Profiling the X Protocol

John Danskin * Pat Hanrahan

May 16, 1994

Abstract

This paper describes a system for tracing the X Window System Protocol. We have analyzed these traces by breaking the protocol down into discrete instructions or messages, and analyzing the frequency and network resource requirements of these messages. This analysis has resulted in preliminary results in 4 areas: the relative importance of X messages, how X usage changes during different application stages, differing application idioms for accomplishing similar tasks, and the size distributions of X messages and groups of X messages. Some of these results have architectural implications for network protocols and implementations.

1 Introduction

Window systems are a major system component of modern operating systems. With windows, a user can display multiple work sessions at once on a single computer display. One of the most powerful attributes a window system can have is network transparency. With a network transparent window system, the user can run window programs on any machine on the network, interacting with them as if they were local.

The most widespread network transparent window system is the X Window System version 11 [7], commonly known as X11 or just X. The part of X that runs on the machine in front of the user is known as the X server. The X server draws on the display and handles input devices such as the mouse and the keyboard. The user programs that connect to the X server are called clients. Probably the most commonly run X client is xterm, the X terminal emulator. Other popular clients include clocks, text editors (WYSIWYG and otherwise), and drawing programs.

Most X connections are either local (no network involved), or over an ethernet connection using TCP/IP [16, 14] and Berkeley sockets. Both of these environments usually provide adequate bandwidth for X [6]. However, X is also used in environments where

*Funded by an AT&T PhD fellowship and a generous grant from Apple Computer

bandwidth is precious, such as over the internet and across public telephone lines using fast modems and a serial line network protocol [5]. Upcoming improvements in telecommunications such as ISDN and improved cable bandwidth will open up more opportunities for network-based window systems and graphics protocols, many over slow links. Measurements taken now will help us adapt graphics protocols to these slow links.

This paper presents a method for capturing and analyzing the X protocol, as well as our results. Figure 1 is an example of a display of a trace which we gathered, showing the time, size, and type of X messages processed during a three minute period. We will present a taxonomy of X messages. This taxonomy will allow a discussion of the relative importance of groups of X messages such as drawing, imaging, and windowing operators. We continue with a discussion of how X applications go through different stages, and how this is more important for interactive applications than for batch applications. After adding the dimension of time, we zoom in to the level of individual X messages, examining X application idioms for implementing keystroke echoing. Finally, we will examine the size distributions of X messages and groups of X messages. To conclude, we will point out some of the implications of these results.

2 Goals for Gathering Data

We wanted to gather traces of network traffic, to actually capture an entire protocol stream. Once we'd captured the stream, we had to save it for later analysis. For the sake of accuracy, the logging process should interfere as little as possible with the content and timing of the data processed.

A unique aspect of tracing a window system running an interactive application is that there is a human in the loop. This has ramifications for our instrumentation: unlike most computer programs, humans behave in a time dependent way. If the tracing system slows down the system too much, the user will act differently. Visual feedback delays can adversely affect hand to eye coordination just as audio feedback delays can make it difficult to talk. For instance, handwriting performance falls off dramatically with visual feedback delays of 150ms [11]. If our instrumentation adds a latency which is a large fraction of 100ms, then many interactive operations (such as keystroke echo) which previously appeared instantaneous will take a noticable amount of time [4]. This would distract the user, affecting typing performance and the content of the trace. The effect of excessive latency would be even more severe for users using drawing programs where mouse tracking has to appear instantaneous. Throughput is also important because reducing throughput can result in increased perceived latency, adversely affecting the performance of the subject.

Another goal was to avoid changing the X server and the X clients. Changing the clients would be a big problem because we wanted to get traces of users in their day to day environment. Some people run dozens of X clients on many machines, including

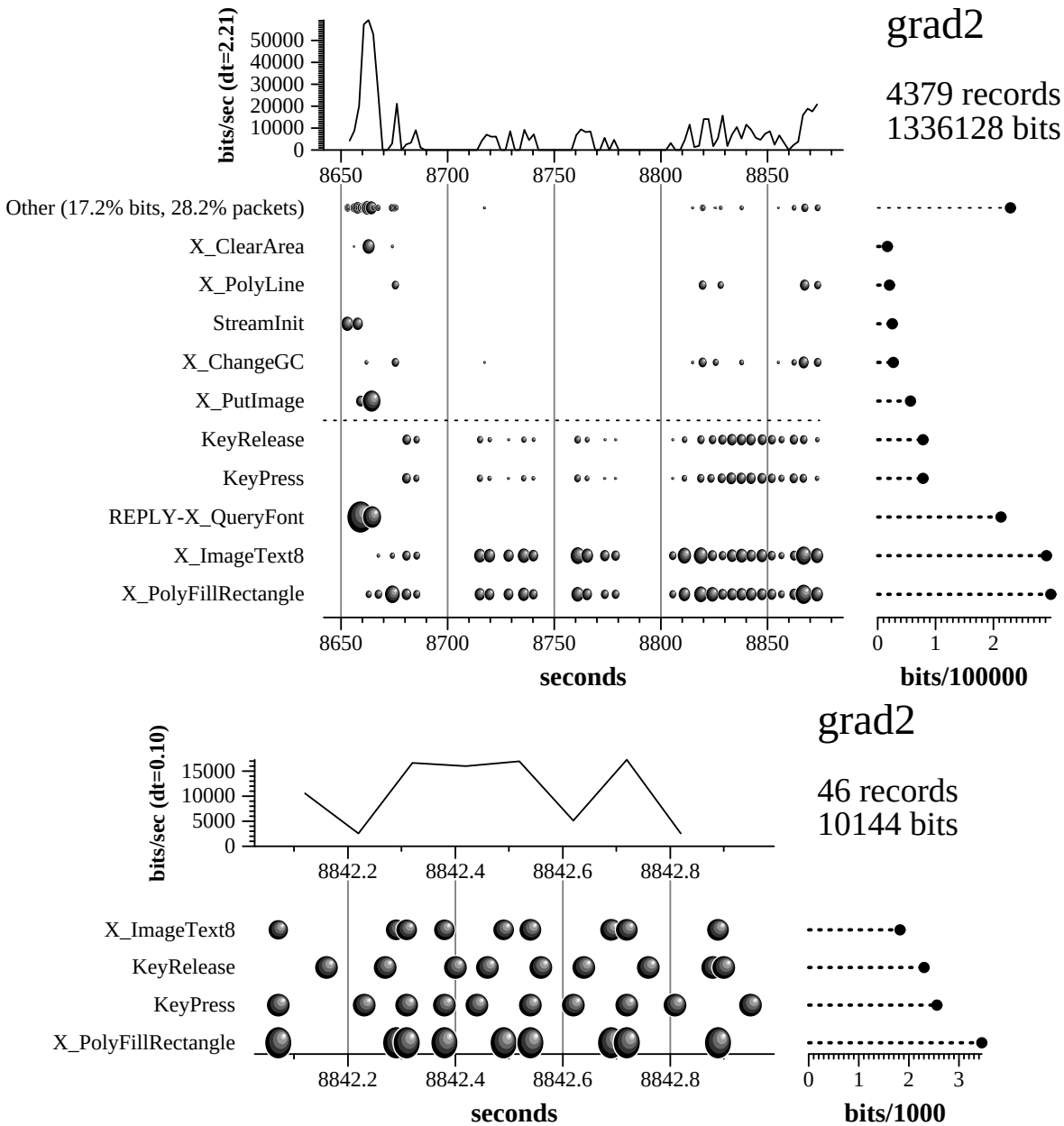


Figure 1:
These graphs are excerpts from the grad2 trace. The top graph covers almost four minutes, while the bottom graph is a one second detail. The X axis along the bottom of the central area in each graph is seconds since session startup. The Y axis in the main area is marked off by message type. The volume of spheres in the data area is proportional to the size of the messages which they represent. Spheres which are vertically stretched represent multiple nearly simultaneous messages, other spheres represent single messages. The dash-dot graph to the right shows the total number of bits for each type of packet. The line graph at the top shows the number of bits per second as a function of time. The integration interval is listed in the Y-axis legend.
At the left of the top graph we see an application starting up. It creates windows, allocates colors, and queries fonts. The instantaneous network load is comparatively high. After about a minute, we settle down to simple typing (keypress, polyfill rectangle, image text, keyrelease, etc.). Polyfill rectangle is used to clear the area behind each character. From the bottom graph, we can tell that the user is pressing keys almost every 100ms, which is extremely fast.

machines on which only they have privileges. It would have been hard to install custom versions of all of the X clients on all of those machines. It would be easier to change the X server, but then our subjects would have had to use a special machine for a few days, since we didn't have source for the X server in all the X terminals and computers which our subjects were using. To avoid modifying the X server and inconveniencing our subjects, we decided to extract the data directly from the communication stream between client and server.

The operating system and file system communities have a lot of experience gathering and analyzing traces, so we decided to look there for ideas. One non-intrusive way to capture a trace would be to eavesdrop on the ethernet capturing all relevant packets and reassembling the protocol stream from the captured data. As long as the data could be logged locally (as opposed to sending it over the network), tracing would have no practical effect on the measured stream. Blaze [3] implemented an eavesdropping system like this for NFS, which uses the UDP datagram protocol [15] for transport. We decided not to implement a passive eavesdropping system for X because we are building our instrumentation as a component of a protocol transformation system, capable of compressing and decompressing the X protocol. In another situation, passive listening would be a very promising technique for making time sensitive measurements.

An approach which would have helped control the size of our traces and the impact of our instrumentation was taken by Osterhout el al [12] and Baker et al [2] who gathered abstracted traces of the UNIX filesystem by recording just open, close, and file pointer repositioning operations, omitting reads and writes. Abstracting their traces allowed much longer traces and reduced the impact of instrumentation tremendously. We didn't end up abstracting our traces because although they were large, they were still manageable. In addition, we needed complete traces for future work in compression.

When gathering complete X traces, privacy is an issue. The X protocol stream includes all of the characters typed by the user and all the characters seen by the user, including passwords, love letters, grades and so on. Eliminating this data from X traces would make it possible to redistribute them and would eliminate the risk of inadvertent or malicious disclosure of confidential information. For some applications it might be possible to replace all of the text with random characters which would provide a much greater degree of privacy. Unfortunately, this technique would perturb the data unacceptably for our ultimate application: network compression. Our subjects were warned that their sessions were not strictly private, but that we would do our best to keep their contents secure.

The tracing technique which we adopted was derived from work by Droms and Dyksen [6]. They realized that if X clients and the X server converse using Berkeley sockets and TCP/IP, it would be easy to interpose a program between client and server which forwarded traffic in both directions while keeping a complete log. Complete data integrity is guaranteed at the price of a little performance. Droms and Dyksen used their traces to characterize the load on an ethernet due to an X session. Another program which uses

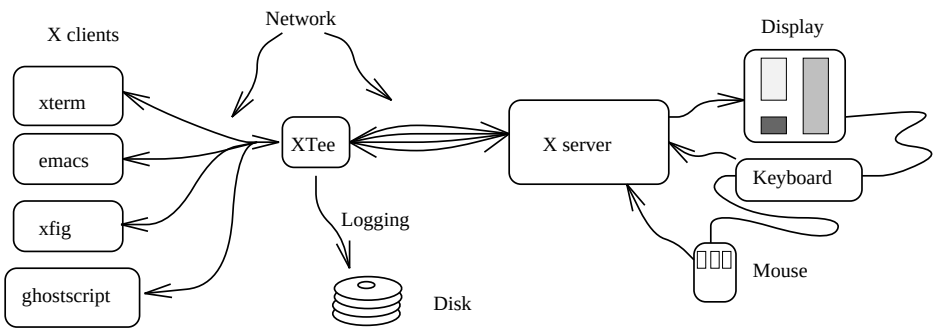


Figure 2: X clients attach to X server through logging process *XTee*

the interposition technique for snooping on X traffic is *xscope* by Peterson [13]. We used Peterson's table lookup technique for matching X queries with X replies.

3 Gathering Data

In order to understand the tracing process, we should discuss the client-server connection. X clients communicate with the X server using a full duplex reliable byte stream protocol, usually TCP/IP and Berkeley sockets. Each client establishes a single connection which is used for all of the client's windows. Communication in X is asynchronous; that is, the client is not suspended while waiting for notification of request completion. The designers of the X protocol went to great lengths to remove unnecessary round trip delays for performance reasons [7]. Even with commands that do require replies, it is possible to send several requests, and then receive several replies, with only one round trip delay. This requires a little bookkeeping on the part of the client, and on the part of any filter that attempts to match requests to replies as ours does.

On UNIX systems X clients determine the address and method for connecting to the X server from an environment variable. The most common type of address is a machine name followed by a colon followed by a display number, for example "foo:0". To the X library, this address means to open a TCP socket to a machine named foo using port $(6000 + displayNumber)$. Usually, $displayNumber$ is 0. Most X servers only have one display and hence are not listening on ports 6001, 6002 etc.. It is easy to write a third program *XTee* which waits for new connections on one of these unused ports. When *XTee* detects a new connection A, *XTee* opens a new connection B to the X server. Input on A is forwarded to B, and input on B is forwarded to A. Input from both connections is logged (See Figure 2). To an X client, *XTee* appears to be the X server, while to the X server, *XTee* appears to be the X client.

Internally *XTee* parses each client-to-server or server-to-client stream into separate messages, and then multiplexes the packetized half duplex streams into one half duplex stream going in each direction. Rather than annotate each message in the merged stream

with a stream identifier, we pass a "ChangeStream" token down the merged stream whenever the current stream changes. New streams are identified with a "NewStream" token, and streams whose sockets have closed are shut down with "CloseStream" tokens. These three extra tokens provide enough information for demultiplexing and stream management. For details of a similar multiplexed X protocol see [5]. Messages in these two half duplex streams are logged into two files (client-to-server and server-to-client) with their length, a global sequence number, a unique timestamp, and a magic number which was used to increase programmer confidence in correct parsing of the logged data. The three stream management tokens are included in the logged data so that the streams can be isolated in later processing.

We ran *XTee* on a DECpc 450ST (50Mhz 486) running CMU Mach version 2.6, except for one trace (*xbattle1*) where we ran *XTee* on a 100Mhz DEC Alpha running OSF1. The X servers used were NCD X terminals, except for the grad2 and grad4 traces, where the X servers were SGI Personal Irises. Most application programs were run on DecStation 5000 model 240s.

To test the impact on performance on tracing, we ran *xbattle* (a game which continuously updates it's display) in two different scenarios:

- competing for cycles with *XTee* on a 50 Mhz 486 PC.
- on the 486 PC, with *XTee* executing on a 100 Mhz DEC Alpha.

Although the games seemed to play the same in both cases, and we played for the same length of time in both cases, the xbattle1 trace is twice as long, indicating twice the update bandwidth.

We were concerned about our tracing system's effect on performance because poor performance will affect our users and thus our traces. In order to determine the effect of our tracing system on performance, we ran *x11perf* (an X benchmark program) on a DEC DS5000 model 240 with display on an NCD X terminal in three configurations:

1. Connecting directly (over the ethernet) to the remote X terminal.
2. Connecting through *XTee* running on a 100 Mhz DEC Alpha to the X terminal.
3. Connecting through *XTee* running on a 50Mhz 486PC to the X terminal.

Table 1 shows the X performance achieved in the three different configurations. Large rectangles show no degradation, even with logging running on the slower 486, which is what we expect because the X server has to update many pixels for each rectangle (250,000), and because rectangles can be concisely specified in X: about 8 bytes each. Large rectangles are server bound, not network bound, even when we degrade the network by introducing a filter.

x11perf	No Tracing		*XTee* on DEC Alpha			*XTee* on 486		
command	prims/s	bytes/s	prims/s	bytes/s	degradation	prims/s	bytes/s	degradation
bigsrect500	41	328	41	328	0%	41	328	0%
f9text	33400	66800	32100	64200	4%	33100	66200	0%
f9itext	61500	123000	59300	118600	4%	54400	108800	12%
putimage100	228	285000	216	270000	5%	75	93750	67%
dot	78600	314400	79100	316400	0%	22600	90400	71%
noop	17800	71200	17500	70000	2%	4320	17280	76%

command	explanation
bigsrect500	500x500 stippled rectangle, 161x145 stipple pattern. A stipple pattern is like a silk screen. Pixels are only updated where the stipple pattern is set.
f9text	Characters in 60-char line (9x15). Background is untouched.
f9itext	Characters in 60-char line (9x15). Background is cleared before character is drawn.
putimage100	PutImage 100x100 square.
dot	Dot.
noop	X protocol NoOperation.

Table 1: Impact on X throuput due to logging with *XTee*

The top table shows X performance as measured by the *x11perf* benchmark program for three test configurations: without tracing, with the trace process running on a DEC Alpha, and with the trace process running on a 486 PC. We show the number of primitives per second achieved, a lower bound on the number of bytes per second implied by this number, and for the tracing configurations, the degradation relative to the untraced configuration ($1 - slow/fast$).

The bottom table is a legend explaining the command names. The names are the same as the *x11perf* commands used to generate the data.

Examining the table, we see that the highest network bandwidth achieved is 316KB/s, except that *XTee* actually doubles network traffic, which means that we actually achieved 630Kb/s over the ethernet. This implies that we are always server limited in the case where there is no tracing. For these tests, the DEC Alpha can always keep up with the X server. The 486 seems to keep up with the X server until the data rate reaches 90-110KB/s, which is as fast as the 486 can go. The exception to this rule is the *noop* test, where the 486 only reaches about 17KB/s. We believe that this is because *XTee* has some per message overhead, and *noop*s come singly. All of the other primitives tested come in large messages (*PolyRectangle, PolyText*, etc.) with many primitives per message,

Configuration	latency	increase
No Tracing	3.6 ms.	0%
XTee on DEC Alpha	4.8 ms.	33%
XTee on 486	5.3 ms.	47%

Table 2: *XTee* latency

Impact on X round trip latency due to logging with *XTee*

allowing us to amortize our overhead over many bytes. X does not support a *PolyNoop* command.

When X11 was designed in 1986, UNIX workstations connected via ethernet typically achieved data rates of from 62.5 K bytes per second to 125 K bytes per second, and peak character display rates were from 5K to 30K characters per second [20]. Our slowest (and major) tracing configuration has lower bandwidth than our untraced setup, but would possibly have been one of the fastest machines available in 1986. It is important to point out, however, that our traces show that most applications spend very little time sending data at rates approaching 100KB/s. In most cases, round trip latency is a more important measure of interactive performance.

Table 2 shows the X round-trip latency in the three configurations (as measured by the *x11perf* benchmark program). In both cases, introducing the filter significantly increases the latency, but the latency never quite doubles as we expected. We conjecture that this means that the 486 and the Alpha handle interrupts more quickly than the X terminal, although we have not investigated further. Note that the new latencies are well below the 100 millisecond human threshold and that it would take many round trips in either configuration before a human could see a difference attributable purely to latency (most user-X interactions are handled in a single round trip).

The X round-trip latency was measured in 1986 in [20]. There the X round trip latency was reported as 15 to 30 milliseconds. We now measure X round trip latency for a traced connection at just over 5 milliseconds. Our current system with tracing installed is substantially faster than the systems available to the X designers in 1986.

4 Traces

We took a number of traces of applications that we thought would be representative: two 2D drawing programs, a postscript previewing program, an X benchmark, some games, and five graduate students doing their daily work. These traces are summarized in Table 3

We chose 2D drawing programs because they are based more on geometry than text and we wanted to characterize geometry based applications. We used two programs hoping that if their traces were very different, we might learn something. The user in both cases was one of the authors, who drew relatively simple pictures consisting of lines and polygons, and then recursively duplicated these pictures several times to build up a large amount of geometry.

The postscript previewer was chosen as an imaging application which we often use in our daily work. We traced the previewer generating a page of text, which it does using image operators.

An obvious trace for a trace suite is a benchmarking program, if only to demonstrate an atypical load. The benchmark we used, *x11perf*, is included in the X distribution.

While *x11perf* can provide a comprehensive performance report after running for hours, we didn't have enough disk space to log a comprehensive trace, so we generated a relatively short trace, only checking the performance of dots.

A few years ago, it might have been considered frivolous to include game traces in a suite like this, but very large companies are now seriously investigating the possibility of charging people to play games over the telephone and cable networks [1]. Probably X will not be the major graphics protocol for graphics games, but X traces may still provide a useful characterization of the traffic. Xtetris animates a simple arrangement of falling rectangles while listening for keyboard and mouse button events. Xbattle animates a map with armies on it, spending most of it's bandwidth redrawing the map grid with lines, and army pieces with rectangles.

Finally, we included traces of graduate students working. We gathered real life traces because they seemed to us to be the most representative of an actual network load. We used graduate students because they were available and because they generate most of the network traffic around here. It turns out that graduate students spend most of their X bandwidth on text. This is why we did not specifically include a text trace.

There are a number of important categories of traces which we've left out: 3D, ECAD (for instance, vlsi design), and color imaging, to name a few. 3D graphics isn't directly supported by X, but it is supported by the PEX extension to X. ECAD is an important native X application. Color imaging is another important X application supported by X and enhanced by imaging extensions. We expect that ECAD applications are similar to 2D drawing programs in their use of X, but that 3D and color imaging applications are completely different, particularly since they use extensions to the core X protocol.

5 X Protocol Activity

In the following sections we will talk about categories of X messages; however, before we can talk about these catagories of X messages, we need to develop a classification scheme or taxonomy defining them. We've built our X message taxonomy as a two level hierarchy, using function as the criteria for making distinctions between messages. Our complete taxonomy is reproduced in Table 4. The 5 top level categories of X messages are summarized below:

1. GEOMETRY includes line polygon, arc and point drawing primitives.

2. IMAGE includes cursor manipulation and image manipulation primitives.

3. INPUT includes events generated directly by the user, such as KeyPress, as well as primitives often used in the inner loop of mouse tracking operations such as X_QueryPointer and X_TranslateCoords.

Trace	color?	Size	Duration	Description
idraw	bw	4.5 Mb	11 min	A 2D drawing program, making a very large model by duplicating a small one repeatedly.
xfig	bw	5.6 Mb	17 min	A 2D drawing program, making a very large model by duplicating a small one repeatedly.
ghostscript	bw	30 Kb	10 sec	A postscript previewer, previewing a single page
x11perf	bw	1.9 Mb	33 sec	An X benchmark program configured to test point drawing performance.
xbattle	bw	7 Mb	14 min	A multiplayer interactive wargame.
xbattle1	bw	15 Mb	14 min	The same game with logging executed on a DEC Alpha at 100 Mhz instead of on a 50 Mhz 486
xtetris	bw	350 Kb	11 min	The classic game
grad1	bw	13 Mb	6 hrs	A graduate student, preparing a paper and previewing graphs.
grad2	color	14 Mb	45 hrs	A fast typing graduate student.
grad3	bw	2.4 Mb	54 min	A graduate student doing software development.
grad4	color	21 Mb	11 hrs	A graduate student, making significant use of a dvi file previewer
grad5	bw	10 Mb	110 hrs	A graduate student who reads mail and writes papers.

Table 3:

4. MISC includes access control, the bell, errors, X extensions, and the X interprocess communication facilities such as cut and paste.

5. TEXT includes font primitives and character drawing primitives. Note that when text is actually drawn, most applications use a rectangle to clear the area behind the new character being, so that text intensive applications will usually see about 50% text and 50% geometry.

6. WINDOW includes colormaps, input focus manipulation, keyboard maps, graphics state changes, X's database facility, and window movement and resizing.

Analyzing our traces lead to preliminary results in four areas:

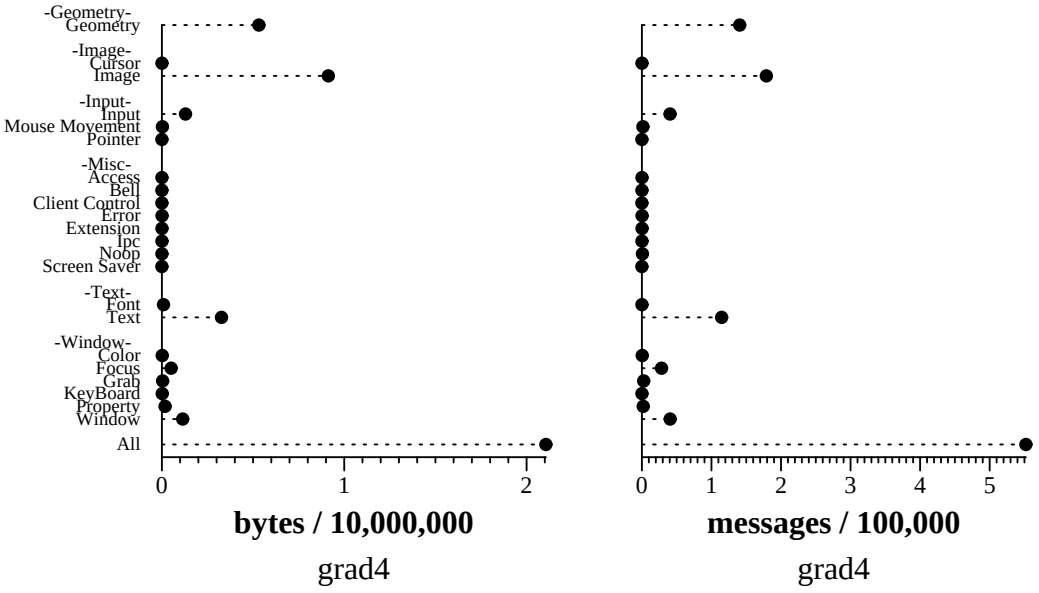
- The relative importance of X messages.
- How X usage changes during application stages.
- Short sequences of messages representing higher-level application idioms.
- Size distribution of X messages and groups (trains) of X messages.

5.1 Relative Importance of X Messages

We began with two questions: which individual messages are important, and which groups of messages are important. We've defined importance in two ways: frequency,

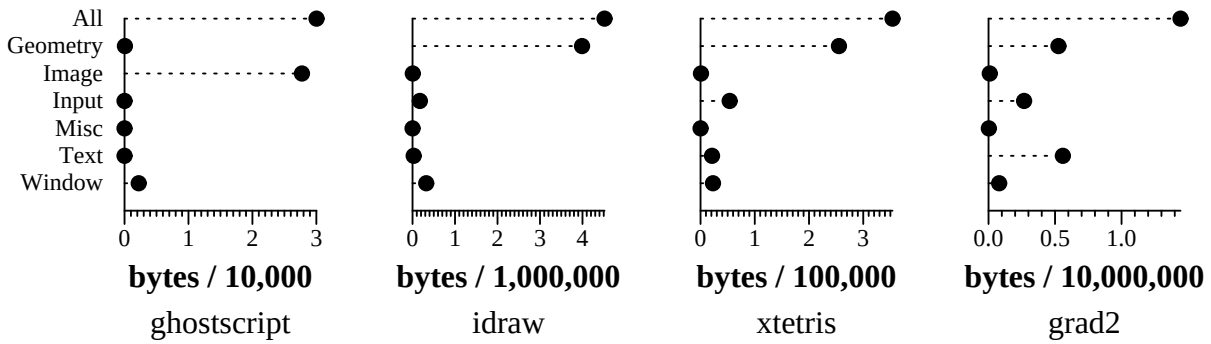
Class	Message	Class	Message	Class	Message
Geometry			X_GetSelectionOwner		X_QueryKeymap
·Geometry	X_ClearArea		X_SendEvent		X_SetModifierMapping
	X_FillPoly		X_SetSelectionOwner	·Property	PropertyNotify
	X_PolyArc	·Noop	X_NoOperation		X_ChangeProperty
	X_PolyFillArc	·Screen Saver	X_ForceScreenSaver		X_DeleteProperty
	X_PolyFillRectangle		X_GetScreenSaver		X_GetAtomName
	X_PolyLine		X_SetScreenSaver		X_GetProperty
	X_PolyPoint	**Text**			X_InternAtom
	X_PolyRectangle	·Font	X_CloseFont		X_ListProperties
	X_PolySegment		X_GetFontPath		X_RotateProperties
Image			X_ListFonts	·Window	CirculateNotify
·Cursor	X_CreateCursor		X_ListFontsWithInfo		CirculateRequest
	X_CreateGlyphCursor		X_OpenFont		ConfigureNotify
	X_FreeCursor		X_QueryFont		ConfigureRequest
	X_QueryBestSize		X_QueryTextExtents		CreateNotify
	X_RecolorCursor		X_SetFontPath		DestroyNotify
·Image	X_CopyArea	·Text	X_ImageText16		Expose
	X_CopyPlane		X_ImageText8		GraphicsExpose
	X_GetImage		X_PolyText16		GravityNotify
	X_PutImage		X_PolyText8		KeymapNotify
Input		**Window**			MapNotify
·Input	ButtonPress	·Color	X_AllocColor		MapRequest
	ButtonRelease		X_AllocColorCells		MappingNotify
	EnterNotify		X_AllocColorPlanes		NoExpose
	FocusIn		X_AllocNamedColor		ReparentNotify
	FocusOut		X_CopyColormapAndFree		ResizeRequest
	KeyPress		X_CreateColormap		StreamInit
	KeyRelease		X_FreeColormap		UnmapNotify
	LeaveNotify		X_FreeColors		VisibilityNotify
	MotionNotify		X_InstallColormap		X_ChangeGC
·Mouse Movement	X_GetMotionEvents		X_ListInstalledColormaps		X_ChangeSaveSet
	X_QueryPointer		X_LookupColor		X_ChangeWindowAttributes
	X_TranslateCoords		X_QueryColors		X_CirculateWindow
·Pointer	X_ChangePointerControl		X_StoreColors		X_ConfigureWindow
	X_GetPointerControl		X_StoreNamedColor		X_CopyGC
	X_GetPointerMapping		X_UninstallColormap		X_CreateGC
	X_SetPointerMapping	·Focus	X_GetInputFocus		X_CreatePixmap
	X_WarpPointer		X_SetInputFocus		X_CreateWindow
Misc		·Grab	X_AllowEvents		X_DestroySubwindows
·Access	X_ChangeHosts		X_ChangeActivePointerGrab		X_DestroyWindow
	X_ListHosts		X_GrabButton		X_FreeGC
	X_SetAccessControl		X_GrabKey		X_FreePixmap
·Bell	X_Bell		X_GrabKeyboard		X_GetGeometry
·Client Control	X_KillClient		X_GrabPointer		X_GetWindowAttributes
	X_SetCloseDownMode		X_GrabServer		X_MapSubwindows
·Error	Error		X_UngrabButton		X_MapWindow
·Extension	Extension-request		X_UngrabKey		X_QueryTree
	X_ListExtensions		X_UngrabKeyboard		X_ReparentWindow
	X_QueryExtension		X_UngrabPointer		X_SetClipRectangles
·Ipc	ClientMessage		X_UngrabServer		X_SetDashes
	ColormapNotify	·KeyBoard	X_ChangeKeyboardControl		X_UnmapSubwindows
	SelectionClear		X_ChangeKeyboardMapping		X_UnmapWindow
	SelectionNotify		X_GetKeyboardControl		
	SelectionRequest		X_GetKeyboardMapping		
	X_ConvertSelection		X_GetModifierMapping		

Table 4: X messages are categorized functionally. We have a two level hierarchy. Top level categories are boxed. Second level categories begin with a · and are on the same line with a message. Messages beginning with "X_" are client requests or queries. Other messages are server events (such as "KeyPress"), "StreamInit" or "Error". "StreamInit" is the connection startup sequence. Server replies to queries are categorized with the queries, and are therefore not listed.



Detailed Application Signatures (bytes and messages), trace=grad4

Figure 3: Messages are categorized according to Table 4.



Application Signatures (bytes)

Figure 4: Messages are categorized according to the high level categories in Table 4. Magnitudes correspond to total bytes in each category.
Note that graphs in this series may have different horizontal scales

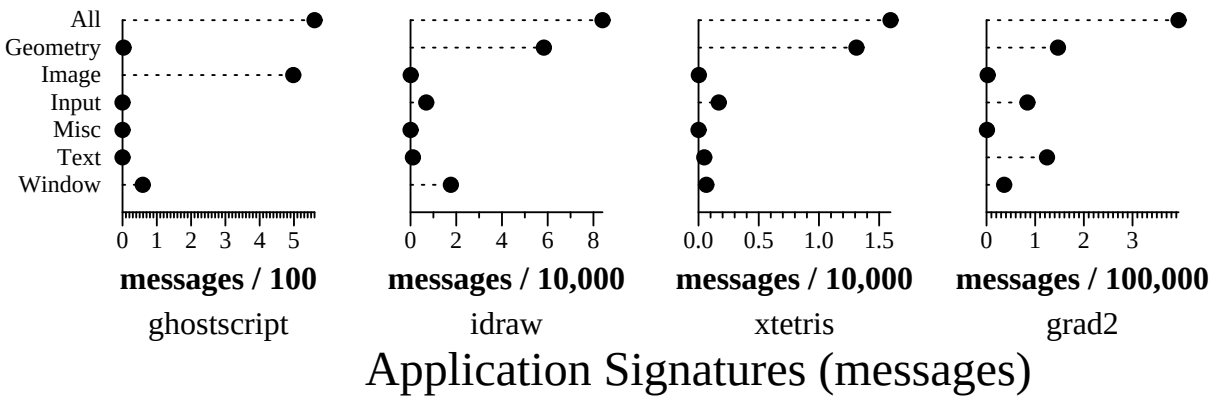


Figure 5: Messages are categorized according to the high level categories in Table 4. Magnitudes correspond to total message count in each category.
Note that graphs in this series may have different horizontal scales

and total bytes. As we see it, frequency is more specific to the instruction set; total bytes is a more general measure. Analogously, in a complex computer instruction set, some string instructions may occur infrequently, and yet account for a large percentage of total cycles. We have included graphs of both measures for most of our studies.

In Figure 3, taken from the grad4 trace, we see that image operations use more bytes than any other category. This is because the subject was using a dvi file previewer (to view TEX output), and X's support for high quality text is such that document previewers usually use the X image operators rather than X's text primitives. Geometry is the next most important category in this trace, although it is likely that about half of the geometry was rectangles clearing letter spaces for text. We also see activity in the window category, which encompasses window manipulation, graphics state changes, input, and focus. Focus events are usually seen when the mouse enters or leaves a window. None of the other 17 categories see any appreciable use.

Comparing bytes per message type with the number of messages of each type in Figure 3, we see that in this trace, image operations were larger on average than the other packet types, since image operations are weighted more heavily in the byte counting graph than in the message counting graph. We can compute from the figures that the average message was about 38 bytes, while the average image operation was about 53 bytes (size distributions can be derived from Figure 6, discussed below). These image operations are actually surprisingly small until we realize that many document previewers (*ghostscript* for example) operate in monochrome, send images to X a single scanline at a time, and only send the span of the scanline updated.

Comparing the detailed breakdown of message types in Figure 3 to the high level breakdown also available in the same graph, little information is lost because most of the categories have so little traffic, and active low level categories are always grouped in different high level categories. This is part of a pattern of X command use: many

trace		N				
		4	8	16	32	64
ghostscript	messages	94%	97%	99%	100%	100%
	bytes	96%	98%	99%	100%	100%
idraw	messages	65%	79%	94%	99%	100%
	bytes	86%	92%	97%	99%	100%
xtetris	messages	91%	98%	99%	99%	100%
	bytes	85%	96%	98%	99%	100%
grad2	messages	82%	91%	94%	98%	99%
	bytes	86%	91%	95%	98%	99%

Table 5: Percentage of messages or bytes accounted for by the top N commands in each trace. Commands are sorted according to command popularity for the "messages" row, and aggregate bytes for the "bytes" row.

commands are used, but only a few are used frequently. Table 5 quantifies this effect for several of our traces.

5.2 Application Stages

Application profiles lose time dependant data, masking temporal changes in running applications. If optimization is applied only according to the static profile, then total run-time may be decreased, but the interactive user may not perceive a positive effect. Frequent operations which already appeared instantaneous may have been improved, leaving irritatingly slow but infrequent operations untouched.

All of the application signatures in Figure 4 have one thing in common; most of the data is spent on directly useful operations: geometry, text, or imaging, rather than IPC, window manipulations, database management or other overhead. This is encouraging, but it doesn't show the whole picture. In Figure 1, the top graph shows the first four minutes or so in the life of an application (xterm). The bottom graph shows a detail from the top graph of about 1 second. In the bottom graph, the user is typing, and only 4 primitives are in use. Anyone wishing to optimize this steady state mode of xterm would know exactly where to start working.

However, looking back to the top graph in Figure 1, almost 20 seconds are spent in application startup. In this mode, otherwise infrequently used commands are used exclusively, and network use is much higher than during normal operation. Perceptually, this startup time is very important. A long startup time can contribute to a perception of sluggishness, even if the startup time is a very small percentage of the total time spent with the application. This sort of problem highlights a fundamental difference between interactive programs and batch mode programs like benchmarks. When optimizing batch mode programs, the object of optimization is to minimize total run time. When optimizing interactive programs, the goal should be to minimize perceived run time: operations should be weighted by their importance to the user before optimization. Infrequent events

such as application startup can be very important to the user; this tends to spread out the area of optimization and makes substantial improvements in perceived performance more difficult to achieve.

Graphs like those of Figure 1 can be very useful for analyzing interactive applications for optimization. Instead of summarizing the whole run-time, an interval of interest is displayed. Once a single interaction is isolated, the interaction can be optimized as if it were a batch mode program, with this graph supplying much of the necessary information.

5.3 Application Idioms

Figure 4 shows differences among the ways applications and groups of applications use X. The *ghostscript* document previewer uses mostly image operations. The drawing programs *idraw* and *xfig* use mostly geometry. The user session traces were mixed between geometry, imaging, and text.

Something we didn't expect, is that on a microscopic level, X applications do things very differently. For example, we examined three applications echoing keystrokes at the end of a line:

1. xwsh (a terminal emulator) clears from the end of the text on the current line to the right edge of the window using X_PolyFillRectangle, draws all of the text for the current line, including the new character using X_ImageText8, and draws a cursor using X_PolyFillRectangle.

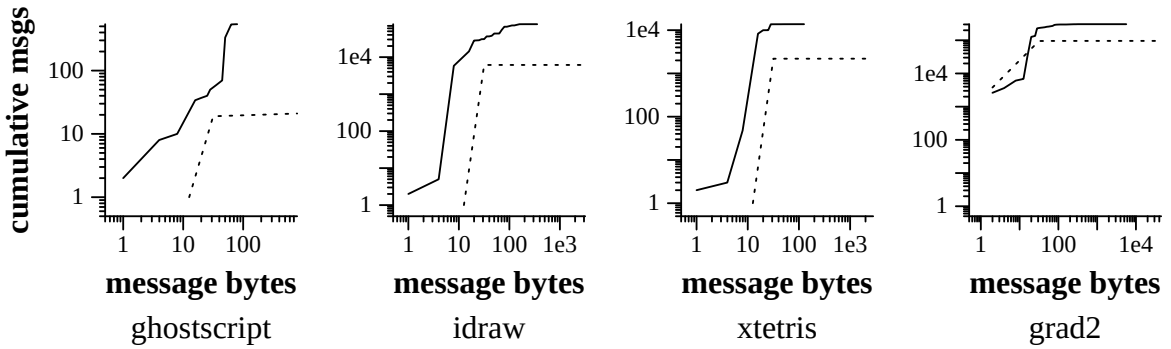
2. xterm (the standard X terminal emulator) clears the old cursor with X_ImageText8, clears a slightly larger area around the new character with X_PolyFillRectangle, draws the new character with X_ImageText8, and draws the new cursor using X_ImageText8.

3. emacs (a native X11 text editor) uses X_ClearArea to remove the old cursor, draws the new character with X_ImageText8, and draws a new cursor with X_ImageText8.

These instruction sequences all produce the same visual result with different methods. X applications have not settled into stable idioms for accomplishing identical trivial tasks. Anyone wishing to optimize some trivial task has to find all of the applications of interest, see how they accomplish the task, and then optimize for each one. Graphics architects are going to have a hard time really optimizing X or developing really good protocol transformations until toolkits develop to the point that most programs use them and idiom settles down.

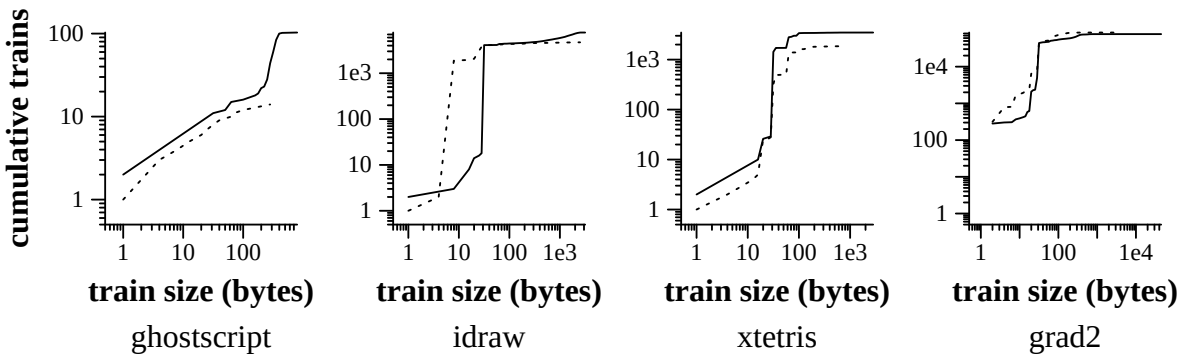
5.4 Size Distribution of X Messages and Trains of Messages

We measured the size distribution of X messages. Results of these measurements are graphed in Figure 6, which plots the cumulative number of messages against message



Cumulative # Messages vs. Message Size

Figure 6: The cumulative number of messages is graphed against message size. We can see that for these traces, the average message size is always less than 100 bytes. Most server messages are 32 bytes, although the X server occasionally sends large messages, usually replies to database queries or requests for font information. The client sends more messages than the server, and is more likely to send large numbers of large messages.
Note that these graphs have varying horizontal and vertical scales.



Cumulative # Trains vs. Train Size

Figure 7: A train is an uninterrupted group of messages arriving from a single source, separated by no more than 1ms. The significance of trains is that some transport layer could collect them and process them as units. In this case, cumulative numbers of trains are graphed against train size. Clients and servers usually end up with nearly the same total number of trains because client commands are provoked by server events, and server replies are provoked by client queries.
Note that these graphs have varying horizontal and vertical scales.

size for four traces. Our result is that although most messages are small, large messages appear occasionally. Client messages in the x11perf trace (not graphed) average almost 1600 bytes (as opposed to less than 100 bytes per message for the traces shown) because *x11perf* is able to send large groups of primitives in each packet. Non-benchmark programs seem to have a harder time generating these large packets.

We think that a more interesting measure than message size is train size. For our purposes, a train of X messages is an uninterrupted stream of X messages all traveling in the same direction, all on the same socket with no inter-message gaps of more than a millisecond. This is roughly the unit of conversation between an X application and the X server. Figure 7 shows that clients and servers usually arrive at similar numbers of trains, even though they have different size distributions. This confirms our expectation of X behavior, that a client usually responds to a server event with a single train. We believe that the client side of the ghostscript trace has more trains because the application was not fast enough to completely update its screen in a single train.

Train information allows us to make some inferences as to good and bad system decisions. We know that the client issues at least one system call per train, and we can estimate TCP/IP [16, 14] header overhead from the train size as was done in [6]. In three out of the four traces shown, most trains are less than 100 bytes long, implying that most TCP packets will also be under 100 bytes. The TCP/IP protocol will add 48 bytes of header information to each packet, implying an overhead of at least 50%. Although this might be tolerable for a fast network, it is a disaster for a slow one. CSLIP/PPP (Compressed Serial line IP [9] layered on the Point to Point Protocol [21]) would be a very efficient network protocol for an X session traversing a slow point to point connection since it only adds about 7 bytes of header information to each packet. A modem that reallocated asymmetric bandwidth after 50 bytes or so would thrash for most interactive use, but would be efficient at application startup when larger transfers dominate, as shown in Figure 1.

6 Conclusions

Our X tracing methodology has allowed us to gather complete traces of X sessions with little impact on performance. A new taxonomy of X messages allowed us to present profiles of X traffic, revealing that X applications spend most of their network bandwidth on useful operations like updating the screen. As a function of the total number of bytes transmitted, X applications spend very little on database operations, window manipulation, or IPC. We've made time dependent snapshots of X activity. These snapshots allow us to see how an application is progressing through it's various stages, and make it easy to zoom in on areas of interest. In particular, we can see that X database operations are clustered at application startup, contributing to a common perception that X applications are sluggish. These operations are important to user perceptions, even though they

are not a large contributor to total bandwidth. Our interest in sending the X protocol over low bandwidth networks led us to to investigate the size distribution of X messages and groups of X messages that could be combined into single network packets (trains). We found that even trains of X messages were usually smaller than 100 bytes, less than a tenth of the maximum ethernet packet size and only twice the TCP/IP header size. We conclude that TCP/IP is not a very efficient way to transport X traffic.

This work applies the careful measurement technology, now so common in computer architecture, to a graphics protocol. The techniques presented here will apply to other graphics protocols. We expect to see further work in this area, resulting in a quantitative understanding of graphics protocols and the systems necessary to support them.

References

[1] Andrews, Edmund L., "A. T. & T. sees the future in in games," The New York Times, Business Day, pages D1 and D5, July 29, 1993

[2] Baker, Mary G., John H. Hartman, Michael D. Kupfer, Ken W. Shirriff, and John L. Osterhout, "Measurements of a distributed file system," Proceedings of the Thirteenth ACM Symposium on Operation System Principles, October 1991, pp 198-212

[3] Blaze, Matt, "NFS Tracing by Passive Network Monitoring," Proceedings of the 1992 USENIX Winter Technical Conference.

[4] Card, Stuart K, Thomas P Moran, and Allen Newell, "The psychology of human-computer interaction," Hillsdale, N.J., L. Erlbaum Associates, 1983

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

[6] Droms Ralph and Wayne R. Dyksen, "Performance measurements of the X window system communication protocol," *Software Practice and Experience* vol. 20, no. S2, May 1991.

[7] Gettys Jim, Phil Karlton, and Scott McGregor, "The X Window System, Version 11," *Software Practice and Experience* vol. 20, no. S2, May 1991.

[8] Hennessy, John L., and David A. Patterson, "Computer architecture: a quantitative approach," Morgan Kaufmann Publishers, Inc., San Mateo California, 1990

[9] Jacobson, V., "Compressing TCP/IP headers for low-speed serial links," SRI International, Menlo Park, CA, Feb. 1990., RFC-1144

[10] Griswold, Ralph E., and Madge T. Griswold, "The icon programming language," Prentice Hall, Englewood Cliffs NJ, 1990

[11] H. Kalmus, D. B. Fry and P. Denes, *Effects of Delayed Visual Control on Writing, Drawing and Tracing.* Language Speech, 1960, v3, pp 96-108

[12] Osterhout, John K., Hervè Da Costa, David Harrison, John A. Kunze, Mike Kupfer, and James G. Thompson "A trace-driven analysis of the UNIX 4.2 BSD file system," Proceedings of the Tenth ACM Symposium on Operating Systems Principles December 1985, pp 15-24

[13] Peterson, James L., "XSCOPE: A debugging and performance tool for X11" Information Processing 89: Proceedings of the IFIP 89 Congress, August 1989, pp 49-54

[14] Postel J. Ed., "Internet Protocol Specification," SRI International, Menlo Park, CA, Sept. 1981. RFC-791

[15] Postel J., "User Datagram Protocol," SRI International, Menlo Park, CA, Aug. 1980. RFC-768

[16] Postel J. Ed., "Transmission Control Protocol Specification," SRI International, Menlo Park, CA, Sept. 1981. RFC-793

[17] Romkey, J., "A Nonstandard for Transmission of IP Datagrams Over Serial Lines," SRI International, Menlo Park, CA, June 1988, RFC-1055

[18] Rubin, Paul and Jay Fenlason, "Gawk - pattern scanning and processing language," man page for gawk, Free Software Foundation, 1993, ftp from prep.ai.mit.edu

[19] Scheifler Robert W., "The X Window System Protocol," M.I.T. Laboratory for Computer Science. 1988.

[20] Scheifler Robert W., "The X Window System," *Transactions on Graphics* vol. 5, no. 2, April 1986, and*Software Practice and Experience* vol. 20, no. S2, May 1991

[21] Simpson, W. A. "Point-to-Point Protocol (PPP) for the transmission of multi-protocol datagrams over point-to-point links," SRI International, Menlo Park, CA, May 1992. RFC-1331