Inverting X: An Architecture for a Shared Distributed Window System

John Menges *Kevin Jeffay*

Department of Computer Science
University of North Carolina at Chapel Hill
Chapel Hill, NC 27599-3175
{menges,jeffay}@cs.unc.edu

Abstract: Shared window systems have become a popular vehicle for supporting distributed, synchronous collaboration. At present they are difficult to build and they support limited paradigms of multi-user interaction with shared applications. We believe this difficulty is largely due to the inverted nature of the client/server architecture of most distributed window systems. The architecture is inverted in the sense that the user is nearer the server than the client; this hampers attempts to share windows. By comparing the traditional client/server architecture of distributed file systems with the inverted architecture of distributed window systems we argue that it is possible to develop window systems where the user is nearer window system clients than servers, and that this architecture greatly facilitates the sharing of windows among users.

Introduction

A cursory comparison of the architecture of a typical distributed file system (DFS) (Figure 1) with the architecture of a typical distributed window system (DWS) (Figure 2) reveals that while both are "distributed" in similar ways, the effect of this distribution on users is quite different. Both systems are object management systems oriented around the client/server model, with objects being represented in and managed by servers and used by clients. Both also allow client access to servers over a network transparently, *i.e.*, in the same way as if the client and server were co-located on one machine. But in the case of the distributed file system (DFS), the user is located nearest the client, while in the distributed window system (DWS) case, he is located nearest the server.

Consider first the trivial case where there is only one server and one client (Figures 1 and 2). In the case of the DFS, distribution allows a client application (and therefore a user) to be in a different location from the files being accessed. This is important in the case of files, because files, unlike applications and users, are often represented on heavyweight physical devices (disks) that have location and movement restrictions. In the DWS case distribution means that an application can be in a different location from the windows (and therefore the user) with which it is associated. This is important because applications often need computational ability not commonly available in a workstation, where the server must reside because of its tight association with the display.

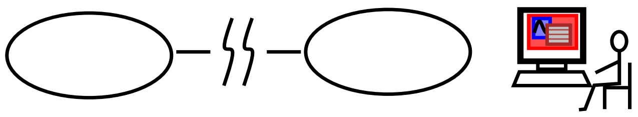

Figure 1: A Typical Distributed File System Architecture.

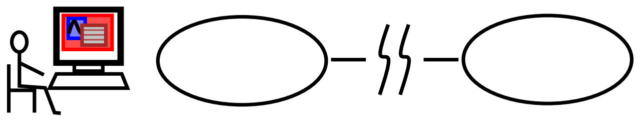

Figure 2: A Typical Window System Architecture.

Now consider the case where there is a single server and multiple clients (Figures 3 and 4). In the DFS case, this allows applications and users in different locations to share files. In the DWS case, different applications can share windows, but not different users. This is of limited utility; it makes client window managers possible but is otherwise rarely used.

Finally, consider the case where there is a single client and multiple servers (Figures 5 and 6). In the DFS case, this allows a single application to access files in different locations. In the DWS case, it allows a single application to access windows on different displays, and therefore allows different users to access the same application. This, then, is the way that real-time collaborative systems are typically built today. What is shared is the application, which must be aware of the various users among which it is being shared. (Collaboration toolkits are often provided to make it possible to isolate this knowledge from the

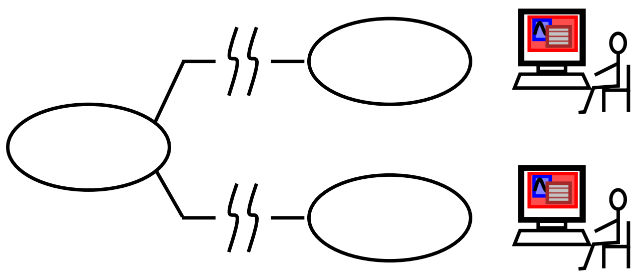
Figure 3: A DFS with Multiple Clients.

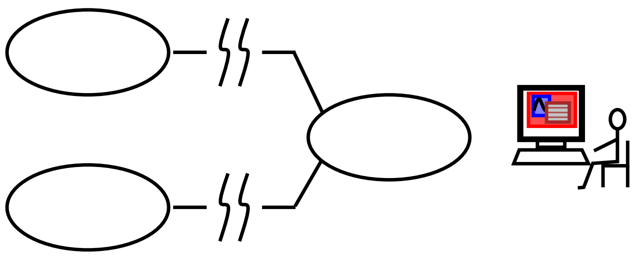
Figure 5: A DFS with Multiple Servers.

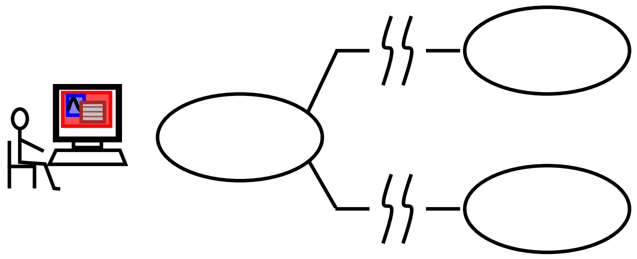
Figure 4: A DWS with Multiple Clients.

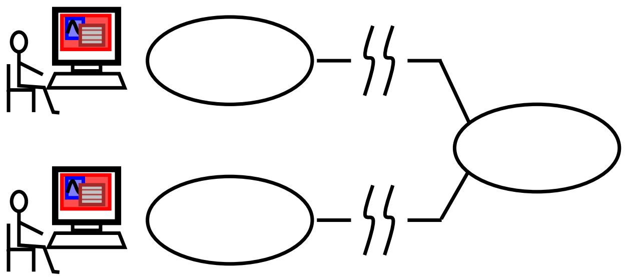
Figure 6: A DWS with Multiple Servers.

application proper.) Windows themselves are not shared, although particular applications can simulate window sharing by duplicating windows on different displays.

In summary, a comparison of the traditional client/server architecture of distributed file systems with the inverted architecture of distributed window systems reveals that, while both allow the separation of applications from the objects being referenced (files and windows, respectively), only the traditional architecture allows the sharing of these objects among multiple users. Window sharing must be accomplished via the sharing of applications that are able to simulate the sharing of windows.

The inverted nature of typical distributed window systems is not, however, intrinsic to window systems. It is forced by the desire (possibly motivated by efficiency considerations) to tightly couple the physical representation of windows on displays with their abstract representation in memory. The physical representation of windows must, of course, be co-located with the users of those windows. The abstract representation, however, can be located anywhere. The traditional client/server model, with its desirable sharing characteristics, can be achieved by placing the abstract representation of windows in a window server and the physical representation in a client of the window server that is closely associated with the display (Figure 7). We call this type of client the display client, to distinguish it from the application client, whose relative position in the architecture is left unchanged.

Note that the server is still between the user and his application, but now the display is managed by a client of the window server rather than by the server itself. This architecture makes it possible to directly share windows among multiple users. In particular, the windows of *any* application can be shared, without the need for special knowledge of window sharing in the application or collaboration toolkit. This is the appropriate architecture for a shared distributed window system (SDWS).

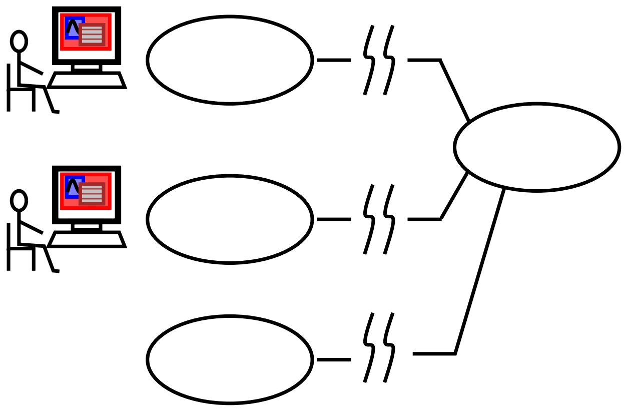
Figure 7: Proposed SDWS Architecture.

2. Characteristics of a SDWS

Given the desirability of a window system directly supporting window sharing, what should be the characteristics of a SDWS? In this section we first discuss those DFS characteristics that are also applicable to the SDWS, and then the characteristics that are unique to the SDWS.

2.1. DFS Characteristics Applicable to the SDWS

Borrowing once again from the DFS, three characteristics come to mind immediately: naming, access control, and concurrency control.

Naming of windows can be similar to naming of files. Windows, like files, are usually arranged in a hierarchy, so the typical character-string representation of a path through a hierarchy of objects can apply to windows as well as files. In current window systems windows are typically not named symbolically. This is primarily because the lack of window sharing renders such naming of little value; windows can easily be identified within a single application by pointers or integer IDs. Windows also open up the possibility of graphical naming (point to the window or some simplified drawing of the window in its context and click), but this is not a substitute for string-oriented naming, because applications need to be able to name windows without user intervention.

Access controls for file systems determine what operations a particular user may perform on a particular file. These operations typically include name lookup, creation, deletion, reading, and writing. Window operations are similar. Drawing by an application client and event generation by a display client are both write operations, and reception of events by application clients and reception of drawing requests by display clients are both read operations. (In fact, there is no reason to distinguish between application and display client types, except to draw analogies to the X Window System. Any client should be able to read and write both events and drawing requests, subject to access controls.) Since reads and writes are more highly structured for window systems than for file systems, finer-grained access controls are desirable. For example, access controls for write requests that make structural changes to a window might be different from access controls for drawing in a window.

Concurrency control for file systems is usually accomplished by the explicit setting of read and write locks. A similar mechanism may also be sufficient for windows. For example, A display client could put a write lock on a window as a means of obtaining floor control.

In file systems, access rights and concurrency controls are sometimes inherited from ancestors in the file hierarchy, to simplify the specification of access rights and to make their implementation more efficient. Similar inheritance mechanisms would also be useful for window systems.

In addition to naming, access control, and concurrency control, one might at first think of persistence as another DFS characteristic applicable to the SDWS. Windows, however, are not intended to represent permanent state information; they are only intended as a means of delivering a graphical interface to some portion of an application's state. There seems therefore to be little point in implementing persistent windows. Applications can use files to store any state specific to windows (their layout, color scheme, access restrictions, names, *etc*.) that must be retained across application instantiations.

Finally, some form of mounting distributed window systems and/or linking between one part of the name space and another will prove to be desirable, but this discussion must be delayed until the concept of a SDWS is further developed.

2.2. Characteristics Unique to the SDWS

Exactly what does it mean to share a window? This question can be answered by defining the coupling possibilities between multiple representations of a shared window. That is, when a window is shared, are the apparent attributes of the various physical representations of the window (size, appearance, colors, etc.) identical? If not, what are the possibilities for variation among physical representations? In many circumstances, a tight coupling between physical representations are essential, but the boundaries of the usefulness of window sharing are determined by the degree to which looser coupling of windows can be supported.

2.2.1. Tightly-Coupled Window Sharing

First consider the tightest coupling possible, where all attributes are shared. In this case, sharing a window means that the window's appearance on multiple displays (including the appearance of any subwindows) must be identical in all respects, *i.e.,* colors, fonts, sizes resolutions, etc. For the moment, let's assume a homogeneous display environment where different display resolutions, color capabilities, etc. are not a problem. The only thing that may vary is the context in which the window is placed. That is, the surrounding environment of the window need not be the same on each display. Once again, we can borrow DFS semantics to accomplish this type sharing. A specially-typed link (or mount) from a stub in one portion of the window hierarchy to a window in another portion fulfills our purpose. The link must be of a special type to ensure that the hierarchical constraints of the name space are not violated. Figure 8 demonstrates such a link: window /L/N/Q is a link to window /A/D/F.

Note that some window system operations require knowledge about the parent of a particular window. For example, a display client needs to know the location of a window relative to its parent's location. The correct parent

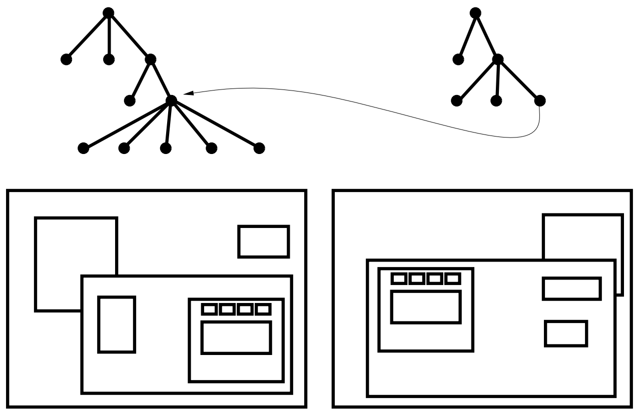

Figure 8: Tightly Coupled Window Sharing.

depends on the path taken to get to the child. In this respect window links are more akin to Network File System (NFS) mounts than links.

2.2.2. Support for Heterogeneity

Of course, assuming display homogeneity is unrealistic. The color capabilities, sizes, and resolutions of displays vary widely. How can we accommodate display heterogeneity in a SDWS? Display accommodation can take place in either the application client or the display client. We will call the ability to make these accommodations *application flexibility* or *display flexibility*, depending on where it is done. Both methods should prove useful, and both require server support. Display flexibility, however, falls under the broader category of display customization, discussed later. For now, let us consider how application flexibility can be accomplished.

A flexible application is aware that its windows may be simultaneously displayed on multiple displays with different capabilities. It takes control of how the window will be physically represented on the different displays. This is accomplished through *conditional window attributes*, supported by the window server. For example, suppose a window is to have a blue background on a color display, but a dark background on a monochrome display. The background attribute for the display is conditional: `IF color THEN blue ELSE dark`.

Note that flexible applications are not the same as applications that simulate window sharing by knowing about the various observers and duplicating windows on multiple displays. Flexible applications need never be aware of the types of displays on which their windows are being rendered at any particular point in time. They simply give their windows conditional attributes so that the server can perform the adaptation for display type.

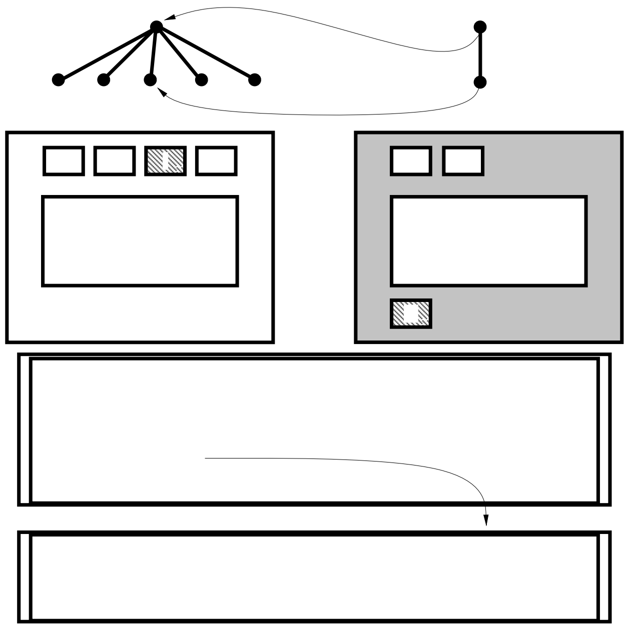

Figure 9: Customization via Attribute Inheritance.

Simple conditional attributes have their limitations, however, since no application can be expected to explicitly specify conditional attribute values for every conceivable type of display. For example, suppose text is to be displayed in a text editor window. One type of display has 300 dots per inch resolution, while another has only 100 dots per inch. If we hold fixed the size of the window in pixels in both cases, the window on the high resolution display will be 1/9th its size on the low resolution display, which is probably unacceptable. If, however, we hold fixed the size of the window in inches, different fonts should be used on the different types of displays to ensure that the fonts are attractive and legible. This type of problem can be resolved by specifying certain attributes as patterns rather than as fixed values. A pattern can specify the characteristics required of a font without explicitly naming the font, and the server or display client can complete the selection of a particular font meeting the requirements.

2.2.3. Display Customization

It is sometimes desirable for the display client rather than the application client to specify coupling constraints on windows, either to accommodate for heterogeneity or to allow a user to customize his view of windows. As an example of the former, the user viewing a window may wish to decide whether the size of the window in pixels or its size in inches is fixed, in the presence of differences between the resolution of the window as created by the application client and that of his display. A user may also want to customize his view of windows in minor ways (*e.g.* by changing colors), or in more elaborate ways by changing the arrangement of subwindows or even the set of subwindows displayed (by adding his own subwindows or excluding certain subwindows of the shared parent window). Some such customization is be automatic, because the viewer may not have permission to access all of the subwindows of a shared window. Other customizations must be performed at the request of the viewer.

Customization by the viewer can be supported in the server by an attribute inheritance mechanism (Figure 9). Windows can specify their own attributes explicitly or inherit individual attributes from other windows. Window Q specifies its background color to be gray, but otherwise inherits all attributes from F (except for its children, which are also attributes). It specifies that it does not have windows I or J in order to avoid inheriting them directly from F, and specifies a new window, R. R, in turn, inherits everything from F/I except its location, which is specified explicitly.

Since the contents (pixels) of window Q are inherited from window F, drawing into either window has exactly the same effect: any other physical windows inheriting the contents of either of these windows will see the drawing requests. Events work similarly; events sent to either window are sent to both and to any other window inheriting from either. Window contents and event inheritance can be overridden by setting explicit attribute values. We have not yet developed a complete understanding of the proper exact semantics of various types of attribute inheritance.

Now note that tightly-coupled window sharing, discussed in Section 2.2.1, is just a special case. It can be accomplished by simply having one window inherit all the attributes of another without specifying any explicitly. Conditional attributes still apply, so that even tightly-coupled windows can be viewed on displays with different capabilities if the application client creating the window is flexible.

Note also that attribute inheritance can be used to implement certain characteristics of existing window systems, such as inheriting attributes (*e.g.*, background color) down and propagating events (*e.g.*, button presses) up the window hierarchy.

2.2.4. Window Management

Given a SDWS with the characteristics described in the preceding sections, window management (as viewed by the X Window System) becomes trivial. It is simply a matter of customizing the top levels of the window hierarchy to add title bars and borders and the operations associated with them (window placement, movement and resizing, for example). Window management, traditionally restricted to top-level application windows, can also be combined with management of sub-windows for customization purposes.

As mentioned earlier, there is really no difference between application and display clients, other than to make a rough distinction between the clients that are creating “original” windows, drawing in them, and responding to events occurring within them (application clients) and clients primarily responsible for creating a physical image of a window hierarchy (display clients). In the above discussion, we have chosen to assign display customization to the display clients. Customization can, however, be performed by a third party client (as is done with window management in X), since it just involves manipulating the window database in a server; the display client can just display the hierarchy constructed by the third party. Using a third party client might be desirable if, *e.g.*, customizations themselves are to be shared.

3. Designing a Non-Inverted Window System

We claim that the architecture of the X Window System in particular, and most distributed window systems in general, makes sharing windows among users difficult because the window database is maintained by a server that is tightly coupled with an individual display. Separating the window server from the display and implementing display-specific functions in a client of this window server facilitates direct sharing of windows among users.

We are currently implementing a shared distributed window system prototype having the proposed architecture, and demonstrating that it has substantially better window-sharing properties than X. In particular, we will demonstrate the following, none of which are available in X:

- A hierarchical window name space for accessing windows from application and display clients.
- A suite of access controls for windows sufficient for implementing a variety of volatile and non-volatile window system operations and conference control (*e.g.*, floor control) functions.
- Support for heterogeneity (*e.g.*, accommodating displays with different color attributes and resolutions) through conditional object attributes.
- An attribute inheritance mechanism for window attribute sharing — a powerful tool for supporting display customization in the form of both non-structural (*e.g.*, color) and structural (*e.g.*, subwindow rearrangement) customization.

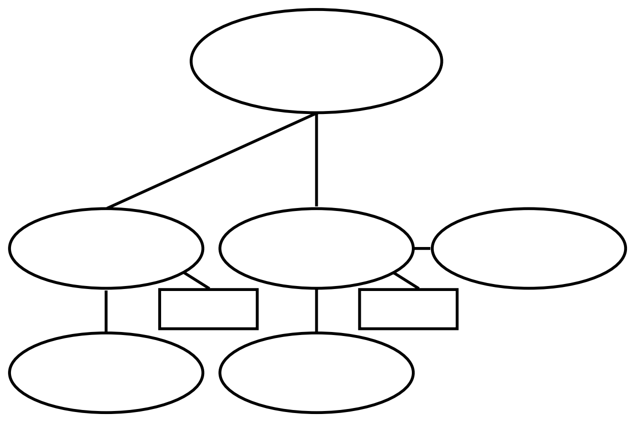

Figure 10: Logical Implementation of the SDWS Prototype.

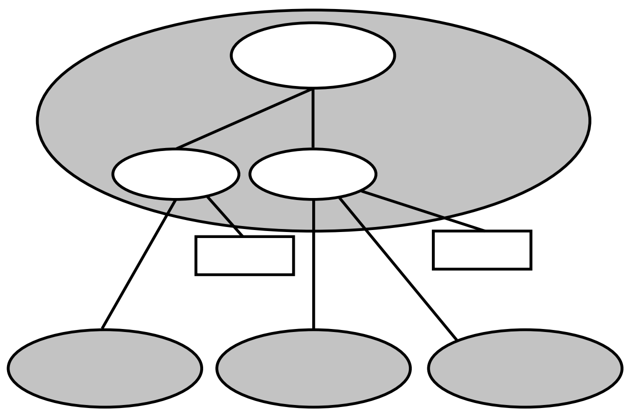

Figure 11: Process Architecture of the SDWS Prototype.

- Window management mechanisms that can be naturally extended to include customization of the views of all windows, not just top-level application windows.
- Overall flexible and lightweight window-sharing, the ability to drag arbitrary windows into and out of shared virtual screen windows and to “park” a virtual screen and pick it up again on a different display.

3.1. Overall Implementation Strategy

We now describe the overall implementation of our SDWS prototype. The system uses the X Protocol Engine Library (XPEL) [13], a C++ class library we’ve developed for constructing X pseudo-servers (XPSes).

The prototype SDWS is built top of the existing X Window System, utilizing existing X clients and servers. Logically, X clients and servers are augmented by XPSes (Figure 10) to make them application and display clients, respectively. (An XPS is a process placed on the communication link between an X client and server, which appears to the client to be a server, and vice versa. It manipulates the X protocol streams between client and server to add capabilities to the window system.) Information not supplied by existing servers and clients is provided by initialization files for the XPSes. Window management functions are performed by the display client, perhaps utilizing an existing third party window manager connected to the display client. The SDWS server itself is implemented as another XPS.

To reduce round-trip latency to 4 hops instead of 8 between X clients and servers in the implementation (in order to get the same number of hops as an SDWS implementation “from scratch”), the process architecture for the prototype is as indicated by Figure 11, with the gray ovals representing processes. XPEL supports collapsing adjacent XPS processes in this way.

The XPS process (the combined SDWS Server, Application Client, and Display Client, in the figure) is implemented using XPEL. The SDWS Server, Application Client, and Display Client are each a composite filter composed of some network of lower-level filters.

3.2. Limitations of the Chosen Strategy

The decision to build on top of X is a practical one; it eliminates the need to implement applications and display drivers and also makes the prototype useful in existing environments. The primary drawback to this approach is that certain desirable window sharing functions cannot be fully implemented using this approach.

For example, one property a SDWS must have is the ability to perform request/reply transactions in both directions between clients and servers (Figure 12). (Such request/reply transactions are only supported from client to server and back in the X protocol. This has proven inadequate even in X, as Inter-Client Communication Conventions above and beyond the X protocol have had to be developed to implement copy and paste operations.)

For an example of how request/reply transactions from server to client and back are needed, consider implementing efficient and non-distracting window damage repair of shared windows. To avoid storing huge numbers of pixel values in the server (or display client), window systems like X require that, on request, an application redraw portions of a window that have been damaged by being obscured by another window and then revealed. In X, applications are notified of window damage by a one-way event. The drawing requests sent in

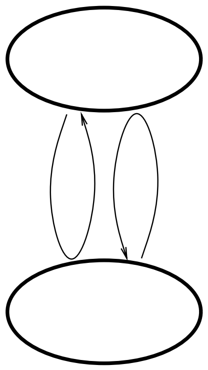

Figure 12: Symmetric Request/Reply Transactions.

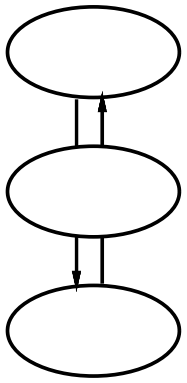

Figure 13: An X Pseudo-Server.

response to the event are not distinguished from other drawing requests. In a shared window environment, this means that the redrawing must be done on every physical display, not just the one where the damage occurred; this is inefficient and distracting. A two-way window damage request/reply transaction would enable redrawing to occur only on the damaged display.

A more elaborate example is when a user pulls down a menu using a mouse button. Under certain window coupling models, only the user invoking the action should see the menu; the others should only see the effect of the action invoked (if any) on less transient windows. This suggests that the response of a client to any event should indicate the event that triggered the response.

It is impossible in general to implement such request/reply transactions from server to client and back using existing X clients. To demonstrate the desirability of the proposed approach, we make an attempt to identify client responses to events using per-application heuristics described in the application client's initialization file.

3.3. The X Protocol Engine Library

The X Protocol Engine Library (XPEL) is a C++ class library supporting the design and implementation of XPSes using the X Protocol Engine Architecture (XPEA).

An XPS is a process interposed between an X server and client that adds capabilities to the X window system by manipulating the protocol streams passing between client and server (Figure 13). It is a common means of adding window sharing and tracking/replay capabilities to X. The advantage of this approach is that existing X clients and servers need not be changed to take advantage of the new capabilities, so the new capabilities are made available to all existing clients and servers. The primary limitation of this approach is that it is not adequate for implementing capabilities that require applications to be aware that they are being used collaboratively.

XPSes can be difficult to design, implement, and maintain. Attempts to date have resulted in systems that are large and complex but low in function and reliability [3]. XPEA was developed to provide an organizing framework for the design of XPSes. XPEL supports the XPEA architecture and dramatically simplifies the implementation of XPSes by enabling the programmer to work at a higher level of abstraction and by encouraging code reuse.

XPEA is an object-oriented architecture that uses a model of interconnected protocol manipulation filters, like that sometimes used in the implementation of inter-process communication facilities [12, 8, 14, 18]. The task of implementing a particular window system capability can be broken down into component tasks performed by small filters. These filters can then be combined in series to accomplish the larger task (Figure 14).

The XPEL class library includes predefined filter classes to perform various X protocol manipulation functions. Programmers can choose from these filters and write filters of their own to piece together into solutions to larger problems. The object-orientation of the class library also allows decomposition of solutions in the class inheritance dimension. Thus, all user-defined filters are sub-classes of at least the top-level filter super-class in the library, which implements generic filter functions.

XPEL also includes a message class for each protocol unit (X message type). Message objects are created as the messages arrive on the communications link; they also can be created by filters and inserted into a message stream. The object-oriented nature of XPEL makes it possible for many message manipulations to be performed polymorphically, where the individual message types implement generic operations as they apply to that particular message type. A scheduler is provided that

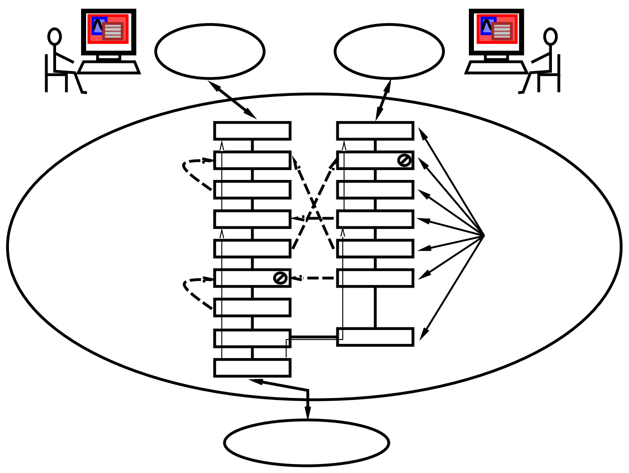

Figure 14: An XPEA-based XPS.

schedules the handling of events such as message arrivals, and the movement of messages through the filter network.

In the XPSes developed to date, message streams have been passed all the way through the XPS, with the streams being monitored and/or multiplexed along the way, as show in Figure 14. Thus, the sharing or tracking/replay capabilities have been implemented on a per application basis; all windows created by a given application are affected together. In the SDWS, individual windows must be shared rather than whole applications. This requires the SDWS to be an endpoint of the message streams (Figure 15). Conversion to this architecture would likely be very difficult in existing XPSes; the organization imposed by the XPEA makes it easy.

4. Related Work

There are three distinct areas of related work. The first is the area of collaboration support systems for collaboration-aware applications. The second area covers the work in shared window systems, which have been designed primarily to allow single applications to be used by multiple users. The last area involves filter-based protocol manipulation systems similar in architecture to XPEL.

4.1. Supporting collaboration-aware applications

A collaboration-aware application is an application that is aware that it is being used collaboratively. This usually means that it is a multi-user application, although single-user applications utilizing a shared data object can also be collaboration-aware. Collaboration-aware applications are sometimes implemented in an ad-hoc manner without the support of collaboration toolkits, but a better approach is to isolate the data- and view-sharing functions into a collaboration toolkit. It is sometimes possible to keep

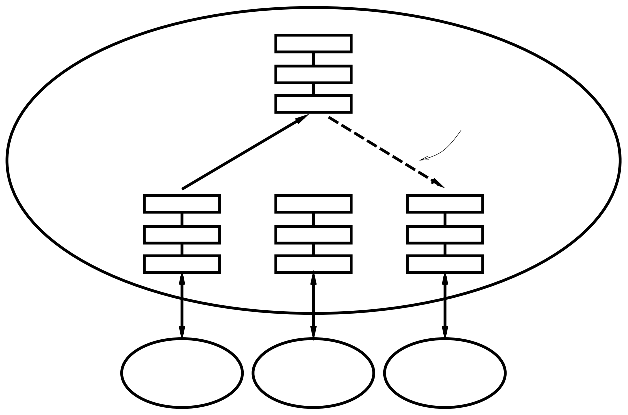

Figure 15: Filter Network Structure in an XPEA-based SDWS.

collaboration awareness out of the toolkit's programming interface, thus removing it from the application proper.

A number of collaboration support systems (*e.g.*, Rendezvous [17], MMConf [5], and Suite [6]) facilitate the development of collaboration-aware applications via language extensions or toolkits that allow the programmer to develop his applications at a high level of abstraction, leaving details such as synchronization and shared state management to the support system. Most such systems provide both user interface and data sharing functions, since the user interface components are often displaying shared data; they may even *be* the "data" that is to be shared.

The SDWS described in this paper attempts to separate the sharing of user interface components (windows) from the sharing of data. It does not provide mechanisms for sharing data objects other than windows. But it is not possible to solve all collaboration problems by simply sharing single-user applications or their windows; often other data objects must be shared. Thus a SDWS cannot take the place of collaboration support systems for collaboration-aware applications. It can, however, ease the development of such support systems by taking over the problems of window naming, access control, concurrency control for shared windows, replication of shared windows, display heterogeneity, and simple view customization. It also has the advantage of being able to bring existing single-user applications into the collaborative environment.

It should be possible to push higher-level user interface abstractions (widgets) into the SDWS server, thus making not only the lower level abstraction of windows sharable, but also the higher-level abstraction of widgets. This in turn would solve a common problem with interactive user interfaces — the fact that they often can be manipulated only by people. Sharable widgets could be directly manipulated by applications.

4.2. Shared Window Systems

Many attempts have been made to implement window sharing using the X Window System: SharedX [7], Shadows [15], and XTV [1], to name a few. Some are implemented as modified window system libraries; as such they need to be linked into an existing application. Most, however, are implemented as XPSes.

The problems encountered when designing and implementing X-based shared window systems are many, but they are fairly well known and documented [15, 16, 9, 10, 2]. They include both problems regarding design decisions and problems in implementing chosen designs. (Implementability, of course, affects the design decisions that are possible.)

Design decisions for shared window systems include:

- **Floor control**. Who can interact with a window at a given point in time?
- **Annotation**. Can a user make notes over the top of a window?
- **Telepointing**. How many pointers are visible to all users? None? One? One per user?
- **Workspace Management**. Can shared windows be independently positioned by each user, or is a whole workspace containing multiple windows shared [11]?
- **Cut and Paste**. What should be the cut and paste semantics in a shared window environment?
- **Window Sharing Granularity**. Are all the windows associated with an application shared as a unit, or top-level windows individually, or all windows individually?
- **Window attribute coupling** [6]. To what degree is there flexible coupling in the sharing of window attributes, both structural (*e.g.*, subwindow arrangement) and non-structural (*e.g.*, colors)?

Implementation issues include:

- **Late-comers, Spontaneous Sharing, and User Migration.** How can a new participant be added to the set of users sharing a set of windows after a conference has begun [4]? How can a non-shared window be made sharable (and vice-versa)? How can a user move a window from one display to another (park it and pick it up elsewhere)? These are all similar problems with similar solutions.
- **Heterogeneity.** How can we accommodate displays with different capabilities?
- **Window Damage Control.** How can window damage (which occurs when windows are partially obscured and then revealed) be repaired efficiently and non-disruptively?
- **Coordinate Mapping.** If windows are not positioned identically on different displays, how can we get the coordinates right? This is especially difficult with transient windows, which have an implied relationship to non-transient windows that is not reflected in the X protocol stream.
- **Resource Mapping**. Different X servers have different identifiers for windows and other resources. How can we map between these name spaces?
- **Sequence Numbers.** Since X uses an asynchronous protocol, X clients rely on sequence numbers to match messages from servers to the corresponding client request. When message streams are modified (and especially when they are multiplexed), how can the sequence numbers be kept correct so clients don’t get confused?

Even though these problems have been known for some time, existing window-sharing systems continue to be large, complex, and unreliable, while they are difficult to use and the features they support are few [3]. Why is this the case? We suggest the following three reasons:

- **Lack of a Good XPS Architecture**. Even when the set of problems and their solutions are well known, combining them into a functioning shared-window system is complex. The XPEA architecture addresses this problem by providing a structuring mechanism that allows problems to be decomposed into small, easily understood components that can be solved individually and pieced together into the solutions to larger problems.
- **Lack of a Good Shared Window System Architecture**. The common view of a shared window system implementation as an XPS that simply modifies and distributes X protocol streams between clients and servers is deficient. The client/server architecture of the window system itself needs to be reorganized as proposed in this paper. This reorganization allows individual windows rather than entire applications to be shared.
- **The X Protocol is Inadequate for Window Sharing**. Certain problems simply cannot be solved in systems built on top of the X protocol, without creating elaborate extensions to X that all clients must be modified to use. An example is the fact that reply/request transactions directed toward applications are not supported in the protocol. We hope to define the necessary characteristics of a shared window system protocol.

In summary, the proposed SDWS is a new shared window system, mostly but not totally implementable as an XPS.

Our prototype promises to do a better job of providing window sharing capabilities because it is modeled on a better window system architecture and a better XPS implementation architecture than previous attempts. It also points the way toward how a complete SDWS might be implemented.

4.2. Other Filter-Based Architectures.

Filter-based protocol manipulation systems like XPEL have been used in several inter-process communication (IPC) facility implementations. Three of these are the *x*-Kernel [8], Packet Filters [14], and Eighth Edition Unix Streams [18]. (For a detailed comparison of XPEL and these systems, see [12].) Briefly, the high-level architectures of all four systems are similar; all use a network of filter-manipulation objects through which message streams are passed and by which they are manipulated. XPEL, because it is the latest of the systems to be designed, makes better use of object-oriented concepts. This is most notable in the use of “smart” messages that can be manipulated polymorphically. Several concepts used in some of the other systems pose intriguing possibilities for future enhancements to XPEL; these include special in-stream control messages, multi-protocol support (required for the SDWS implementation), and multi-thread support.

At first it appears that XPEL is the only one of the systems not primarily designed for implementing IPC facilities. The proposed SDWS architecture, however, suggests a different view. Shared windows can be, and perhaps should be, viewed as a high-level, specialized IPC facility tailored to the sharing of the window abstraction among processes.

5. Summary

To date, shared window systems have been inflexible in their support for synchronous collaboration and difficult to build. We have argued that the essential problem with such systems lies in their inability to directly support the sharing of windows as abstract objects. This sharing is unnecessarily impeded by the inverted nature of the client/server architecture of the underlying distributed window system.

By applying standard, well established abstractions and mechanisms from the distributed file system domain, we have developed an architecture for a true shared, distributed window system wherein all objects managed by the window system can be shared in a flexible, straightforward manner. A prototype of a such is shared distributed window system is being developed on top of the X Window System using tools for the efficient, modular construction of pseudo servers.

Acknowledgments

This work is supported in part by the National Science Foundation (Grant # IRI-9015443), and a Graduate Fellowship from the IBM Corporation.

6. References

[1] 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.

[2] Hussein M. Abdel-Wahab and Kevin Jeffay. Issues, Problems, and Solutions in Sharing X Clients on Multiple Displays. *Internetworking — Research and Practice*, 5(1), (March 1994) pp. 1-15.

[3] John Eric Baldeschweiler, Thomas Gutekunst, and Bernhard Plattner. A survey of X Protocol Multiplexors. *ACM SIGCOMM Computer Communication Review*, pages 16-24, April 1993.

[4] Goopeel Chung, Kevin Jeffay, and Hussein M. Abdel-Wahab. Dynamic Participation in a Computer-based Conferencing System. *Computer Communications*, 17(1), (January 1994) pp. 7-16.

[5] Terrence Crowley, Paul Milazzo, Ellie Baker, Harry Forsdick, and Raymond Tomlinson. MMConf: An Infrastructure for Building Shared Multimedia Applications. In *CSCW ‘90 Proceedings*, 1990.

[6] Prasun Dewan and Rajiv Choudhary. Flexible Interface Coupling in a Collaborative System. In *Proceedings ACM CHI ‘91*, pages 41-48, New Orleans, LA, April 1991.

[7] P. Gust. Shared X: X in a Distributed Group Work Environment. Unpublished paper presented at the Second Annual X Technical Conference, January 1988.

[8] Norman C. Hutchinson and Larry L. Peterson. The *x*-Kernel: An Architecture f or Implementing Network Protocols. *IEEE Transactions on Software Engineering*, 17(1):64-76, January 1991.

[9] 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.

[10] J. Chris Lauwers and Keith A. Lantz. Collaboration Awareness in Support of Collaboration Transparency: Requirements for the Next Generation of Shared Window Systems. In *CHI '90 Proceedings*, 1990.

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

[12] John Menges. A Comparison of the Architectures of the X Protocol Engine Library and Three Related Systems. Technical Report TR93-047, University of North Carolina, Chapel Hill, North Carolina, May 1993.

[13] John Menges. The X Engine Library: A C++ Library for Constructing X Pseudo- Servers. In *Proceedings of the Seventh Annual X Technical Conference*, January 1993, pp. 129-141.

[14] Jeffrey C. Mogul, Richard F. Rashid, and Michael J. Accetta. The Packet Filter: An Efficient Mechanism for User-level Network Code. In *Proceedings of the Eleventh ACM Symposium on Operating System Principles*, pages 39-51, November 1987.

[15] 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.

[16] John F. Patterson. What Does Window Sharing Say About Virtual Terminals? Unpublished paper written at Bellcore.

[17] John F. Patterson, Ralph D. Hill, Steven L. Rohall, and Scott W. Meeks. Rendezvous: An Architecture for Synchronous Multi-User Applications. In *CSCW '90 Proceedings*, 1990.

[18] D. L. Presotto and D. M. Ritchie. Interprocess Communication in the Eighth Edition Unix System. In *Proceedings of the 1985 USENIX Association Summer Conference*, pages 309-316 , June 1985.