected, the router increases the variation in rate going from input to output.

7. Network Device Maximum Throughput Measurements

7.1. Objective

The objective of this section is to determine the maximum throughput of the routers in bits/sec in the LAN configuration. This value is necessary in the construction of the queuing model.

7.2. Flooding Experiment Procedure

The technique used in measuring the network device maximum throughput was to send (flood) streams of Ethernet packets at different packet rates and sizes until the router is overwhelmed. A *ping* program with a special flooding feature which sends *ICMP*[5] messages as fast as possible was used by several "packet generator" machines concurrently on the local area network to achieve the input bit rate required for router saturation.

The saturation point was considered reached when the router started dropping packets to protect itself. The observed bit rate at the saturation point is then an estimate to the network device maximum bit rate throughput. This experiment was done at times when the network load was very low from sources other than the test sources so as to provide a control environment for the experiment and also to avoid inconveniences to network users caused by the routers momentary saturation.

Different number of packet generating machines were chosen to flood the router concurrently. During the flooding period another machine not involved in generating packets was acting as an Network Management Station (NMS) and querying the router via SNMP messages for statistics on the interface (physical) layer which includes total number of bytes/packets received or sent by the network router.

By increasing the ICMP packet size sent by each machine to the router the overall throughput was increasing until the saturation point was reached. Beyond that point the router performance degraded significantly as majority of the ICMP messages coming from "packet generators" were left unreplied.

The SNMP messages were sent to the router's SNMP agent at fixed intervals. It was assumed that the responses from the SNMP agent would also be received at fixed interval. This, however, turned out to be wrong assumption, since during the flooding period the querying SNMP packets were either lost or the SNMP agent was slow in responding which resulted in getting responses at variable time interval. Without an accurate time interval one could not obtain an accurate rating values from the SNMP messages.

[5]Internet Control Message Protocol. This protocol is used to handle error and control information between gateways and hosts. The *ping* program sends ICMP messages to the hosts or gateways and awaits reply messages from them. The ICMP messages are usually generated and processed by the TCP/IP networking software itself, and not user processes

At this point it was decided to proceed with experiment with a passive approach. In this approach while the ICMP message generating machines were flooding the router and the router was reaching its saturation point another machine not involved in packet generation was designated to capture packets that were destined to or sourced from the router with the help of the *tcpdump* application described earlier.

The output of the *tcpdump* application was saved in a packet trace file for different experiments run.

The trace file includes one line of information about every packet that was transmitted by the packet generating machines or the flooded router during the flooding experiment. The summary line shows the packet's source and destination IP address, timestamp, length, and type.

Figure 16 illustrates flooding experiment set up.

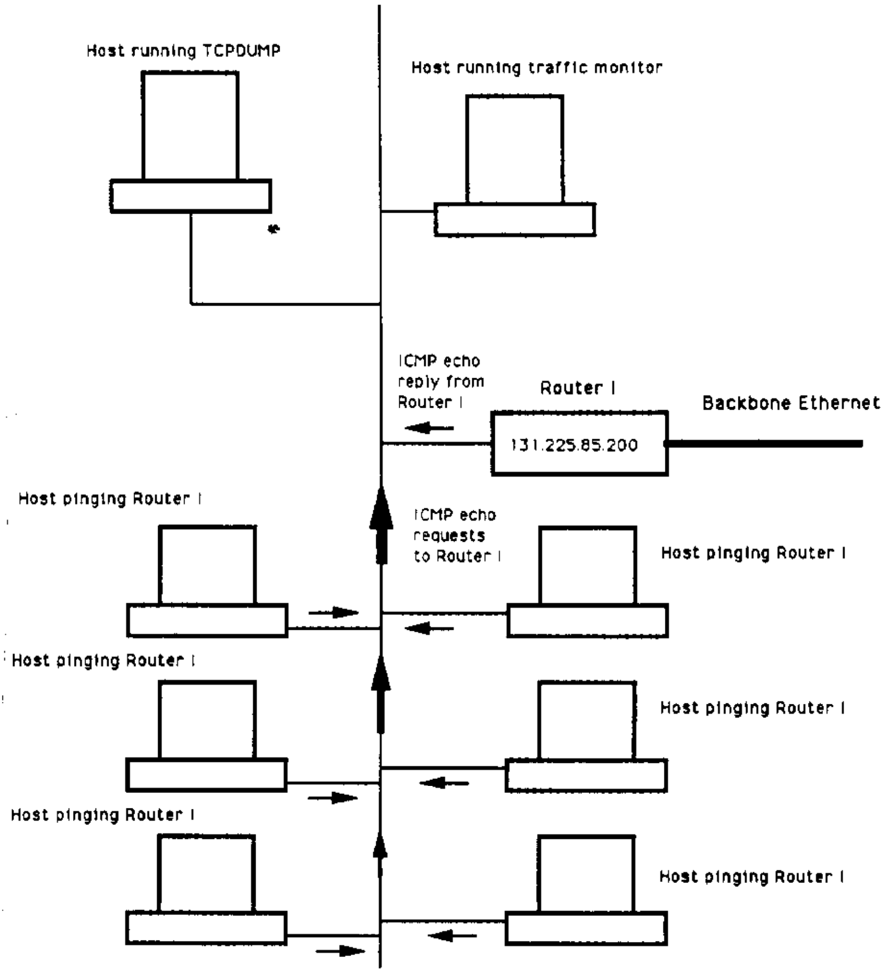

Figure 16: Flooding Experiment Test Bed

An analysis program was created to parse the trace file for a summary of input packet rate to the router during the flooding experiment. The input bit rate to the router was also determined from the length field in each packet summary line.

The trace file was also parsed to obtain a summary of output packet rate from the router. The output bit rate was determined from the length field in each packet summary line.

The maximum throughput of the router in bit rate is therefore the maximum observed traffic originating from the router. This approach assumes that the observed bit rate on one interface is the maximum achievable throughput by the router and that the traffic on the other interface is insignificant. Indeed prior to the experiment the traffic on the other interface was measured and confirm to be minimal. This was due to the fact the experiment was done at times when there were almost no network traffic present in the LAN.

The following observations should be noted for the preceding experiment set up. First of all, the current set up does not allow to observe the effect of packet rate and bit rate on the router separately. A better approach would be to create the packet stream with a single source with ability to control the interpacket gaps and packet sizes. This way the packet rate and bit rate can be change by changing the gap size and packet size independently. This approach could yield a better estimate for packet and bit rate saturation points.

Secondly, in the current set up the saturation point could also be affected by the Ethernet channel acquisition delay. That is because the ICMP replies are sent from the router to the ICMP "packet generators" on the same segment.

Finally, it should be noted that this experiment only measures the router maximum throughput when packet generating machines are flooding it with ICMP messages created by the *ping* application. This means there are some extra processing involved in the ICMP packet processing at the network layer (IP). This extra processing includes packet assembly for the incoming messages and fragmentation for reply messages. It is quite possible that the observed saturation point due to ICMP packet processing is lower than saturation point due to IP packet forwarding process.

A more ideal experiment would be to measure the router maximum throughput when input packets are simply routed or forwarded at the IP layer to another subnet on the other side of the router. This way the channel acquisition delay effect is out of the picture since the outgoing packets are forwarded to the segment with no Ethernet traffic. The IP packet routing which is viewed as any router's main activity requires less processing that ICMP reply processing and this could result in a higher saturation point and consequently greater device throughput.

Such an ideal experimental set up is currently under development.

7.3. Flooding Experiment Results

The following table summarizes the observations made during the flooding experiment.

		Input To Router-I		Output From Router-I	
Number of Hosts	ICMP Message [Bytes]	Pkt-rate [Pkt/sec]	Bit Rate [Mbps]	Pkt-rate [Pkt/sec]	Bit Rate [Mbps]
1	700	90	0.54	90	0.54
1	1400	81	0.94	80	0.93
1	2000	195	1.62	192	1.60
2	700	162	0.97	162	0.97
2	1400	147	1.70	147	1.70
2	2000	396	3.30	120	1.00
4	700	257	1.53	184	1.10
4	1400	410	4.74	241	2.80
4	2000	585	4.91	66	0.43
7	700	454	2.70	166	0.99
7	1400	606	7.01	236	2.73

Figure 17 provides a graphical observation made from the flooding experiment result in the above table.

This graph represents the router's observed output bit rate during the experiment as a function of observed packet rate traffic into the router. Three plots for different message sizes (700, 1400, and 2000 bytes) are presented in order to also establish bit rate saturation effect.

As was suspected the service rate capacity of the router is a function of byte processing (packet size) as well as packet processing (the number of incoming packets).

The router showed its best performance (output bit rate) when 4 machines were flooding it with the ICMP packet size of 1400 bytes.

The maximum throughput for the router in processing 1400 ICMP packets is 2.8 Mbps.

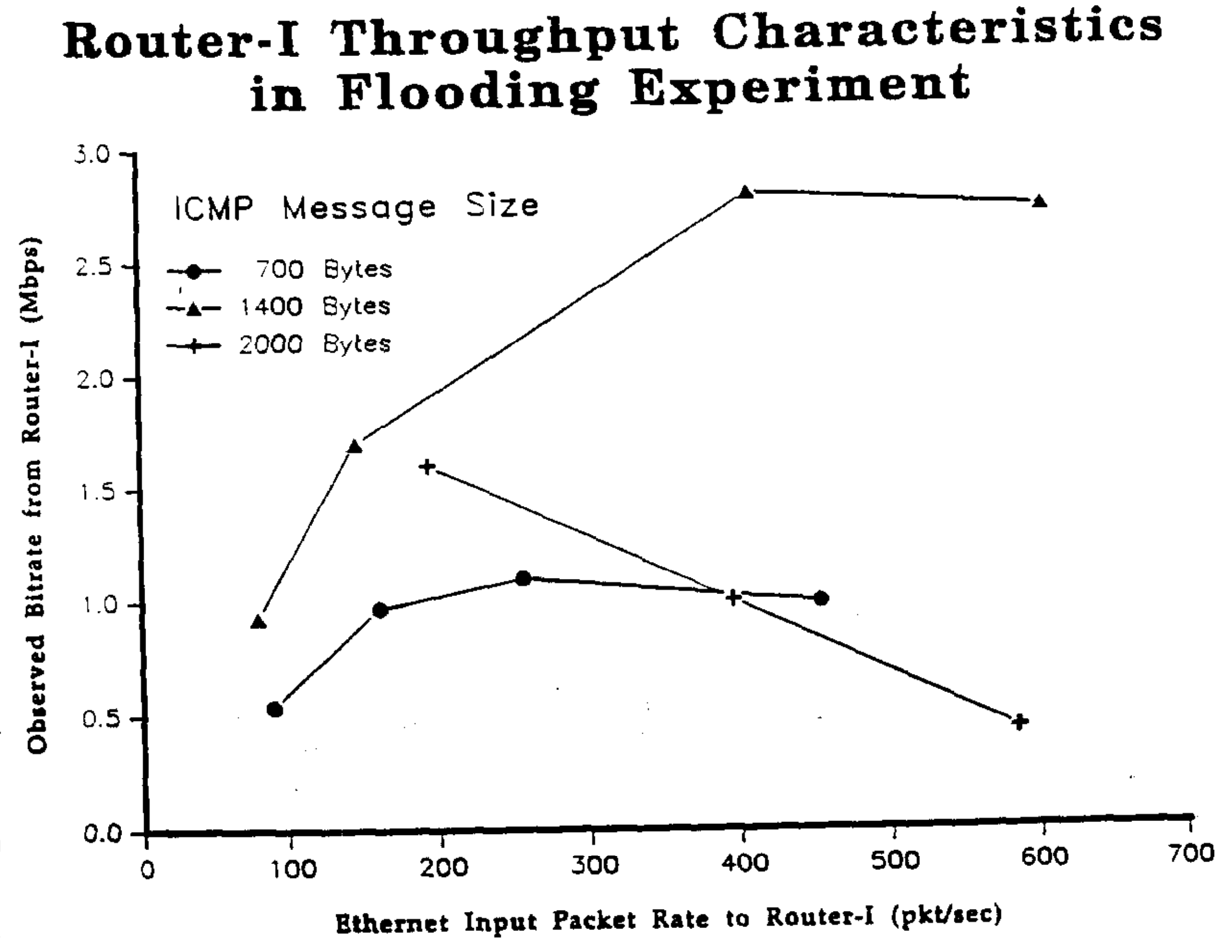

Figure 17: Router-I Throughput Characteristics in Flooding Experiment

8. Analytical Modeling of LAN Configuration

8.1. Objective

The objective of this section is to create an analytical queueing model of the network between the X server and client processes and obtain a closed form solution for the network delays experienced. The aim is to create the model for a small network, validate its predictions and then expand the model for larger networks.

8.2. Queuing Model Introduction

Queuing theory is one of the most important tool for quantitative analysis of computer networks. A simple queueing model consists of a a waiting line (buffer) where customers (packets) are awaited for services, a server with some service rate capacity *mu*, and some customer (packet) arriving rate. A more complete treatment of the queuing theory and its applications to networking can be found in Schwartz [SCHWARTZ]. Such queuing models can be used to quantify the time delay, blocking performance, and packet throughput of a networking system. These performance parameters are known to depend on the probability of the state of the queue or in other words the number of the packets on the queue (including the one in service). To calculate the probability of number of the packets in the queue one must have the knowledge of:

1. The packet arrival rate.
2. The packet length distribution, assuming that service time is directly proportional to the packet length.
3. The service discipline.

<u>M/M/1 Queue</u>

The M/M/1 queue is the simplest model of a queue. This model implies that:

- The interarrival time has a Poisson distribution.
- The service distribution (Packet length) is exponential.
- There is only one server 'and the service discipline is First Come First Served (FCFS).
- The queue has infinite buffer.

In analysis of the M/M/1 queue one can derived the mean number of packets in the system (N) to be [TANENBAUM]:

(1)

$$N=\frac{\rho}{(1-\rho)}$$

Where: ρ (Traffic Intensity) is defined as: Packet-Rate/Service-Rate.

D.C. Little [LITTLE] provided the well known **Little's results** which states that a queuing system with average packet arrival rate (λ) and mean time delay (T) through the system, has an average queue length (N) given by:

(2)

$$T=\frac{N}{\lambda}$$

where:

T:	average delay.
N:	average number of packets in the queue.
ρ:	average packet-rate.

Combining equations **(1)** and **(2)** one could get the total waiting time, including the service time to be:

$$Mean\_delay=\frac{1}{(average\_service\_rate-average\_packet\_arrival\_rate)}$$

<u>Networks of M/M/1 Queues</u>

The results derived from the previous section can be directly applied to our LAN configuration if each router can be modeled as an M/M/1 queue.

Kleinrock [KLEINROCK] was one of the first to apply the application of queuing theory to a communication channel. In this application the mean queuing and transmission delay for every node on the network is derived to be:

(3)

$$T=\frac{1}{(\mu C-\lambda)}$$

where:

$1/\mu$:	mean packet size [bits/packet]
C:	capacity of communication channel [bits/sec]
λ:	packet arrival rate [packet/sec]

T: mean delay per packet[sec]

The service rate capacity (C) in equation (**3**) was taken to be equal to the communication channel capacity rate according to Tanenbaum [TANENBAUM]. This assumes that the packet processing time is negligible compare to the transmission delay and as a results the service time per packet is mainly due to the packet transmission time.

This assumption does not apply to our LAN environment where transmission rate is no longer the communication bottleneck. A study of the network which encompasses our LAN configuration has shown that on the entire network the average utilization is about 5.94% of Ethernet channel with the peak of 8.49% [PABRAI].

By changing the definition of the C parameter in the equation (**3**) from communication channel capacity to network device maximum throughput one could derive the queuing delay (packet processing as well as packet transmission) for every node on the network.

Additional delay is involved in our Ethernet LAN environment and that is the delay associated with the channel acquisition. There have been many studies on the subject of analytical modeling of CSMA/CD protocols.

Metcalfe and Boggs derived a simple formula for prediction of the capacity of finite-population Ethernets [METCALFE]. Lam used a single-server queueing model in which the server is the shared channel to obtain expressions for delays and throughput [LAM]. Tobagi and Hunt applied the method of embedded Markov chains [TOBAGI] to more accurately model CSMA/CD. Franta also obtain solution for delay by modeling the communication channel as a single server queueing model. The service time for a packet is given by the time required to send the message over the channel; a time determined by the length of the message and the speed of the channel [FRANTA].

In our LAN configuration the output of several lines are converged to the routers in the LAN. Thus the input to one router is no longer a Poisson process outside the network, but the sum of the outputs of several other networks as describe in [TANENBAUM] section A.3. However, Jackson [JACKSON] has shown that an open network of M/M/1 queues can be analyzed on individual basis and the total delay would be equal to the sum of all delays experienced in each queue.

In general Poisson streams have the following properties:

- Merging of k Poisson streams with mean rate results in a Poisson stream with a mean rate equal to the sum of the input rates.

- The arrival of a Poisson stream to a service center with exponential service time results in the departure Poisson stream with the same mean rate.

8.3. LAN Configuration: A Network of M/M/1 Queues

Our LAN configuration consists of two hosts running X client and server processes in separate Ethernet subnets. The subnets are connected to each other with two routers and a backbone Ethernet. The message delays between the two processes is the sum of queuing delays in the routers (two router, two queuing delays), the CSMA/CD packet transmission delays (there are three of these), and packet processing delays in the network and interface (physical) layers of each hosts.

It should be noted that simple summation of individual delays to get the overall delay in the system is based on the assumption that all queues within our system can be characterize as M/M/1 queue.

Figure 18 is a graphical representation of our LAN configuration analytical model.

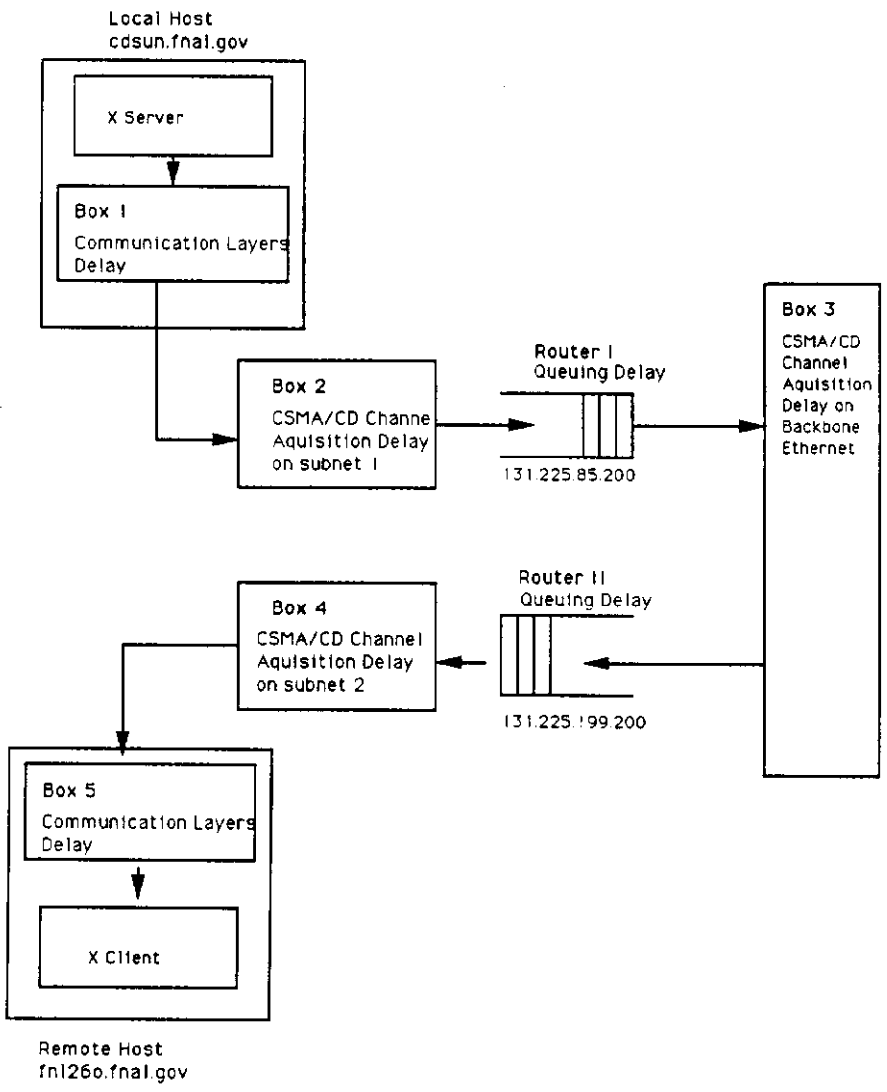
Local Host
cdsun.fnal.gov
X Server
Box 1
Communication Layers Delay
Box 2
CSMA/CD Channel Aquisition Delay on subnet 1
Router I
Queuing Delay
131.225.85.200
Box 3
CSMA/CD Channel Aquisition Delay on Backbone Ethernet
Router II
Queuing Delay
131.225.199.200
Box 4
CSMA/CD Channel Aquisition Delay on subnet 2
Box 5
Communication Layers Delay
X Client
Remote Host
fnl26o.fnal.gov

Figure 18: Analytical Modeling of LAN Configuration

8.4. Routers Queueing Delay

As is shown in the Figure 18, the routers in our LAN configuration are modeled as a network of M/M/1 queues. The followings are the arguments for such assumptions:

Poisson Arrival Rate Assumption

The assumption of an exponential interarrival probability (Poisson arrival) is reasonable for any system that has a large number of independent customers [TANENBAUM]. The Ethernet subnet in our LAN configuration contains over 100 computers communicating with each other and other subnets through router-I.

Exponential Service Time Assumption

It is assumed that the service time is directly proportional to the packet size and if the packet size distribution is exponential one can confidently argue that the service time on each packet is also exponential.

Figures 14 and 15 provide a graphical representation of average-packet-size distribution in our LAN routers for a period of four weeks. As is shown the average packet-size distributions have close resemblance to an exponential distribution. Several studies [SURI] have shown that a slight departure from the exponential assumption does not make a significant difference in the result.

Queuing Delay on Router-I

```
Average Packet Size: 109.7 bytes
Service Rate Capacity: 2.8 Mbps
Average Arrival Rate: 166.6 pkt/sec
```

From equation (3) one can calculate the average delay each packet would experience in this router to be 0.33 msec.

Queuing Delay on Route-II

```
Average Packet Size: 128.2 bytes
Service Rate Capacity: 2.8 Mbps
Average Arrival Rate:  111.4 pkt/sec
```

From equation (3) one can calculate the average delay each packet would experience in this router to be 0.38 msec.

Total LAN Configuration Queuing Delay

The overall average delay experienced by each packet in our LAN configuration routers is simply the sum of the two delays, 0.71 msec. Again this simple addition is possible since the routers were shown to be correctly modeled as M/M/1 queues.

8.5. Communication Layers Delay

The X messages communicated between the X client and server experience certain delay as they become subject to processes within the communication layers. These delays are represented as Box-1 and Box-5 in Figure 18.

To measure the time spent on these layers, the client and server applications in *emulX* were started on the same host and the elapsed time between requests sent and replies received were logged in.

In such arrangement a message sent from the X client to an X server on the same machine will go down the socket and network layer in the client side and up the socket and network layer to reach the server port.

The average time for a packet going through the local and remote communication layers is thus the time spent between the client and server communicating over the same machine assuming that the local and remote hosts are of the same kind. This delay, however, does not include the time spent in the interface layer. The average delay for a packet of size 100 bytes to go through the socket interface layer, the transport layer (TCP), the network layer (IP) was measured to be 0.55 msec. The packet of size 100 bytes will experience the same average delay as it travels upward from the network layer through transport layer to the socket layer.

A packet size of 100 bytes is close to the average packet size observed in our LAN configuration.

Figure 19 illustrates the test set up used in measuring the communication layers delay in each host.

Is it a correct assumption to characterize the communication layers within the local host as a pair of M/M/1 queues, one for incoming packets and another for the outgoing ones.

Further investigation (measurement or simulation) is required to check the validity of such assumption.

8.6. Ethernet Channel Acquisition Delay

Boxes 2, 3, and 4 in Figure 18 represent the average delays associated with the channel acquisition delays in CSMA/CD protocol for our LAN configuration.

The equation for the normalized transfer time in the CSMA/CD protocol is given by:

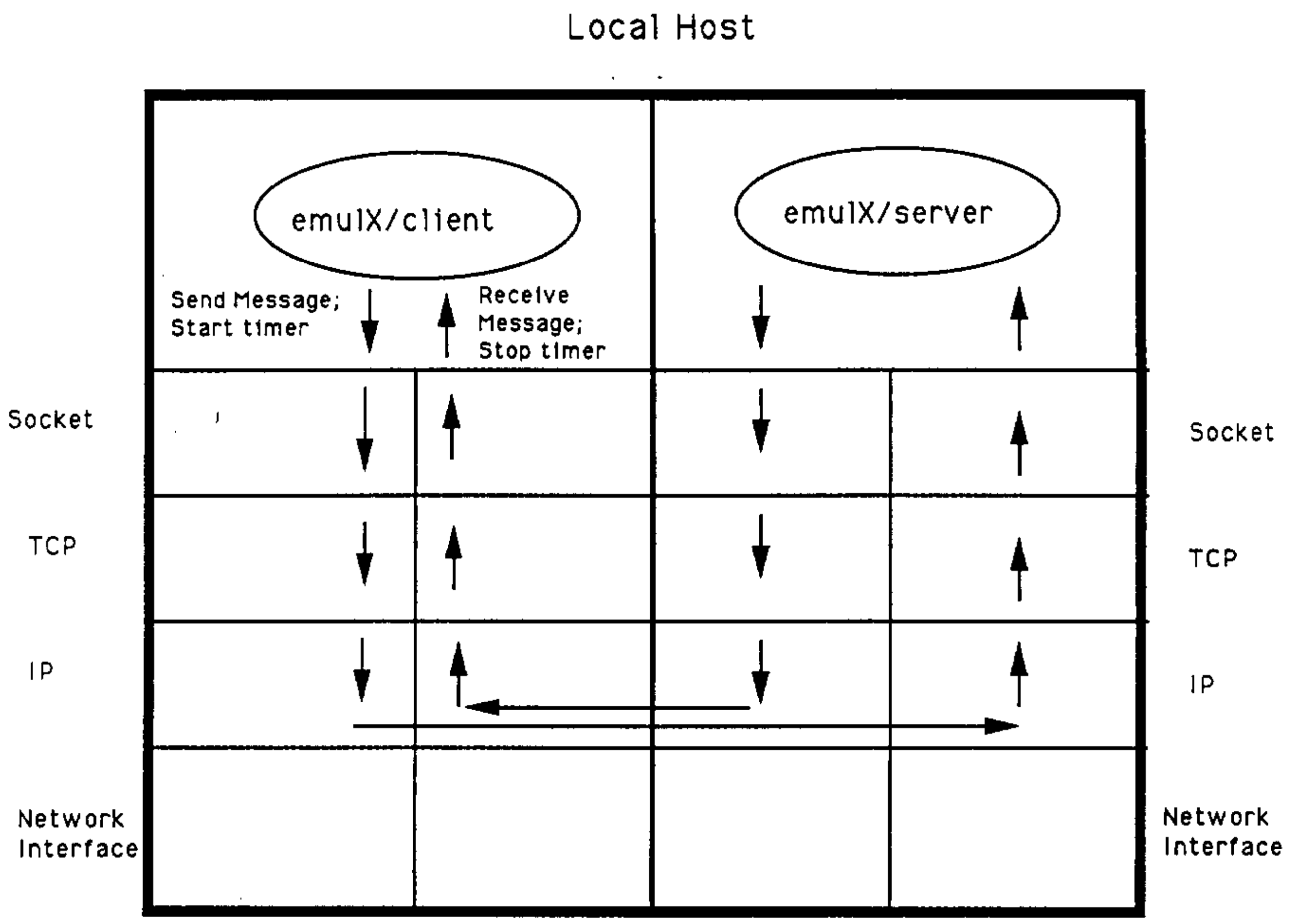

Figure 19: Test Setup to Measure Communication Layers Delay

(4)

$$t_f/m=\rho\frac{[(\bar{m}^2/m^2)+(4e+2)a+5a^2+4e(2e-1)a^2]}{2\{1-\rho[1+(2e+1)a]\}}$$
$$+1+2ea-\frac{(1-e^{-2a\rho})(2ae^{-1}-6a+2/\rho)}{2[F_p(\lambda)e^{-\rho a-1}-1+e^{-2\rho a}]}+\frac{a}{2}$$

where:

t_f	Transfer delay in seconds.
m	The average frame (packet) length (data plus overhead), in units of time.
ρ	Defined as λm.
λ	Total average traffic, in packets/sec.
$\bar{m}^2$	The second moment of the packet length distribution.

a	The ratio of the propagation delay τ to the message transmission length m, τ/m.

The function $F_p(\lambda)$ is the Laplace transform of the packet length distribution $f(t)$:

$$F_p(\lambda)=\int_0^\infty f(t)e^{-\lambda t}dt$$

Assuming the frame length is exponentially distributed, with average length m, we then have;

$$F_p(\lambda)=\frac{1}{(1+\rho)}$$

and

$$\hat{m}/m^2=2$$

Equation **(4)** is credited to Bux [BUX] which is modified slightly from the original derivation by Lam [LAM]. Lam used a discrete-time analysis based on slots 2τ units of time wide, where τ is the end-to-end delay along the bus.

Average Transfer Delay in Subnet 1

The network traffic in subnet 1 represented by **Box-2** in Figure 18 was monitored for eight weekdays between 10:00-11:00 A.M. The end-to-end distance for this subnet was estimated at 500 meters.

With the help of the Novell's LANALYZER (a dedicated machine for network traffic monitoring) overall traffic of our subnet was recorded at one minute time interval.

The following observation were made at the end of our test;

Average Packet Rate [Packet/Sec]: 217.5

Average Packet Length [Bytes]: 151.7

Given the above information and equation **(4)** the average transfer delay per packet in subnet 1 (Box-2) is estimated at 0.126 msec.

Average Transfer Delay in Subnet 2

The network traffic in subnet 1 represented by **Box-4** in Figure 18 was monitored for five weekdays between 10:00-11:00 A.M. The end-to-end distance for this subnet was estimated at 500 meters.

The following observation were made at the end of our test with the help of Novell's LANALYZER;

Average Packet Rate [Packet/Sec]: 186.5

Average Packet Length [Bytes]: 250.6

Given the above information and equation **(4)** the average transfer delay per packet in subnet 2 (Box-4) is estimated at 0.210 msec.

Average Transfer Delay in Backbone Ethernet

Box-3 in Figure 18 represents the backbone Ethernet for the entire site connecting various subnets together.

Previous studies [Pabrai] provided us with the following observation:

Average Packet Rate [Packet/Sec]: 455.28

Average Packet Length [Bytes]: 163.2

Given the above information and equation **(4)** the average transfer delay per packet in the backbone Ethernet (Box-3) is estimated at 0.142 msec.

9. Model Validation

Using the *emulX* application, series of measurements were done to obtain the average delay for an Ethernet packet of size 100 bytes which is close to the average packet size observed in our LAN configuration. The program simply sends Ethernet packets of 100 bytes at 1 minute interval awaits a response from the server which is told to reply with a message of the 100 bytes also. The program was instructed to send messages in one minute interval. Such long gap between each measurement was necessary in order to not violate the Poisson arrival assumption used in our M/M/1 queueing model.

The *emulX* client records and displays the elapsed time (round trip delay) between the request sent and reply received. The average delay experienced by such packet going from the client to the server is then half of the round trip delay.

Using a data set of 40 points the average delay measured for a packet going from the client application to the server application in our LAN configuration is estimated to be 2.75 msec with standard deviation of 1.63.

Total delay predicted from our LAN analytical model (queuing delay in the routers (0.71 msec), channel acquisition delays (0.48 msec) , and communication layer delays (1.1 msec)) is 2.28 msec. The average analytical delay estimate is about 16% away from direct transfer delay measurement which could be the result of following assumptions:

- **Exponential Distribution**: In modeling of the Ethernet Channel acquisition the assumption of exponential packet length distribution was used. Many studies have shown that the packet length distribution in a LAN is more likely to be a bimodal than an exponential one. Direct measurement of subnet 1 and 2 confirmed such observation. The exponential packet length distribution assumption was used in order to reach a close form solution in our queuing analysis.

  The same argument applies also to the exponential packet length distribution assumption used in modeling of our routers as M/M/1 queues. However, as shown earlier in Figures 14 and 15 the observed packet distribution is not that far away from the exponential distribution one.

- **Communication Layer Delay Estimate**: Boxes 1 and 5 in Figure 18 represent the communication layer delays a packet would experience within each host. In combining the average communication layer delay estimate in each host to the average estimates predicted by other queues in a sense we are modeling Boxes 1 or 5 as a M/M/1 queue. There are no indications as to correctness of this assumption. Again this assumption was necessary in order to reach a simple close form solution for our analytical model.

Finally due to the highly dynamic nature of our LAN configuration it is essential to increase the frequency and number of observations made in obtaining the routers and the subnets average traffic profiles as well as the average round trip delay estimate used in validation of the analytical model.

10. Conclusion

More measurements and sample points are required to obtain better estimates on average values used in our model analysis and validation. Random and bursty activities are typical in a LAN environment and a large amount of measurement is really required to obtain a reasonable and repeatable average value.

The flooding experiment should also be modified as described earlier for a better estimate on the router's throughput.

More experiments are needed for larger compilation of *graphical primitives.* However, given the network load and delays involved with these few graphical primitives, one can clearly see the advantages of distributed application paradigm employed in design of X protocol. The low network load and delay associated with the graphical primitives such as resizing a window, moving a window, copying from one window to another, and drawing points or lines are the result of distribution of the graphic task between the client (X application) and the server (X display).

X protocol was designed to operate efficiently for a bitmapped display with predefined geometrical shapes such as lines, points, and rectangles of which a two dimensional bitmapped display is is often comprised.

One can also see from tables 1 and 2 that drawing images could impose a hefty load on the network with long delays specially for the WAN model. This is partially due to their inherent large size, (a Group 3 fax is about 1 Mbytes of data and a digitized color photograph is easily over 10 Mbytes) but also due to the lack efficient mechanism in handling the images.

There are many possibilities for reducing network load and delay. Data Compression can be implemented at the hardware level between the client and the server to reduce the data transfer. Display PostScript allows arbitrary shapes to be scaled rotated, and clipped. Downloading procedures to handle pointer and device request locally and thus eliminating round trip delays for each increment of the pointer movement or to handle the redrawing of the objects are all solutions this problem.

The X protocol high degree of extensibility, hardware independence, network transparency, and its good use of bit mapped display are some of the important reasons for its success. It has clearly filled a void in the distributed graphical applications. However, as Ritchie[6] cleverly points out, "Sometime, even when you fill a vacuum, it still sucks."

[6] Dennis Ritchie's keynote speech at 1991 Summer USENIX conference

11. Future Direction

More sampling of the network traffic is needed to obtain a better estimate on key network characteristics in our LAN configuration. Specifically the measurement on the average delay between client and server should be conducted for a longer period of time as this value is the only way of validating the analytical model.

There is also a definite need for adopting an accurate model for prediction of Ethernet channel acquisition delay.

The analytical analysis could be extended to include WAN configuration. This requires further investigation on the packet distributions in our WAN model.

Finally it would be very interesting to extend the modelling analysis to include the queueing delays (Figure 6) involved in the X application and X server.

12. References

1. [LIDINSKY] "Data Communication Needs," Lidinsky, IEEE Network Magazine, March 90

2. [NYE] "Networking and the X Window System," Adrian Nye

3. [SCHEIFLER] The X Window System, by Robert Scheifler and Jim Gettys, ACM Journal Transactions on Graphics, Vol. 5, No.2, April 1987.

4. [MIT] "X Window System Protocol," MIT X Consortium Standard, X Version 11, Release 4

5. [X11PERF] ftp anonymous uunet.uu.net, UUNET Communications Services

6. [TCPDUMP] ftp-anonymous sol.ctr.columbia.edu

7. [XSCOPE] ftp-anonymous sol.ctr.columbia.edu

8. [TRACEROUTE] ftp-anonymous sol.ctr.columbia.edu

9. [SNMP] SNMP RFC 1098

10. [ROSE]
Rose, M.T.: *The Simple Book: An Introduction to Management of TCP/IP-based internets*, Prentice Hall, Englewood Cliffs, New Jersey, 1991.

11. [SCHWARTZ] Telecommunication Networks, Mischa Schwartz, Addison Wesley, 1988.

12. [TANENBAUM] Computer Networks, Second Edition, Andrew Tanenbaum, PRENTICE HALL, 1988.

13. [LITTLE] Little, D.: "A Prof for the Queuing Formula: $L=\lambda*W$," Oper. Res., vol. 9, pp. 383-387.

14. [KLEINROCK] Communication Nets, Kleinrock, L. New York: Dover, 1964.

15. [PABRAI] Chris O'Reilly and Uday Pabrai, Report on the X terminal Xhibition (EN00290), Computing Division, Fermi National Accelerator Laboratory, 1990.

16. [METCALFE] R.M. Metcalfe and D. R. Boggs, "Ethernet: Distributed packet switching for local computer network," Commun. Ass. Comput. Mach., vol. 19, pp. 395-404, July 1976.

17. [BUX] W. Bux, "Local-Area Subnetworks: A Performance Comparison," *IEEE Trans. on Comm.*, vol. COM-29, no. 10, Oct. 1981, 1465-1473.

18. [LAM] S. S. Lam, "Carrier sense multiple access protocol for local networks," *Computer Networks*, vol. 4, pp. 21-32, Feb. 1980. [TOBAGI] F. A. Tobagi and V. B. Hunt, "Performance analysis of carrier sense multiple access with collision detection," Comput. Networks, vol. 4, pp. 245-259, Oct./Nov. 1980.

19. [FRANTA] Local Networks, W.R. Franta, Imrich Chlamtac, Lexington Books, 1981.

20. [JACKSON] J. R. Jackson, " Job Shop-like Queueing Systems," Management Science, vol. 10, no. 1, pp. 131-142, 1963,

21. [SURI] R. Suri, RObustness of queueing Network Formulas, Journal of the ACM, 30(3), 564-594.

22. [JAIN] Raj Jain, THE ART OF COMPUTER SYSTEMS PERFORMANCE ANALYSIS, Wiley, 1991.

13. Bibliography

1. "An Analysis of TCP Processing Overhead," IEEE Communication Magazine, June 1989.

2. Computer Networks, Tanenbaum

3. "Data Communication Needs," Lidinsky, IEEE Network Magazine, March 90

4. UNIX NETWORKING PROGRAMMING, STEVENS.

5. Probability & Statistics with Reliability, Queuing, and computer Science Applications, Trivedi

6. Queuing Network, Walrand

7. Telecommunication Networks, Schwartz

8. "User-Process Communication Performance in Networks of Computers," IEEE Transactions on Software Engineering, 1988.

9. X Window System Protocol, MIT X Consortium Standard, X Version 11, Release 4

10. X Version 11, Release 4 Source Code. MIT Project Athena.

11. X WINDOW SYSTEM, Scheifler and Gettys

12. X Protocol Reference Manual Vol. 0, O'Reilly

13. Networking and the X Window System, Adrian Nye

14. XLIB programming Manual Vol 1, O'Reilly

This Page Intentionally Left Blank

Appendix A

emulX: X Client-Server Emulator

This appendix provides the source code for the emulX application.

```
/* The following code measures the network delays associated
 * between an X client and an X server
 */

#include <stdio.h>
#include <string.h>
#include <stdlib.h
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netinet/tcp.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <rpc/rpc.h>
#include <math.h>
#include <netinet/in.h>
#include <sys/param.h>

int readn();
int readline();
int writen();
int client_session ();
int fltcompare();

void usage();
void Xemu_server();
void Xemu_server_session ();
void Xemu_client();
int Xemu_client_session();

void report_stat();      /* does statistical analysis */

void t_start();          /* start timer */
void t_stop();           /* stop timer */
void daemon_start();     /* make this a daemon */
double t_getrtime();     /* return real time (elapsed) in seconds */
char *getargu ();        /* get the arguments fromo command line */

#define MAXSAMPLE 5000
#define MAXQUE 5
#define MAXLINE 80
#define SERVER_PORT 7001
#define NUMCELL 10
#ifndef INADDR_NONE
#define INADDR_NONE 0xffffffff
```

```
#endif

int replysize, reqsize, interval;
int count=1;
char *display;
double t_arr[MAXSAMPLE];        /* array of sample network trip delays in msecond

main(argc, argv)
int argc;
char *argv[];

{

  char *p;

  system("date");

  if (argc==1)
    usage(argv[Ø]);
  else if ( argc == 2 && !strcmp(argv[1], "server")) {
    /* must act as an X server */
    printf("\nEmulating an X server Networking Interface\n\n");
    daemon_start();    /* make this a daemon server*/
    Xemu_server();
  }
  else { /* must act as an X client */

   if ( !(p=getargu("-d",argc,argv)) )
     usage(argv[Ø]);
   display = p;

   if ( !(p=getargu("-q",argc,argv)) )
     usage(argv[Ø]);
   reqsize = atoi(p);

   if ( !(p=getargu("-r",argc,argv)) )
     usage(argv[Ø]);
   replysize = atoi(p);

   if ( !(p=getargu("-c",argc,argv)) )
     usage(argv[Ø]);
   count = atoi(p);

   if ( !(p=getargu("-i",argc,argv)) )
     usage(argv[Ø]);
   interval = atoi(p);


   printf("\nEmulating an X client Networking Interface\n\n");
   printf("Sending requests to: %s \n",display);
   printf("request size: %d [byte] \n",reqsize);
   printf("reply size: %d [byte] \n",replysize);
   printf("count: %d \n",count);
   printf("interval [sec]: %d \n\n",interval);
   Xemu_client();
 }
}

void Xemu_client()

{

  int flag, length, sockfd;
  unsigned long inaddr;
  struct sockaddr_in mapper_addr,serv_addr;
  struct hostent *hostptr;
  char *message;


  if ( (sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0){
    fprintf(stderr,"error in creating stream socket \n");
    exit(1);
  }

  bzero((char *) &serv_addr, sizeof(serv_addr));
  serv_addr.sin_family = AF_INET;
  serv_addr.sin_port = htons(SERVER_PORT);

if ( (inaddr = inet_addr(display)) != INADDR_NONE ) {
  /* it is in dotted-decimal format */
  bcopy((char *) &inaddr, (char *) &serv_addr.sin_addr,
        sizeof(inaddr));
}
  else {

  /* it is not in  dotted-decimal format */
  if ( (hostptr = gethostbyname(display)) == '\0'){
    fprintf(stderr,"gethostbyname: error for server host %s \n", display);
    return;
  }
  bcopy(hostptr->h_addr, (char *) &serv_addr.sin_addr, hostptr->h_length);
}


  if( connect(sockfd, (struct sockaddr *) &serv_addr, sizeof(serv_addr)) < 0) {
    fprintf(stderr,"error in connecting to server \n");
    exit(1);
  }
    length=sizeof(serv_addr);
    if (getsockname(sockfd,(struct sockaddr *)&serv_addr,&length) < 0 )
      fprintf(stderr," error in getting socket name \n");

  printf("\n\nclient listening on ephemeral port: %d \n\n", serv_addr.sin_port);

  if ( reqsize > replysize )
    length = reqsize;
  else
    length = replysize;

  if ( (message=malloc(length+1)) == 0 ){
    printf("error in by malloc() \n");
    exit(1);
  }
  message[reqsize+replysize]='\0';

  if (Xemu_client_session(sockfd,message))
    report_stat();
  else
    err("error in client_session");

  close(sockfd);
}

int Xemu_client_session(sockfd, message)

int sockfd;
char *message;
```

```
{
  int n;
  int to_go;
  char line[MAXLINE];

  line[MAXLINE-1]='\0';

  sprintf(line,"%d %d\n",reqsize,replysize);
  n = strlen(line);

  if (writen(sockfd,line,n) != n){
    /* send the request and reply size to server*/
    fprintf(stderr,"session: writen error to socket \n");
    return(0);
  }

  to_go = count;

  while(to_go){

    n=reqsize;
    t_start();
    if (writen(sockfd,message,n) != n){
      fprintf(stderr,"client_session: writen error to socket \n");
      return(0);
    }

    n= readn(sockfd, message, replysize);
    t_stop();

    printf(".");
    fflush(stdout);

    if ( (n != replysize) || (n<0) ){
      printf("client_session: error in reading the server reply \n");
      return(0);
    }

    t_arr[to_go--]=t_getrtime()*1000.0; /* save the elapsed time in msec */

    if (interval) sleep(interval);

  }
  return(1);
}

void Xemu_server()
{

  int sockfd,newsockfd;
  int flag;
  int clilen;
  struct sockaddr_in serv_addr, cli_addr;

  if ( (sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0){
      fprintf(stderr,"error in creating stream socket \n");
    exit(1);
  }

  bzero((char *) &serv_addr, sizeof(serv_addr));
  serv_addr.sin_family = AF_INET;
  serv_addr.sin_addr.s_addr = htonl(INADDR_ANY);
  serv_addr.sin_port = htons(SERVER_PORT);
  if ( bind(sockfd, (struct sockaddr *) &serv_addr, sizeof(serv_addr)) < 0 ){
      fprintf(stderr,"error in binding \n");
      exit(1);
    }

  flag=1;
  if ( setsockopt(sockfd, IPPROTO_TCP, TCP_NODELAY, (char *) &flag,
                  sizeof (flag)) < 0 ) {
    fprintf(stderr,"error in setting stream socket \n");
    exit(1);
  }

  if ( listen(sockfd,MAXQUE)<0) {
    printf("error in listening \n");
    exit(1);
  }

  for(;;){
    clilen = sizeof(cli_addr);
    newsockfd = accept(sockfd, (struct sockaddr *) &cli_addr, &clilen);
           /* blocking */

    if ( newsockfd > 0 ){
      Xemu_server_session(newsockfd);
      close(newsockfd);
    }
    else{
      printf("accept error\n");
      sleep(1);
    }
  }
}

void Xemu_server_session (sockfd)
int sockfd;
{
  int i, n, size, fd;
  int upcoming, outgoing;
  char line[MAXLINE];
  char *buff, *ptr;

  line[MAXLINE-1]='\0';
  n=readline(sockfd,line,MAXLINE); /* read the number of bytes coming */
  if (n == 0 ){
    printf("session: Unable to read number of upcoming bytes \n");
    return;
  }
  upcoming = strtol (line, &ptr, 10);
  if ( upcoming == 0 )
    return;
  outgoing = strtol (ptr, &ptr, 10);

  if ( upcoming > outgoing )
    size = upcoming;
  else
    size = outgoing;

  if ( (buff=malloc(size+1)) == 0 ){
    printf("error by malloc() \n");
    exit(1);
  }

  for(;;){
    if ( (n=readn(sockfd, buff, upcoming)) != upcoming || (n == 0) )
      return;   /* EOF */

    if ((n=writen(sockfd,buff,outgoing)) != outgoing){
      fprintf(stderr,"session: writen error to socket \n");
      return;
    }
  }
}

int writen(fd, ptr, nbytes)

register int fd;
register char *ptr;
register int nbytes;

{

  int nleft, nwritten;

  nleft = nbytes;
  while ( nleft > 0 ) {

    nwritten = write(fd,ptr,nleft);
    if ( nwritten <= 0 )
      return(nwritten);

    nleft -= nwritten;
    ptr += nwritten;

  }

  return(nbytes-nleft);
}


int readn (fd, ptr, nbytes)

register int fd;
register char *ptr;
register int nbytes;
{

  int nleft, nread;

  nleft = nbytes;

  while ( nleft > 0 ) {

    nread = read(fd,ptr,nleft);
    if (nread < 0 )
      return(nread);    /* error, return < 0 */
    else if ( nread == 0 )
      break;            /* EOF */

    nleft -= nread;
    ptr += nread;
  }

  return(nbytes - nleft);       /* return >= 0 */
}

int readline (fd, ptr, maxlen)
```

```
register int fd;
register char *ptr;
register int maxlen;
{

  int n,rc;
  char c;

  for (n=1; n<maxlen; n++) {
    if ( (rc=read(fd,&c,1)) == 1) {
      *ptr++ = c;
      if ( c == '\n' )
        break;
    } else if (rc == Ø ) {
      if ( n == 1 )
        return(Ø);      /* EOF, no data read */
      else
        break;          /* EOF, some data was read */
    } else
      return(-1);       /* error */

  }

  *ptr = Ø;
  return(n);
}


void report_stat()
{

  int i, imin, imax, j, histog[NUMCELL+1];
  double subrange = Ø, avg=Ø, std_dis=Ø;

/* sort it */

  qsort(t_arr+1,count,sizeof(double),fltcompare);

  for(i = 1; i<=NUMCELL; i++)
    histog[i]=Ø;

  imin = count * .1;
  imax = count * .9;

  subrange = (t_arr[imax] - t_arr[imin]) / NUMCELL;

/* get the average */

  printf("\n \nSample time delay points in msec\n");

  for (i=1; i <= count; i++){
    if ( i%1Ø != Ø )
      printf("%.1f ",t_arr[i]);
    else
      printf("%.1f\n\n",t_arr[i]);
    if ( i >= imin && i <= imax ){
      histog[(int)((t_arr[i]-t_arr[imin])/subrange) + 1]++;
      avg += t_arr[i];
    }
  }


  avg = avg / (imax-imin+1);
  for (i=imin; i <= imax; i++)
    std_dis += pow( (t_arr[i]-avg),(double)2.0);

  std_dis = pow( (std_dis /(imax-imin)), (double)0.5);

  printf("\n\n Number of sample points: %d\n", imax-imin+1);

  printf(" Total number of cells: %d\n Cell size: %6.1f\n",NUMCELL, subrange);

  for ( i = 1; i<= NUMCELL; i++){
    printf("\n%6.1f-%6.1f ",t_arr[imin]+(i-1)*subrange,
           t_arr[imin]+(i)*subrange);
    for (j = 1; j <= histog[i];j++)
      printf("*");
  }
  printf("\n\n Median delay: %.3f \n",t_arr[(imax-imin+1)/2]);
  printf("\n Average delay (msec): %.3f \n Standard deviation: %.3f \n\n", avg, s

  printf(" Overall bit-rate[Kb/sec]: %.3f \n", (reqsize*8/1024)*1000.0/avg);

}


#include  <stdio.h>
#include  <signal.h>
#include  <sys/param.h>
#include  <errno.h>
extern int errno;

#ifdef SIGTSTP /* true if BSD system */

#include  <sys/file.h>
#include  <sys/ioctl.h>

#endif


/* detach a daemon process from login session context */

void daemon_start()

{
  register int childpid, fd;


  if (getpid() == 1)
    goto out;

  /* Ignore the terminal stop signals (BSD) */

#ifdef SIGTTOU
  signal(SIGTTOU, SIG_IGN);
#endif

#ifdef SIGTTIN
  signal(SIGTTIN, SIG_IGN);
#endif

#ifdef SIGTSTP
  signal(SIGTSTP, SIG_IGN);
#endif

#ifdef SIGHUP
```

```
  signal(SIGHUP,SIG_IGN);
#endif

  if ( (childpid = fork()) < 0 )
    fprintf(stderr,"can't fork first child \n");
  else if ( childpid > 0 )
    exit(0);     /* parent */

  /* first child process */

#ifdef SIGTSTP  /* BSD */
  if (setpgrp(0, getpid()) == -1)
    exit(1);
  if ( (fd=open("/dev/tty",O_RDWR)) >= 0 ) {
    ioctl(fd, TIOCNOTTY, (char *)NULL); /* lose controling terminal */
    close(fd);
  }

#else   /* SYSTEM V */

  if (setpgrp() == -1)
    exit(1);
  signal(SIGHUP,SIG_IGN);

  if ( (childpid = fork()) < 0)
    exit(1);
  else if (childpid > 0)
    exit(0);     /* first child */

  /* second child */

#endif

 out:

  for (fd=0; fd < NOFILE; fd++)
    close(fd);

  errno=0;

  chdir("/");

  umask(0);
}


 int    fltcompare(i,j)
  double *i, *j;
  {
       return((int)(*i - *j));
  }




char *getargu(key, argc, argv)
char *key;
int argc;
char *argv[];
  {

    int i;
    for ( i=1; i<argc; i++ ){

      if ( strcmp(key, argv[i]) == 0 ){

        if ( strchr(argv[i+1], '-') == 0 )

          return(argv[i+1]);

        else{
          fprintf(stderr,"error in argument syntax \n");
          exit(1);
        }
      }
    }
      /* found nothing */

      return (0);
  }


void usage(name)

char *name;

{
    printf("\n\nUsage: %s server (Network Emulate an X server) \n\n",name);
    printf("\n\nUsage: %s options (Network Emulate an X Client) \n\n",name);
    printf("options: -d display-name (Xserver name) \n");
    printf("         -q request-size [byte] \n");
    printf("         -r response-size [byte] \n");
    printf("         -c count \n");
    printf("         -i interval(sec) \n\n");
    exit(0);
  }
```

The following is the source code for the time function

```
#include <stdio.h>
#include <sys/time.h>
#include <sys/resource.h>

static struct timeval time_start, time_stop; /* for real time */
static struct rusage  ru_start, ru_stop; /* for user and sys time */

static double   tstart, tstop, seconds;

/*
 * start the timer.
 * We don't return anything to the caller, we just store some information for the
 * stop timer routine to access.
 */

void err();

void
t_start()

{

  if (gettimeofday (&time_start, (struct timezone *) 0 ) < 0 )
    err("t_start: gettimeofday() error");

  if (getrusage(RUSAGE_SELF, &ru_start) < 0)
```

```
    err("t_start: getrusage() error");

}

/*
 * Stop the timer and save the appropriate information.
 */

void
t_stop()
{

  if (getrusage(RUSAGE_SELF, &ru_stop) < 0)
    err("t_stop: getrusage() error");

  if (gettimeofday (&time_stop, (struct timezone *) 0 ) < 0 )
    err("t_stop: gettimeofday() error");
}

/*
 * Return the user time in seconds.
 */

double
t_getutime()
{

  tstart = ((double) ru_start.ru_utime.tv_sec) * 1000000.0 +
    ru_start.ru_utime.tv_usec;

  tstop = ((double) ru_stop.ru_utime.tv_sec) * 1000000.0 +
    ru_stop.ru_utime.tv_usec;

  seconds = (tstop - tstart) / 1000000.0;

  return(seconds);
}

/*
 * return the system time in seconds.
 */

double
t_getstime()
{

  tstart = ((double) ru_start.ru_stime.tv_sec) * 1000000.0 +
    ru_start.ru_stime.tv_usec;

  tstop = ((double) ru_stop.ru_stime.tv_sec) * 1000000.0 +
    ru_stop.ru_stime.tv_usec;

  seconds = (tstop - tstart) / 1000000.0;

  return(seconds);

}


/*
 * return the real (elapsed) time in seconds.
 */

double
```

```
t_getrtime()
{

  tstart = ((double) time_start.tv_sec) * 1000000.0 +
    time_start.tv_usec;

  tstop = ((double) time_stop.tv_sec) * 1000000.0 +
    time_stop.tv_usec;

  seconds = (tstop - tstart) / 1000000.0;

  return(seconds);
}


void
err(s)
char *s;
{
  fprintf(stderr,"%s \n",s);
}
```

Appendix B

X Requests synopsis and Raw Data

This appendix provides the synopsis and the raw data collected for all the X requests generated by the modified *x11perf*.

Common X Setup Calls

The following is the listing of common setup X calls within the *x11perf* program.

- *XOpenDisplay*: Connects the client to the server controlling the hardware display through TCP, or UNIX or, DECnet streams.

  Synopsis

  ```
  Display *XOpenDisplay(display_name)

          char* display_name;
  ```

  Arguments

  - *display_name* Specifies which server to connect to. If display_name is NULL then the Xlib will try to connect to the X server specified by the environment variable DISPLAY with the format of *host:server.screen*, where *host* is the Internet address or name of the X server, *server* is the server number on that machine (for single user workstation the *server* is set to 0) and optional *screen*, the screen number on that server.

  Experimental Data:

```
XOpenDisplay()

 Xclient.Sun > Xserver.SiliconG: S 686080000:686080000(0) win 4096 <mss
 Xserver.SiliconG > Xclient.Sun: S 1363008000:1363008000(0) ack 6860800
 Xclient.Sun > Xserver.SiliconG: . ack 1 win 4096
 Xclient.Sun > Xserver.SiliconG: P 1:13(12) ack 1 win 4096
 Xserver.SiliconG > Xclient.Sun: P 1:185(184) ack 13 win 16384
 Xclient.Sun > Xserver.SiliconG: P 13:57(44) ack 185 win 4096
 Xserver.SiliconG > Xclient.Sun: P 185:217(32) ack 57 win 16384
 Xclient.Sun > Xserver.SiliconG: . ack 217 win 4096
```

- *XCreateSimpleWindow*: Creates an unmapped *InputOutput* subwindow of the specified parent window.

Synopsis

```
Window XCreateSimpleWindow(display, parent, x,y, width, height, border_
        border, border, background)

        Display display;
        Window parent;
        int x,y;
        unsigned int width, height, border_width;
        unsigned long border;
        unsigned long background;
```

Arguments

- *display* Specifies a pointer to the *Display*; returned from *XOpenDisplay*.
- *parent* Specifies the parent window ID.
- *x,y* Specifies the x and y coordinates of the upper-left pixel of the new window's border relative to the origin of the parent.
- *width, height* Specify the width and height, in pixels, of the new window.
- *border_width* Specifies the width, in pixels, of the new window's border.
- *border* Specifies the pixel value for the border.
- *background* Specifies the pixel value for the background of the window.

Experimental Data:

```
XCreateSimpleWindow()

 Xclient.Sun > Xserver.SiliconG: P 89:133(44) ack 345 win 4096
 Xserver.SiliconG > Xclient.Sun: P 345:377(32) ack 133 win 16384
 Xclient.Sun > Xserver.SiliconG: . ack 377 win 4096
```

- *XChangeWindowAttributes*: Changes any or all of the window attributes that can be changed.

Synopsis

```
XChangeWindowAttributes ( display, w, valuemask, attributes)

        Display *display;
        Window w;
        unsigned long valuemask;
        XSetWindowAttributes *attributes;
```

Arguments

- ○ ***display*** Specifies a pointer to the *Display*; returned from *XOpenDisplay*.
- ○ ***w*** Specifies the window ID.
- ○ ***valuemask*** Specifies which window attributes are defined in the ***attributes*** argument. The mask is made by combining the appropriate mask symbols listed in the Structure section defined by XSetWindowAttributes using bitwise OR (|).
- ○ ***attributes*** Window attributes to be changed. The *valuemask* indicates which members in this structure are referenced.

Experimental Data:

```
XChangeWindowAttributes ()

 Xclient.Sun > Xserver.SiliconG: P 133:161(28) ack 377 win 4096
 Xserver.SiliconG > Xclient.Sun: P 377:409(32) ack 161 win 16384
 Xclient.Sun > Xserver.SiliconG: . ack 409 win 4096
```

- *XMapWindow*: Maps a window, making it eligible for display.

Synopsis

```
XMapWindow(display, w)

       Display *display;
       Window w;
```

Arguments

- *display* Specifies a pointer to the *Display*; returned from *XOpenDisplay*.
- *w* Specifies the window ID.

Experimental Data:

```
Xclient.Sun > Xserver.SiliconG: P 161:173(12) ack 409 win 4096
Xserver.SiliconG > Xclient.Sun: . ack 173 win 16384
Xserver.SiliconG > Xclient.Sun: P 409:441(32) ack 173 win 16384
Xclient.Sun > Xserver.SiliconG: . ack 441 win 4096
```

- *XWarpPointer*: Move the pointer suddenly from one point on the screen to another.

Synopsis

```
XWarpPointer(display, src_w, dest_w, src_x, src_y, src_width,
       src_height, dest_x, dest_y)

       Display *display;
       Window src_w, dest_w;
       int src_x, src_y;
       unsigned int src_width, src_height;
       int dest_x, dest_y;
```

Arguments

- *display* Specifies a pointer to the *Display*; returned from *XOpenDisplay*.
- *src_w* Specifies the ID of the source window. Could be *None*.
- *dest_w* Specifies the ID of the destination window. Could be *None*.
- *src_x, src_y* Specify the x and y coordinates within the source window.

- *src_width, src_hight* Specify the width and height in pixels of the source area.

- *dest_x, dest_y* Specify the destination x and y coordinates within the destination window. If *dest_w* is *None*, these coordinates are relative to the root window of *dest_w*.

Experimental Data:

```
XWarpPointer()

 Xclient.Sun > Xserver.SiliconG: P 281:309(28) ack 569 win 4096
 Xserver.SiliconG > Xclient.Sun: . ack 309 win 16384
 Xserver.SiliconG > Xclient.Sun: P 569:601(32) ack 309 win 16384
 Xclient.Sun > Xserver.SiliconG: . ack 601 win 4096
```

- *XCreateGC*: Creates a new graphics resource in the server.

Synopsis

```
GC XCreateGC(display, drawable, valuemask, values)

        Display *display;
        Drawable drawable;
        unsigned long valuemask;
        XGCValues *values;
```

Arguments

- *display* Specifies a pointer to the *Display*; returned from *XOpenDisplay*.

- *drawable* Specifies a drawable. The created GC can only be used to draw in drawables of the same depth as this *drawable*.

- *valuemask* Specifies which members of the GC are to be set using information in the *value* structure.

- *values* Specifies a pointer to an *XGCValues* structure which will provide components for the new GC.

Experimental Data:

```
XCreateGC()

 Xclient.Sun > Xserver.SiliconG: P 309:337(28) ack 601 win 4096
 Xserver.SiliconG > Xclient.Sun: P 601:633(32) ack 337 win 16384
 Xclient.Sun > Xserver.SiliconG: . ack 633 win 4096
```

- *XDestroySubwindows*: Destroys all descendants of the specified window (recursively).

Synopsis

```
XDestroySubwindows(display,w)

        Display *display;
        Window w;
```

Arguments

- *display* Specifies a pointer to the *Display*; returned from *XOpenDisplay.*
- *w* Specifies the window ID.

Experimental Data:

```
XDestroySubwindows()

 Xclient.Sun > Xserver.SiliconG: P 493:505(12) ack 793 win 4096
 Xserver.SiliconG > Xclient.Sun: P 793:825(32) ack 505 win 16384
 Xclient.Sun > Xserver.SiliconG: . ack 825 win 4096
```

- *XClearWindow*: Clears a window, but does not cause exposure events.

Synopsis

```
XClearWindow(display,w)

        Display *display;
        Window w;
```

Arguments

- *display* Specifies a pointer to the *Display*; returned from *XOpenDisplay*.
- *w* Specifies the window ID.

Experimental Data:

```
XClearWindow()

 Xclient.Sun > Xserver.SiliconG: P 505:525(20) ack 825 win 4096
 Xserver.SiliconG > Xclient.Sun: . ack 525 win 16384
 Xserver.SiliconG > Xclient.Sun: P 825:857(32) ack 525 win 16384
 Xclient.Sun > Xserver.SiliconG: . ack 857 win 4096
```

- *XGetWindowAttributes*: returns the *XWindowAttributes* structure containing the current window attributes.

Synopsis

```
Status XGetWindowAttributes (display, w, window_attributes)

        Display *display;
        Window w;
        XWindowAttributes *window_attributes; /* RETURN */
```

Arguments

- *display* Specifies a pointer to the *Display*; returned from *XOpenDisplay*.
- *w* Specifies the window ID.
- *window_attributes* Returns a filled *XWindowAttributes* structure, containing the current attributes for the specified window.

Experimental Data:

```
XGetWindowAttributes ()

 Xclient.Sun > Xserver.SiliconG: P 525:533(8) ack 857 win 4096
 Xserver.SiliconG > Xclient.Sun: P 857:901(44) ack 533 win 16384
 Xclient.Sun > Xserver.SiliconG: P 533:541(8) ack 901 win 4096
 Xserver.SiliconG > Xclient.Sun: P 901:933(32) ack 541 win 16384
 Xclient.Sun > Xserver.SiliconG: P 541:545(4) ack 933 win 4096
 Xserver.SiliconG > Xclient.Sun: P 933:965(32) ack 545 win 16384
 Xclient.Sun > Xserver.SiliconG: . ack 965 win 4096
```

- *XFreeGC*: Frees all memory associated with a graphics context, and removes the GC from the server and display hardware.

Synopsis

```
XFreeGC(display, gc)

        Display *display;
        GC gc;
```

Arguments

- *display* Specifies a pointer to the *Display*; returned from *XOpenDisplay*.
- *gc* Specifies the graphics context to be freed.

Experimental Data:

```
XFreeGC()

 Xclient.Sun > Xserver.SiliconG: P 545:557(12) ack 965 win 4096
 Xserver.SiliconG > Xclient.Sun: P 965:997(32) ack 557 win 16384
 Xclient.Sun > Xserver.SiliconG: . ack 997 win 4096
```

- *XDestroyWindow*: The window and all inferiors (recursively) are destroyed, and a *DestroyNotify* event is generated for each window.

Synopsis

```
XDestroyWindow(display, w)

        Display *display;
        Window w;
```

Arguments

- *display* Specifies a pointer to the *Display*; returned from *XOpenDisplay*.
- *w* Specifies the window ID to be destroyed.

Experimental Data:

```
XDestroyWindow()

 Xclient.Sun > Xserver.SiliconG: P 569:581(12) ack 1029 win 4096
 Xserver.SiliconG > Xclient.Sun: . ack 581 win 16384
 Xserver.SiliconG > Xclient.Sun: P 1029:1061(32) ack 581 win 16384
 Xclient.Sun > Xserver.SiliconG: . ack 1061 win 4096
```

- *closing a network connection.*

Experimental Data:

```
 Xclient.Sun > Xserver.SiliconG: F 597:597(0) ack 1093 win 4096
 Xserver.SiliconG > Xclient.Sun: . ack 598 win 16384
 Xserver.SiliconG > Xclient.Sun: F 1093:1093(0) ack 598 win 16384
 Xclient.Sun > Xserver.SiliconG: . ack 1094 win 4096
```

Graphical Primitives X Calls

The following is the listing of all the *graphical primitives'* X calls involved in the *x11perf* program:

- *XDrawPoints*: draws one or more points into the specified drawable.

Synopsis

```
XDrawPoints(display, drawable, gc, points, npoints, mode)

        Display *display;
        Drawable drawable;
        GC gc;
        Xpoints *points;
        int npoints;
        int mode;
```

Arguments

- *display* Specifies a connection to an X server; returned from *XOpenDisplay*.
- *drawable* Specifies the drawable (e.g., window).
- *gc* Specifies the graphics context.
- *points* specifies a pointer to an array of *XPoint* structures containing the positions of the points.
- *npoints* Specifies the number of points to be drawn (**npoionts= 1000**).
- *mode* Specifies the coordinate mode.

Experimental Data:

```
Xclient.Sun > Xserver.SiliconG: . 281:1741(1460) ack 581 win 4096
Xclient.Sun > Xserver.SiliconG: . 1741:3201(1460) ack 581 win 4096
Xclient.Sun > Xserver.SiliconG: P 3201:4293(1092) ack 581 win 4096
Xclient.Sun > Xserver.SiliconG: P 4293:4297(4) ack 581 win 4096
Xserver.SiliconG > Xclient.Sun: . ack 4297 win 16384
Xserver.SiliconG > Xclient.Sun: P 581:613(32) ack 4297 win 16384
Xclient.Sun > Xserver.SiliconG: . ack 613 win 4096
```

- *XDrawLines*: Draws a series of lines joined end-to-end.

Synopsis

```
XDrawLines(display, drawable, gc, points, npoints, mode)

Display *display;
Drawable drawable;
GC gc;
Xpoints *points;
int npoints;
int mode;
```

Arguments

- *display* Specifies a connection to an X server; returned from *XOpenDisplay*.
- *drawable* Specifies the drawable (e.g., window).
- *gc* Specifies the graphics context.

- *points* specifies a pointer to an array of *XPoint* structures containing the positions of the points.
- *npoints* Specifies the number of points in the array, (**npoionts= 1000**).
- *mode* Specifies the coordinate mode.

Experimental Data:

```
XDrawLines()

 Xclient.Sun > Xserver.SiliconG: . 281:1741(1460) ack 581 win 4096
 Xclient.Sun > Xserver.SiliconG: . 1741:3201(1460) ack 581 win 4096
 Xclient.Sun > Xserver.SiliconG: P 3201:4297(1096) ack 581 win 4096
 Xclient.Sun > Xserver.SiliconG: P 4297:4301(4) ack 581 win 4096
 Xserver.SiliconG > Xclient.Sun: . ack 4301 win 16384
 Xserver.SiliconG > Xclient.Sun: P 581:613(32) ack 4301 win 16384
 Xclient.Sun > Xserver.SiliconG: . ack 613 win 4096
```

- *XLoadQueryFont*: loads a font and fill information structure.

Synopsis

```
XFontStruct *XLoadQueryFont (diplay, name)

        Display *display;
        char *name;
```

Arguments

- *display* Specifies a connection to an X server; returned from *XOpenDisplay*.
- *name* Specifies the name of the font, (**name = "6x13"**).

Experimental Data:

```
XLoadQueryFont()

 Xclient.Sun > Xserver.SiliconG: P 261:285(24) ack 473 win 4096
 Xserver.SiliconG > Xclient.Sun: . 473:1933(1460) ack 285 win 16384
 Xserver.SiliconG > Xclient.Sun: P 1933:2237(304) ack 285 win 16384
 Xclient.Sun > Xserver.SiliconG: . ack 2237 win 4096
 Xclient.Sun > Xserver.SiliconG: P 285:289(4) ack 2237 win 4096
 Xserver.SiliconG > Xclient.Sun: P 2237:2269(32) ack 289 win 16384
 Xclient.Sun > Xserver.SiliconG: . ack 2269 win 4096
```

- *XChangeGC*: Changes the components of a given graphics context.

Synopsis

```
XChangeGC(display, gc, valuemask, values)

Display *display;
GC gc;
unsigned long valuemask;
XGCValues *values;
```

Arguments

- *display* Specifies a connection to an X server; returned from *XOpenDisplay*.
- *gc* Specifies the graphics context.
- *valuemask* Specifies the component in the graphics context to be changed, (**valuemask = GCFont**).
- *values* Specifies a pointer to the *XGCVlaues* structure.

Experimental Data:

```
XChangeGC()
 Xclient.Sun > Xserver.SiliconG: P 309:329(20) ack 2301 win 4096
 Xserver.SiliconG > Xclient.Sun: . ack 329 win 16364
 Xserver.SiliconG > Xclient.Sun: P 2301:2333(32) ack 329 win 16384
 Xclient.Sun > Xserver.SiliconG: . ack 2333 win 4096
```

- *XDrawString*: Draw an 8-bit text string, foreground only.

Synopsis

```
XDrawString(display, drawable, gc, x, y, string, length)

Display *display;
Drawable drawable;
GC gc;
int x, y;
Char *string;
int length;
```

Arguments

- *display* Specifies a connection to an X server; returned from *XOpenDisplay.*
- *drawable* Specifies the drawable.
- *gc* Specifies the graphics context.
- *x, y* Specify the x and y coordinates of the baseline starting position for the character, relative to the origin of the specified drawable.
- *string* Specifies the character string.
- *length* SPecifies the number of characters in *string.*

Experimental Data:

```
XDrawString()

 Xclient.Sun > Xserver.SiliconG: P 349:453(104) ack 2441 win 4096
 Xserver.SiliconG > Xclient.Sun: P 2441:2473(32) ack 453 win 16384
 Xclient.Sun > Xserver.SiliconG: . ack 2473 win 4096
```

- *XFreeFont:* Unloads a font and free storage for the font structure.

Synopsis

```
XFreeFont(display, font_struct)

Display *display;
XFontStruct *font_struct;
```

Arguments

- *display* Specifies a connection to an X server; returned from *XOpenDisplay.*
- *font_struct* Specifies the storage associated with the font.

Experimental Data:

```
XFreeFont()

 Xclient.Sun > Xserver.SiliconG: P 473:485(12) ack 2505 win 4096
 Xserver.SiliconG > Xclient.Sun: . ack 485 win 16372
 Xserver.SiliconG > Xclient.Sun: P 2505:2537(32) ack 485 win 16384
 Xclient.Sun > Xserver.SiliconG: . ack 2537 win 4096
```

- *XCopyArea:* Combines (copies) the specified rectangle of *src* with the specified rectangle of *dest.* Both *src* and *dest* must have the same root and depth.

Synopsis

```
XCopyArea(display, src, dst, gc, src_x, src_y, width, height,
dest_x, dest_y);

        Display *display;
        Drawable src, dest;
        GC gc;
        int src_x, src_y;
        unsigned int width, height;
        int dest_x, dest_y;
```

Arguments

- *display* Specifies a connection to an X server; returned from *XOpenDisplay.*
- *src, dest* Specify the source and destination rectangles to be combined.
- *gc* Specifies the graphics context.
- *src_x, src_y* Specify the x and y coordinates of the upper-left cornet of the source rectangle.
- *width, height* specify the dimension in pixels of both the source and destination rectangles, (**width=height=100 pixels**]).
- *dest_x, dest_y* Specify the x and y coordinates within the destination window.

Experimental Data:

```
XCopyArea()

 Xclient.Sun > Xserver.SiliconG: P 697:729(32) ack 613 win 4096
 Xserver.SiliconG > Xclient.Sun: P 613:645(32) ack 729 win 16384
 Xclient.Sun > Xserver.SiliconG: . ack 645 win 4096
```

- *XPutImage* : Draws a section of an rectangle in a window or pixmap.

Synopsis

```
        XPutImage(display, drawable, gc, image,  src_x, src_y, dst_x,
                dst_y, width, height)

        Display *display;
        Drawable drawable;
        GC gc;
        XImage *image;
        int src_x, src_y;
        int dst_x, dst_y;
        unsigned int width, height;
```

Arguments

- *display* Specifies a connection to an X server; returned from *XOpenDisplay*.
- *drawable* Specifies the drawable.
- *gc* Specifies the graphics context.
- *image* Specifies the image to be combined with the rectangle.
- *src_x, src_y* Specify the x and y coordinates of the upper-left corner of the rectangle to be copied.
- *dst_x, dst_y* Specify the x and y coordinates relative to the origin of the drawable, where the upper-left corner of the copied rectangle will be place.
- *width, height* specify the width and height in pixels of the rectangle area to be copied. (**width=height=100 pixels**|).

Experimental Data:

```
XPutImage()

 Xclient.Sun > Xserver.SiliconG: . 10769:12229(1460) ack 360741 win 409
 Xclient.Sun > Xserver.SiliconG: . 12229:13689(1460) ack 360741 win 409
 Xclient.Sun > Xserver.SiliconG: P 13689:14865(1176) ack 360741 win 409
 Xserver.SiliconG > Xclient.Sun: . ack 14865 win 16384
 Xclient.Sun > Xserver.SiliconG: . 14865:16325(1460) ack 360741 win 409
 Xclient.Sun > Xserver.SiliconG: . 16325:17785(1460) ack 360741 win 409
 Xclient.Sun > Xserver.SiliconG: P 17785:18961(1176) ack 360741 win 409
 Xserver.SiliconG > Xclient.Sun: . ack 18961 win 16384
 Xclient.Sun > Xserver.SiliconG: . 18961:20421(1460) ack 360741 win 409
 Xclient.Sun > Xserver.SiliconG: P 20421:20793(372) ack 360741 win 4096
 Xclient.Sun > Xserver.SiliconG: P 20793:20797(4) ack 360741 win 4096
 Xserver.SiliconG > Xclient.Sun: P 360741:360773(32) ack 20797 win 1638
 Xclient.Sun > Xserver.SiliconG: . ack 360773 win 4096
```

- *XGetImage*: Dumps the contents of the specified rectangle, a drawable, into a client-side *XImage* structure, in the format specified.

Synopsis

```
XImage *XGetImage(display, drawable, x, y, width, height,
        plane_mask, format)

Display *display;
Drawable drawable;
int x, y;
unsigned int width, height;
unsigned long plane_mask;
int format;
```

Arguments

- *display* Specifies a connection to an X server; returned from *XOpenDisplay.*
- *drawable* Specifies the drawable.
- *x, y* Specify the x and y coordinates of the upper-left corner of the rectangle.
- *width, height* specify the width and height in pixels of the image, (**width=height=100 pixels**|).
- *plane_mask* Specifies a plane mask that indicates which planes are represented in the image, (**plane_mask = ˜0**).
- *format* Specifies the format for the image, (**format=ZPixmap**).

Experimental Data:

```
XGetImage()

 Xclient.Sun > Xserver.SiliconG: P 721:741(20) ack 360677 win 4096
 Xserver.SiliconG > Xclient.Sun: . 360677:362137(1460) ack 741 win 1638
 Xserver.SiliconG > Xclient.Sun: . 362137:363597(1460) ack 741 win 1638
 Xclient.Sun > Xserver.SiliconG: . ack 363597 win 4096
 Xserver.SiliconG > Xclient.Sun: . 363597:365057(1460) ack 741 win 1638
 Xserver.SiliconG > Xclient.Sun: . 365057:366517(1460) ack 741 win 1638
 Xclient.Sun > Xserver.SiliconG: . ack 366517 win 4096
 Xserver.SiliconG > Xclient.Sun: . 366517:367977(1460) ack 741 win 1638
 Xserver.SiliconG > Xclient.Sun: . 367977:369437(1460) ack 741 win 1638
 Xclient.Sun > Xserver.SiliconG: . ack 369437 win 4096
 Xserver.SiliconG > Xclient.Sun: P 369437:370709(1272) ack 741 win 1638
 Xclient.Sun > Xserver.SiliconG: P 741:745(4) ack 370709 win 4096
 Xserver.SiliconG > Xclient.Sun: P 370709:370741(32) ack 745 win 16384
 Xclient.Sun > Xserver.SiliconG: . ack 370741 win 4096
```

•

XResizeWindow : Changes the inside dimension of the window. The

Synopsis

```
XResizeWindow (display, w, width, height)

        Display *display;
        Window w;
        Unsigned int width, height;
```

Arguments

- *display* Specifies a connection to an X server; returned from *XOpenDisplay.*
- *w* Specifies the ID of the window to be resized.
- *width, height* Specify the new dimension of the window in pixels.

Experimental Data:

```
XResizeWindow()

 Xclient.Sun > Xserver.SiliconG: P 469:493(24) ack 741 win 4096
 Xserver.SiliconG > Xclient.Sun: P 741:773(32) ack 493 win 16384
 Xclient.Sun > Xserver.SiliconG: . ack 773 win 4096
```

- *XMoveWindow*: Changes the position of the origin of the specified window relative to its parent.

  Synopsis

```
XMoveWindow (display, w, x, y )

        Display *display;
        Window w;
        int x,y;
```

  Arguments

  - *display* Specifies a connection to an X server; returned from *XOpenDisplay*.
  - *w* Specifies the ID of the window to be moved.
  - *x, y* Specify the new x, and y coordinates of the upper-left pixel of the window's border, relative to its parent.

  Experimental Data:

```
XMoveWindow()

 Xclient.Sun > Xserver.SiliconG: P 541:565(24) ack 837 win 4096
 Xserver.SiliconG > Xclient.Sun: P 837:869(32) ack 565 win 16384
 Xclient.Sun > Xserver.SiliconG: . ack 869 win 4096
```

Appendix C

X Requests Listings and Descriptions

This appendix provides a summary of X requests grouped by functionality.

Each X request is followed by a brief description, it opcode, number of data bytes used, number of padding bytes used for alignment requirements, total length of the request including the header, and its corresponding reply length.

Colors and Colormaps

AllocColor:

> **Description:** Allocate a read-only colorcell specifying the color with RGB values.
>
> **opcode:84**
> **data:10**
> **padding:2**
> **total length:16**
> **reply length:32**

AllocColorCells:

> **Description:** Allocate read/write colorcells. This request does not set the colors of the allocated cells.
>
> **opcode:86**
> **data:8**
> **padding:0**
> **total length:12**
> **reply length: 32+4n**
> n: number of pixels values
> m: numer of masks values

AllocColorPlanes:

> **Description:** Allocate read/write colorcells for overlays. This request does not set the colors of the allocated cells.
>
> **opcode:87**
> **data:12**
> **padding:0**
> **total length:16**
> **reply length:32+4n**
> n: number of pixels values

AllocNamedColor:

Description: Allocate a read-only colorcell specifying the color with a color name.

opcode:85
data:8+n
    n: length of string
padding:p
total length:12+n+p
reply length:32

CopyColormapAndFree:

Description: Copy into a new colormap the colorcells that one client has allocated, and free these colorcells in the old colormap.

opcode:80
data:8
padding:0
total length:12
reply length:0

CreateColormap:

Description: Create a virtual colormap.

opcode:78
data:12
padding:0
total length:16
reply length:0

FreeColormap:

Description: Free a virtual colormap.

opcode:79
data:4
padding:0
total length:8
reply length:0

FreeColors:

Description: Deallocate colorcells.

opcode:88
data:8+4n
    n: number of pixel values
padding:0
total length:12+4n
reply length:0

InstallColormap:

Description: Copy a virtual colormap into the display hardware, sothat it will actually be used to translate pixel values.

opcode:81
data:4
padding:0
total length:8
reply length:0

ListInstalledColormaps:

Description: List the IDs of the colormaps installed in the hardware.

opcode:83
data:4
padding:0
total length:8
reply length:32+4n
n: number of COLORMAPs

LookupColor:

Description: Return the RGB values associated with a color name, and return the closest RGB values available on the display hardware.

opcode:92
data:8+n
n: length of string
padding:p
total length:12+n+p
reply length:32

QueryColors:

Description: Return the colors in the specified cells of a colormap.

opcode:91
data:4+4n
n: number of pixel values
padding:0
total length:8+4n
reply length:32+8n
n: number of RGBs in colors

StoreColors:

Description: Store colors into cells allocated by AllocColorCells or AllocColorPlanes.

opcode:89
data:4+12n
n: number of COLORITEMs
padding:0
total length:8+12n
reply length:

StoreNamedColor:

Description: Store colors into cells allocated by AllocColorCells or AllocColorPlanes.

opcode:90
data:12+n
n: string length
padding:p
total length:16+n+p
reply length:0

UninstallColormap:

Description: Remove a virtual colormap from the display hardware, so it will not be used to translate pixel values.

opcode:82

data:4
padding:0
total length:8
reply length:0

Cursors

CreateCursor:

Description: Create a cursor resource from characters in a special cursor font.

opcode:93
data:28
padding:0
total length:32
reply length:0

CreateGlyphCursor:

Description: Create a cursor from characters in any font.

opcode:94
data:28
padding:0
total length:32
reply length:0

FreeCursor:

Description: Destroy a cursor resource.

opcode:95
data:4
padding:0
total length:8
reply length:0

RecolorCursor:

Description: Change the foreground and background colors of a cursor.

opcode:96
data:16
padding:0
total length:20
reply length:0

Drawing Graphics

ClearArea:

Description: Clear an area of a window.

opcode:61
data:12
padding:0
total length:16
reply length:0

CopyArea:

Description: Copy an area of a window to another area in the same or a different window. If the source area is obscured, this request will generate a GraphicsExpose event to identify the area of the destination for which the source is not available.

opcode:62
data:24
padding:0
total length:28
reply length:0

CopyPlane:

Description: Copy a single plane of one drawable into any number of planes of another, applying two pixel values to translate the depth of the single plane.

opcode:63
data:28
padding:0
total length:32
reply length:0

FillPoly:

Description: Fill a polygon, without drawing the complete outline.

opcode:69
data:12+4n
    n: number of points
padding:0
total length:16+4n
reply length:0

PolyArc:

Description: Draw one or more arcs, each of which is a partial ellipse aligned with the x and y axis.

opcode:68
data:8+12n
    n: number of ARCs
padding:0
total length:12+12n
reply length:0

PolyFillArc:

Description: Fill one or more arcs, without drawing the arc itself.

opcode:71
data:8+12n
    n: number of ARCs
padding:0
total length:12+12n
reply length:0

PolyFillRectangle:

Description: Fill one or more rectangles, without drawing the entire outline.

opcode:70

data:8+8n
    n: number of RECTANGLEs
padding:0
total length:12+8n
reply length:0

PolyLine:

Description: Draw one or more connected lines.

opcode:65
data:8+4n
    n: number of POINTs
padding:0
total length:12+4n
reply length:0

PolyPoint:

Description: Draw one or more points.

opcode:64
data:8+4n
    n: number of points
padding:0
total length:12+4n
reply length:0

PolyRectangle:

Description: Draw one or more rectangles.

opcode:67
data:8+8n
    n: number of RECTANGLEs
padding:0
total length:12+8n
reply length:0

PolySegment:

Description: Draw one or more disconnected lines.

opcode:66
data:8+8n
    n: number of SEGMENTs
padding:0
total length:12+8n
reply length:0

Events

GetInputFocus:

Description: Return the current keyboard focus window.

opcode:43
data:0
padding:0
total length:4
reply length:32

GetMotionEvents:

Description: Some servers are equipped with a buffer that records the position history of the pointer. This request will return segments of this history for selected time periods.

opcode:39
data:12
padding:0
total length:16
reply length:32+8n
n: number of TIMECOORD

SetInputFocus:

Description: Set a window and its descendants to recieve all keyboard input.

opcode:43
data:0
padding:0
total length:4
reply length:32

Fonts and Text

CloseFont:

Description: Disclaim interest in a particular font. If this is the last client to be using the specified font, the font is unloaded.

opcode:46
data:4
padding:0
total length:8
reply length:0

GetFontPath:

Description: Get the path that the server uses to search for fonts.

opcode:52
data:0
padding:0
total length:4
reply length:32+n+p
n: number of STR
p: padding

ImageText8:

Description: Draw text string in 8-bit font. The bounding rectangle of the string is drawn in the background color from the GC before the text is drawn.

opcode:76
data:12+n
n: length of string
padding:p
total length:16+n+p
reply length:0

ImageText16:

Description: Draw text string in 16-bit font. The bounding rectangle of the string is drawn in the background color from the GC before the text is drawn.

opcode:77
data:12+2n
    n: number of 2-byte characters
padding:p
total length:16+2n+p
reply length:0

ListFonts:

Description: List the fonts available on a server.

opcode:49
data:4+n
    n: length of pattern
padding:p
total length:8+n+p
reply length:32+n+p
        n: number of names

ListFontsWithInfo:

Description: List the fonts available on a server, with information about each font.

opcode:50
data:4+n
    n: length of pattern
padding:p
total length:8+n+p
reply length:32+28+4m+n+p
        m: number of FONTPROPS in properties
        n: length of name in bytes

OpenFont:

Description: Load a font for drawing. If the font has already been loaded, this request simply returns the ID.

opcode:45
data:8+n
padding:p
total length:12+n+p
reply length:0

PolyText8:

Description: Draw text items using 8-bit fonts. Each item can specify a string, a font, and a horizontal offset.

opcode:74
data:12+n
padding:p
total length:16+n+p
reply length:0

PolyText16:

Description: Draw text items using 16-bit fonts. Each item can specify a string, a font, and a horizontal offset.

opcode:75
data:12+n
padding:p

total length:16+n+p
reply length:0

QueryFont:

Description: Get the table of information describing a font and each character in it.

opcode:47
data:1
padding:0
total length:8
reply length:32+28+8n+12m
n: numbber of FONTPROPs in properties
m: number of CHARINFOs

QueryTextExtents:

Description: Calculate the width of a string in a current font.

opcode:48
data:4+2n
n: length of string
padding:p
total length:8+2n+p
reply length:32

SetFontPath:

Description: Set the path that the server uses to search for fonts.

opcode:51
data:4+n
n: path length
padding:p
total length:8+n+p
reply length:0

The Graphics Context

ChangeGC:

Description: Change any or all characteristics of an existing GC.

opcode:56
data:8+4n
n: number of values
padding:0
total length:12+4n
reply length:0

CopyGC:

Description: Copy any or all characteristics of one GC into another.

opcode:57
data:12
padding:0
total length:16
reply length:0

CreateGC:

Description: Create a graphics context, and optionally set any or all of its characteristics. If not set, each characteristic has a reasonable default.

opcode:55
data:12+4n
  n: number of values
padding:0
total length:16+4n
reply length:0

FreeGC:

Description: Free the memory in the server associated with a GC.

opcode:60
data:4
padding:0
total length:8
reply length:0

SetClipRectangles:

Description: Set the clip region of a GC to the union of a set of rectangles.

opcode:59
data:8+8n
  n: number of RECTANGLEs
padding:0
total length:12+8n
reply length:0

SetDashes:

Description: Set the dash pattern for lines, in a more powerful way than is possible using CreateGC or ChangeGC.

opcode:58
data:8+n
  n:length of dashes
padding:p
total length:12+n+p
reply length:0

Images

GetImage:

Description: Place an image from a drawable into a representation in memory.

opcode:73
data:16
padding:0
total length:20
reply length:32+n+p
  n: number of bytes

PutImage:

Description: Dump an image into a drawable.

opcode:72

data:20+n
    n: number of bytes
padding:p
total length:24+n+p
reply length:0

Interclient Communication

ChangeProperty:

Description: Set the value of a property.

opcode:18
data:20+n
    n: number of bytes
padding:p
total length:24+n+p
reply length:0

ConvertSelection:

Description: Request that the owner of a particular selection convert it to a particular format, then send an event informing the requestor of the conversion's success and the name of the property containing the result.

opcode:24
data:20
padding:0
total length:24
reply length:0

DeleteProperty:

Description: Delete the data associated with a particular property on a particular window.

opcode:19
data:8
padding:0
total length:12
reply length:0

GetAtomName:

Description: Get the string name of a property given its ID.

opcode:17
data:4
padding:0
total length:8
reply length:32+n+p
    n: length of name

GetProperty:

Description: Get the value of a property.

opcode:20
data:20
padding:0
total length:24
reply length:32+n+p
    n: number of bytes

GetSelectionOwner:

Description: Get the current owner of a particular selection property.

opcode:23
data:4
padding:0
total length:8
reply length:32

InternAtom:

Description: Get the ID of a property given its string name, and optionally create the ID if no property with the specified name exists.

opcode:16
data:4+n
n: length of name
padding:p
total length:8+n+p
reply length:32

ListHosts:

Description: Obtain a list of hosts having access to a display.

opcode:110
data:0
padding:0
total length:4
reply length:32+n
n: number of hosts

ListProperties:

Description: List the IDs of the current list of properties.

opcode: 21
data:4
padding:0
total length:8
reply length:32+4n
n: number of ATOMs

RotateProperties:

Description: Rotate the values of a list of properties.

opcode:114
data:8+4n
n: number of ATOMs
padding:0
total length:12+4n
reply length:0

SetSelectionOwner:

Description: Set a window as the current owner of a particular selection property.

opcode:22
data:12
padding:0
total length:16

reply length:0

Keyboard and Pointer

AllowEvents:

Description: Release events queued in the server due to grabs with certain parameters.

opcode:35
data:4
padding:0
total length:8
reply length:0

Bell:

Description: Ring the keyboard bell.
opcode:104
data:0
padding:0
total length:4
reply length:0

ChangeActivePointerGrab:

Description: Change the events that are sent to a window that has grabbed the pointer or keyboard.

opcode:30
data:12
padding:0
total length:16
reply length:0

ChangeKeyboardControl:

Description: Change personal preference features of the keyboard such as click and auto-repeat.

opcode:102
data:4+4n
n: number of VALUEs
padding:0
total length:8+4n
reply length:0

ChangeKeyboardMapping:

Description: Change the keyboard mapping seen by all clients.

opcode:100
data:4+4nm
n: keycode-count
m: keysyms-per-keycode
padding:0
total length:8+nm
reply length:0

ChangePointerControl:

Description: Change personal preference features of the pointer, such as acceleration (the ratio of the amount the physical mouse is moved to the amount the cursor moves on the screen).

opcode:105
data:8
padding:0
total length:12
reply length:0

GetKeyboardControl:

Description: Get personal preference features of the keyboard such as click and auto-repeat.

opcode:103
data:0
padding:0
total length:4
reply length:32+20

GetKeyboardMapping:

Description: Return the keyboard mapping seen by all clients.

opcode:101
data:4
padding:0
total length:8
reply length:32+4nm
nm: number of KEYSYMs

GetModifierMapping:

Description: Get the mapping of physical keys to logical modifiers.

opcode:119
data:0
padding:0
total length:4
reply length:32+8n
n: number of keycodes

GetPointerControl:

Description: Return personal preference features of the pointer.

opcode:106
data:0
padding:0
total length:4
reply length:32

GetPointerMapping:

Description: Get the mapping of physical buttons to logical buttons.

opcode:117
data:0
padding:0
total length:4
reply length:32+n+p
n: number of MAPs

GrabButton:

Description: For all pointer events (button presses and motion) occurring while the specified combination of buttons and modifier keys are pressed, declare that these pointer events will be delivered to a particular window regardless of the pointer's location on the screen.

opcode:28
data:20
padding:0
total length:24
reply length:0

GrabKey:

Description: For all keyboard events occurring while the specified combination of buttons and modifier keys are pressed, declare thatthese keyboard events will be delivered to a particular window regardless of the pointer's location on the screen.

opcode:33
data:12
padding:0
total length:16
reply length:0

GrabKeyboard:

Description: Declare that all keyboard events will be delivered to a particular window regardless of the pointer's location on the screen.

opcode:31
data:12
padding:0
total length:16
reply length:32

GrabPointer:

Description: Declare that all pointer events (button presses and motion) will be delivered to a particular window regardless of the pointer's location on the screen.

opcode:26
data:20
padding:0
total length:24
reply length:32

QueryKeymap:

Description: Get the current state of the entire keyboard.

opcode:44
data:0
padding:0
total length:4
reply length:32+8

QueryPointer:

Description: Get the current pointer position.

opcode:38
data:0

padding:0
total length:4
reply length:32

SetModifierMapping:

Description: Set the mapping of physical keys to logical modifiers such as Shift and Control.

opcode:118
data:8n
    n: number of KEYCODEs
padding:0
total length:4+8n
reply length:32

SetPointerMapping:

Description: Set the mapping of physical buttons to logical buttons.

opcode:116
data:n
    n: number of MAPs
padding:p
total length:4+n+p
reply length:32

UngrabButton:

Description: Release a grab on a button.

opcode:29
data:8
padding:0
total length:12
reply length:0

UngrabKey:

Description: Release a grab on a button.

opcode:34
data:8
padding:0
total length:12
reply length:0

UngrabKeyboard:

Description: Release a grab on the keyboard.

opcode:32
data:4
padding:0
total length:8
reply length:0

UngrabPointer:

Description: Release a grab on the pointer.

opcode:27
data:4
padding:0
total length:8

reply length:0

WarpPointer:

Description: Move the pointer.

opcode:41
data:20
padding:0
total length:24
reply length:0

Security

ChangeHosts:

Description: Modify the list of hosts that are allowed access to a server.

opcode:109
data:4+n
n: number of addresses
padding:p
total length:8+n+p
reply length:0

SetAccessControl:

Description: Turn on or off the mechanism that checks the host access list before allowing a connection.

opcode:111
data:0
padding:0
total length:4
reply length:0

WindowCharacteristics

ChangeWindowAttributes:

Description: Set any or all window attributes. For a brief description of the window attributes.

opcode:2
data:8+4n
n: number of VALUEs
padding:0
total length:12+4n
reply length:0

GetGeometry:

Description: Return the position, dimensions, border width, and depth of a window; return the ID of the root window at the top of the window's hierarchy.

opcode:14
data:4
padding:0
total length:8
reply length:32

GetWindowAttributes:

Description: Get the current values of some of the window attributes described for ChangeWindowAttributes; also find out the characteristics of the window that were set when it was created (InputOnly or InputOutput, and visual), whether its colormap is installed and whether it is mapped or viewable.

opcode:3
data:4
padding:0
total length:8
reply length:32+12

Window Manipulation by the Client

CreateWindow:

Description: Create a window.

opcode:1
data:28+4n
n: number of VALUEs
padding:0
total length:32+4n
reply length:0

DestroySubwindows:

Description: Destroy an entire hierarchy of windows.

opcode:5
data:4
padding:0
total length:8
reply length:0

DestroyWindow:

Description: Destroys a window.
opcode:4
data:4
padding:0
total length:8
reply length:0

MapSubwindows:

Description: Map all subwindows of a window.

opcode:9
data:4
padding:0
total length:8
reply length:0

MapWindow:

Description: Mark a window as eligible for display.

opcode:8
data:4
padding:0
total length:8
reply length:0

UnmapSubwindows:

Description: Remove all subwindows of a window, but not the window itself, from the screen.

opcode:11
data:4
padding:0
total length:8
reply length:0

UnmapWindow:

Description: Remove a window and all its subwindows from the screen.

opcode:10
data:4
padding:0
total length:8
reply length:0

Window Manipulation by the Window Manager

ChangeSaveSet:

Description: Add or remove windows from a save-set.

opcode:6
data:4
padding:0
total length:8
reply length:0

CirculateWindow:

Description: Lower the highest window on the screen or raise the lowest one, depending on the parameters of this request.

opcode:13
data:4
padding:0
total length:8
reply length:0

ConfigureWindow:

Description: Allow the window manager to move, resize, change the border width, or change the stacking order of a window.

opcode:12
data:8+4n
    n: number of VALUEs
padding:0
total length:12+4n
reply length:0

QueryTree:

Description: Allow the window manager to get the window IDs of windows it did not create.

opcode:15
data:4
padding:0

total length:8
reply length:32+4n
n: number of WINDOWs in children

ReparentWindow:

Description: Allow the window manager to change the window hierarchy to insert a frame window between each top-level window on the screen and the root window. The window manager can the decorate this frame window with a title for the application, buttons for moving and resizing the window, etc.

opcode:7
data:12
padding:0
total length:16
reply length:0

CreatePixmap:

Description: Create an off-screen drawable.

opcode:53
data:12
padding:0
total length:16
reply length:0

ForceScreenSaver:

Description: Activate or reset the screen saver.

opcode:115
data:0
padding:0
total length:4
reply length:0

FreePixmap:

Description: Free the memory associated with an off-screen drawable.

opcode:54
data:4
padding:0
total length:8
reply length:0

GetScreenSaver:

Description: Get the characteristics of the mechanism the blanks the screen after an idle period.

opcode:108
data:0
padding:0
total length:4
reply length:32

GrabServer:

Description: Initiate a state where requests only from a single client will be acted upon. The server will queue events for other clients and requests made by other clients until the grab is released.

opcode:36
data:0
padding:0
total length:4
reply length:0

KillClient:

Description: After a client exits because of the SetCloseDownMode request, kill the resources that remain alive.

opcode:113
data:4
padding:0
total length:8
reply length:0

ListExtensions:

Description: List the extensions available on the server.

opcode:99
data:0
padding:0
total length:4
reply length:32+n+p
n: length of list of names

NoOperation:

Description: The minimum request, it contains only the opcode and request length.

opcode:127
data:0
padding:0
total length:4
reply length:0

QueryBestSize:

Description: Query the server for the fastest size for creating tiles or stipples or the largest support size for cursors.

opcode:97
data:8
padding:0
total length:12
reply length:32

QueryExtension:

Description: Determine whether a certain extension is available in the server.

opcode:98
data:4+n
n: length of name
padding:p
total length:8+n+p
reply length:32

SendEvent:

Description: Send any type of event to a particular window.

opcode:25
data:40
padding:0
total length:44
reply length:0

SetCloseDownMode:

Description: Determine whether resources created by a client are preserved after the client exits. Normally, they are not, but if the client can reclaim its resources in a later incarnation, the client can use this request.

opcode:112
data:0
padding:0
total length:4
reply length:0

SetScreenSaver:

Description: Set characteristics that blank the screen after an idle period.

opcode:107
data:8
padding:0
total length:12
reply length:0

TranslateCoordinates:

Description: Translate coordinates from a window frame of reference to a screen frame of reference.

opcode:40
data:12
padding:0
total length:16
reply length:32

UngrabServer:

Description: Release the grab on the server, process all outstanding requests, and send all queued events.

opcode:37
data:120
padding:0
total length:4
reply length:0

Appendix D

Listing of X Requets with Replies

This appendix provides a summary of X requests that have replies.

- AllocColor
- GetAtomName
- GetGeometry
- GetImage
- GetKeyboardControl
- GetKeyboardMapping
- GetModifierMapping
- GetMotionEvents
- GetPointerControl
- GetPointerMapping
- GetProperty
- GetScreenSaver
- GetSelectionOwner
- GetWindowAttributes
- GrabKeyboard
- GrabPointer
- InternAtom
- ListExtensions

- ListFonts
- ListHosts
- ListInstalledCollormaps
- ListProperties
- LookupColor
- QueryBestSize
- QueryColors
- QueryExtensions
- QueryFonts
- QueryKeymap
- QueryPointer
- QueryTextExtents
- QueryTree
- SetModifierMapping
- SetPointerMapping
- TranslateCoordinates

Appendix E

X Events

The following is a listing of all event types, what they signify, and any special notes about how they are selected.

ButtonPress, ButtonRelease

A pointer button was press or released. These events include the pointer position and the state of the modifier keys on the keyboard (such as shift).

CirculateNotify, ConfigureNotify, CreateNotify, DestroyNotify, MapNotify, UnmapNotify

This event is generated when one of these requests is actually made on a window. These are used to tell a client when some other client has manipulated a window. Usually this other client is the window manager. All these events and **GravityNotify** and **ReparentNotify** can only be selected together.

CirculateRequest, ConfigureRequest, MapRequest, ResizeRequest

These events are selected by the window manager to enforce its window management policy. Once selected by the window manager, any request to resize, remap, reconfigure, or circulate the window by any client other than the window manager will not be acted on by the server but instead will result in one of these events being sent to the window manager. The window manager then can decide whether to allow, modify, or deny the parameters of the request given in the event and then reissue the request to the server.

ClientMessage

These events, or any other type, can be sent from one client to another using the **SentEvent** request. This event type is for client-specific information.

ColormapNotify

This event tells a client when a colormap has been modified or when it is installed or unistalled from the hardware colormap.

EnterNotify, LeaveNotify

The pointer entered or left a window. These events are generated even for each window not visible on the screen that is an ancestor of the orgin or destination window.

Expose

Expose envents signify that a section of a window has become visible and should be drawn by the client.

FocusIn, FocusOut

The keyboard focus window has been changed. Like **EnterNotify** and **LeaveNotify**, these events can be generated even for invisible windows.

GraphicsExpose, NoExpose

GraphicsExpose and **NoExpose** are generated only as the result of **CopyArea** and **CopyPlane** requests. If the source area specified in either request is unavailable, one or more GraphicsExpose events are generated, and they specify the area of destination that could not be drawn. If the source area was available, a single NoExpose event is generated. GraphicsExpose and NoExpose events are not selected normaly but instead are turned on or off by a member of the graphics context.

GravityNotify

This event notifies a client when a window has been moved in relation to its parent because of its window gravity attribute. This window attribute is designed to alllow automatic positioning of subwindows in certain simple cases when the parent is resized.

KeymapNotify

Always following EnterNotify or FocusIn, KeymapNotify gives the complete status of all the keys on the keyboard.

KeyPress, KeyRelease

A keyboard key was pressed or released. Even the Shift and Control keys generate these events. Thereis no way to select just the events on a particular keys.

MappingNotify

The pointer moved. MotionNotify events can be selected such that they are deliverd only when certain button are pressed or regardless of the pointer buttons.

PropertyNotify

This event is issued whenever a client changes or deletes a propert, even if the change is to replace data with identical data.

SelectionClear, SelectionNotify, SelectionRequest

These three events are used in the selection method of communicating between clients. These events are not selected, but are always generated by the requests involved in the selection procedures.

VisibilityNotify

This event is generated when a window changes from fully obscured, partially obscured, or unobscured to any other of these states and also when this window becomes viewable.

This Page Intentionally Left Blank

Appendix F

X Errors

The following error codes can be return by the various requests.

Access:

An attempt is made to grab a key/button combination already grabbed by another client.

An attempt is made to free a color map entry not allocated by the client.

An attempt is made to store into a read-only or an unallocated colormap entry.

An attempt is made to modify the access control list from other than the local host (or otherwise authorized client).

An attempt is made to select an event type that only one client can select at a time when another client has already selected it.

Alloc:

The server failed to allocate the requested resource.

Atom:

A value for an ATOM argument does not name a defined ATOM.

Colormap:

A value for a COLORMAP argument does not name a defined COLORMAP.

Cursor:

A value for a CURSOR argument does not name a defined CURSOR.

Drawable:

A value for a DRAWABLE argument does not name a defined WINDOW or PIX-MAP.

Font:

A value for a FONT argument does not name a define FONT.

A value for a FONTABLE argument does not name a defined FONT or a defined GCONTEXT.

GCONTEXT:

A value for a GCONTEXT argument does not name a defined GCONTEXT.

Implementation:

The server does not implement some aspect of the request. A server that generates this error for a core request is deficient.

Length:

The length of a request is shorter or longer than that required to minimally contain the arguments.

The length of a request exceeds the maximum length accepted by the server.

Match:

An InputOnly window is used as a drawable.

In a graphic request, the GCONTEXT argument does not have the same root and depth as the destination DRAWABLE argument.

Some argument has the correct type and range, but it fails to match in some other way required by the request.

Name:

A font or color of the specified name does not exist.

Pixmap:

A value for a PIXMAP argument does not name a defined PIXMAP.

Request:

The major or minor opcode does not specify a valid request.

Value:

Some numeric value falls outside the range of values accepted by the request. Unless a specified range is specified for an argument, the full range defined by the argument's type is accepted.

Window:

A value for a WINDOW argument does not name a defined WINDOW.