The X-Engine Library: A C++ Library for Constructing X Pseudo-servers

TR93-019

John Menges

The University of North Carolina at Chapel Hill
Department of Computer Science
CB#3175, Sitterson Hall
Chapel Hill, NC 27599-3175
919-962-1792
jbs@cs.unc.edu

A TextLab/Collaboratory Report

Portions of this work were supported by the National Science Foundation (Grant #IRI-9015443 and by IBM Corporation (SUR Agreement #866).

The X Engine Library: A C++ Library for Constructing X Pseudo-servers

John Menges

Abstract

A common technique for adding window sharing and tracking/replay capabilities to the X Window System is the use of an X Pseudo-server. An X Pseudo-server is a process placed between an X server and X clients. It adds capabilities to X by monitoring and manipulating the X protocol stream between X servers and clients.

X Pseudo-servers are typically large, complex programs, and can be difficult to design, implement, and maintain. The X Engine Library described in this paper is a C++ library that greatly simplifies these tasks.

Introduction

A common technique for adding window sharing and tracking/replay capabilities to the X Window System is the use of an X Pseudo-server (XPS) [Pat90] [Lau90] [AWF91] [Lin92]. An XPS is a process placed between an X server and X clients. Figure 1 shows a simple XPS and its relationships to X servers and clients. The XPS monitors and/or modifies the X protocol message streams between the server and clients to add capabilities to the window system. The main advantage of XPS-based approaches to adding such capabilities (as opposed to toolkit- and server-based approaches) is that the new capabilities are applicable to existing applications and servers, without any need to modify, recompile, or relink existing code. (The main disadvantage to the XPS-based approach is that the X protocol is at too low a level to implement certain desirable application-sharing and tracking/replay capabilities.)

John Menges (menges@cs.unc.edu) is a graduate student in Computer Science at the University of North Carolina at Chapel Hill.

This paper appears in the Proceedings of the 7th Annual X Technical Conference, Boston MA (Jan 1993), published in The X Resource, Issue 5.

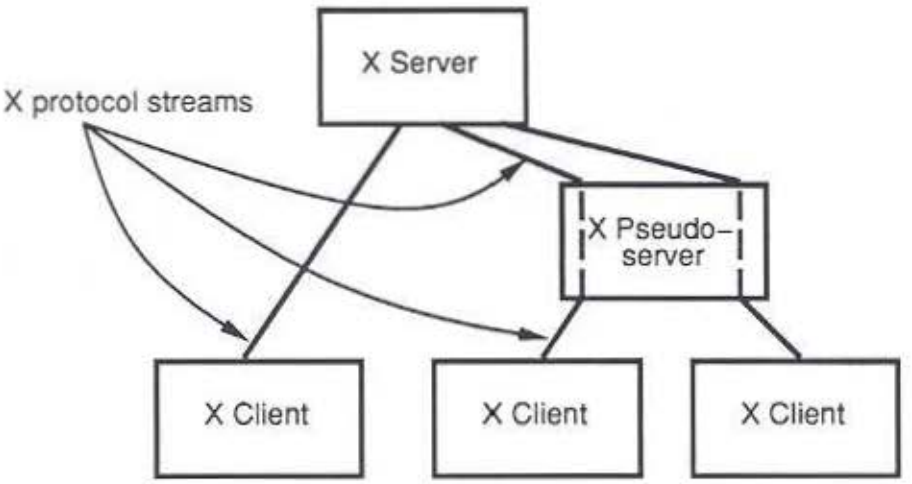

Figure 1: An X Pseudo Server and Its Relationships to X Servers and Clients

As part of our Artifact-Based Collaboration System (ABC) [SS91], we are implementing window sharing and tracking/replay for X applications using an XPS. In previous work here and at Old Dominion University, we have written two XPSs. The first, XTV [AWF91], was originally based on the idea of sharing individual windows. The second, Virtual Screen [Lin92], was based on the idea of sharing a collection of windows within an encompassing "virtual" screen. Here, window positions within the virtual screen and a window manager for the virtual screen are shared, as well as the windows themselves.

An XPS that supports, e.g., window sharing between users on different workstations, is more complex than one might at first expect. There are many difficult problems to be solved, as Patterson [Pat90] and Lauwers [Lau90], among others, attest. These problems include mapping resource IDs and absolute coordinates, accommodating late-comers (sharing a window with a new user after the window has been created), sharing pointers and cursors, implementing floor control (determining which user(s) can interact with which windows at a given point in time), resolving various sequencing and synchronization issues, making window damage repair efficient and nondisruptive, and resolving various heterogeneity issues such as differing screen resolutions, color capabilities, and font sets.

Thus, an XPS is a large program, often written by multiple authors over a long period of time. Both XTV and Virtual Screen are monolithic systems written in C. Both were written by and/or need to be maintained by multiple graduate students over many years. The big learning curves and short tenures of graduate students, coupled with our need to do rapid prototyping of new ideas and options, has led us to consider how XPSs might be implemented more easily and flexibly. To address this problem, I have designed and implemented the X Engine Library (XEL), a C++ library for constructing XPSs.

An Overview of The X Engine Library

The X Engine Library (XEL) is a library of C++ classes representing an event scheduler, event handlers, X messages, and filters for monitoring and/or manipulating X messages. A scheduler is combined with event handlers and filters to construct an XPS. The classes used

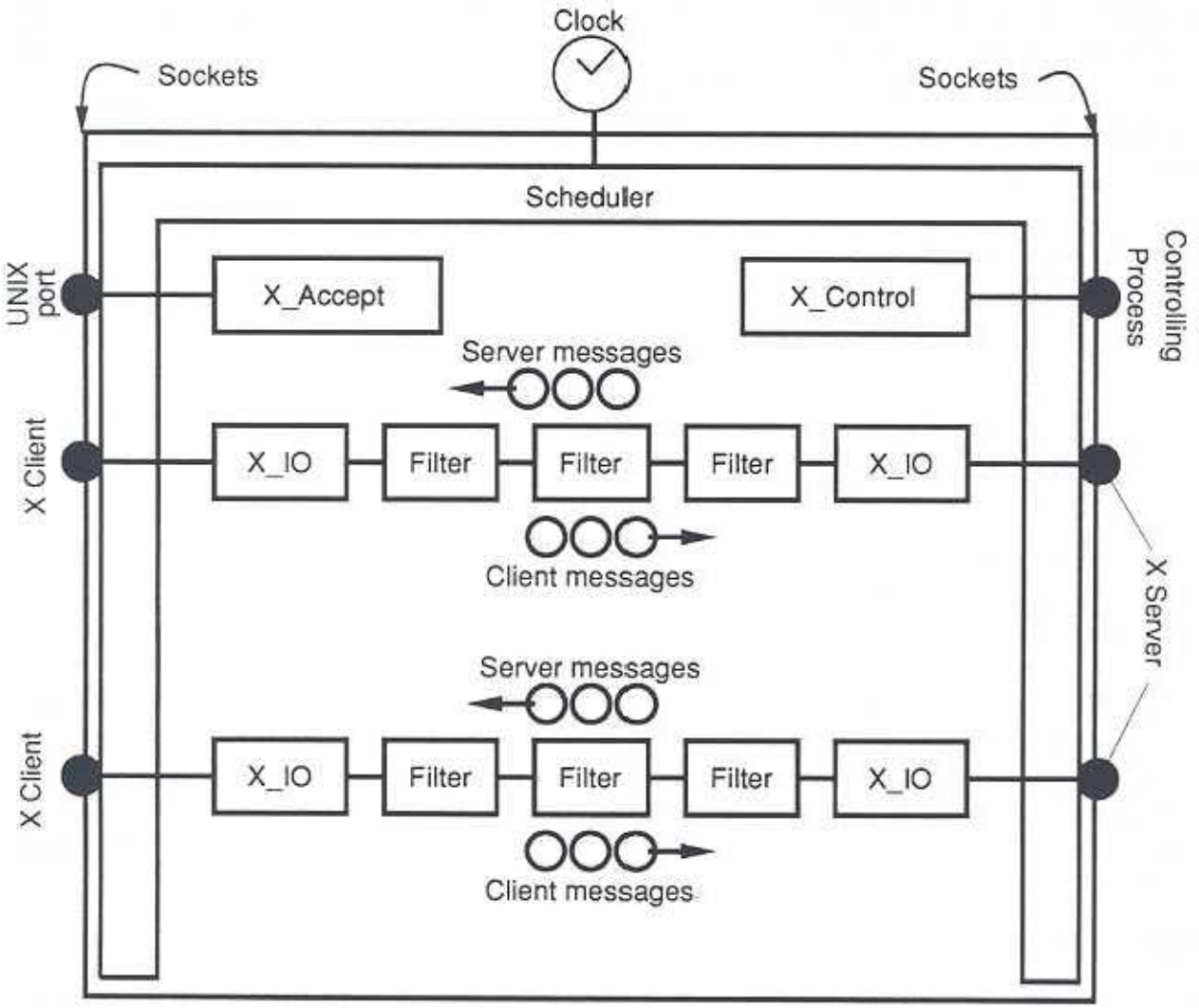

Figure 2: An Overview of an XEL-based XPS

to construct an XPS can be taken from the XEL library directly or created as subclasses of classes in XEL.

Figure 2 is an overview of a simple XEL-based XPS. The X_Accept event handler accepts connections from X clients and builds a filter network (here, a list) to handle the new client's X protocol message stream. The X_IO handler/filters at the ends of the filter lists do the translations between X messages on a socket stream and X Message objects. The message objects are passed from filter to filter through the network in the appropriate direction. (The X_IO handler/filters are both event handlers and filters. This is accomplished via multiple inheritance in C++.) The X_Control handler reads commands from a socket and executes them, which results in method calls on other objects in the XPS. This is how the operation of the XPS is controlled from other processes. The Scheduler translates operating system events into calls on event handler triggering methods, guided by priorities assigned to events.

The structure provided by the XEL library and the predefined classes it provides greatly

simplifies the task of designing and implementing a given XPS. This, in turn, makes it feasible to experiment more freely with XPS alternatives, and to add capabilities to X that have hitherto been too difficult to attempt to provide.

The XEL classes are described in more detail below, after which a concrete example XPS is presented and discussed.

The XEL Classes

The classes in the XEL library represent a scheduler, several types of event handlers, numerous filters, and X messages. The scheduler and some of the event handlers and filters can be used as-is or as superclasses for programmer-defined classes. Other event handlers and filters are intended to be used only as superclasses. The message classes are intended to be used as-is. The most important classes provided by XEL are described in this section.

The Scheduler

The scheduler class implements event scheduling. Event handlers are registered with the scheduler, and are triggered when the associated events occur. Most event handlers handle socket events such as "input available" or "write possible", the latter being necessary to avoid blocking writes. "After_input" event handlers are triggered after a set of input events are handled, as determined by the `select()` system call. An After_input event handler may be used, e.g., to move messages through a network of filters, as described in more detail below. Timer event handlers are triggered at preset times or intervals. Process handlers are triggered when there are no other events to be handled, so that background processing may be done. The scheduler supports a strict priority scheme for prioritizing the handling of events.

Event Handlers

Event handlers react to operating system events when triggered by the scheduler. Several important event handlers are provided by XEL.

The X_Accept event handler accepts connections from X clients on a specified port. It is intended as a superclass for a programmer-defined event handler that accepts client connections. Accepting a connection yields a new socket connected to the client. This socket is passed to the `create_filter_network` member function of the programmer-defined subclass of the X_Accept handler. It is the duty of this member function to create a filter network for the new client. The filter network created includes an I/O event handler/filter for the socket to the new client, and may include another I/O event handler/filter connected to an X server, as in Figure 1. These I/O handler/filters are typically instances of the X_IO class described below. The `create_filter_network` member function must register these I/O handlers with the scheduler.

The X_IO event handler translates between X protocol messages on a socket stream and X message objects in the XPS. It puts messages objects it creates onto an adjacent filter's input queue, and receives message objects from other filters, buffering them and later writing them to its socket. Since objects of C++ classes can be instantiated using different constructors, one constructor is used to create an X_IO handler for the client end, where the socket is

already available, and another constructor is used to create an X_IO handler for the server end. In the latter case, the constructor sets up the connection to the appropriate server. The X_IO handler is actually a filter and three types of event handlers, all in one. It is a filter because it needs to be connected to other filters and to perform filter functions such as sending and receiving message objects. It is an "input available" event handler because it needs to read messages from its socket when they arrive. It is an "After_input" event handler, because once a set of messages has been passed through the filter network, outgoing messages received by the X_IO filter should be sent as a unit, for performance reasons. It is also a "write possible" event handler, as it may not be possible to send all outgoing messages on the first attempt, due to flow control on the socket connection. To avoid blocking, the X_IO filter must then buffer some output and wait for notification of when writing is again possible.

The X_Move_Messages event handler is typically registered with the scheduler as an After_input handler before the scheduler is started. Its purpose is to service filter queues by passing X message objects from the queues to the associated filters. There are two queues per filter, one for client messages (messages from clients) and one for server messages. When an X message object is placed into an empty queue, the queue is normally put into a global list of queues that need to be serviced by the X_Move_Messages handler. Filters may, however, block input on either queue, in which case the blocked queue doesn't get put into the service queue for X_Move_Messages. When a queue becomes empty, it is removed from the service queue.

The X_Control handler is an "input available" event handler that reads TCL [Ous90] commands from a socket and passes them to a TCL interpreter. Commands are registered with the interpreter by filters, handlers, or other objects in the XPS. Interpreting a command results in a method call on the object registering the command. This is how the behavior of the XPS is controlled by other processes. This controlling mechanism might be used, e.g., to ask the XPS to create a shared virtual screen or to add viewers to an existing one, to move windows into or out of a shared virtual screen, or to turn tracking on or off.

Filters

A filter's purpose is to monitor and/or modify a stream of X message objects. All filter classes are subclasses of the Filter superclass, which implements functions needed by filters in general. Filters can pass on X message objects without change (i.e., they can monitor or just ignore them), or modify them before passing them on. They can destroy messages (not pass them on), clone them and pass them to multiple filters, or create new ones to insert into a stream. They can also double as event handlers, as does X_IO above. For example, a filter that prints message contents might be implemented as a filter/process handler so that it can do asynchronous output during pauses in X message traffic in order to avoid slowing down the X protocol stream.

Filters can be interconnected in various ways to provide different capabilities. Filter links can be static or dynamic; that is, messages can be sent to a fixed set of adjacent filters (utilizing Filter superclass support), or the set of filters to which a particular message is sent

can be computed on-the-fly. Static links can be created using a UNIX[1] pipe-like syntax. For example, a simple string of filters can be created like this:

```
client_io_filter | filter | ... | filter | server_io_filter;
```

and a simple tree of filters can be created like this:

```
client_io_filter | branch | filter | ... | filter | server_io_filter1;
branch | filter | ... | filter | server_io_filter2;
```

A filter composition capability is envisioned for a future release. This would enable a network of filters to be viewed as a single filter.

Filters can be selective about which messages they want to process. This selection is facilitated by the C++ member function overloading mechanism. For example, a filter can define a member function to handle all server messages:

```
void My_filter::filter(Xserver_msg& message) {...}
```

or only replies:

```
void My_filter::filter(XReply& message) {...}
```

or only InternAtom replies:

```
void My_filter::filter(XInternAtomReply& message) {...}.
```

Or, it can define all three. In this case, all InternAtom replies will be sent to the `filter` member function for InternAtomReply messages, other replies will be sent to the member function for replies, and all non-reply server messages will be sent to the server message member function. Messages that are not explicitly handled are automatically forwarded to the next filter(s) via the static link(s). Filters can explicitly send messages to the static links using the `send(message)` member function of the Filter superclass, or they can send them explicitly to arbitrary filters using `send(message, filter)`. Server messages and client messages are always handled by separate member functions (using C++ overloading). This makes use of the C++ type system to ensure that messages always flow in the proper direction.

A future version of XEL will implement Condition classes. Each filter will have two Condition objects, passed as parameters to its constructor (with appropriate defaults). One condition object will screen client messages, and the other, server messages. Messages that don't pass the test will be sent to the next filter(s) via the static links, bypassing the current filter. This will make it possible to define filters that can be combined with different conditions to achieve different effects, which in turn encourages code reuse.

[1] UNIX is a registered trademark of AT&T.

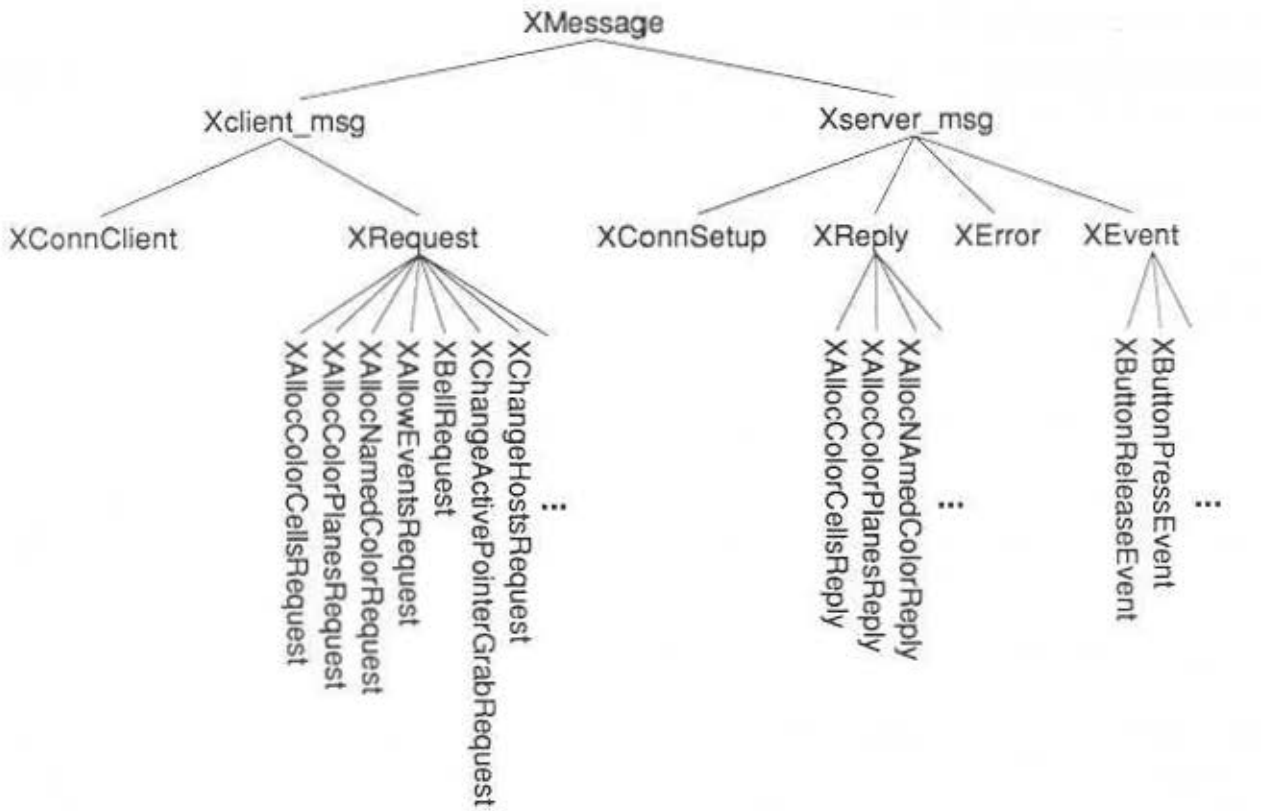

Figure 3: The X Message Class Hierarchy

X Messages

XEL provides message classes for each major type of messages (i.e., server message, client message, request, reply, error, and event), and for each individual message type (e.g., InternAtomRequest or ExposeEvent). These message classes are arranged in an inheritance hierarchy as shown in Figure 3, which facilitates the selection of message types by filters, as described above.

The C++ code for classes for specific message types is automatically generated from the structure definitions (similar to those in Xproto.h), using csh and awk scripts. Having a separate message class for each message type (about 200 in all) requires a lot of code, but it is useful in several ways. As mentioned above, the message types can be used to distinguish between overloaded function definitions, making it possible to write filters that explicitly mention only the message types they need to process. Message objects know how to print themselves, and the member function that does so takes optional arguments to restrict the amount and type of output. This makes it easy to debug XPSs; just insert filters at strategic points that ask certain messages to print themselves. Note that this makes it rather easy to construct an XPS that implements a rather flexible Xscope-like tracing capability. Message objects also know how to apply a filter member function to particular sets of their fields. For example, one can apply a filter member function to all resource IDs in a message, or to all graphics context IDs, or to all atoms, or to all pixels. This facilitates the implementation

of resource mapping filters.

Message objects can be created by passing an array of bytes read from a socket to a static member function of the Xserver_msg or Xclient_msg class:

```
serverMessagePointer := Xserver_msg::create(bytearray);
```

This is how X_IO creates messages. Alternatively, messages can be created by using a specific message type's constructor without arguments, which creates a message of that type with suitable defaults:

```
createWindowRequestPointer = new XCreateWindowRequest;
```

Once a message object has been created, its fields can be referenced and modified. Messages have a fixed-length part, followed by zero or more variable length fields. Fields in the fixed-length part can be referenced like this:

```
mapWindowRequest->window := windowID;
```

It would be costly at message object creation time to initialize a separate member in the message object for each field in the fixed portion of the message, because these fields would have to be separately initialized from the array of bytes read from a socket. To avoid this, the `->` operator of the message classes is overloaded to provide access to the fields of the fixed portion in place in the byte stream. (Note that mapWindowRequest is an object, not a pointer to an object.) Each variable-length field in a protocol message, however, has a corresponding member in the message class. These members are accessed like this:

```
polyFillArcRequest.arcs[i].height
```

In the example above, `arcs` is a variable-length array of `xArc` structures. If `arcs[i]` does not already exist, space is made for it in the message object, it is initialized appropriately, and all necessary fields in the fixed part of the message object (e.g., the request length) are updated accordingly. This is accomplished by overloading the `[]` operator.

Server messages contain a pointer to the client message referred to by the sequence number in the server message. (Technically, server messages contain a client message pointer member, which is set to the appropriate client message by the X_IO filter.) This relieves filters, in many cases, from having to monitor and manipulate sequence numbers. It also enables the X_IO filter on the client end to get the sequence numbers right for the client, in the event that filters insert client messages into the stream.

An XPS Example

As an example, consider the filter network in Figure 4. This filter network was created by an instance of a subclass of the X_Accept event handler when the client on the top connected to the XPS. The XPS implements a very simple "conference" between two participants by

sharing all windows created by all clients that connect to the XPS. The X_IO filters are instances of the X_IO class described above. They translate X protocol messages between byte streams and X message objects. The Gate, Close, and Open filters implement synchronization constraints. The Resource filters do resource translation. The Branch filter clones client messages and sends them down both branches of the network. The Screen filter allows only Event (and not Reply or Error) server messages to flow from the right branch to the client. The Seq filter ensures that sequence numbers in server messages are monotonically increasing as they are sent to the client. The thick, solid arrows indicate static inter-filter links. The thin arrows indicate that all messages sent in the direction indicated by the arrow are passed through the filters indicated without being monitored or modified. The various filter types and the dashed arrows are described in more detail below.

The Gate filters have two states, open and closed. When open, they allow all server messages to pass. (Note that the thin arrows indicate that they *always* allow client messages to pass.) When closed, they allow all server messages to pass until a Reply message is received. The Reply and all subsequent messages are then blocked. The Close filters close the Gate filters indicated by the dashed arrows, when they receive a Reply message. They are given a pointer to the associated Gate filter via their constructor. The Open filters similarly open the corresponding Gate filters. Gate filters marked with a "Do Not Enter" symbol (slashed circle) are closed initially, and the others are open initially. This initial state is set by the Gate filter constructor.

The lower set of Gate, Close, and Open filters implements alternation of Reply messages going through the Resource filters. First a Reply from Server 1 passes, then the corresponding Reply from Server 2, and then back to Server 1. Intervening Event and Error messages are passed in order, but without otherwise being blocked. The Resource filters capture the resource ID base and mask from the XConnSetup messages and store them locally. For resources not created by the client (such as the root window ID) that need to be mapped, the right Resource filter creates a mapping table that maintains the correspondence between the resource IDs on Server 2 and those on Server 1. To perform this task, the right Resource filter needs to see corresponding replies from both branches at the same time. This is the reason for the alternation of Reply messages described above. The left Resource filter keeps a pointer to the current Reply. The right Resource filter, when it gets the corresponding Reply, can look at both messages to get the resource IDs that need to be paired. (The right Resource filter has a pointer to the left one, which it got via its constructor, and it uses this access the Reply on the left.) The Resource filter also performs resource mapping on client and server messages by asking the messages to apply specified filter member functions to all resource IDs in the messages. The code for doing the mapping and any state needed by this code (e.g., resource mapping tables and the resource ID bases and masks) are kept in the filter.

The upper set of Gate, Close, and Open filters ensure that a reply will not go to the client before the right Resource filter has a mapping for any IDs in the reply. This ensures that the client will not use any resource IDs for which a mapping does not yet exist.

The Screen filter ensures that replies to requests only come from one branch, the left one. (Reasonable floor control mechanisms require more elaborate screening than this.) The Seq

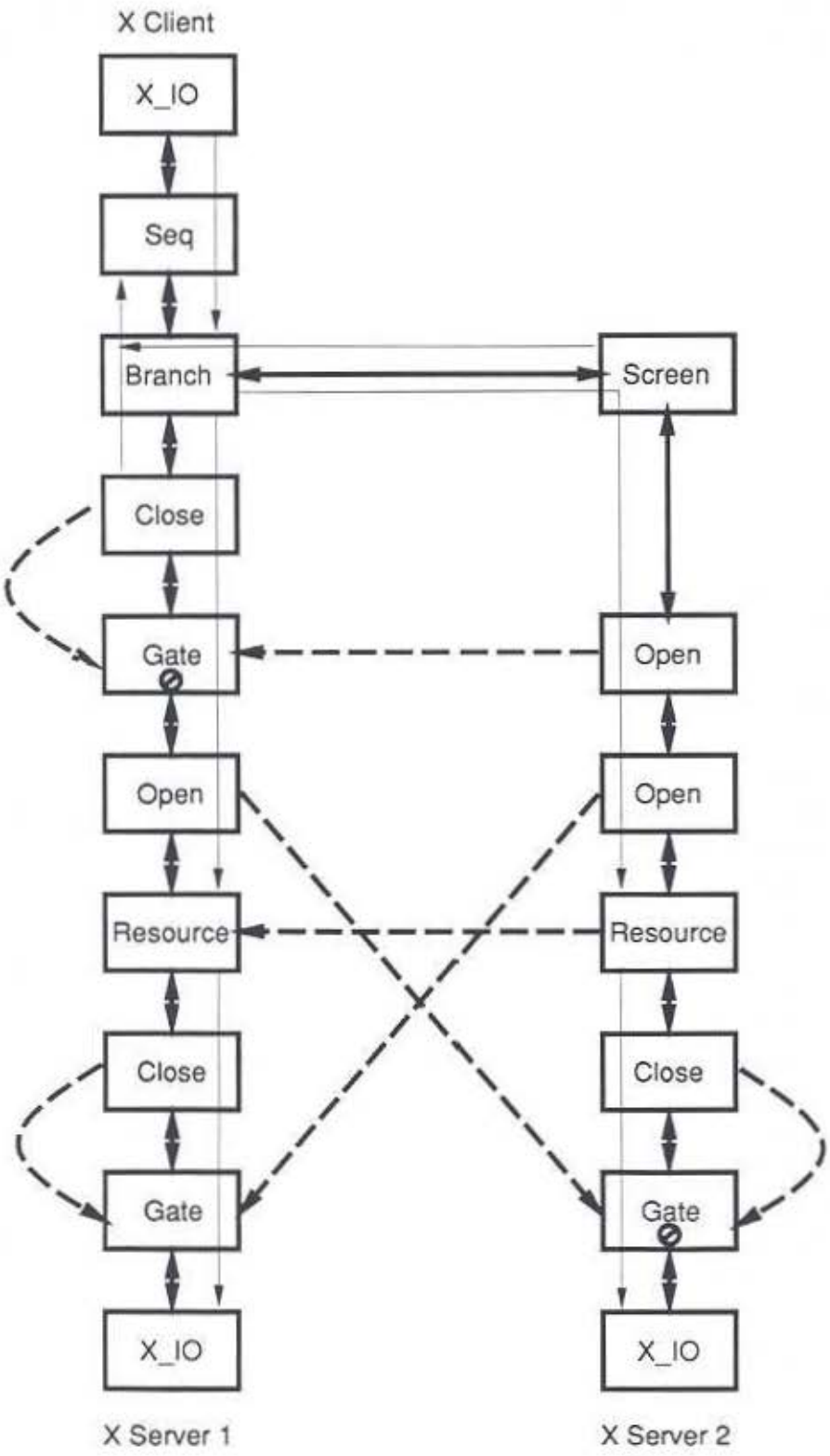

Figure 4: A Simple Example XPS for Window Sharing

filter ensures that sequence numbers going back to the client are monotonically increasing. Sequence numbers on events can end up being out of order at this point, because we are receiving events from both branches. Once again, stronger assurances than this about sequence numbers are typically necessary.

Note that the synchronization of messages could have been implemented entirely within the Resource and Branch filters. There is a good reason for not taking this approach. The XEL was designed to make it easy to implement XPSs. Factoring out synchronization makes it easier to write the Resource Mapper in this example, since alternation can be assumed. Keeping filters small and simple decreases the difficulty of writing XPSs and increases code reuse.

Similarly, notice that Gate filters are always followed immediately by Close filters, and the two could be combined. This has not been done, so that other types of synchronization can be implemented with these same classes. But in this case, I've made the assumption that once X_Move_Messages starts moving a message, it continues to move it as far as possible until it is blocked. This is indeed the case with the current version of XEL, and, since it is a useful assumption, it will probably become part of the XEL specification.

Summary and Conclusions

X Pseudo-servers (XPSs) are an important means of adding capabilities to the X Window System. But they are difficult to design, implement, and maintain. I have addressed this problem by creating the X Engine Library (XEL), a C++ library for constructing XPSs.

XEL simplifies the construction of XPSs for a number of reasons. Code reuse is high, because scheduling, event handling, filtering, and structuring capabilities are part of the library. These capabilities can be encorporated into an XPS by selecting concrete (complete) classes from the library and by subclassing off of concrete or abstract (incomplete) classes. Filters can often be connected in a simple sequence, using a UNIX pipe-style syntax. Inter-filter communication (IFC) needs are normally small, and when IFC is required, it can be accomplished via method calls between filters or via shared objects.

Filter code only needs to explicitly mention messages types it must monitor or modify. Entire classes of messages (e.g. Requests, or Replies) can often be handled as a unit, because there is a separate message class for each X protocol message that knows the particulars of that message and can apply functions to its particular members by type. C++ operator overloading makes selecting message classes easy.

Filters are typically small. The Resource filter is about 200 lines of code and the others (excluding X_IO) are around 100 lines, mostly boiler-plate. If we assume that the Seq, Screen, and Resource filters, the subclass of X_accept, and the main program are all that a programmer needs to write in the example XPS described above, this XPS was implemented with about 800 lines of C++ code, most of which is boiler-plate.

Flexibility, modularity, simplicity, and efficiency are primary design criteria for our ABC system. Flexibility is important because it facilitates the prototyping and implementation of a wide range of ABC design options. Modularity and simplicity are critical given the big learning curves and short tenures of graduate student programmers; they also contribute to

flexibility. Efficiency is important both because we need to conduct meaningful user studies of group collaboration and because an efficient prototyping mechanism eases the conversion to efficient products. The XEL gives the XPS component of the ABC system great flexibility, modularity, and simplicity. Early XPSs we have developed with XEL make us confident that implementing an efficient XPS with XEL is possible, though this has yet to be proven for an XPS with a reasonably complete set of capabilities.

The XEL structure has also proved useful as a common language and framework for discussing alternative XPS capabilities, and how they might be implemented. Its flexibility has encouraged us to investigate more flexible window sharing and tracking/replay systems. For example, we are now working on a drag-and-drop style of conferencing, where an arbitrary window can be picked up and placed into a conference (shared virtual screen) or taken out of one, or cloned and placed in different locations, in and out of conferences. We have not seen this type of per-window conferencing in other XPSs. Most XPS-based conferencing systems (including XTV and Virtual Screen) share client connections to an X server, rather than individual windows. Unlike XTV, Virtual Screen, and some others, ABC uses only a single XPS per X server, and all clients connect to our XPS instead of to the real X server. That is, we implement multiple conferences within a fixed set of XPSs, one per workstation, rather than having a single XPS per conference. This, combined with the use of XEL to construct our XPSs, facilitates more flexible window sharing capabilities. It also makes conferences more lightweight (an important goal), because it avoids process instantiations when a conference is created.

References

[AWF91] H. M. Abdel-Wahab and Mark A. Feit. XTV: A Framework for Sharing X Window Clients in Remote Synchronous Collaboration. In *Proceedings of Tricomm '91*, April 1991.

[Lau90] J. Chris Lauwers. Collaboration Transparency in Desktop Teleconferencing Environments. Technical Report CSL-TR-90-435, Computer Systems Laboratory, Departments of Electrical Engineering and Computer Science, Stanford University, Stanford, California 94305-4055, July 1990.

[Lin92] Jin-Kun Lin. Virtual Screen: A Framework for Task Management. In *Proceedings of the Sixth Annual X Technical Conference*, January 1992.

[Ous90] John K. Ousterhout. TCL: An Embeddable Command Language. In *Proceedings of the 1990 Winter USENIX Conference*, 1990.

[Pat90] J. F. Patterson. The Good, the Bad, and the Ugly of Window Sharing in X. In *Proceedings of the Fourth Annual X Technical Conference*, January 1990.

[SS91] J. B. Smith and F. D. Smith. ABC: A Hypermedia System for Artifact-Based Collaboration. In *Proceedings of Hypertext '91*, December 1991.