An Update on Low Bandwidth X (LBX)

A Standard for X and Serial Lines

Jim Fulton[†]
Chris Kent Kantarjiev[‡]

Abstract

The question of how to run X applications over telephone lines and other low bandwidth channels has gained much attention recently. In response, the X Consortium is working to define a standard called Low Bandwidth X (LBX) that will allow the X protocol to be transmitted more efficiently over connections that have much lower bandwidth and much higher latency than Ethernet or Token Ring. This effort is building on the concepts pioneered by NCD's XRemote `[Cornelius92]` and other similar systems, and is incorporating a variety of caching techniques to give LBX higher performance than its predecessors.

Introduction

One of the main strengths of the X Window System `[Scheifler92]` is the ability to connect applications to any display using a reliable byte stream. Although X was originally developed for local area networks (LANs), many people have come to realize that, with a small amount of effort,

[†] *Jim Fulton (**jim@ncd.com**) heads the Standards and Extensions group at Network Computing Devices, Inc., where he focuses on future directions in desktop technologies, user interface design, and relations with various standards organizations. He has worked with and participated in the design of many of the various pieces of the X Window System since its inception at Project Athena in 1984. He was the architect of the X Font Service and was one of the original members of the X Consortium staff. He holds a B.S. in Computer Science and Engineering from MIT.*

[‡] *Chris Kantarjiev (**cak@parc.xerox.com**) is member of the research staff in the Computer Sciences Lab at the Xerox Palo Alto Research Center (PARC), where he works on ubiquitous computing. He has been involved with the X Window System since the first VS100s went to Project Athena, most notably as an architect of the Display PostScript Extension to X. He holds a B.S. in Physics from Xavier University and a Ph.D. in Computer Science from Purdue University.*

it can be used quite effectively over serial lines and other low bandwidth channels. This enables applications to be used in remote offices, at home, and in environments where LANs are not feasible. Furthermore, with the advent of notebook personal computers, it makes mobile computing with X a reality.

History

Previous efforts in this area have typically used serial lines in one of the following ways:

- The X server runs on a host and sends escape sequences to a graphics terminal.
- Raw X protocol is sent over PPP or CSLIP (raw SLIP is unacceptably slow), often with compressing modems when used over phone lines.
- The X protocol streams from the various applications are squeezed using techniques such as caching, delta replacement, and compression prior to transmission.

The first method is rapidly disappearing because it does not allow for X applications that are on the same side of the network as the display. Since X terminals, PCs, and workstations all tend to provide local clients of some form (especially window managers), a true X server in the display is required.

The second method is acceptable for some types of applications, but it has several drawbacks. The X protocol encoding is optimized for local area networks, where it is more important to reduce the amount of time spent encoding data than the amount of data transmitted. In contrast, when dealing with serial lines there is typically much more time for packing and unpacking packets and much less space for data. This makes it very important to squeeze as much redundancy out of the protocol as possible.

Further disadvantages of relying on special hardware to compress the data include added expense, increased transmission latency, and decreased interactivity. These problems are caused by the need to send extra bytes out the serial port and wait for them to accumulate in the modem before they can be compressed, plus the inverse operations on the other side.

The third method involves using a special program to collect the X protocol packets on either side of the network, removing data that can be recreated from caches, reencoding data that can be represented more efficiently, and compressing the results as tightly as possible. This generates far less data than does sending raw protocol while still maintaining a full X server for local clients.

Architecture

LBX builds on this third approach (see Figure 1). Existing networking standards will be used when possible, particularly for the transport layers (*e.g.,* PPP, CSLIP and DDCMP). Since the display will have a real network address, other types of services can be used by the X server (*e.g.,* font and authentication services) and the user (*e.g.,* file, printing, naming and remote login services).

PCs, X terminals, and workstations will all be able to be LBX displays. The host-side program that converts X to LBX protocol will be able to run in terminal servers (see `[Herbert91]`),

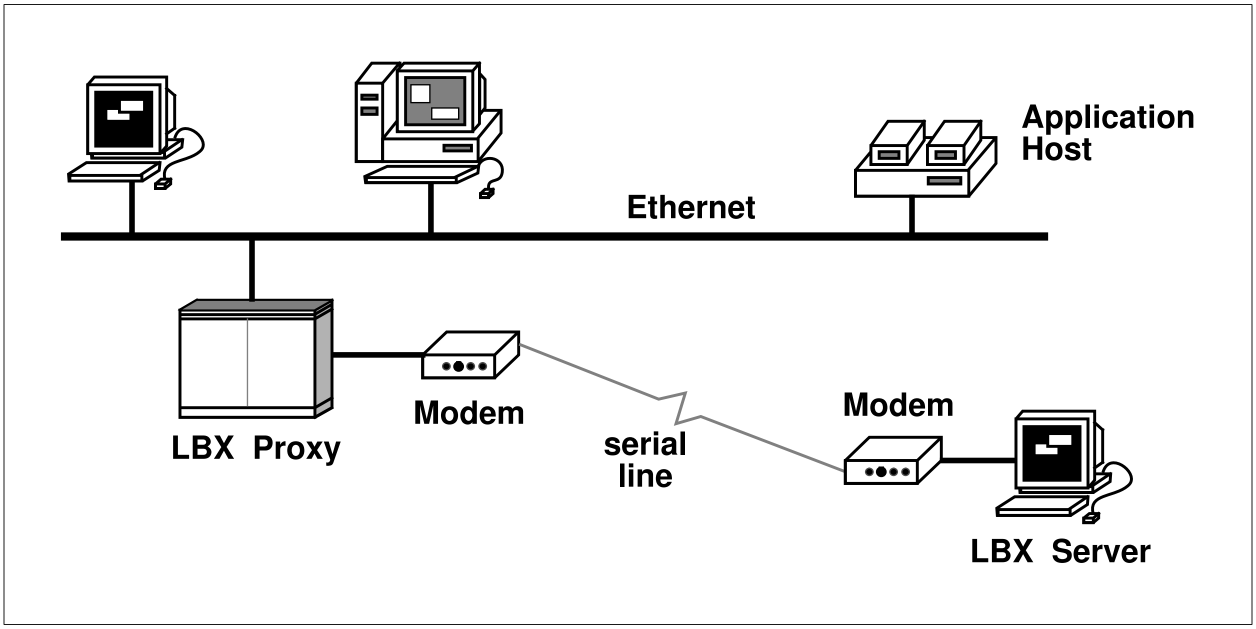


Figure 1: Example of an LBX Connection

workstations, and even mainframes. Hardwired serial lines, phone lines with 9600-baud modems, packet radios, and ISDN are all envisioned as ways of connecting the ends together.

Host-side applications will communicate with what appears to be normal X server running on the same side of the serial line. However, rather than controlling a display, this program, called a "proxy," sends LBX protocol over a serial line or other low bandwidth channel to the real X server (which is enhanced to understand LBX).

X protocol packets are converted into LBX protocol through the following steps (see Figure 2):

1. Any X requests for information that can be answered from the proxy's caches are replied to directly. This allows the serial line to be bypassed in many common cases (*e.g.,* font metrics, atoms, color cell allocations, keysym tables).
2. Packets are reencoded, when possible, to take less space (*e.g.,* images, font metrics).
3. The remaining request streams are multiplexed together for transmission. Access to the serial line is allocated in a round robin basis, with large packets being split up so that a single client doesn't monopolize the transport.
4. Packets that can be more efficiently represented as changes against previous packets (called "deltas") are replaced.
5. The entire stream is then compressed and transmitted.

To give implementations the flexibility to incorporate new technologies and to trade cost for performance, LBX will allow the display and the proxy to negotiate which compression algorithms are used and how much caching is done.

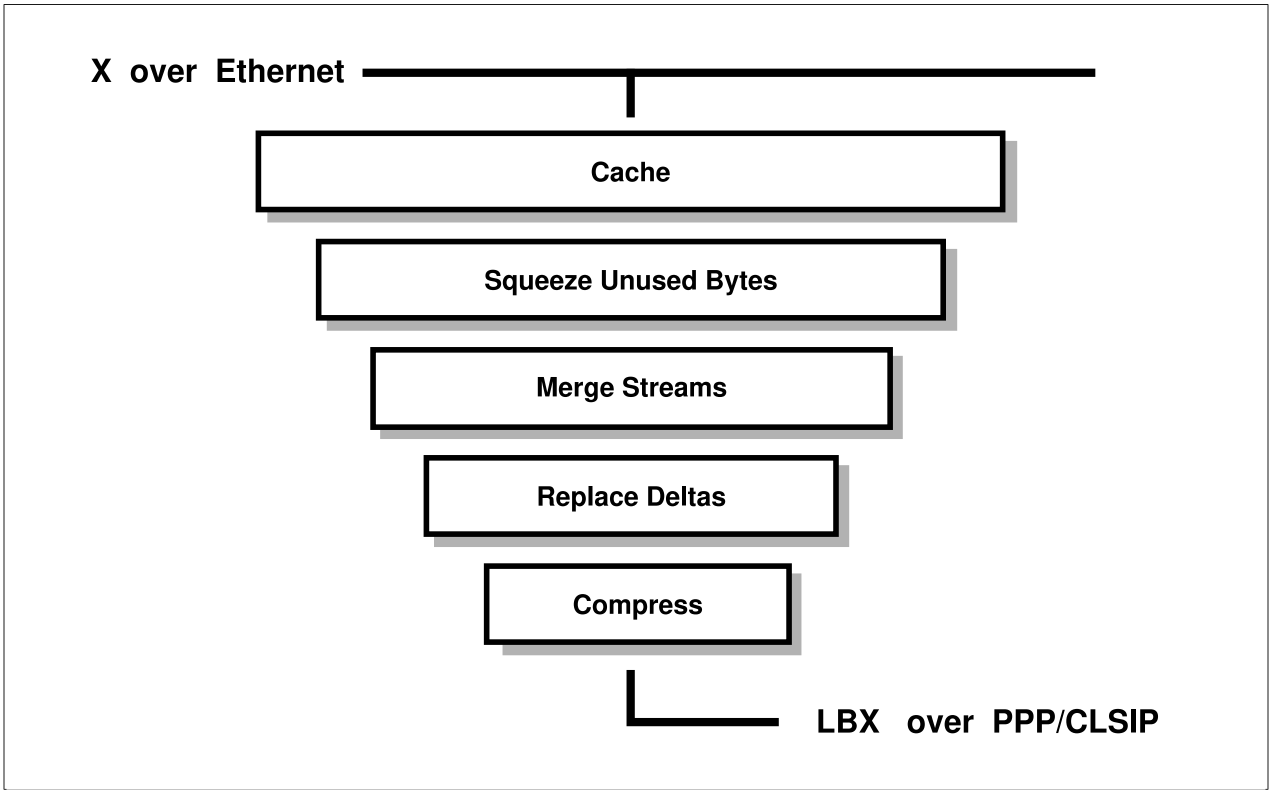


Figure 2: LBX Proxy Architecture

Caching Techniques

The major feature that distinguishes LBX from its predecessors is its use of various caching techniques to reduce redundancies in the X protocol. In many situations, this can dramatically improve interactive response and application performance.

The general purpose of caching is to find ways to avoid sending data down the serial line. This is particularly important when dealing with requests for which the application is blocked pending a reply. Many people are willing to accept that output will be slower when running on a serial line; what bothers them is the delay, or latency, with which applications respond. One of the keys to improving the perceived performance is to get rid of round trips whenever the user is forced to wait for the application (particularly at start-up and in response to user input).

The X protocol provides opportunities for almost arbitrary amounts of caching at a price of equally arbitrary amounts of complexity. At a minimum, LBX will cache the types of information that do not require a lot of high-speed synchronization with the X server:

- Connection Setup Information
- Window Property Data
- Atom Values

- Font Metrics
- Color Cells and Names
- Keyboard Mappings

Further gains will most likely require keeping a shadow of the window hierarchy in the proxy. The amount and types of information to keep and how to avoid thrashing is still under analysis.

Connection Setup Information

Before an X client can request any services from a display, it must:

- establish a connection to the display,
- have that connection be accepted by the X server, and
- receive a variety of information describing the display and its capabilities.

The proxy will intercept the connection request by pretending to be a local X server. It will then perform the security checks needed for the second step since the real X server may be unable to decrypt the authorization data provided by the client. The proxy will work with the X server to decide whether or not to accept the connection so that a variety of access policies can be supported.

Once the proxy has decided to accept the connection, it will send the X server setup information back to the client. This data tends to be the same across most connections and can easily be cached so that it doesn't have to be transmitted over the serial line each time. The time that is saved will be particularly important at the beginning of a session when many applications are being started.

Window Property Data

After connecting to the display, all Xlib-based applications fetch the contents of the `RESOURCE_MANAGER` property off of the root window of screen zero. Newer Xt-based applications also fetch the contents of the `SCREEN_RESOURCES` property off of the root window of the default screen. Since either or both of these properties often contains large amounts of data, application start-up time can be significantly reduced if the property values are cached in the proxy.

This technique can also speed up the transfer of large selections. For example, when cutting and pasting a large block of text between two remote applications, data would normally be sent down to the server and then back up to the other application. But with LBX, the proxy will have the option of caching and delivering the data directly.

Atom Values

With the growing complexity of the Inter-Client Communications Conventions Manual `[Rosenthal91]` and most graphical user interfaces, many applications intern a large number of common atoms at start-up. Since atoms cannot be deleted or changed, a simple cache of known values will eliminate a significant number of round trips and decrease the overhead of starting subsequent applications.

Font Metrics

In order to present a consistent user interface, many applications use a common set of fonts. Each of these fonts is usually queried once per application, often when the client starts up. This results in `80 + 8p + 12c` bytes of data being sent to the client, where "`p`" is the number of properties in the font and "`c`" is the number of characters for which metrics are supplied. Most fonts follow the X Logical Font Description standard `[Flowers91]` and have an average of 25 properties. Far more significant, however, is the number of characters in the font. Latin-1 fonts tend to have 220 characters (of which 32 do not actually encode any glyph); Kanji fonts have on the order of 7900 characters (of which a little over 1000 that do not encode any glyph). Thus, uncompressed font descriptions tend to require between 2900 bytes in Latin-1 environments and an astounding 95000 bytes in Kanji environments.

Thankfully, fonts are immutable and thus are an ideal candidate for caching. The proxy can store a copy of the font metrics so that additional requests for them can be answered directly. Thus, once a commonly-used font has been loaded, new applications will be able to request it with no delay.

Unfortunately, there is one situation in which the font data must be transmitted twice. If the actual font data is not local to the display it must be read over the serial line from the network and then sent back to the proxy through LBX. Although it is possible that the proxy might sometimes be able to locate the font data and determine the character metrics itself, this cannot be guaranteed.

There is another possible solution to this problem. LBX could include protocol that would allow the X server to ask the proxy to obtain raw font data and cache it immediately. However, this would violate a goal of LBX that existing network standards (such as the X Font Service `[Fulton91]`) be used whenever possible. Regrettably, this may become necessary if access to font data is ever dependent upon the authorization information provided by the client when it connected.

Color Cells and Names

Most applications also use common sets of read-only color cells to present a consistent appearance. Although graphical user interfaces are slowing moving towards providing automated interfaces to color selection, many people also use server-resolved color names instead Xlib-resolved RGB or color-space values to specify the contents of those cells. Conveniently, in both cases the resulting information is immutable and can easily be cached.

Keyboard Mappings

One of the more annoying artifacts of using X over slow networks is the pause that often occurs the first time a key is pressed. Xt-based applications, for example, wait until the first key is pressed before fetching the table maps keycodes into keysyms. On modern servers, this table tends to have 4 columns and anywhere between 100-247 rows (some servers leave room for "virtual" keycodes). Although extra rows tend to be full of contiguous 0's (which compress well), even a 100 row table occupies 1600 bytes. Fortunately, this data tends to be relatively static once the user's session has been initialized.

Request-Specific Compression

Although all data in LBX will be run through a general stream compressor, some requests will benefit by changing the way they are encoded. In particular, bitmap and pixmap images typically contain a substantial amount of blank space that can be compressed using a variety of standard techniques (*e.g.,* CCITT Group 4, run-length encoding, etc.).

As discussed on the previous page, font metrics can occupy an enormous amount of space. The normal X protocol uses 6 signed 16-bit numbers for each character (see Figure 3):

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 3: 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 reencode 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.

The keyboard mapping table mentioned in the previous section also provides significant room for improvement. Like the atom values mentioned above, keysyms are 4 bytes long and often contain zero in the 2-3 most significant bytes. An alternate representation that removes these common bytes could save 800-1600 bytes, depending on the number of rows in the table.

†*Similar techniques will be applied to the X Font Service protocol, including the ability to fetch sets of compressed glyph images.*

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

Figure 4: Unused Bytes in Various X Events

Finally, 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 4). Reencoding these events to take less room will speed up their delivery and improve application response times.

Issues Still Under Discussion

Although the techniques described above will provide significant improvements, it is not clear yet whether or not they will be enough. A number of the harder issues are still under consideration:

- How Many Proxies
- Authorization
- Font Access
- Shadowing the Window Hierarchy in the Proxy
- Drag-and-Drop
- Motion Events
- Distributed Resource Allocation and Modification
- Grouped Events

The deciding factor for each of these issues will be whether or not they will provide enough of an improvement in performance to justify the complexity that they will add.

How Many Proxies

Figure 1 shows a single LBX proxy connected to a display, and all the discussions so far have (implicitly) assumed this as well. Proponents of ISDN foresee environments in which an LBX display may have multiple ISDN connections, one to each of several information providers. If there is no high-speed path between the information providers, there is no one best place for the proxy to run. Still, each of those information providers may wish to use LBX to decrease the bandwidth required to communicate with its customers (particularly when the cost of the service is related the amount of network traffic it generates).

A more familiar example comes in the form of wide-area networks (WANs) that in common use today. A display user in Chicago might wish to run applications simultaneously in San Francisco and Boston and would ideally like to use LBX for both. Even more immediately, LBX proxies can improve the latency and throughput between clients connected to the same proxy, so even in the case of a singly-connected display, it might make sense to place a proxy near a group of applications running at a site which is distant in the WAN topology. In this situation, the whole network is assumed to be a low-bandwidth channel.

Keeping the proxies synchronized with the X server and with each other grows in complexity in direct proportion to the aggressiveness of the caching algorithms. Although adding extra proxies would appear to further complicate matters, allowing clients local to the display already mandates the need for careful synchronization and arbitration over who is in control of a given resource. Thus, allowing multiple proxies may not add much complexity beyond that inherent in generalizing from “two” to “more than two.” However, the total complexity is not yet understood; in the end, LBX may end up avoiding proxy allocation and modification, instead relying solely on read-only caching. This will be the subject of very careful study in the coming months.

Authorization

After a client specifies its byte-order, it sends a description of the version of the protocol it wishes to use as well as authorization data to use in determining whether or not it should be granted access. In the current X protocol, the server sends a boolean response indicating success or failure; the ANSI draft standard for X (which has stalled in committee) adds a third type of reply may be returned indicating that additional authorization information is desired.

Although one could envision the multi-trip authorization exchange being used to tell the client of the existence of the X server, current X applications assume that the authorization data will be decoded by the recipient connected to the client. In LBX, this means that the proxy has to handle the decoding. Will the proxy have enough information available to implement access control? Will this conflict with any authorization policies? Some servers may wish to somehow reuse the authorization data (*e.g.,* for communicating with font servers), so it will need to at least be passed through. But should this pass-through be done in encoded or decoded form?

The proxy could pass the authorization data through to the X server via an LBX request, wait for a reply from the server, cache the results, and then pass them through to the client. This would allow the proxy to be separated from the authorization policy of the X server while still enabling both the proxy and server to know the outcome of the authorization request.

However, this won’t work for all types of authorization mechanisms. If the client has encrypted some data that only the proxy is supposed to be able to decrypt via a third party such as Kerberos, then passing it to the display won’t do any good.

Font Access

Obtaining font data that has license restrictions is tied up with how authorization information is handled. In particular, it determines whether the display needs to have a *direct* link to the font server or an *indirect* link through the proxy.

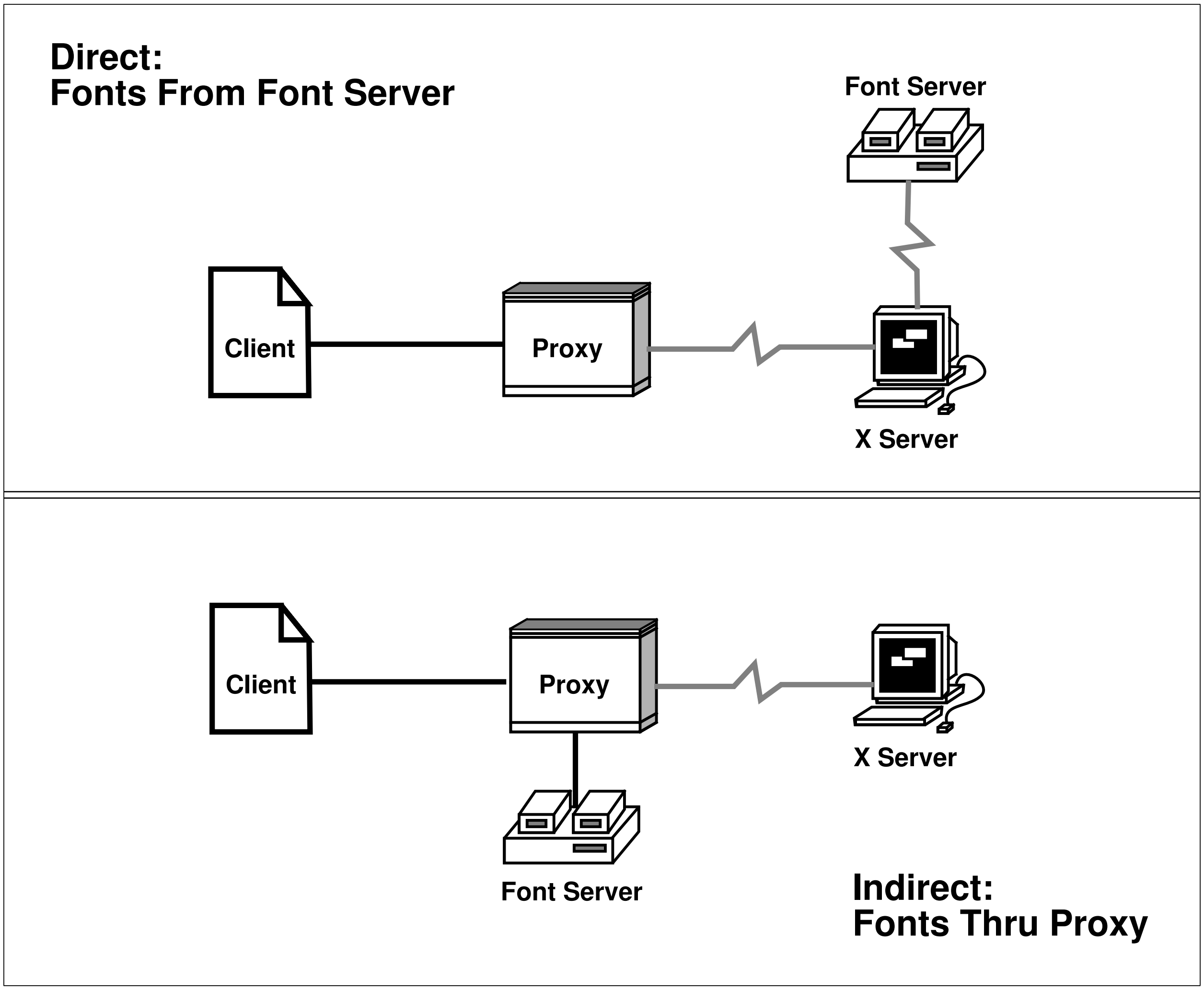


Figure 5: Direct and Indirect Font Access

Since the client believes that the proxy is the actual X server, it will generate authorization information encoded for the identity (*i.e.*, the address) of the proxy. If an outside service such as Kerberos is involved in the authorization process (typically to authenticate the sender of the data), the proxy will have to be involved. Furthermore, if the display wishes to somehow reuse the authorization information, it will probably have to forward its request through the proxy.

If communication with a font service is done through the proxy, font information can be used immediately without being retransmitted back from the display. But, it is unclear whether or not this savings (a little under 50% if glyph bitmaps are not fetched at the same time) justifies having font access requests in the LBX protocol. It would not only add redundancy, but complexity as well. Whenever the display needed font data, it would have to explicitly “call out” to the proxy to invoke a particular font service request so the proxy could match up the font service reply to a particular X request/reply (and the associated resource IDs).

When a font server with real licensing constraints is involved, including the proxy in the authorization loop adds even more complexity. For example, if the proxy is not authorized (can not get a license) to access the same font data as the display, there must also be a mechanism to ensure that the proxy and the server have the same sets of font available. Fingerprinting (computing some small value such as a checksum or hash value that can easily be transmitted and compared) is one approach that would allow the display and proxy to quickly verify that they have the same font, but this wouldl require support in the font server. Treating the font path as a shared resource, with caching, synchronization and notification of changes, is another approach. However, this doesn’t begin to solve the problem of fonts that are licensed on a per-use basis.

If the font server is on the same side of the serial line as the display, including the proxy in the font data access loop will just slow things down. If the proxy is making authorization decisions, the server may not have client-specific authorization data to access fonts that the client has licensed. A worse situation will arise whenever the proxy is able to access font services on networks that are not reachable by the display. The only options will be to have the proxy forward font requests and replies to the display, or to disallow access to such fonts.

These are only a few of the scenarios that can be constructed. This situation will almost certainly be resolved by policy, not technology; certain combinations will just not work with LBX. The trick will be to find the policy that is least astonishing to users.

Shadowing the Window Hierarchy in the Proxy

Another technique that would enable additional optimizations would be to have the proxy maintain a shadow copy of the window hierarchy. This would the proxy to generate events directly (especially **Expose** events, see below), and would allow faster response for **QueryTree**, **GetWindowAttributes**, **GetGeometry**, and **TranslateCoordinates** requests.

One way to implement this would be to assign ownership of various resources to a “token” that is passed back and forth between the proxy and the server. The holder of the token is allowed to modify the tree, and has the true data; the other side maintains only a cache of the data and must be aware that the held data may be invalidated at any time. So, for example, the proxy would keep ownership of client windows, but ownership for the root would remain at the server. This would allow queries about the state of a client window to be very quick, since no round trip to the server is necessary, but **TranslateCoordinates** would still be slow since the true information needed for that only lives at the server.

An extremely aggressive design would generate events for window tree changes at the proxy and suppress the transmission of the request completely.

So far, though, this discussion has ignored window managers running local to the display. By keeping ownership of the client window tree at the proxy, every interaction with the window manager that involves a change to the tree involves a round trip up to the proxy. This will reduce the responsiveness achieved by keeping the window manager close to the X server, but it is hard to predict by how much. More difficult is the fact that a reparenting local window manager may arbitrarily alter the client requests to change the window tree, and there is no good mechanism for the proxy to find out exactly what those changes were and reflect them in the true data.

Drag-and-Drop

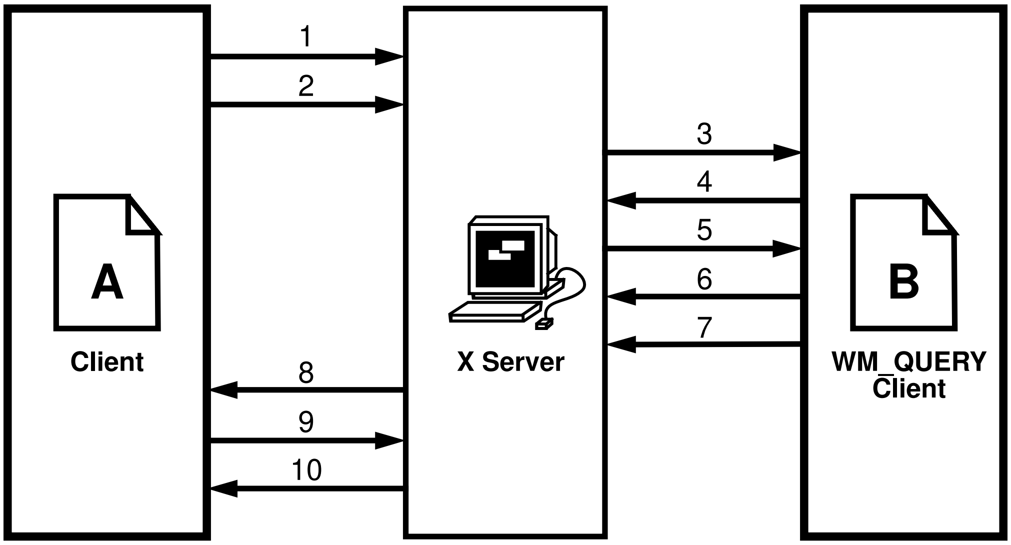
Direct manipulation of objects on the screen has become one of the important interactive feedback techniques in modern window systems. Allowing users to “drag and drop” windows, icons, graphics, and text has dramatically improved the usability of many graphical interfaces. The X Consortium is also working on standard that will allow applications that wish to be a source of data o locate potential destinations. The current proposal provides two alternative mechanisms:

- Preregistering: The source collects the geometry information at the beginning of the drag operation by reading a list of drop sites out of various windows, grabs the server, provides feedback to help the user select a destination, and finally notifies the destination when the drop occurs. The goal of this approach is to avoid having all the applications swap in as the drag moves over the screen.
- Dynamic: The source does a WM_QUERY of the window manager to find out where the cursor is. Then, as the pointer moves, the source does uses **SendEvent** to notify the destination that it should do feedback (see Figure 6).

It would be nice to insist that only the Preregistered mode should be used with LBX. Unfortunately, most applications will not have a window per drop object and will be forced to be Dynamic. This case will be important to consider since the initial sample implementation of a WM_QUERY handler will be a separate client that probably should be run on the side of the network closest to the client.

The main requests to optimize for better drag-and-drop performance are **ChangeProperty**, **SendEvent**, and **GetProperty**. Drag-and-drop doesn’t cause selection ownership to change, but it does use **ConvertSelection**. Although the proxy could conceivably track ownership of various selections, there is currently a debate within the X Consortium on whether or not a value of *CurrentTime* in a **ConvertSelection** request is supposed to be replaced by the server in the resulting **SelectionRequest** event that is sent to the owner of the selection. If so, this would force the proxy to track the server’s notion of the current time.

One of the results of the way drag-and-drop works is that there is at least one property read on every mouse motion. LBX could try to short-circuit the property reads by caching the contents of the properties, but it then must guarantee that the server cannot later generate an *Alloc* error for a request that the proxy has serviced locally. This could be done by having the server and proxy negotiate a pool of preallocated server memory that is guaranteed available to the proxy. The server may also need to regularly send updated timestamps and explicit notifications of success so that the proxy can resynchronize with the server.



1 **ChangeProperty** to set WM_QUERY property information
2 **ConvertSelection** to ask for data
3 **SelectionRequest** to notify owner of the WM_QUERY that data is desired
4 **GetProperty** to retrieve WM_QUERY property information
5 reply for previous **GetProperty** request
6 **ChangeProperty** to send selection data to requestor
7 **SendEvent** of **SelectionNotify** to tell requestor that data is ready
8 previous **SelectionNotify** forwarded to requestor
9 **GetProperty** to retreive selection data
10 reply for previous **GetProperty** request

Figure 6: Dynamic Drag-and-Drop Protocol

Several techniques can be applied to reduce the time spent waiting for a response:

- Caching property data in the proxy for steps 4 and 9.
- Deferred sending of property data to the server for steps 1 and 6.
- Caching the ownership of selections for steps 2 and 3.
- Caching the contents of client event masks for steps 7 and 8.

Thus, with a lot of work, it would be possible to do the entire drag-and-drop dialogue in the proxy. However, this would increase the complexity of fully featured LBX implementations; leaving out any one of these techniques implies that the dialogue must make at least one round trip to the server.

Motion Events

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 favored scheme is to use a liberalized variation of **PointerMotionHint**, in which the display and proxy take into account the typical latency and bandwidth available, and try to keep a reasonable number of motion events in flight between the server and the proxy. Some sort of credit-based acknowledgment would be 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 would add few enough events that it wouldn't cause much trouble if either side were to fall behind, and enough latency that under low-load conditions, the acknowledgment for the first motion event would be received at the display just in time to avoid dropping any events. If the load either way on the wire were to increase such that one end would fall behind (for example if the client were to paint a complex image in response to motion events), the acknowledgment would get delayed, and only some number of updates per second would be lost, rather than responsiveness (although the updates could track a few events behind).

Distributed Resource Allocation and Modification

As suggested in previous sections, in many situations, round trips to the display could be avoided if the proxy were able to allocate or modify resources directly. For example, if the proxy were able to take responsibility for allocating cells in a colormap, it could assign pixel values quickly. This would improve the start-up time of most applications.

One approach for allowing either the proxy or the server to take ownership of a given resource would be to provide a token that would be passed back and forth. When one side wished to modify a resource, it would ask for the token and begin its operations. Any changes would be sent to the other side at a later time, such as whenever there was extra space in the network or whenever the token changed hands.

In a more aggressive strategy, the proxy could cache the contents of a colormap and avoid sending requests to the server altogether. For example, if a client requested a read-only cell that matched an entry in the cache, the proxy could immediately return the information about that cell to the client, without waiting for a round trip.

Although this is a highly desirable effect, it is not without cost: The proxy is operating on cached data, so there is a potential race condition between a local client freeing the color cell and decrementing the server's reference count to zero, and the proxy telling the client to use that cell and sending a message that will increment the reference count again. For this to work, it will be necessary to either cache only those cells that are allocated through the proxy, or invent some synchronization messages around the last freeing of color cells. Analogous mechanisms will be needed for other resources.

Grouped Events

The X protocol relies upon events to notify applications of user input, changes in the state of the window hierarchy, and requests for information from other applications. Although events typically do not involve a lot of data, they are frequently sensitive to delays in delivery. For example, during the time between when an **Expose** event is generated and the resulting drawing requests are received, the display will appear to the user to be damaged. Human factors experts typically cite response times of 1/3 of a second as the threshold at which people begin to become distracted.

At 9600-baud, this is enough time to transmit 330 bytes which could reasonably be expected to expand to 500 bytes when uncompressed. Even ignoring the networking headers that will typically add 5 or more bytes to each packet (which is why it will be important to coalesce as many events as possible into a single packet), at 32 bytes per event this is only enough time to send 15 full-sized events. If the server were sending a large number of events at once (for example, exposures or configuration notifications to a large number of subwindows), it would impossible for the application to even receive all the events within 1/3 of a second. More efficient event encodings will help, but a new type of “grouped” event that the proxy could send to multiple windows would provide even faster delivery of events to applications.

The down side of grouped events is that would most likely require the proxy to track window geometry, clip lists, and other attributes. The proxy would have to either duplicate the efforts of the server or ask it to describe the results of any window manipulations.

Summary

The goal of the LBX effort is to produce a standard for running X over low bandwidth networks that incorporates the lessons that the industry has learned over the last several years. It will combine the best features of the existing technologies (particularly local clients and network addressability) with a variety of techniques for improving performance and providing room for future expandability.

If all goes well, the specification for the LBX protocol will be made available for public review during the summer of 1993, and a sample implementation will be provided with the release of X11R6 from the X Consortium at the end of the year.

References

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

[Flowers91] Jim Flowers, *The X Logical Font Description Conventions*, X Consortium X11R5, 1991.

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

[Herbert91] Kevin Paul Herbert, *Xremote and Terminal Servers*, Proceedings of the Silicon Valley Networking Conference, 1991.

[Rosenthal91] David S. H. Rosenthal, *Inter-Client Communications Conventions Manual,* X Consortium X11R5, 1991.

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