XCB: An X Protocol C Binding

Bart Massey Jamey Sharp
Computer Science Department
Portland State University
{bart,jamey}@cs.pdx.edu

October 6, 2001

Abstract

The standard X Window System binding for the C programming language, Xlib, is a very successful piece of software across a wide range of applications. However, for many modern uses, Xlib is not an ideal fit: its size, complexity, difficulty of extension and synchronous interface are particularly problematic.

The XCB "X Protocol C Binding", currently under development, is a response to these and other concerns. XCB is intended to be a simpler and more direct binding of protocol objects to C functions; careful design of the XCB API and internal data structures and thorough modularization provides a solution that is size and time efficient, maintainable, easy to use both by single-threaded and multi-threaded applications and easily extensible. Some features of Xlib have been sacrificed in achieving these goals, notably i18n support and convenience buffering and caching features, but the cost will likely be worth it for toolkit developers and for those looking to write specialized applications layered atop the XCB API.

1 Xlib

Perhaps the oldest software in modern X Window System [SG86] distributions is Xlib [SG92]: the oldest files in the current XFree86[1] distribution have a 1985 copyright. Xlib has played an important role in the development of the X Window System in several ways: as the standard binding of X protocol requests and responses to an API for the C programming language [KR78]; as the repository of a number of performance optimizations of the C API, such as caching and grouping of requests; as a base for GUI toolkit implementations; and as a focal point for convenience routines and extensions necessary to build functional, internationalized and standards-conforming standalone programs without toolkit support.

Unfortunately, these roles have been somewhat contradictory, and the long history of Xlib has not generally worked in its favor. The current Xlib suffers from a number of outright defects.

1.1 Xlib Features

Xlib has a number of notable features, resulting both from an excellent initial design and the maturity attained through 15 years of large-scale use. Perhaps the most important is its completeness: the XFree86 Xlib release includes support for every protocol request provided by X11R6 and libraries for every protocol extension request supported by XFree86, as well as support for a rich set of convenience features. Among these convenience features are automatic marshaling of multiple requests (for example, combining a series of `DrawPoint` requests into a `PolyPoint` request), automatic decomposition of large requests and tools for automatic input canonicalization and internationalization (i18n) of both input and output.

The large-scale usage of Xlib has also made it a thoroughly use-tested piece of software. It has been carefully maintained, and its reliability is currently quite high. In particular, its code embodies many

[1] http://xfree86.org

clever solutions to subtle problems in X protocol interaction. Since the protocol has co-evolved with Xlib, both have been adjusted to work well together.

The X protocol is bandwidth efficient. Requests, replies and events are parsimonious in size. The simple representations of the protocol allow most transactions to be easily compressed by a general purpose method such as `gzip` [Gai93], further improving performance. Xlib contains a number of optimizations designed to improve this bandwidth efficiency. Notable among these are buffered output of requests[2] and a reasonable amount of local caching of server state.

The choice of the C programming language as the standard binding language for X clients was a natural one, given the C server implementation and the efficiency and accessibility of the language. It has since proven a fortuitous decision. Because most other programming languages provide an interface to C source or binaries, Xlib has been used as a binding by toolkits and programs written in languages ranging from C++ and Java to Scheme and Prolog.

1.2 Xlib Areas For Improvement

While Xlib is highly successful, there are still areas that could use improvement. Xlib is arcane: even experienced X programmers must frequently refer to the documentation in order to use it successfully. This is due to several factors: first, the difficulty of building a transparent API in C; second, accretion; but third and especially the fact that Xlib tries to be a general purpose interface suitable for writing everything from small toolkit components to large applications directly.

Because of XLib's generality and emphasis on convenience functions, the library is complex and oversized; little of its code is regularly used by applications. Work by Jim Gettys[3] has recently reduced this problem, but ultimately it seems difficult to seriously shrink Xlib without significantly changing the API and excising substantial parts of its functionality.

Because of the size, complexity and ubiquity of Xlib, it is quite difficult to maintain, especially to extend. Potential authors of new X protocol extensions are often deterred not by the difficulty of the server-side work, but by the difficulty of adding library support for the extension: there is little support in Xlib for extensions, so a great deal of new code typically needs to be written on the client side. This may be a major factor in the dearth of new extensions over the last 10 years or so.[4]

Xlib can also be difficult to use. For example, many X protocol items are XIDs, small integers. Unfortunately, C provides no way to declare incompatible types isomorphic to small integers. This occasionally leads to type errors in Xlib usage that are not statically detected: passing a window ID where a font ID was required, for example. The use of C structure and union types to "wrap" small integers can solve this problem, but in 1985 a few C compilers still had trouble treating structures in the largely first-class fashion required by the standard: structure return was particularly problematic, and some compilers even had problems with structure copying.

As another example, user memory management of Xlib data is complicated by a couple of factors. First, because the `XAlloc()` and `XFree()` interfaces are used instead of their normal `malloc()` and `free()` counterparts, traditional memory allocation debugging and leak-detection tools are difficult to use to detect and diagnose Xlib usage errors. This may be one of the reasons why X applications so commonly leak storage. Another is that Xlib routines occasionally return allocated storage containing pointers to other unallocated storage: it is the responsibility of the Xlib user to free the referents before freeing the referencing block, by calling the appropriate destructor routine[5] rather than `XFree()`. Needless to say, this is error-prone, and

[2] Xlib contains a support for a timer that periodically flushes the output stream to reduce latency. While XCB takes the position that this is an activity better performed by the API client, which may be able to do a better job with reduced complexity, this mechanism in any case does help control request latency.

[3] Personal communication, June 2001.

[4] Keith Packard, personal communication, August 2001.

[5] Often the destructor's name contains "destroy" rather than "free", adding to the confusion.

the resulting errors are difficult to detect.

The design goals of Xlib are somewhat contradictory. Modern toolkits such as Gtk and Qt eschew most of the special features of Xlib, such as its complicated input model and i18n semantics, and use just the protocol binding. This is not merely wasteful: interference from Xlib also makes it difficult to do certain styles of toolkit optimization, such as latency hiding and the use of multiple threads.

While Xlib attempts to be re-entrant, its complexity and the "retrofitted" nature of the reentrancy support make exploiting this feature difficult. In particular, the Xlib API is not easily suited to thread-based use. For example, it is difficult to obtain the sequence number of a request in a multi-threaded environment, as the sequence counter may be advanced by a second request between the time the first request is sent and the counter is queried. In addition, many Xlib calls that retrieve information, such as GetNextEvent, come in two basic forms: a blocking form that locks critical data structures and a non-blocking form that is subject to races with other threads.

While Xlib and the protocol are bandwidth-efficient, Xlib encourages a style of programming that tends to have high latency. Because Xlib requests requiring a reply are generally synchronous, blocking until the reply is available, Xlib often blocks for a full round trip time. This is not generally an issue when the X connection is local, and even for remote connections a round trip can often be avoided due to XLib's extensive caching. However, Packard's recent critique of LBX [Pac01] has shown that latency, rather than bandwidth, is the major contributor to poor remote performance of X applications, and that Xlib is a real contributor to this excess latency.

2 XCB

When faced with multiple conflicting design goals and excessive design complexity, one good design solution is often to modularize the implementation, separately implementing each desired piece of functionality. The design of XCB takes this approach. 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.

It is assumed that the principle clients of XCB will be of two types: toolkits intended for higher-level program implementation and small libraries intended to ease the direct use of XCB. Thus, certain constraints of the Xlib design disappear. Features such as i18n and most caching, that can be better managed at a higher layer, may be eliminated. Controlling the syntactic details of the API interface also becomes slightly less critical (although the current design seems rather pleasant in this regard).

2.1 XCB Structure

The basic structure of XCB is in two layers, as illustrated in Figure 1. A lower layer, XCB_Connection, supports the establishment of an X server connection and handles buffering and batching of requests and responses. XCB_Connection exports a simple API to the upper XCB_Protocol layer. XCB_Protocol, in turn, provides a quite direct C API for the core X Protocol. The ability for extension client code to sit atop XCB_Connection, together with the automatic code generation features discussed in Section 2.3 below, should make adding extension support quite easy.

A key feature of XCB is the thread-safety of the entire API. This is implemented via the locking mechanisms of POSIX Threads [NBF96]. Specifically, each connection is locked against concurrent access with a `pthread_mutex` and blocking during API calls is supported by `pthread_cond` condition variables.

The XCB API allows (indeed, encourages) a style of interaction in which one thread makes requests and handles replies and another thread processes events. The availability of locking mechanisms for thread safety was difficult to ensure 15 years ago. Now that threads are widely available to C programmers, they should probably be taken advantage of: the standard window systems in the Smalltalk and Java environments, among others, have exploited this approach with notable suc-

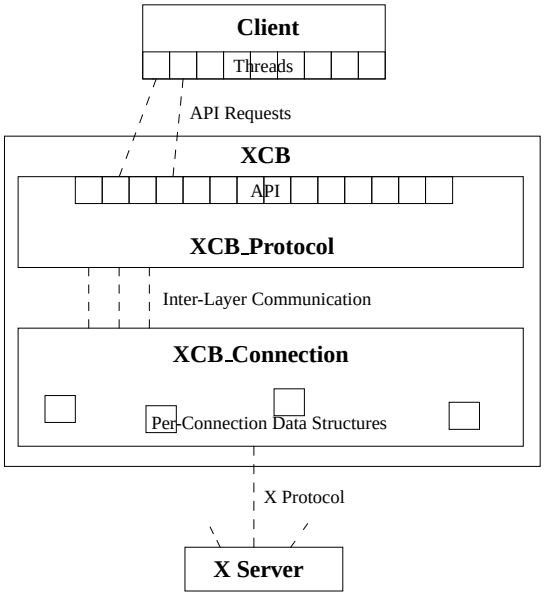

Figure 1: XCB structure and usage.

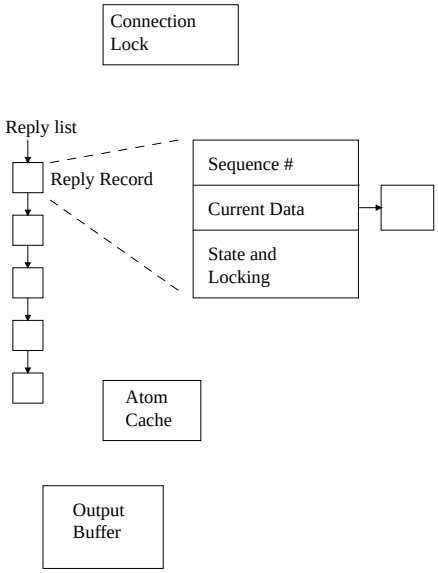

Figure 2: XCB data structures.

cess. The XCB API also permits a more traditional single-threaded "event loop" style: this enables single-threaded usage of XCB, and should allow re-implementation of the Xlib API atop XCB if desired.

Central to the XCB API is the use of "reply cookies" to permit latency hiding. Protocol requests requiring a reply do not block. Instead of returning reply data that is not yet available, the XCB nonblocking request returns a reply cookie, that can be converted into the reply data on demand, blocking if it is not yet available. This mechanism eases latency hiding without greatly distorting expected C API calling conventions, easing the X latency problems described in Section 1.2.

2.2 XCB Data Structures

XCB has a reasonably simple set of data structures and interfaces that interact to provide the desired functionality. Figure 2 gives an overview of these data structures and their interaction; the rest of this section attempts to describe some of the more interesting details.

The fundamental unit of interaction in XCB is the server connection. For thread safety, there are no global variables in XCB: all state for a connection is stored in an opaque, locked connection structure. Elements of the connection structure include

- The connection mutex.
- The socket for the connection.
- The current request sequence number.
- The request queue, a simple character buffer for outgoing requests.
- The reply list, that tracks the state of pending requests requiring a reply.
- The event queue, containing processed but unconsumed events.
- A dictionary to cache interned atoms. This is currently the only non-mandatory client-side caching performed by XCB: it avoids a common cause of high latency and bandwidth, and is easy to add. (Connection startup data is also cached, but this is essentially mandatory.)

For user compatibility with Xlib applications, a couple of its external formats are supported, namely the `DISPLAY` and `XAUTHORITY` environment variables and the `.Xauthority` file. Nonetheless, the XCB notion of a connection is somewhat

different than that of Xlib. In particular, there is expected to be just one connection per display: there is no special notion of a "screen" in XCB outside of the root window concept supported by the protocol.[6]

The current XCB implementation supports TCP/IP and UNIX-domain sockets: obtaining a socket is treated separately from creating a connection, so other types of file descriptor may be used as well, although some applications may require sockets with specific capabilities (such as non-blocking I/O).

The request queue is, as noted above, a simple character FIFO used to hold formatted requests for grouping into larger packets via a mechanism of write batching. (This is necessary for XCB TCP/IP sockets, since the Nagle mechanism [Nag84] is turned off: a latency-reducing technique borrowed from Xlib.) There are three conditions under which the lower layer will flush the request queue:

1. The request queue is about to become over full.
2. The user has requested a flush via the XCB_Protocol API.
3. The user has requested the reply to a request enqueued but not yet delivered.

This third reason is the most interesting, reflecting the fact that, unlike in Xlib, batching a request requiring a reply does not require flushing the current buffer.

The reply list of an XCB connection is a reflection of XCB's asynchronous reply delivery: replies arrive in request order but may be accessed out-of-order. The reply list is a simple singly-linked list of `malloc`-ed reply records: this data structure suffices since it is expected that a small number of replies will normally be outstanding. A reply record is created and placed in the reply list when a request requiring a reply is enqueued. The reply record will remain on the list until the reply is received and delivered to the user, at which point it will be freed by the upper layer of XCB. Thus, at most one thread can receive the reply to a request (although it need not be the thread from which the request was issued). Reply records that are of no further interest to the XCB user may (and should) be discarded by sequence number via the XCB_Protocol API.

Each reply record contains just a few pieces of critical information:

- The sequence number of the request.
- An indication of the request's status. A request in the list is either pending a reply, pending delivery of a reply to the XCB user, or both.
- A pointer that points to one of two quantities, depending on whether a reply has been received.
  - If a reply is pending, the reference is to a block of request-specific data allocated by XCB_Protocol and needed to construct the result delivered to the user (for example, the address of a user-supplied buffer into which a portion of the reply is to be delivered).
  - Once the reply has been received, this pointer will be converted by the upper layer to a pointer to the reply itself, awaiting delivery to the user.
- A condition variable on which at most one user thread may block when requesting a result before the reply has been received. The user thread will then be awakened when the result it desires has been generated from a received reply.

In addition to requiring careful management of replies, the asynchrony of reply and event delivery also mandates the implementation of a simple event queue. In order to receive a specific reply desired by the user, the lower layer may have to read past event responses sent by the server between the time the request was received and the reply sent. For a single-threaded XCB client, this presents a

[6] In fact, the screen portion of the DISPLAY environment variable is reported to the client by the XCB convenience routine, but is otherwise ignored.

problem. XCB cannot block until the reply is read from the input stream: it may deadlock the client. However, a single-threaded XCB client cannot process intervening events until control has been returned to it. While it would be possible to have the XCB reply-access API return an `EWOULDBLOCK`-like value, this would complicate both the API and the client. Instead, XCB enqueues events in a simple FIFO limited only by available memory. For a multi-threaded client, events will be delivered to a waiting event-handling thread rather than being enqueued.

The reply cookies described in Section 2.1 are implemented as transparent structures containing a single element: the sequence number of the referenced request. There is one such structure for each expected reply type, which helps solve the type-compatibility problem described in Section 1.2. Because the sequence number of a request is visible in the reply cookie, the application can correlate a server-side error with the request that produced it. For this reason, as well as for uniformity, requests not requiring a reply also return a reply cookie. However, this cookie is of a structure type for which no further processing is available in the API. Hence erroneously expecting a reply to a request that will not produce one is an error that should be detected at compile time.

One subtlety of the X protocol is that many requests are "pseudo-synchronous". Instead of returning a reply (and thus requiring a round trip) with the server-side ID of a newly created resource such as a window or font, a numeric ID, known as an XID, is created on the client side and passed to the server with the request. A careful specification of the ID space ensures that the ID is globally unique for the server. This latency-hiding optimization is concealed by Xlib, where XID generation is implicit and pseudo-synchronous and synchronous API requests are syntactically indistinguishable.

XIDs are explicitly requested as part of the XCB API: a created XID is wrapped in a structure or union and must be passed with the correct type to the XCB request API. An interesting type-safety issue arises here: some X protocol requests accept "subtyped" requests. For example, some protocol

```
typedef struct XCB_Pixmap {
  int xid;
} XCB_Pixmap;

typedef struct XCB_Window {
  int xid;
} XCB_Window;

typedef union XCB_Drawable {
  int xid;
  XCB_Pixmap pix;
  XCB_Window win;
} XCB_Drawable;
```

Figure 3: Declaration of the `pixmap`, `win`, and `drawable` datatypes.

requests require the XID of a window, some the XID of a pixmap and some the XID of a "drawable", which will denote either a window or a pixmap.[7] Since the protocol itself deals with window and pixmap types as XIDs, there is no type issue for the server, which correctly tracks types at runtime. The static typing issue is a bit harder, given the limitations of the C type system. XCB implements the drawable XID type as shown in Figure 3. The user is required to pass either `drawable`, `drawable.win`, or `drawable.pix` to the XCB API as appropriate. This provides a statically type-safe interface, without greatly inconveniencing the XCB user. Note that the ANSI C [ANS] `union` semantics essentially guarantee that the correspondence between the various `xid` fields will hold.

As a general rule, XCB tries not to provide transparent client-side caching of server-side data: doing so is normally difficult, expensive and of little benefit to most applications. However, one dataset that is transparently cached by XCB is the mapping from strings to interned "atoms", a globally unique server-side mapping which is cooperatively set up by clients. The atom cache is very effective in reducing latency and has low overhead. In addition, because it was explicitly designed to be cached on

[7] Similarly, there is exactly one X protocol request that accepts either a font or a GC—a "fontable".

the client side, the complexity of correctly maintaining a client side cache of the interned atom table is low.

2.3 The XCB API

The XCB API consists of several kinds of calls:

- *Creation functions* for such things as connections and XIDs. These functions return typed values.
- *Non-blocking requests*, for example

```
XCB_Void_Cookie
XCB_DrawPoint(
      XCB_Connection c,
      XCB_Drawable d,
      int x, int y)
```

- *Blocking requests*, for example

```
XCB_Window_Cookie
XCB_GetInputFocus(
      XCB_Connection c)
```

- *Extraction functions* for retrieving and converting reply data, for example

```
XCB_XID
XCB_GetWindowID(
      XCB_Window_Cookie c)
```

- *Response processing calls* to support retrieving events and errors. The XCB event-processing interface is via a call to `XCB_Wait_Event()`: it will block until an event is available and return it.
- A number of miscellaneous primitives. For example, single-threaded X applications often need to be able to call `select()` on the socket for a protocol connection in order to be able to multiplex their inputs correctly. XCB thus supports retrieving the socket underlying a connection, appropriately manipulated to be suitable for `select()`.

The connection creation interface to XCB consists of several routines. First, convenience functions are provided to obtain a file descriptor attached to a server via TCP/IP or UNIX-domain networking. This file descriptor, or any other file descriptor of a server connection, is then passed to a second routine which takes care of initializing the connection data structures and performing the initial protocol handshake. This approach is convenient for such tricks as running X directly over a serial line, and helps to isolate XCB from long-term changes in X and networking environments and conventions.

A principle goal of XCB is to provide as lightweight a layer atop the protocol as reasonably possible. Simply marshaling the approximately 120 requests of the X protocol and their underlying data structures should be a large enough job for a single library. The bulk of the XCB implementation consists of request and response marshaling stubs. The approach taken by Xlib to these stubs is a traditional one: they are individually hand-coded in C. XCB takes a metalevel approach: a stub description processor implemented using the `m4` [KR77] macro preprocessor translates custom stub descriptions written in a specialized macro language into C code automatically. This approach has several advantages: it helps reduce the likelihood of defects in the stubs, allows automatic generation of documentation and is easier to read and understand than C code. Perhaps the most important advantage, however, is the reduced programmer burden: this approach has been essential to a timely implementation of XCB, and should make it much easier to implement extension libraries atop XCB_Connection.

2.4 XCB In Action

It is interesting to trace the flow of a a client request and response through the XCB machinery. Consider the case of a client request to identify the window with the current input focus. Figure 4 illustrates the flow of data and control through XCB. The chains of arrows in the figure show the flow of control of a single thread.

First, the client issues an `XCB_GetInputFocus` API request with the target connection as an argument. This API call returns immediately, deliver-

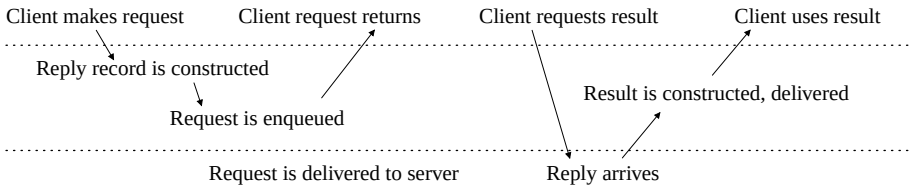

Figure 4: Flow of data and control through XCB.

ing a reply cookie of type `XCB_Window_Cookie` to the client. The upper layer of XCB delivers this cookie by asking the lower layer to allocate a reply record with the current sequence number. The request is also encoded at this time and placed in the lower layer's output buffer for eventual delivery. Finally the cookie containing the sequence number is returned to the client.

The pending request is eventually shipped to the X server, when the buffer is flushed as described in Section 2.2. By this time, the server may have generated and shipped several events, which are enqueued ahead of the server-generated reply. When the user calls XCB_GetWindowID() with the reply cookie as an argument, the lower layer uses the sequence number of the reply cookie to index the reply list, eventually finding the reply record. It then notes that no reply has yet been received from the server. Initiating a blocking read from the server to obtain the data, the lower layer enqueues the events read from the connection, then reads the reply. The resulting reply buffer, together with the connection socket, is passed up to the XCB upper layer, which assembles the requested window ID from the given data. The result is placed in the reply cookie and the blocked request returns to the upper layer. The upper layer removes the relevant reply from the reply record, frees this record and returns the window requested by the XCB_GetWindowID() call.

Alternatively, the reply may be received before the result is requested. In this case, the result is nonetheless constructed as the reply is received, and delivered to the client as requested. The reply record tracks the current state, recording whether a reply has been received, and whether a result has been requested.

3 Xlib and XCB Compared

Having discussed both Xlib and XCB in some detail, it is useful to summarize some of the comparisons and contrasts between the two. XCB introduces several novel features, borrows some nice features from Xlib, and foregoes some Xlib functionality.

Principal new features of XCB include latency hiding through reply cookies, amenability to use by threaded clients and better static typechecking. Because of its extensive use of structure and union types, XCB interfaces are well protected against parameter mismatches. In general, the syntactic and semantic regularity of XCB, at least in contrast to Xlib, should greatly ease its use.

A variety of good ideas from Xlib were incorporated into XCB. The buffering of output to form large packets increases effective bandwidth and reduces network load. Similarly, doing large reads on connection input when possible may greatly reduce syscall overhead in the presence of large numbers of events. XLib's sequence number management is quite clever: in particular, the use of a dummy request requiring a reply to both skip a sequence number and establish a synchronization point is imitated by XCB in sequence number management. XLib's proviso for direct `select()` access to the connection file-descriptor has proven to be important for single-threaded applications and was included in XCB for this reason.

Some features of Xlib, including some quite useful ones, were dropped on the theory that they could not carry their own weight in implementation size and complexity. Notably, XCB is a fairly direct binding to the X protocol: it does not send multiple requests for large inputs, directly marshal multiple results, or cache request or reply information

locally (with the exception of the atom cache). The complicated input processing of Xlib is not provided, nor is any direct support for i18n features.

Perhaps the most notable feature of Xlib missing in XCB is Xlib compatibility in the API: it would be nice to be able to achieve some of the XCB gains by relinking existing applications. While, for the reasons described immediately above this is not a feasible goal for XCB itself, it is believed that with reasonable effort a lightweight Xlib compatibility layer could be placed atop XCB.

4 Status and Future Work

An XCB code base is currently under development. Simple examples work, but much work remains to complete the implementation. Once the implementation of the core protocol is complete, work will commence on the implementation of many of the X Consortium and XFree86 extensions. Finally, the longer-term goal of layering font and rendering support atop XCB should eventually lead to an Xlib compatibility library, a redesigned C interface for standalone programs and a toolkit.

Availability

When complete, the XCB implementation will be made freely available under the XFree86 License. More information should appear on the web at `http://xcb.cs.pdx.edu` in the near future.

Acknowledgments

The author gratefully acknowledges the advice and assistance of Keith Packard in the analysis, design, and implementation of XCB.

References

[ANS] X3.159-1989.

[Gai93] Jean-loup Gailly. *Gzip: the data compression program*, 1.2.4 edition, July 1993.

[KR77] Brian W. Kerninghan and Dennis M. Ritchie. *The M4 Macro Processor*. AT&T Bell Laboratories, 1977. Unix Programmer's Manual Volume 2, 7th Edition.

[KR78] Brian W. Kerninghan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, 1978. ISBN 0-13-110163-3.

[Nag84] John Nagle. RFC896: Congestion control in IP/TCP internetworks. RFC 896, Ford Aerospace and Communications Corporation, 1984.

[NBF96] Bradford Nichols, Dick Buttlar, and Jacqueline Proulx Farrell. *Pthreads Programming, A POSIX Standard for Better Multiprocessing*. O'Reilly & Associates, Inc., first edition, September 1996.

[Pac01] Keith Packard. An LBX postmortem. `http://xfree86.org/~keithp/talks/lbxpost`, January 2001.

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

[SG92] Robert W. Scheifler and James Gettys. *X Window System*. Digital Press, third edition, 1992.