Designing LBX

An Experiment Based Standard

Keith Packard[†]

Abstract

Unlike other X Standards efforts, the design of Low Bandwidth X (LBX) `[Fulton93]` could not be done in a vacuum. With the goal of LBX to provide a usable X environment via an extremely limited bandwidth channel, the only way to effectively design the system was to experiment with different ideas and measure which actually worked better. Some of these experiments could be done with pencil and paper, other required extensive development. Many design decisions were made using parts of the eventual implementation. The process, along with the experiments, results and design will be presented together.

Introduction

History

The X Window System `[Scheifler92]` was designed to provide a network based window system in an environment where the available network was Ethernet and the available processing power was a VAX 11/750 or perhaps a bit better. The original X encoding was the result of a compromise between the available network bandwidth and the processing power. As such, it was not originally intended to run over low bandwidth network connections. However, because it requires only a reliable duplex bytestream to operate, there has been a continued interest in reaching down to these low performance networks and making it work in spite of the difficulties.

[†]*Keith Packard (**keithp@ncd.com**) has been a member of the Network Computing Devices engineering staff since June 1992. Prior to that, he was the senior staff member at the X Consortium. He has worked with and participated in the design of much of the X Window System since early days of the X Consortium.*

SLIP, CSLIP and PPP

These systems encapsulate the IP protocol suite and transmit them over standard RS-232 wiring. This allows any IP application to run, albeit slowly, over any serial link, from dialup modems to dedicated serial lines. As X is just another IP protocol, it too can run in this environment. However, none of these link-layer protocols takes care of ameliorating the low bandwidth/high latency of the underlying serial transport. Using the original encoding for X over a 9600 baud modem is excruciating in these environments.

Worse yet, when compressing modems are used, the actual performance decreases as compression inside the modem must wait for a chunk data to be transmitted before it can compress and send it to the remote modem effectively doubling the latency of the system, not to mention the delay while the modem discovers that there isn't any more data coming soon.

Xremote and Tek eXpress

Xremote `[Cornelius92]` is a proprietary serial-line protocol available in NCD X terminals, NCD PC X servers and many network terminal concentrators. Tek eXpress is a proprietary serial-line protocol available in Tektronix X terminals.

Both of these systems implement a link level protocol which provides for a single reliable duplex bytestream over a serial line. Layered on this bytestream is a slightly compressed version of the X protocol, some simple difference encodings for small packets and Lempel-Ziv compression on both directions of the bytestream. Eliminating the generality of IP reduces the packet headers to a few bytes; compressing the X protocol decreases the bandwidth consumption a bit more. Both of these systems are usable for basic text-based operations; in particular, the performance of xterm over this link is nearly identical to the available underlying serial performance.

Neither of these systems attack the problem of increased latency, which has been demonstrated to significantly limit many aspects of application performance, especially application start-up. Starting up a new xterm in this environment can take as long as 10 seconds.

Because there is no underlying IP transport available, the only system which can gain network access is the window system. Remote file access, terminal sessions and even name resolution are not available. The remote X server is isolated from the network, instead of connected to the network. So much for network computing.

LBX Requirements

These requirements directly address the problems of existing systems.

Use existing network protocols when available

To make the X server host able to be fully integrated into a network computing environment, it is essential that the network be available to applications on that host. Even if the host is an X terminal, remote configuration, font access and network authentication are denied to a non-networked system.

When Xremote was designed, SLIP was the only available serial link level protocol. SLIP does not compress the IP or TCP headers at all, making the typically small X packet double in size. Since then, CSLIP and PPP have demonstrated that general network link level protocols needn't be significantly more expensive than specialized ones.

However, imagine trying to administer a collection of 1000 remote X systems accessing a network via dial-up modems; each X system would need to have an IP address allocated and each serial link would need a separate network allocated. In such an environment, the capability of a full network are dwarfed by the amount of administrative work involved. Such an environment still needs a specialized remote X access system.

Support multiple X applications over a single bytestream

Both Xremote and eXpress implement a single reliable duplex bytestream. To have more than one X client running at a time, they take the single bytestream and multiplex it between the different clients. This is fundamentally different from how an IP based system works. It is, however, easier to design and potentially much more efficient as contention for bandwidth can be tightly controlled at either end by agents with a better idea of what is going on.

For LBX to operate more efficiently over either a specialized link layer protocol, or existing single-connection oriented network systems like X.25, it also supports multiplexing many clients over a single connection.

Beneficial in low-bandwidth as well as high latency environments

With the pervasiveness of IP access, it is commonplace to want to run applications over extremely long distances through networks which may have high bandwidth, but which suffer from high latency. In these environments X applications spend quite a bit of time waiting for replies to X requests. This problem is aggravated by the synchronous nature of Xlib in these situations.

The only way to decrease the effect of latency is to reduce the number of round trips made between the application and the X server.

Faster than currently available systems

LBX is quite different from other X Consortium standards in that it does not attempt to provide new functionality for the X Window System. The ability to run X over low bandwidth links has existed ever since it was designed. Instead, LBX must strive to simply be better than existing solutions, enough better to make it worth the cost of conversion.

LBX will be useful only when it is demonstrably more usable than existing systems. Therefore, LBX has been designed in response to specific performance measurements and analysis, rather than in isolation from implementation as with most other X Consortium standards.

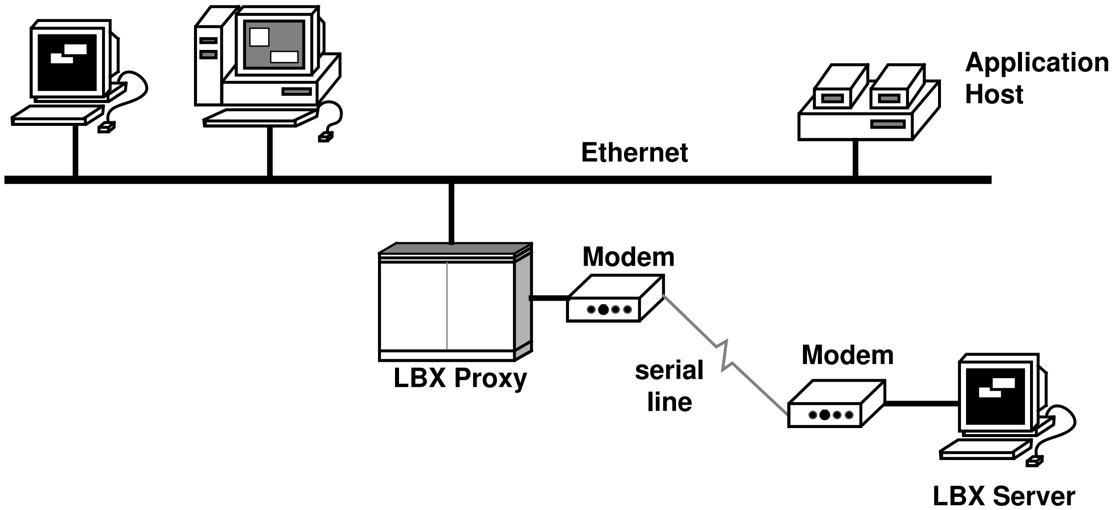

Figure 1: Example of an LBX system

Overall architecture of an LBX system

The architecture of an LBX system consists of several major pieces (See Figure 1).

- The LBX capable X server,
- A communications link capable of supporting LBX
- The LBX proxy which presents a standard X service to the network.
- Regular X applications running on machines connected to the proxy

The only difference between this any regular X environments, other than the different encoding running over the communications link to the X server, is that an LBX proxy is interposed between the X applications and the LBX capable X server. This proxy advertises X services to its local network just like an X server would. The proxy transforms the regular X encoded protocol from X clients into the LBX encoding running over the serial wire in a series of steps (See Figure 2).

Experimental design process

Many experiments were made during the design of LBX. Until a working LBX prototype was available, they were done in a variety of environments, from modified versions of Xremote to massaging X protocol dumps generated with xscope. Some of the results of these experiments have directed the design of LBX and are described in detail.

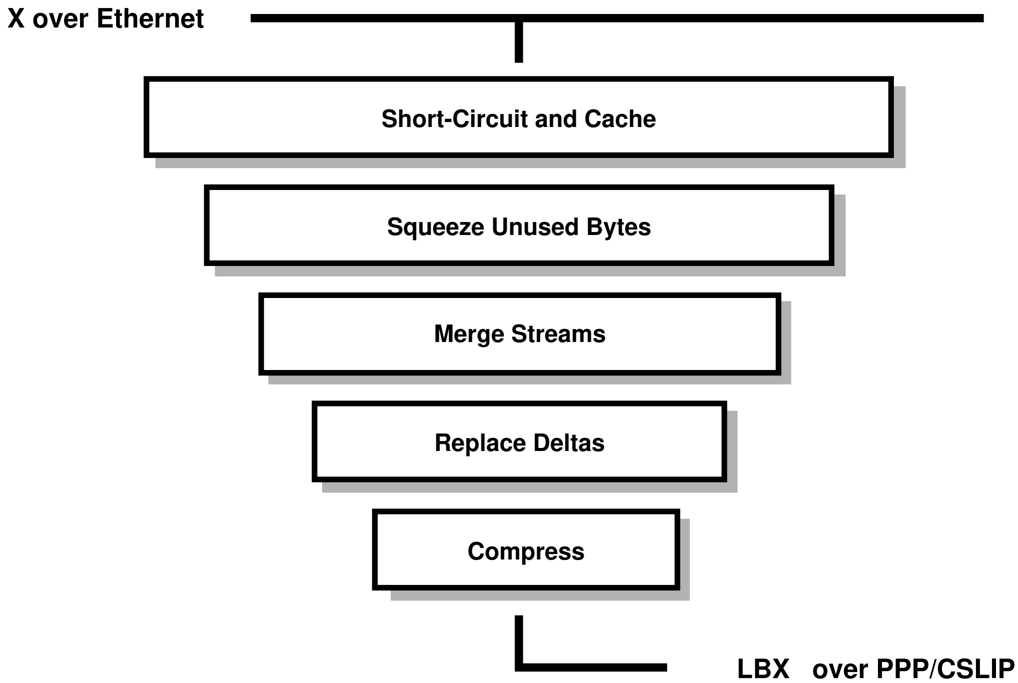

Figure 2: LBX Proxy Architecture

Reducing data transmission by using Tags

Tags were designed to reduce bandwidth consumption, and potentially latency as well, by distributing knowledge of information without distributing the information itself. In requests and replies where largely static information would normally be sent, LBX re-encodes the information to deliver a tag instead. The data which would have been sent are referred to as the *tag contents*. If the receiver already has the contents for that tag, no additional information need be sent. If the receiver ends up never needing the tag contents, again, no additional information need be sent. However, if the contents are needed, a separate message is transmitted requesting it.

As an example, consider the connection setup information sent by the X server whenever a new client connects. With the exception of the root-window event masks, this information is typically static. When the first client connects, the proxy will not have the contents of that tag, and will request it. Thereafter, if the connection setup information doesn't change, the X server will continue to send the old tag in place of the connection setup block, and the proxy will simply deliver its copy to the client. If the information changes, the server delivers a new tag and the proxy gets the new contents. At connection setup, this eliminates 200-300 bytes of information on a typical display. There are several cases where tags are used in the LBX protocol are (See Figure 3).

At start-up, Frame maker (a typical Motif application) gets 5 properties, opens 3 fonts and gets the keyboard mapping twice. The savings over regular X are substantial (See Figure 4).

	Typical size
Connection setup	250 bytes
Keyboard mapping	1000 bytes
Keyboard modifier mapping	16
Large window properties	?
Font information	2800 bytes for ISO8859-1 encoding
	82660 bytes for JISX0208.1983-0 encoding

Figure 3: Tag usage in LBX

Connection setup	272
Keyboard mapping	2064
Properties	232
Font information	8820
Total	11660

Figure 4: Typical savings from tag usage

Eliminating round trips through short-circuiting

Many X applications spend a considerable amount of time at start-up making small synchronous requests getting unchanging information from the X server. These include interning various ICCCM atom names not included in the predefined atom list and allocating common colors. Short-circuiting is a technique where these synchronous X requests are turned into either asynchronous requests, or eliminated altogether using data stored in the proxy.

Atom related requests are extremely easy to short circuit. The relationship between an atom name and value is fixed until the server resets. By remembering each atom used by every X client the proxy can quickly return the appropriate response for future requests. In this case, the atom related requests are elided from the protocol forwarded to the server, along with the necessity of waiting for the reply.

To discover the effect of atom caching on LBX performance, an experiment was run on a modified version of Xremote. No encoding change was required as the atom request was simply replaced with a noop request. The following results were obtained using x11perf to measure the number of GetAtomName requests for the same atom executed per second (See Figure 5). These results aren't very helpful without some idea of how many atom requests a typical application uses (See Figure 6).

	9600 baud modem	38400 direct
Without atom caching	8.0 ops/sec	45.0 ops/sec
With atom caching	687.0 ops/sec	687.0 ops/sec

Figure 5: Short Circuiting Performance

Application	InternAtom	GetAtomName
FrameMaker	29	0
Xlogo	5	0

Figure 6: Typical Atom Request Usage

If it takes 100ms to execute a synchronous X request, FrameMaker will start 3 seconds faster just given this speedup.

Read-only color allocation is a bit more complicated. The proxy tracks allocations by name or RGB value for read-only color cells in each colormap. When a duplicate name or RGB value is received by another client for the same colormap, it returns the previous pixel value to the client and replaces the synchronous alloc color request with a new asynchronous increment pixel reference request. As the client receives the reply directly from the proxy, it can continue executing without delay.

Reducing the number of delivered MotionNotify events

X version 11 specifically allows the server to limit the number of MotionNotify events delivered to a client, the only guarantee is that when the pointer moves and then stops, at least one MotionNotify event will be delivered. To make finer position information available, a history of recent pointer positions at as fine a granularity as the driver supports is available through separate requests.

This mechanism is included with the knowledge that applications frequently don't really care where the pointer has been, only where it is right now; many applications read motion events and throw away all but the most recently received event. This is especially true for window managers positioning windows, and for menu tracking.

When available network bandwidth is high, frequent MotionNotify events yields smooth animation on the screen. With less bandwidth, frequent MotionNotify events ends up making the application lag to far behind the pointer, increasing perceived latency in the system.

There is no way to reduce the latency of accurate mouse-tracking below a single round trip time, but many factors can conspire to make it rise up to two round-trip times. Moreover, the bandwidth required just to support mouse motion might reasonably be up around 800 bytes/second, assuming that motion events were compressed down to a half-dozen bytes of packet overhead, and another eight bytes or so of data. On a 9600-baud line, this is a very large portion of the available bandwidth, and on a 4800-baud line, it is more than is available. Thus, some form of motion compression between the display and the proxy seems mandated. The desirable properties of such a scheme are that it not consume more bandwidth than it can reasonably get while continuing to allow processing of client requests, and that it keep the rate of motion events high enough that the latency is closer to one round-trip per tracking update than to two.

The currently scheme is a liberalized variation of **PointerMotionHint**, in which the server and proxy take into account the typical latency and bandwidth available, and try to keep a reasonable number of motion events in flight between them. A credit-based acknowledgment is used to keep from overloading the link when some other source of traffic takes a lot of the bandwidth away.

Experiments at DEC’s Systems Research Center with local 9600-baud lines found good results with eight motion event tokens and a single type of acknowledgment from the proxy that resets the token counter to its maximum.

This adds few enough events that it doesn't cause much trouble if either side falls behind, and enough latency that under low-load conditions, the acknowledgment for the first motion event should be received at the display just in time to avoid dropping any events. If the load either way on the wire increases such that one end falls behind (for example if the client paints a complex image in response to motion events), the acknowledgment is delayed, and only some number of updates per second are lost, rather than responsiveness (although the updates could track a few events behind).

To measure the effects of motion compression, the current LBX system was compared with the original X protocol. The application under test is xterm, with 2000 lines of history filled with the contents of /etc/termcap. The network is the NCD corporate backbone, a 56Kbaud leased telephone line. The scrollbar was slowly dragged from bottom to top and the number of motion events and returning request traffic was measured (See Figure 7).

	Total motion events	Total request traffic
Without motion compression	207	67128
With motion compression	196	35348

Figure 7: Effect of LBX Motion Compression Technique

With motion compression enabled, only a few motion events were eliminated; actual statistics gathered indicated that the server disposed of 16 motion events (out of about 200) in this particular example. Even still, the pointer tracking lagged significantly behind. The significant reduction in request traffic is probably the result of xterms own motion compression acting on the larger batches of motion events it receive, rather than from the reduction in delivered motion events.

While the experiments using a 9600 baud direct serial line yielded encouraging results, the final LBX architecture does not demonstrably reduce the number of motion events delivered to the client. More work is clearly needed to better determine how many motion events to keep in flight between the server and proxy; obviously some sort of dynamic protocol is necessary.

Re-encoding the X protocol

While LBX is designed to operate in conjunction with a general stream compressor, much data in the X protocol can benefit from protocol-specific compression. Typical usage patterns can be exploited to significantly reduce the amount of data even before presenting it to the stream compressor. In other cases, typical usage patterns can be exploited when evaluated in conjunction with the stream compressor to increase the efficiency of the compressor.

The former case is easy to analyze. Simply compute the number of bytes presented before and after compression. The latter is much more difficult, and has been analyzed by performing experiments using Xremote augmented with the new encodings.

Font metric compression

Font metrics can occupy an enormous amount of space. The normal X protocol uses 6 signed 16-bit numbers for each character (See Figure 8):

Field	Type	Bytes
Left side bearing	int	2
Right side bearing	int	2
Character width	int	2
Ascent to top	int	2
Descent to bottom	int	2
Attributes	unsigned	2

Figure 8: Per-Character Information

Fortunately, a variety of techniques can be used to reduce their size. Most fonts' bearings, ascents, descents, and character widths could easily fit within signed 8-bit numbers. In fact, many fonts could even encode the horizontal and vertical bearing into to a pair of 12 bit values, saving yet another byte. The contents of the 2-byte character attributes field are not defined by X but are typically zero and can usually be omitted. Terminal emulator fonts can further be squeezed by removing the character width field, since it is constant across all encoded characters. Thus, it is possible to re-encode most sets of font metrics to use only 3-5 bytes per character instead of 12, saving 1500 bytes for Latin-1 fonts and over 50,000 bytes for Kanji fonts.[†]

Similar reductions can also be applied to some of the font header information. Furthermore, within the list of font properties, 3/4 of the data is made up of 4-byte atom values of which the 2-3 most significant bytes are typically zero. By removing the unnecessary values, as many as half of the 280 bytes in the font header and property data could potentially be eliminated.

Event compression

All X events are normally padded to 32 bytes. Although this simplifies their reading from the network and processing within the client, it adds a lot extra data (See Figure 9). Re-encoding these events to take less room speeds up their delivery and improves application response times.

MappingNotify	26	CirculateNotify	16
VisbilityNotify	24	CirculateRequest	16
Focus	23	Expose	15
NoExposure	22	PropertyNotify	15
DestroyNotify	21	GraphicsExposure	12
MapRequest	21	ReparentNotify	12
ResizeRequest	21	CreateNotify	10

Figure 9: Unused Bytes in Various X Events

[†]*Similar techniques should be applied to the X Font Service protocol* `[Fulton91]`, *including the ability to fetch sets of compressed glyph images.*

MapNotify	20	SelectionNotify	9
UnmapNotify	20	ConfigureNotify	6
ColormapNotify	19	SelectionRequest	5
GravityNotify	17	ConfigureRequest	4
SelectionClear	17	Key and Button	1

Figure 9: Unused Bytes in Various X Events

As an example of what effect this might have, consider opening a new file in Frame maker. It receives 269 events for a total of 8608 bytes. Simply removing the unused bytes in these events will reduce that to 5633 bytes.

Image data re-encoding

LBX will take advantage of existing image compression techniques, like CCITT G4 for bitonal images and at least RLL encoding for multi-bit images. However, the combination of these compression techniques along with a general bytestream compression system frequently yields less-than-ideal results. For this to work as well as it can, the general byte-stream compression will be disabled during transmission of compressed images. Experiments with SIE, NCD's imaging extension, prove that G4 compression can dramatically reduce the size of bitonal images.

Resource ID reencoding

Although X provides requests which can operate on a large chunk of data at one time, many X applications tend to send numerous small requests in which the data is but a small piece of the transmission cost. The overhead of transmitting several resource IDs can dwarf the actual application data. Echoing a character in Xterm is a classic example; to get a single character on the screen requires no fewer than three requests totaling 60 bytes, only 3 of which are actually client data (two space characters to erase and redraw the cursor). 24 bytes of this is consumed with resource IDs.

In these examples, a trace of (what else) frame maker was captured, and the various caching schemes were analyzed against the same trace. This was intended to eliminate any other effects which might influence the amount of data transmitted. In this sample run basic statistics were gathered (See Figure 10).

Number of Drawable IDs sent	4670
Number of Root Windows sent	1548
Number of non-client Windows	46
Number of client Drawables	10709
Number of client Drawables < 7f	462
Number of client Drawables < 3fff	10247
Maximum client Drawable ID	467 (0x1d3)
Bytes used in raw transmission	50188
Number of GC IDs sent	6004

Figure 10: Resource ID usage profile

Number of GC IDs < 7f	2273
Number of GC IDs < 3fff	3731
Maximum GC ID	436 (0x1b4)
Bytes used in raw transmission	24016

Figure 10: Resource ID usage profile

Two different schemes were designed to reduce this amount. The first was a simple reencoding scheme, taking advantage of the fact that most resource IDs are small in value (See Figure 11). The results from this were encouraging, reducing the number of bytes to 27332 for windows and 9735 for GCs, for a 50% reduction in bandwidth; a slight variation, where the 7 bit form indicates server resource IDs yields similar results (23150 windows, 12008 gcs or 53% reduction).

Value	Byte 1	Byte 2	Byte 3	Byte 4
< 0x7f	bits 0-6	not sent	not sent	not sent
< 0x3fff	0x80 | bits 0-6	bits 7-13	not sent	not sent
< 0x1fffff	0x80 | bits 0-6	0x80 | bits 7-13	bits 14-20	not sent
	0x80 | bits 0-6	0x80 | bits 7-13	0x80 | bits 14-20	bits 21-31

Figure 11: Simple reencoding of XIDs

As an alternative, an adaptive least-recently-used caching strategy was also investigated. Keeping a small cache of 15 elements each for Drawable and GC IDs, a rendering request is encoded using 1 byte containing the two 4-bit cache entry values for Drawable and GC ID. A cache miss is indicated by a value of 15 in the appropriate entry. On cache miss, the actual resource IDs follow the 1-byte field, yielding a 1, 5 or 9 byte encoding for both values. For requests using only a Drawable or GC ID, the cache would again be used, except this time only a single 4 bit value would be encoded in the first byte, and the optional 4 bytes following would supply any missing value. To evaluate this, some different statistics from the sample run were generated (See Figure 12). In this example the number of bytes used to transmit the resource ID data would be:

```
1*(num requests) + 4*(num gc miss + num draw miss)
```

for a total of 19797 bytes, or a 73% savings over the uncompressed encoding. This is clearly better than the other proposed technique, having the additional benefit of only two or three possible request header lengths, instead of four or seven.

Number of rendering requests	3574	
Number of other GC requests	2430	
Number of other Drawable requests	1096	
Number of other Window requests	7877	
Total number of these requests	14977	
Number of GC cache hits/misses	5538	466
Number of Drawable/Window cache hits/misses	11808	739

Figure 12: Cache resource usage profile

LBX Implementation

Taking the collection of techniques listed above and putting together an implementation has been an interesting challenge. There are two major pieces of the system, the LBX proxy, and the LBX additions to the server. Instead of implementing a completely new protocol for LBX, a simple extension to the core protocol was designed which incorporates the missing functionality required to manipulate multiple clients on a single byte stream, query and enable various options like compression and the alternate X encoding. This allows LBX to leverage the existing X authorization mechanisms, to use the same registered network name (e.g. TCP Port 6000). In addition, LBX will run over any transport which can support X.

LBX proxy implementation

In order to reduce the amount of code produced for the proxy, instead of simply copying source code from the X server for managing multiple network connections, the proxy actually links in the OS library from the X server directories. To do this, it simply provides all of the interfaces required by that library. This inherits support for multiple networks, authorization, XDMCP and handles the operating system dependencies surrounding network communications. On the other end, when connecting to a remote X server, the proxy uses Xlib. Again, this reduces the amount of machine-specific code in the proxy, and increases its functionality, taking advantage of the authorization mechanisms supported by Xlib, and performing the proxy initialization steps with the X server.

The proxy examines each X request separately and performs short-circuiting and reencoding on each request. The resulting requests are forwarded to the X server for processing. Large chunks of requests are sent in batches to increase the efficiency of the compression system.

LBX server implementation

The changes to support LBX in the X server were relatively minor. The OS layer was augmented to handle non-standard X encodings, compression, and to provide new interfaces for multiplexing clients on a single connection. Some DIX hooks were required to implement tagging. The alternative encoding for LBX is supported by having new functions which implement the new encoding, and replacing the requestVector in each LBX client with a new value.

References

`[Cornelius92]` David Cornelius, *XRemote™: A Serial Line Protocol for X*, 6th Annual X Technical Conference, Boston, MA, 1992.

`[Fulton91]` Jim Fulton, *The X Font Service Protocol*, X Consortium X11R5, 1991.

`[Fulton93]` Jim Fulton and Chris Kent Kantarjiev, *An Update on Low Bandwidth X (LBX)*, 7th Annual X Technical Conference, Boston, MA, 1993.

`[Scheifler92]` Robert W. Scheifler and James Gettys, *X Window System*, Digital Press, 1992.