How Xlib is Implemented
(And What We're Doing About It)

Jamey Sharp
Computer Science Department
Portland State University
Portland, OR USA 97207–0751
jamey@cs.pdx.edu
http://xcb.freedesktop.org

Abstract

The X Window System is the *de facto* standard graphical environment for Linux and Unix hosts, and is usable on nearly any class of computer one could find today. Its success is partially due to its flexible, extensible design.

Unfortunately, as research proceeds on cutting-edge window system functionality, the brittleness of the underlying software is a critical impediment to progress. Xlib, the client-side implementation of the network protocol that underlies X, is one source of these issues. Many developers working on new features in the X protocol are discovering that Xlib requires changes to support these features, but Xlib makes those changes difficult. For more than 15 years, new features have been added to Xlib by accretion, rather than with careful design.

We discuss the implementation of Xlib and analyze some specific difficulties in it that cause problems in understanding and maintaining this code base. We also present our current work on migrating the X Window System to a more maintainable, carefully designed architecture.

1 Introduction

At the core of the X Window System [SG86] is a network protocol, allowing any number of X applications running on far-flung machines to interact with a single keyboard, mouse, and monitor. Nearly all applications currently use Xlib [SGFR92], a C library dating to the mid-1980s, to interact with the X protocol.

Software developers have collectively learned a lot about the engineering of software in the decades since X was created. This fact explains some, though not all, of the difficulties that users and developers experience when working with X. (Other issues are explained later in this paper.) Our previous work on a new X-protocol C Binding [MS01] and subsequently an Xlib Compatibility Layer [SM02] were efforts to apply current best practices in software engineering to the core of the client-side implementation of the X protocol, with goals of improving the usability of X in several cases:

- Resource-constrained environments, such as PDAs.
- Developers wanting to understand how X works.
- Developers implementing X protocol extensions.
- Users desiring better-performing applications.

Xlib spans more than 400 source files, contains more than 150,000 lines of code, and compiles to a roughly 750kB shared library on Linux/i386. On a typical Linux system, this puts Xlib among the top libraries as measured by code size.

This paper has several major parts: a tour of Xlib, an explanation of several current efforts to improve X, some observations on software engineering, and a brief glimpse at the future.

2 The Architecture of Xlib

As we are not aware of a comprehensive tour of the Xlib source, we present one here.

Xlib stores more than a kilobyte of data about each X server connection in a structure named a `Display`. This includes

- The file descriptor and other information about the transport.
- Other file descriptors to monitor for new data.
- An assortment of values cached from the server.
- Pointers used for internal memory management.
- Function pointers to hooks.

The hooks in Xlib allow extensions and applications to modify the way Xlib handles

- Thread synchronization.
- Conversion between wire protocol and C structures.
- XID allocation.
- Management of graphics contexts and server fonts.
- Buffer flushes.
- Connection close.

When an X client successfully establishes a connection to a server, the server sends several hundred bytes

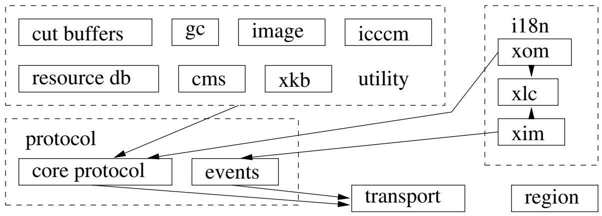

Figure 1: Notional Xlib components: arrows indicate "using" relationships

of information about the capabilities of the display. Xlib copies some of that information into the Display; the rest is kept in other structures pointed to by the Display.

Once connection setup is complete, requests from the client and responses from the server are the fundamental elements of the X protocol. Requests always have a "major opcode" identifying the kind of request, and a length field that measures the number of four-byte words needed to contain the entire packet. Responses come in three forms: replies, events, and errors. Replies and errors are sent in response to requests, while events are sent spontaneously.

Only some request types call for a reply, but for any of those requests, a reply is always sent, unless an error is sent instead. Errors can also occur for other requests. Since X has a network-transparent protocol, it may be run on high-latency connections such as dial-up or DSL Internet links, and on these links replies take long enough to arrive that the delay may have a noticeable effect on application performance. In analyzing this aspect of X performance, we speak of round-trip latency, and look for techniques to avoid or hide it [PG03].

Xlib was originally written without much concern for type-safety, but with great care to minimize the number of function calls. As a result, the C preprocessor gets heavy use when compiling Xlib. Many of the most commonly executed statements in Xlib are macros.

2.1 Xlib Layers

As shown in Figure 1, Xlib can be thought of as having several distinct layers and components. An analysis of the size of these notional components is in Section 6.2. Unfortunately, within the code the boundaries are not as clear as is suggested by the figure. In fact, we are not aware of any previous efforts to describe Xlib in this manner. However, these boundaries will be useful in our analysis of Xlib, and correspond to the vocabulary commonly used in conversation among Xlib developers.

2.1.1 Transport Layer

The transport layer is responsible for conveying requests and responses between the X client and server. This layer is independent of the semantics of those packets.

Much of the code in this layer comes from xtrans, a module comprising several `.c` and `.h` files. Code that uses xtrans instantiates it in one of several variants by defining an appropriate C preprocessor macro before including not just `Xtrans.h`, but also `transport.c`. Available variants are X11, XSERV, Xim, FS, Font, Ice, TEST, and LBXPROXY: while there are occasional subtle differences, these variants differ mostly in symbol names.

Xlib uses two variants of xtrans, X11Trans and XimTrans, meaning that all of the code in xtrans is linked into Xlib twice. The first variant is used in the transport layer, while the second is used by X input methods. We focus on the first of these here.

Nothing in the extension libraries and applications that we have tested uses X11Trans directly. Within Xlib, calls to X11Trans are confined to four source files: XlibInt.c, OpenDis.c, ConnDis.c, and ClDisplay.c. As a result, rewriting Xlib to eliminate references to X11Trans is a relatively straightforward task.

2.1.2 Protocol Layer

On a Cray supercomputer, memory is not addressable in single-byte increments: in fact, it is addressable only in 64-bit increments. In X requests and responses, however, 32-bit values are aligned to 32-bit boundaries, 16-bit values to 16-bit boundaries, and so on. On a Cray, then, accessing individual components of a request or response is inefficient.

Xlib is designed to unpack the wire protocol data from responses into structures that may be efficiently accessed by the host, and to similarly pack data into requests. Many parts of the protocol are represented by a pair of structures in Xlib, namely the wire protocol structure and the host structure. By convention, the names of wire structures are of the form x*ID*Req for requests and x*ID*Reply for replies, with various *ID*s, and in general components of these structures correspond only to arguments to functions. The core event wire type is xEvent and corresponds to host structures named X*ID*Event. The core error wire type is xError, and corresponds to XErrorEvent.

For all of the wire protocol structures, C preprocessor symbols with names like `sz_xEvent` are defined equal to the number of bytes that the structure occupies on the wire. Request structures also have their opcode (major if core, minor if extension) in `#defines`. Structures have fixed-length parts, but no variable-length representation.

When Xlib accesses a Display, perhaps to construct a request or process a response, it must protect the Display against simultaneous accesses by other threads. The LockDisplay and UnlockDisplay macros are provided for this purpose. All access to a Display must be bracketed by a LockDisplay and UnlockDisplay pair.

To deliver a request to the server, Xlib must allocate both a sequence number and a block of memory to construct the request in. Both of these allocations are handled by the GetReq macro or its variants: GetResReq, GetEmptyReq, and GetReqExtra. Any variable-length parts of the request are then delivered with the Data macro, its variants Data16 and Data32, or the _XSend function. The Data macro will copy into the output buffer if there is enough room, or call _XSend with its arguments otherwise. The _XSend function uses the writev system call to write both the buffer and the extra data in one system call, without further copying. Finally, if a reply is expected from the server, _XReply is called to wait for the round-trip to complete so the library can return the data to the caller.

The Display must be locked during this process for two reasons:

1. The block of memory is allocated directly from the output buffer of the Display, and then written to by subsequent instructions.
2. The "current sequence number" is stored in the Display. It is updated by GetReq, but then if a reply is expected, that number must remain constant until _XReply is called.

Xlib offers a feature called synchronous mode. The X manual page says,

> Since Xlib normally buffers requests to the server, errors do not necessarily get reported immediately after they occur. This option turns off the buffering so that the application can be debugged. It should never be used with a working program.

Inside of Xlib, this is implemented with a function pointer (synchandler) stored in the Display structure and called at the end of every protocol stub. (The function pointer is occasionally used for other purposes, and at those times the original pointer is saved in the saved-synchandler pointer.) The SyncHandle macro hides the details of that function call.

In the core X protocol, any drawing request requires a graphics context (GC) identifier as one of its parameters. GCs encapsulate a good deal of state that often should persist between drawing requests, and so are both convenient and bandwidth-efficient. A small challenge lies in creating a convenient C-language interface for setting the many properties of a GC, and Xlib provides a collection of functions allowing the application to set one property at a time. Of course, it would be inefficient to generate a request every time any of these functions is called, so Xlib only constructs a request modifying the GC when some other request that uses that GC is about to be constructed, and then only if the properties of the GC are truly different on the client than on the server. The FlushGC macro hides these details.

It is frequently convenient to have a standardized set of strings for inter-client communications, but preferable to compress the representation of the strings. The X server maintains a list of "atoms", which are numbers that are uniquely mapped to strings for the lifetime of the X server process. Applications may use the InternAtom request to get (or create, if necessary and desired) the atom associated with a string, and the GetAtomName request to get the string for a particular atom. Since most applications participate in a significant number of inter-client communications using a number of atoms, Xlib caches part of the mapping held by the server. The size of the cache is currently hard-coded to be 64 atoms.

2.1.3 Utilities Layer

From the point of view of the X protocol, any code not providing transport or stubs for requests and responses is utility code. Some functions contain both protocol and utility functionality. XPutImage is an example of this. In addition to delivering image data to the X server, XPutImage splits the image if it exceeds the maximum request length; byte-swaps individual pixels; and does other conversions as needed.

2.2 Xlib Modules and Interfaces

Since Xlib contains a wide variety of distinct interfaces and functionality, here is a list of the major components.

locking: Thread synchronization primitives, serving to protect the Display structure.

transport: Low-level communication with one or more X servers, encapsulating support for a variety of reliable stream protocols and handling buffering and connection setup.

core protocol: Stubs for all requests in the core X protocol.

cut buffers: Utility functions for manipulating selections, also known as cut buffers.

gc: Graphics context cache, allowing applications to change the many properties of a graphics context while only generating requests when the graphics context is used.

image: An assortment of utility functions for operating on client-side image buffers, including loading bitmaps from files.

events: Xlib provides a variety of event queue search functions, matching on criteria such as whether an event is related to a specific window.

icccm: The Inter-Client Communications Conventions Manual [Ros] standardizes such things as communicating window titles to window managers. Xlib provides utility functions for the operations documented in the manual.

region: The region data structure describes arbitrary sets of pixels. It has been re-implemented several times in different parts of the X Window System because it has broad utility; but in Xlib it is not useful for much more than setting arbitrary graphics context clipping regions.

resource db: Xt [AS90], the original X Toolkit, gets application preferences from the "resource manager database", which is stored as a property on the root window of the first screen of each server. Xlib provides the low-level support for this mechanism, although it is not generally used in new applications or toolkits.

xom: The X Output Methods provide support for displaying text encoded using ISO 2022, also known as ECMA-35. This standard, which predates Unicode, provides escape sequences for switching between character sets.

cms: A "color management system" (CMS) provides mechanisms for transforming colors between differing color spaces, including device-dependent spaces. With a CMS, users may calibrate scanners, printers, and monitors so that colors appear identical. Xcms [Ber95] can only use monitor calibration data.

xkb: The X Keyboard Extension generalizes the keyboard model of the core protocol. Xlib contains code for interacting with this extension if supported on the server and compatibility code for applications and servers unaware of XKB.

xlc: The X Locale implementation maps strings between a variety of encodings and formats. Supported formats include C strings, wide character strings, and UTF-8, among others.

xim: The X Input Methods allow a user to input text using alphabets not physically present on his or her keyboard. Japanese text, for instance, can be input by typing the phonetic pronunciation of a word and searching for a character with that pronunciation and the intended meaning.

3 Some Xlib Issues

Given skill at reading software written in C and a little understanding of the X protocol, almost any individual Xlib source file may be understood without too much effort. Difficulty in comprehending the whole is due primarily to bulk of Xlib. The broad scope of Xlib, together with the engineering needed to make it work on computers of the 1980s, led to a large implementation, and this implementation has grown dramatically with time. Also, since all of the source of Xlib was written by hand, revising design decisions that affect any significant part of the code is an exceedingly difficult task.

As a brief example, xtrans (covered in Section 2.1.1) is included in Xlib in two forms: X11Trans and XimTrans. These two forms produce nearly identical compiled code, so they are essentially redundant. They are compiled into Xlib using a technique strongly discouraged in C programming: the C preprocessor is used to include source from a .c file into another, nearly empty, .c file. It takes a good deal of time for even an experienced programmer to understand this particular component.

3.1 Inflexible Implementation

Xlib provides two supported ways of accessing information stored in the Display structure: functions and macros. This means that offsets in the Display structure must remain constant to maintain binary compatibility with any code using these macros. Unfortunately, the specification [SGN88, P12] says:

> The macros are used for C programming, and their corresponding function equivalents are for other language bindings.

Because of this advice, X applications, which are predominantly written in C and C++, generally use the macro variants of the accessors. They leave Xlib developers little flexibility to revise design decisions. For binary compatibility, neither the macros nor the data structures that they access can ever be changed.

This problem is aggravated by the fact that Xlib has traditionally installed the "private" header file Xlibint.h alongside the public header files intended to be used by applications and libraries. It was made available for the sole purpose of providing extension libraries access to functions, macros, and data structures convenient for processing X protocol messages. However, toolkit and application writers have taken advantage of the direct data structure access, and now some code depends on vagaries of Xlib implementation.

3.2 Unpredictable Requests

Immediately after connection setup, Xlib automatically generates several requests regardless of whether the application needs those steps taken. The steps are

- Setting up support for requests larger than 256kB.
- Creating a default graphics context for each screen.
- Retrieving the resource database.
- Initializing the XKEYBOARD extension.

If the server supports both the BIG-REQUESTS and XKEYBOARD extensions, then this process will block application startup for the duration of five round-trips to the server. On modern configurations, the resource database can be easily 30 times as large as the data returned by the server during connection setup.

This is only one example of the larger problem that Xlib generates X protocol requests that are not obviously related to the needs of the application. Another example is that it is impossible to use the public API of Xlib to send a GetWindowAttributes request without simultaneously sending a GetGeometry request. Between this sort of opportunistic request generation and the assortment of caches implemented in Xlib, an application cannot effectively predict what protocol requests Xlib will produce on its behalf. That, in turn, suggests that these features would have been more useful as an extra layer built on top of functions providing direct control over protocol generation.

3.3 Threading

One aspect of Xlib that is not at all straightforward is the support for multi-threaded applications. It is perhaps partly due to the complexity of this aspect that the only application we could find to test Xlib in a multi-threaded setting is ico, a sample program from the reference implementation. Whatever the reasons, the thread support in Xlib is poorly tested (though we have yet to be able to demonstrate any actual faults), almost never used, and very difficult to understand in depth.

This is unfortunate. Threads provide a powerful way to organize computation and I/O with a minimum of programmer effort. Major X applications like Mozilla use threads to great effect, yet are careful to make all Xlib calls from a single thread. Users see this as their applications freezing occasionally, for example while rendering a new web page in a browser.

This section provides a sample of confusing thread synchronization situations in Xlib, with explanations of how those situations work. This is by no means an exhaustive list, however.

3.3.1 XLockDisplay

Nearly every function in Xlib invokes the LockDisplay and UnlockDisplay macros. Between LockDisplay and UnlockDisplay, the function is permitted to make any changes to the state of the Display structure. This lock is implemented using a mutex.

In addition, the documented interface to Xlib includes XLockDisplay and XUnlockDisplay. The documentation says that XLockDisplay may be called multiple times from the same thread without deadlock, and XUnlockDisplay must be called an equal number of times before the display is actually unlocked. The UNIX98 standard calls this a recursive mutex; it is also known as a counting mutex.

However, XLockDisplay and XUnlockDisplay are not implemented using a recursive mutex. Instead, they use condition variables together with a non-recursive ("fast") mutex. Pseudo-code for this algorithm is provided in Figure 2.

A fair amount of experience at working with concurrent software is required to understand this algorithm. While it does seem to satisfy its specification, a clearer equivalent is desirable. It would be preferable to use a standard mutex to implement this counting mutex, and in fact we devised and tested an algorithm to do that.

```
_XInitDisplayLock
  level = 0

LockDisplay
  lock mutex
  while level > 0 && thread != self
    wait on condition variable

UnlockDisplay
  unlock mutex

XLockDisplay
  LockDisplay
  ++level
  thread = self
  UnlockDisplay

XUnlockDisplay
  LockDisplay
  --level
  if level == 0
    wake up all waiting threads
  UnlockDisplay
```

Figure 2: Xlib thread-synchronization primitives

Unfortunately, our algorithm still was not perfect under the metric of clarity. That algorithm had enough problems to fall back to an even simpler plan: just use recursive mutexes. The UNIX98 standard offers them, and using them reduces all four of the synchronization primitives in Xlib to single-line calls to pthread_mutex_lock or pthread_mutex_unlock.

3.3.2 GetReq and _XReply

Anyone reading Xlib sources could be forgiven for thinking that the constraints are very strict on functions that may be called in between calls to the GetReq family of macros and to the _XReply function. After all, _XReply discovers the sequence number that it is waiting for by checking the value in `dpy->request`, which was set by GetReq, so clearly nothing should be allowed to touch that value during that interval.

Now note that several functions call Data (Xlibint.h) between GetReq (Xlibint.h) and _XReply (XlibInt.c); Data may call _XSend (XlibInt.c); _XSend may call _XWaitForWritable (XlibInt.c); and _XWaitForWritable may release the display lock while calling select. It should seem disturbing that the display lock might be released during this period where `dpy->request` must not be touched.

```
#include <X11/Xlib.h>
#include <pthread.h>

static char atom_name[32768];

void *event_loop(void *arg)
{
  Display *dpy = arg;
  XEvent evt;
  while(1)
    XNextEvent(dpy, &evt);
}

void main(void)
{
  Display *dpy;
  pthread_t event_thread;
  Atom atom;
  int i = sizeof(atom_name) - 1;
  atom_name[i--] = '\0';
  while(i >= 0)
    atom_name[i--] = '1';
  XInitThreads();
  dpy = XOpenDisplay(0);
  pthread_create(&event_thread, 0,
                 event_loop, dpy);
  atom = XInternAtom(dpy, atom_name,
                     False);
}
```

Figure 3: Does this program work? (Yes.)

This situation is very difficult to reason about: The chain of calls is long, the functions are complex and far apart in the source, and the circumstances under which the display lock might be released are complicated.

Consider Figure 3. In this example we send a request, InternAtom, which will send back a reply; and we ensure that the atom name sent in the request is larger than the output buffer in Xlib (which defaults to 16kB). That will cause _XSend to be called before _XReply in XInternAtom. However, prior to sending this request, we ensure that XNextEvent is waiting in another thread to read any responses that become available on the wire.

What if the event thread reads from the connection while _XSend is trying to deliver the InternAtom request? Could XNextEvent read the reply accidentally? If so, it would not know what to do with it.

The trick is that the reply cannot come back until the entire request has been written, and _XSend was carefully written so that it would return with the lock held and without reading as soon as it finishes writing its single request. Therefore, no other thread has an opportunity to read the reply.

This leaves open the question of whether there is some other fault in this part of Xlib. For instance, perhaps GetReq could be invoked from another thread while _XWaitForWritable has the display unlocked, and perhaps the sequence number stream would be corrupted as a result. We continue to inspect the source of Xlib for cases like this.

4 Starting Over: XCB

We wanted a simpler, smaller base for X development than Xlib, so we wrote XCB. As we explained in [MS01],

> XCB is intended to be a simple and direct binding of X protocol transactions to C language function calls, with the minimum amount of machinery necessary to achieve this aim.

As a result, the XCB interface consists of little more than functions that send requests to the X server and a bit of machinery to handle the responses. This makes its interface much smaller than that of Xlib. The most noticeable benefits of this limited interface are that XCB has much less code than Xlib and a much simpler implementation. When built with reasonable compiler optimizations, XCB is 26kB compared with 750kB for Xlib.

Layers and components are organized in XCB in a manner similar to that presented for Xlib in Figure 1. In contrast with Xlib, however, the boundaries are more rigidly enforced, and in fact XCB offers only a minimal utilities layer. Most utility functionality is expected to be provided by separate libraries.

4.1 X Protocol Description Language

The C programming language is not ideal for the task of describing the X protocol. Patterns emerge in the code that cannot be eliminated using C language constructs, and logically related definitions are forced to be split between header and source files. These issues make maintenance difficult and cause problems for those attempting to understand the functioning of any particular protocol request.

We created a domain-specific language to describe the essence of the protocol. The immediate benefits are these: all information about a request can be found in one convenient bundle, and the implementation is easy to change without changing hundreds of protocol stubs by hand. Over the longer term, these protocol descriptions have further value because they may be reused in a number of ways, including but not limited to automatically producing

- Bindings for other languages.

- Protocol documentation.
- A text representation for use in debuggers like xscope and xev.
- Server-side protocol bindings.

This re-usability has been important in our work, and we discuss it in more detail in section 5.2.2.

4.2 Constructing Requests

In the initial development of XCB, we followed the lead of Xlib on a number of implementation choices. For the most part, these choices were harmless, but one example is instructive. Like Xlib, XCB needs to allocate blocks of memory for request construction. Like the GetReq family of Xlib macros, we initially allocated these blocks directly out of the output buffer. Everyone involved at that time thought this was a perfectly reasonable choice, if for no other reason than that it avoids copying data from some other block into the output buffer.

Eventually, however, we came to feel that this interface was excessively constraining on the implementation of the transport layer. It assumed that memory allocated for requests did not need to be deallocated, and it required that the output buffer be protected against concurrent access for the entire duration of request construction.

We decided instead to allocate request buffers on the stack. This had one immediate advantage: every protocol stub for XCB requires only one function call to deliver a request to the X server, unifying the functionality of GetReq, Data/_XSend, and _XReply. That function is called XCBSendRequest, and it hides the details of computing the length field of requests, including requests larger than 256kB.

In the end, replacing the GetReq model with XCBSendRequest simplifies the interface between protocol stubs and the transport layer of XCB and reduces the duration that locks are held and the number of function calls in many protocol stubs. This change even reduces the size of the compiled code by a small margin. A similar approach would have produced very good results for Xlib as well, at the cost of an extra function call for many protocol stubs, which in the past would have been considered prohibitive. Unfortunately, since Xlib protocol stubs are entirely hand-coded and use macros like GetReq that have all of the problems of inflexible implementation discussed in Section 3.1, changing the design of this part of Xlib at this point would require massive effort.

5 Porting Xlib to XCB

Given the many benefits of XCB, we would like to see applications and libraries use XCB as their low-level interface to the X server. However, nearly every X application in existence uses Xlib, whether through one of the many toolkits or directly. Quite a few of these applications, and some of the toolkits, are closed-source; in some cases the source has been lost. As for the open source X applications, there are too many to count, let alone port.

Clearly, some means of transitioning Xlib applications to XCB is needed. It should be fully binary compatible with existing Xlib-based libraries and applications while allowing XCB to manage the connection to the X server. In [SM02], we described such a library, and called it XCL: the "Xlib Compatibility Layer".

XCL was intended to be a drop-in, source compatible replacement for Xlib, adding an extra interface to allow access to the underlying XCB connection. As it turned out, binary compatibility was easy to achieve as well. The intent was that applications would be able to take advantage of some of the benefits of XCB without any modifications, and then use more features of XCB as portions of the source were ported. As a practical matter, this would mean that applications and libraries that use Xlib may be mixed with those that use XCB.

5.1 Some XCL Issues

The first attempted implementation strategy for XCL was to start from scratch, adding support for more of the Xlib interface as applications that actually needed it were discovered. This effort was not expected to result in a full re-implementation of the Xlib interface, which is much too large to create with a reasonable amount of effort. In fact, that issue was a show-stopper. Every part of the Xlib interface is probably used by some application, somewhere, and unless an XCB-based version of Xlib supports every one of those applications, the traditional Xlib must continue to be maintained in parallel. The duplicate effort inherent in such a plan led to the eventual dismissal of the XCL implementation strategy.

5.2 Some Lessons from Current Work

We are now on our second try, and are approaching the problem from the opposite side. Beginning with the full freedesktop.org implementation of Xlib, we are stripping out and re-writing small parts. This led within a few days to a prototype XCB-based library that supported the full Xlib interface, as well as having limited support for the extra XCL API. The tradeoff is that it is not optimal in code size, clarity, or other measures.

In the version of Xlib at freedesktop.org, the directory layout of the source is notably different than the traditional style. Developers at freedesktop.org have converted the build system from Imake to autoconf, automake, and libtool, and at the same time adopted a layout familiar to anyone working with modern open source software. Public headers may be found in `include/`, while internal headers and all source code are in `src/`. In addition, while all core X software has traditionally been maintained together in a single source tree larger than 600MB, at freedesktop.org it has been split into many smaller modules and several CVS repositories. Xlib is in the `X11` module in the `xlibs` repository.

5.2.1 Transport Layer

Our current work on Xlib began with a focus on the transport layer.

Because of the mismatch between the internal architectures of Xlib and XCB, we provide a minimal set of hooks in XCB so that Xlib may make use of the transport layer of XCB in place of xtrans. (This interface remains a work in progress; we hope to find a clean design that could be useful to callers besides Xlib, but such a design is not yet apparent.) These hooks enable XCB to replace xtrans in a manner transparent to everything other than four Xlib transport-layer source files.

5.2.2 Protocol Layer

Replacing the transport layer with XCB provides notable improvements in code size and clarity, as we report in section 6.2. Yet replacing the protocol layer with XCB offers much more significant gains, at the cost of quite a bit of additional development effort. Fortunately, a significant part of this extra effort is already done: The old XCL work amounted to little more than glue code between the core protocol interfaces of Xlib and XCB.

Many functions in the protocol layer serve exactly the same purpose as their counterparts in XCB. In XCL, we built part of the infrastructure needed to automatically generate the code for these functions. That infrastructure combined information from several sources to produce its stubs:

- The descriptions of the protocol from XCB.
- A machine-readable description of the Xlib interface.
- A little bit of hard-coded knowledge about how to convert between data types used by XCB and Xlib.

This illustrates one benefit of encoding knowledge, like the structure of the X protocol, in a domain-specific language. If done well, that knowledge is reusable in a variety of projects.

Other functions require careful inspection when porting them to XCB, and are not expected to benefit from automatic code generation.

6 Results

We have tested our version of Xlib on real workloads, including:

- Mozilla, with the Xt-based plug-in Adobe Acrobat Reader.
- Many standard KDE and Gnome applications.
- A variety of window and display managers.

We also ran part of the X Test Suite, which is a comprehensive test suite of Xlib and the X server, created from the well-documented specifications for both.

The three metrics of interest in comparing traditional Xlib with an XCB-based Xlib are

- Correctness: has the behavior of the code changed?
- Code size: do any changes in code size justify the effort required to achieve them?
- Performance: what effect has the work had on speed?

6.1 Correctness

For our real workloads, we initially found quite a few bugs both in our version of Xlib and in XCB. However, a rapid series of small fixes resulted in the ability to run a complete desktop environment using XCB. This software is still in the debugging phase, but that phase is mostly done and proceeding well.

6.1.1 Observed Causes of Bugs

To transform Xlib into a library built around XCB, the semantics of Xlib must first be well understood, so that those semantics may be maintained. For reasons typified by the examples in Section 3, that task alone is nontrivial. Most bugs in the new version of Xlib result from failures to understand the intended semantics of functions that we have replaced. Naturally, the remaining bugs are due to a failure to correctly re-implement the original semantics.

6.1.2 X Test Suite

The X Test Suite reported that there were some defects in error handling. Unfortunately, when tests that failed were re-run individually, they succeeded. It remains unclear where the bug lies, but failure to use the test suite in the manner for which it was designed seems like the most probable suspect.

6.2 Code Size

For each source file, the number of lines of code (LOC) were measured by running the file through the C preprocessor, eliminating blank lines and lines from .h files, and counting the remaining lines. This approach was taken because preprocessor conditionals have a significant effect on the number of lines compiled into Xlib, and because it accounts nicely for the inclusion of xtrans. The number of bytes of compiled code due to each source file was computed by examining object files intended for a statically linked library, which do not have the extra code generated for position-independent code (PIC). (The overhead of PIC was deemed uninterest-

Component	Xlib						XCB	
	LOC	%	Δ LOC	bytes	%	Δ bytes	LOC	bytes
locking	125	0.20	-166	1109	0.14	-1348	n/a	
transport	2100	3.35	-2261	17886	2.24	-21264	1005	11250
core protocol	6003	9.57	-560	58839	7.36	-3683	2619	19665
extensions	no change						1442	15080
Total	62702	100.00	-2987	799678	100.00	-26295	5066	45995

Table 2: Code size for Xlib with XCB

Component	LOC	%	bytes	%
locking	291	0.44	2457	0.30
transport	4361	6.64	39150	4.74
core protocol	6563	9.99	62522	7.57
cut buffers	99	0.15	599	0.07
gc	370	0.56	2494	0.30
image	1027	1.56	9550	1.16
events	1112	1.69	6762	0.82
icccm	1160	1.77	8847	1.07
region	1290	1.96	9504	1.15
resource db	2554	3.89	65719	7.96
xom	2891	4.40	24938	3.02
cms	6478	9.86	62188	7.53
xkb	10824	16.48	104430	12.64
xlc	12323	18.76	312602	37.85
xim	13615	20.73	106578	12.90
Other	731	1.11	7633	0.92
Total	65689	100.00	825973	100.00

Table 1: Code size for traditional Xlib

ing for this analysis.) These object files were built for Linux/x86 without optimization. The Unix `size` command was run on each of these object files, and the value from the "dec" column was taken, which includes code, string literals, and any other data. Finally, each source/object file pair was assigned to a component to produce summary results per component.

In Table 1, the number of lines of code and compiled bytes are given for traditional Xlib in two forms: raw, and as a percentage of the total for that library. The revised version of Xlib, together with XCB, is covered in Table 2. In that table, the number of lines of code and compiled bytes are given in raw and percentage form as before, plus the change (Δ) relative to traditional Xlib. Additionally, lines of code and compiled bytes are given for XCB in raw form. Xlib components unaffected by this work were omitted from Table 2.

XCB provides a substantial improvement in code size to the transport layer of Xlib, and in the future is expected to do the same for the protocol layer and extension implementations that have been built on top of Xlib. Yet these improvements comes at relatively little cost in human programmer time: much of the code is automatically generated from straightforward declarative descriptions. Our current work has added less than 700 lines of new hand-written code to Xlib, while making thousands of existing lines irrelevant.

The core protocol layer of XCB was automatically generated from 1,700 lines of protocol description. The current 55 automatically generated Xlib stubs that delegate to XCB average 7 lines each, for a total of 442 lines. Generation of more stubs is planned.

Similar benefits await client-side extension implementations. Current extension implementations span the protocol and utilities layer for the same reasons that the core protocol implementation does. As a result, some portions of these extensions will be good candidates for automatic code generation by the same techniques that we have used in Xlib, and other portions will gain smaller benefits through hand-porting.

XCB itself probably also has opportunities for reduction in code size. We can experiment with different implementations easily, because nearly 80% of the code in XCB is generated automatically. By happy accident, the introduction of XCBSendRequest – made to improve modularity and code clarity – also reduced code size by a small yet noticeable margin. However, the XCB code base is so small already that we have put in only cursory effort to find further savings there.

6.3 Performance

User-visible performance with Xlib is expected to be largely unchanged by the transition to XCB. Rewriting Xlib to use XCB has little effect on those portions of Xlib intended to improve performance, such as the various caches. Patterns of communication between the client and the server should be identical in most cases to traditional Xlib. Xlib cannot use the latency hiding features of XCB to re-order requests and still remain within the constraints of the documented Xlib interface. Some slight performance improvements might be anticipated due to better cache utilization and reduced lock contention, but this is not expected to be significant. Informal testing supports the hypothesis that the difference between traditional Xlib and XCB-based Xlib is not apparent to end-users.

7 Related Work

At freedesktop.org, others are currently doing experimental work toward redesigning the utility layer of Xlib. This layer is a prime target for code size improvements, because it

- Is more than 80% of Xlib.
- Is more clearly separable into components.
- Overlaps most with common toolkit functionality.
- Contains the least frequently used code in Xlib.

Current efforts focus on the xim, xom, xlc, cms, and xkb components. Together, xim/xom/xlc make up about 44% of the lines of source in Xlib, and about 54% of the compiled size. The cms component contributes roughly 10%, and xkb contributes 15%, to the size of Xlib. The present build system allows Xlib to be built without each of these components, producing a build of Xlib that is about 75% smaller.

According to Jim Gettys [Get03], color management was broken in the XFree86 [xfr] implementation of Xlib for about half a year, and nobody noticed. Apparently that code goes unused.

Unfortunately, some applications and libraries that are in common use do depend on some of this functionality. For instance, Gdk 2.0 uses the xlc component to set window titles. For this reason, entirely removing that code from Xlib is not currently feasible. Fortunately, the character set translation tables that occupy a significant portion of xlc are no longer necessary, as more general libraries such as GNU libiconv provide the same services. One project presently awaiting developers is to remove these tables from Xlib and replace them with whatever implementation of iconv is available.

8 Software Engineering Observations

A number of observations may be made about software engineering in general, illustrated by the examples of Xlib and XCB. These observations have been made often in code style guides and software engineering publications, yet code continues to be written that exhibits these problems.

Remember that software is written for two audiences. While a computer must be able to execute the software, it is also necessary that humans be able to read, understand, and modify that software. Much of software engineering comes down to dividing software into manageable chunks, pieces that a human can keep entirely in his or her head long enough to work with them.

Functions should be kept simple, possibly by dividing complicated tasks into several simple functions. Functions that interact strongly should be kept close together, preferably in a single source file. Interactions can and should be weakened where possible through careful modular design, giving callers few opportunities to make mistakes. All of these principles are violated, for example, by the design of GetReq, Data, and _XReply, as explained in Section 3.3.2.

When multiple functions have similar code, a new parameterized function should be created that is the union of the similar blocks. For example, Xlib protocol stubs always call _XSend/Data, _XReply, both, or neither after GetReq. A single parameterized function, similar to XCBSendRequest, would have been better; some reasons were given in Section 4.2.

Functionality like the macros in the C preprocessor should be avoided in modern code. Some reasons for this were given in Section 3.1. Even automatic generation of code, a technique with significant benefits including those described in Sections 4.1 and 5.2.2, has hazards of this sort and should be used with the caution that the generated code should not have redundant similar blocks if the underlying language provides a reasonable mechanism for abstracting them.

Any time a significant chunk of software performs an independent task, that chunk should be an independent module, perhaps encapsulated in its own library. As libraries like XCB and libiconv demonstrate, Xlib contains many components that would have value as stand-alone libraries. Each module and library in a system should be focused on a single reasonably-sized task; have a minimal, orthogonal, and well-defined interface; and be implemented in a readable and maintainable manner.

9 The Future of Xlib

Given the benefits of XCB, new X toolkits and applications are anticipated that will use pure XCB rather than Xlib. Legacy Xlib code is expected to slowly migrate to mixed Xlib/XCB and eventually pure XCB. Development of Xlib is expected to slow. Even though the work described in this paper is not ready for widespread release as of this writing, there are already signs that developers are, indeed, moving their focus to XCB.

Xlib development will certainly not cease for some time yet, and is expected to focus on reducing the installation footprint of Xlib. A small number of new features continue to be planned, however. Since the reference implementation of Xlib is open source, Xlib is sure to be supported and maintained until it has no more users.

10 Conclusion

We have identified some significant areas of inefficient and confusing design and implementation in Xlib, and presented our efforts to repair this core element of the X Window System. Combined with the efforts of others, we believe that the installation footprint of Xlib may be reduced significantly, while the clarity, maintainability, and extensibility of the X client library stack are improved tremendously.

Guiding our efforts are current best practices from the software engineering and formal methods communities, and our work may be taken as a case study in the practical value of these techniques. For that matter, this work would be completely infeasible if the reference X Window System implementation were not open source, and illustrates one of the many benefits of an open development model.

This is an ongoing process. The X Window System has shown an unlimited capacity for extension and innovation. The general techniques of careful modular design, domain specific languages, and others are broadly applicable. We hope we have provoked interest in software engineering in general, and development of the X Window System in particular.

Availability

XCB and the version of Xlib described here are both hosted by freedesktop.org. XCB is available from `http://xcb.freedesktop.org`. Xlib source is at `http://freedesktop.org/Software/X11`, and can be compiled to use XCB by specifying the `--with-xcb` configure option.

Acknowledgements

This paper was shepherded by Carl Worth, and his patience and insight have been greatly appreciated.

Many thanks to Keith Packard for helping us comprehend Xlib and the X Window System. We would not have gotten this far without him.

Contributions by Professor Bart Massey have been invaluable as well, providing much-needed guidance and insight for the design and implementation of XCB, as well as mentoring while writing reports on our progress.

Thanks also go to Jim Gettys for his continued support of our efforts.

Finally, Sheridan Mahoney and Mick Thomure provided valuable feedback on drafts of this paper.

References

[AS90] Paul J. Asente and Ralph R. Swick. *X Window System Toolkit: The Complete Programmer's Guide and Specification*. Digital Press, Bedford, MA, 1990.

[Ber95] David T. Berry. Integrating a color management system with a Unix and X11 environment. *The X Resource*, 13(1):179–180, January 1995.

[Get03] Jim Gettys. Size of Xlib..., October 2003. Web Document. URL `http://pdx.freedesktop.org/pipermail/xlibs/2003-October/000001.html` accessed April 8, 2004 04:30 UTC.

[MS01] Bart Massey and Jamey Sharp. XCB: An X protocol C binding. In *Proceedings of the 2001 XFree86 Technical Conference*, Oakland, CA, November 2001. USENIX.

[PG03] Keith Packard and Jim Gettys. X Window System network performance. In *FREENIX Track, 2003 Usenix Annual Technical Conference*, San Antonio, TX, June 2003. USENIX.

[Ros] David Rosenthal. Inter-Client Communication Conventions Manual. In [SGFR92].

[SG86] Robert W. Scheifler and Jim Gettys. The X Window System. *ACM Transactions on Graphics*, 5(2):79–109, April 1986.

[SGFR92] Robert W. Scheifler, James Gettys, Jim Flowers, and David Rosenthal. *X Window System: The Complete Reference to Xlib, X Protocol, ICCCM, and XLFD*. Digital Press, third edition, 1992.

[SGN88] Robert W. Scheifler, James Gettys, and Ron Newman. *X Window System: C Library and Protocol Reference*. Digital Press, 1988.

[SM02] Jamey Sharp and Bart Massey. XCL: An Xlib compatibility layer for XCB. In *FREENIX Track, 2002 Usenix Annual Technical Conference*, Monterey, CA, June 2002. USENIX.

[xfr] The XFree86 project. Web document. URL `http://www.xfree86.org` accessed April 8, 2004 04:30 UTC.