Beyond Window Sharing Hacks: Support for First-Class Window Sharing

John Menges

Technical Computing R&D Center
Hewlett-Packard Company
Corvallis, OR 97330

Kevin Jeffay

Department of Computer Science
University of North Carolina
Chapel Hill, NC 27599

Abstract

Over the past 8 years there have been numerous attempts to develop add-ons to the X Window System that enable basic sharing of arbitrary X applications. These systems have never been much more than interesting and marginally useful incremental improvements to the capabilities of X. It is our thesis that there are as-yet-unexplored opportunities for the development of new window systems that provide integral, full-featured support for collaborative work and retain substantial backward-compatibility with X. In this paper we present our design for such a window system.

1 Introduction

Suppose you are collaborating with a colleague to write a paper, and you and your colleague both have access to a full-featured distributed file system. Further assume that this is a paper that you don't want to allow others to see until it is published. You can provide drafts of the paper to each other by placing them in the file system and setting the appropriate access controls to allow read permission only to yourself and your colleague. If your editor supports file locking, both authors can have write access to the file, so that you can take turns editing it, or you can break down your paper into component files, where each author has write access to the parts of the paper over which he has ownership. If you are temporarily seated at a different work station, you can still access the files as you could from your office.

Now suppose you prefer to collaborate more interactively by sharing the editor window on your display rather than going through the file system. Current, widely-deployed window systems do not support this type of sharing. Even if you are fortunate enough to have an X-based shared-window system available for your use, it is not likely to provide you with much more than simple window sharing, a floor passing mechanism to enable you to take turns driving the editor, and primitive forms of access control and window naming. If you move from one work station to another, you are likely to either have to run back and forth between your two work stations to set up window sharing to yourself, or to be out of luck altogether.

Ideally, we would like windows in our window systems to be much like files in our file systems. That is, they should be first-class sharable objects, residing in a hierarchical name space, to which access controls and concurrency controls can be applied. In addition, there are other capabilities specific to sharing windows that we would like our window systems to support.

This paper describes Concur, a new window system specifically designed to support the sharing of windows among users in different locations and the migration of windows from one location to another. Current popular window systems were designed under the assumption that a given window would only be displayed on a single workstation throughout its lifetime. Many attempts have been made to design window sharing and migration systems on top of existing window systems. The X Window System[9] is most often the underlying window system of choice, because it uses a network protocol that is easy to intercept and route to multiple workstations. Unfortunately, there are aspects of the X protocol that make it unsuitable as a basis for implementing natural window sharing and migration paradigms.

In Concur, we have taken a different approach. Concur is a new window system, designed with window sharing and migration in mind from the outset. As an academic exercise this is interesting in itself, and the ideas presented here can contribute to the design of future window systems. But we also demon-

strate that existing window system protocols can be efficiently mapped to a subset of the Concur protocol, making most of Concur's capabilities available to existing window-based applications.

Concur was developed as a research vehicle as we sought to define the boundary line between the capabilities of collaboration aware (i.e., multi-user) and unaware (single-user) applications. We believe that there is a large class of applications that do not need to be collaboration aware, even if it makes sense to use them collaboratively. With Concur we are attempting to expand the usefulness of such collaboration-unaware applications in a collaborative setting. Concur also provides a good foundation for the development of collaboration aware applications.

Concur's architecture is largely based on that of distributed file systems, which have many of the characteristics we desire in a distributed window system. Thus, windows are sharable, first-class objects residing in a hierarchical name space, to which access and concurrency controls can be applied.

2 Systems Support for Synchronous Collaboration

There are two major types of systems support for distributed synchronous collaborative use of applications: collaboration toolkits[3][5], which support the development of collaboration aware applications, and shared window systems[2][1][4], which enable collaboration-unaware applications to be shared.

Collaboration toolkits seek to facilitate the development of collaboration-aware applications by abstracting out the common low-level functions required by most such applications and providing them to programmers in the form of a toolkit for building collaboration-aware software. These toolkits provide communication, coordination, synchronization, and consistency maintenance capabilities to the programmer. They usually implement some notion of shared data objects and provide mechanisms for maintaining multiple distributed concurrent views of these objects. The development of these toolkits has been and continues to be an active area of research. While the toolkit approach is a good one for the development of applications that are necessarily collaboration-aware, it requires such applications to be written in the context of the toolkit. Thus, it does not enable us to use existing collaboration-unaware applications collaboratively, nor does it help us to write new collaboration-unaware applications intended for collaborative use.

These latter concerns are addressed by shared window systems. Traditional window systems assume single-user access to windows. If we implement a new window system that directly supports concurrent multi-user access to windows, applications using this window system can be used collaboratively without being collaboration-aware. This new window system can be viewed as a collaboration toolkit where the shared data objects are windows. Unfortunately our window system has the same drawback as other collaboration toolkits – applications still must be written to conform to the new model. If, however, our new window system is also capable of providing exactly the same interface as an existing window system, we will have met our remaining goal – to make existing collaboration-unaware applications available in a collaborative setting. Furthermore, a shared window system can provide basic window-sharing functionality to collaboration toolkits, making them easier to write and encouraging uniformity among toolkits for window sharing operations.

3 Functional Requirements for Shared Window Systems

Ideally, a shared window system must do more than just make windows accessible to multiple users at different terminals. It must also address some or all of the following issues:

- **Sharing Granularity.** What is the atomic unit of sharing? For example, if the application is the atomic unit, then all of an application's top-level and sub-windows and all of their attributes must be shared as a unit. If, on the other hand, the top-level window is the atomic unit, individual top-level windows can be shared. Possible atomic sharing units include: the application, the top-level window, the window, and the window attribute (e.g., width, height, color, or content).

- **Grouping.** Is it possible to group entities (people and objects) so that operations may be performed on the group as a whole? For example, even if the atomic sharing unit is the subwindow, one will normally wish to share one or more subtrees of windows, each rooted at a top-level widow. A set of applications or top-level windows shared as a unit with a particular set of users is often called a *conference*, and users belonging to a conference are called *participants* in that conference.

- **Naming.** How are shared windows, users, terminals, and sets of these entities named so that they can be referred to as parameters of operations?
- **Access Control.** Who is allowed to perform which operations on which windows? For example, who can become member of a particular conference (i.e., who can view its windows), and which members are permitted to give input through a particular window?
- **Floor Control.** Which terminals are allowed to interact with a particular window at a given point in time? This is particularly important for applications that assume a single pointing device (that is, most current applications). What is the granularity of floor control – is there one floor per conference, per application, or per top-level window?
- **Group Membership Dynamicity.** Is the membership of groups of users, terminals, or shared windows dynamic? For example, can we bring a new or unshared application or a new terminal into an existing conference? Can applications be removed from the conference without being terminated, and can users leave a conference or move to another terminal? Can floor control be passed among conference participants?
- **Workspace Layout.** Are all the windows in a given conference grouped physically or otherwise distinguished as being part of this particular conference? Are they arranged the same way on every display so that relative window positions are shared?
- **Customization.** Is it possible for various participants in a conference to customize their views of shared windows and their ways of interacting with them? For example, can a participant change the colors of his view of a window or choose between *emacs* and *vi* key bindings for an editor window? The ability to customize views is largely determined by the sharing granularity and workspace layout. Customizing interaction (e.g. key bindings) is a difficult issue because key bindings are normally mapped to operations by the application itself rather than the window system.
- **Temporal Coupling.** Must the various copies of shared windows always be kept as up-to-date as possible, or will an occasional update suffice? Can passive participants' displays lag in order to give the user with the floor better performance? Is the degree of coupling adjustable either manually by participants or automatically in response to system or network load?
- **Gesturing.** Is it possible for conference participants to gesture with a pointer that other participants can see? Is there one shared pointer, or one per participant, or are there an unbounded number of shared pointers?
- **Annotation.** Is it possible to annotate an active window by typing or drawing over it? If not, can static snapshots of windows be created and so annotated? Can such annotations be erased, stored, and printed?
- **Cut, copy, and paste.** Do cut, copy, and paste operations work when the source or destination window is shared? Are these operations private to a particular terminal (or user), or are they shared so that one person can copy and another paste?
- **Voice and Video.** Are voice and video linked to a conference or are separate mechanisms (like the telephone) used if these are needed?
- **User Interface.** How are sharing operations invoked? Possibilities include line-oriented commands, control panel operations, and direct manipulation.

Shared window systems must also address the related issues of *performance* and *scalability*. Interaction with unshared windows should not be noticeably degraded by the shared window system. When windows are shared, any degradation relative to unshared windows must be acceptable for common participant set sizes and should increase slowly as the number of participants increases.

Nearly all of the previous X-based shared window systems support the basic sharing of an application's windows among a set of users at different terminals, with each user capable of both viewing and interacting with the windows. Most also provide some sort of floor control. Some provide support for *latecomers*, terminals or applications dynamically added to the conference after it is already in progress. Some provide a *virtual screen* grouping mechanism for windows that keeps the spatial relationships between shared windows consistent for participants. Some provide at least limited support for heterogeneous terminal types, font mapping, X extensions, and snapshot overlays. Most do not support meaningful cut-and-paste operations involving shared windows or a smaller atomic sharing unit than the application. To our knowledge, none

supports a global window name space, user-based access controls on windows, sharing of individual subwindows, customization of views or interactions, direct manipulation techniques for performing common window sharing operations, or active window overlays. Many are too slow to be used for all of a user's windows, making sharing of arbitrary windows impractical. Finally, many such systems scale poorly as more participants are added.

4 The Concur Window System

Concur is being designed with the goal of supporting all of the following capabilities:

- Viewing of and interaction with any window from zero or more terminals simultaneously, subject only to access and concurrency controls (below). The set of viewing terminals is dynamic.
- Sharing at all levels of granularity from the application to the individual window attribute. Fine-grained sharing enables the customization of the views of windows.
- A hierarchical, string-based name space for windows, like that of traditional file systems.
- User-based access controls on windows or window sub-trees, by class of window operation. Windows have owners and access control lists similar to those available in distributed file systems.
- Locks on windows and window sub-trees for concurrency control. These can be used to implement various floor-control policies.
- Customization of user interaction with windows. Concur makes it possible for each user to individualize key and mouse bindings for their interactions with shared applications.
- Sharable pointers that can be made to track a user's cursor, for gesturing purposes.
- Window overlays for annotating active or static windows.
- Cut, copy, and paste operations involving shared windows. These operations can apply to either private or shared clipboards, as specified by the user.
- Intelligent change notification mechanisms that use protocol analysis, load measurements, optional user hints, and heuristics to improve the overall performance and scalability of the window system.
- Conference, workspace, window, and customization management via **ConMan**, a Concur window system application with special duties but no special privileges. Direct manipulation techniques are used wherever possible to provide an intuitive user interface. For example, one can share and unshare arbitrary windows by dragging them into and out of shared virtual screen workspaces.
- A mapping between the X and Concur Window System protocols that enables existing X applications to make use of most of the above capabilities. Optional per-application and per-user initialization files can be used to provide information missing in the X protocol. In the absence of these aids, Concur uses heuristics to fill in the blanks.

5 The Concur Architecture

Figure 1 shows the high-level architecture of the X Window System. The window server maintains the window database, and X clients (applications) communicate with the server over a network connection to create windows, draw in them, and receive input through them. X is a *distributed* window system in the sense that applications need not be running on the host that displays the application's windows. In a distributed file system, on the other hand (Figure 2), the term *distributed* takes on a different meaning. Here it means that the user(s) of the files managed by the file system can be in separate locations; i.e., that the files can be shared by users in different locations. We desire our window systems to be *distributed* in the sense that the word is used when applied to distributed file systems. In an earlier paper[8], we analyzed this problem in detail. The conclusion was that the X Window System's architecture made window sharing difficult because its client/server architecture is inverted with respect to that of distributed file systems. The X server, where windows reside, is co-located with the user, while the file system server, where files reside, is potentially remote (see Figures 1 and 2).

In Concur, the window database still resides in the window server, but the window server does not directly drive the display (Figure 3). The display is

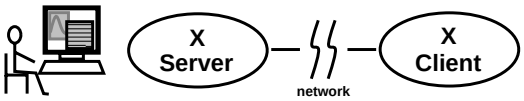


Figure 1: X Window System

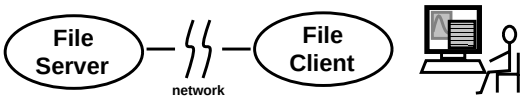


Figure 2: Distributed File System

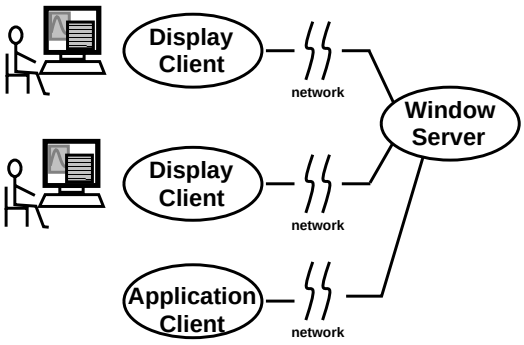
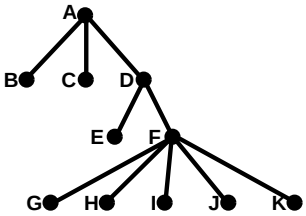


Figure 3: Concur Window System

driven by a special client of the window system, called a *display* client to distinguish it from a normal application client. The display client connects to the window server and expresses an interest in a particular set of windows. The window server then sends the state of these windows to the display client and updates the client when changes are made to the windows. (In order to avoid writing display drivers for all the various kinds of displays, we use X Servers to perform this task. The windows residing in the X servers are only cached copies of those in the Concur server.)

Note that application clients are not aware of how many (if any) displays are rendering their windows, or where these displays reside. Window sharing occurs when more than one display client expresses an interest in the same window. Window migration is accomplished by dropping interest in the window from one display client and picking it up from another.

The application client is also unaware of the particular characteristics of the displays on which their windows are rendered. The Concur window server presents a view of an abstract generic display to the application clients, which they use to build their user interface. Each display client maps this user interface to the particular type of display it is driving.



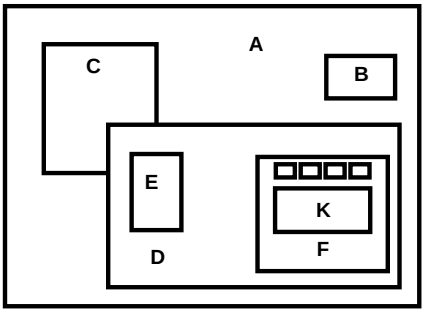


Figure 4: Concur Window Hierarchy

In a practical implementation of the Concur Window System, the window database would be distributed among multiple servers, just as the file database is distributed among multiple file servers in a

distributed file system. This makes it possible to have separate administrative domains for implementing access controls, facilitates good performance by allowing windows to be located near the displays where they are most likely to be rendered, and improves scalability by increasing parallelism. However, the Concur prototype currently only supports a single server.

Concur's window database is similar to the file database in distributed file systems. Windows are arranged hierarchically, such that sub-windows of an enclosing window are represented as children of the enclosing window (Figure 4). Windows have string-based names, defined by the path taken from the root of the hierarchy to the target window. Each window has an owner and an access control list that determines what operations can be performed on the window by which users. The operation types include viewing, modifying the subwindow structure, drawing into the window, sending input through the window, etc.

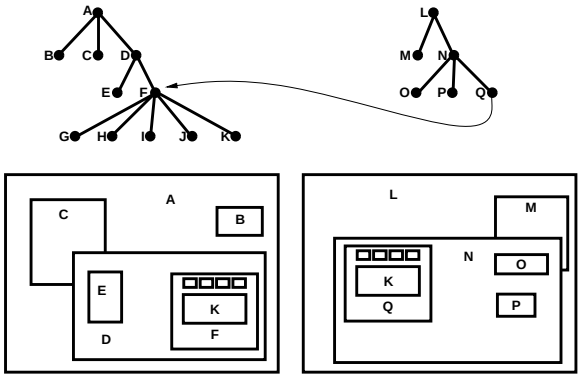


Figure 5: Sharing via Linking

Basic WYSIWIS (what you see is what I see) sharing of a subtree in the window hierarchy can be accomplished in Concur via window links similar to file system links or mounts (Figure 5). Here, window **A/D/F** shown on one display is the same as window **L/N/Q** shown on another.

Finer-grained, customized sharing can be accomplished using an attribute inheritance mechanism as shown in Figure 6. Here, window **Q** is the same as window **F**, except that some of **F**'s attributes have been overridden. On the right, the background color of **F** (as **Q**) has been overridden to make it gray, sub-window **J** has been deleted, and sub-window **I** (as **R**) has been relocated.

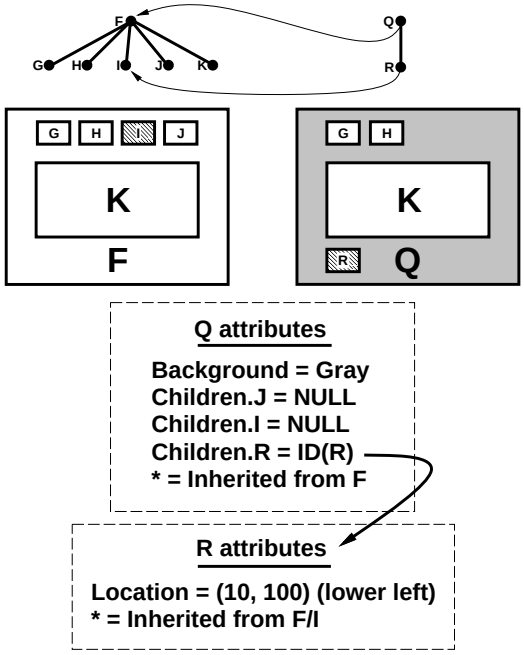


Figure 6: Attribute Inheritance

6 Input Binding in Concur

X Window System applications generally define their own bindings between input events (e.g., keyboard or mouse inputs) and the actions to be taken in response to these events. This is troublesome when applications are shared using a shared window system, because the users interacting with the application are not able to customize the bindings to suit their own preferences. (For example, one might prefer *emacs* key bindings, and another *vi* bindings.) Little can be done to solve this problem for existing X applications shared using Concur.

New applications written for the Concur Window System will, however, be able to take advantage of an improved input event binding mechanism. Concur will support input event binding within the server itself, and will be able to maintain different bindings for different users. The Concur application will export all the operations that can be directly invoked via user interactions to the Concur server, along with default bindings to user events. In the Concur server, these default bindings can be overridden on a per-user basis, as specified by the users. When an input event triggers a particular binding to an operation, a specification of that operation will be sent to the Concur application,

where the actual operation will be invoked. Thus, not only the appearance of windows but also the user's protocol for interacting with the application can be customized.

7 Implementation Framework

The Concur prototype is being implemented using a protocol engine development framework of our own design. A *protocol engine* is a machine whose purpose it is to monitor and/or manipulate conversations among computer system components. A simple protocol engine might monitor and log such conversations for debugging or auditing purposes. A more complex one might serve as a translator among components that use different but equivalent protocols, enabling them to communicate despite the direct incompatibility of their protocols. Protocol engines can also serve as protocol recorder/replayers, component simulators, and protocol multiplexors and demultiplexers.

Concur components (servers and display clients) are protocol engines that intercept and manipulate conversations between graphical applications and the display drivers on which they present their user interfaces. Because of our previous experience building protocol engines[1][6], we were prepared and motivated to begin the work on Concur by creating a general protocol engine development framework which could then be used as a foundation for Concur and other projects. We have called the resulting framework the Protocol Engine Architecture Library, or PEAL. (An early version of this framework is described in [7]). PEAL itself is protocol independent, and particular protocols (such as the X or Concur protocols) are plugged in as required to perform a given task.

The PEAL framework supports the implementation of protocol engines as sets of linearly-interconnected filters, where each filter performs some small portion of the engine's responsibilities (Figure 7). Conversations are implemented as streams of message objects that are passed from one filter to the next. The messages and message streams are monitored and/or modified by the filters through which they pass. Each filter can be viewed as a miniature protocol engine placed between the two communicating filters on either side. At the ends of the linear streams of filters are special filters that perform the conversion from external representations of a conversation (e.g., procedure calls or messages on a byte stream) to the internal message object representation and vice-versa.

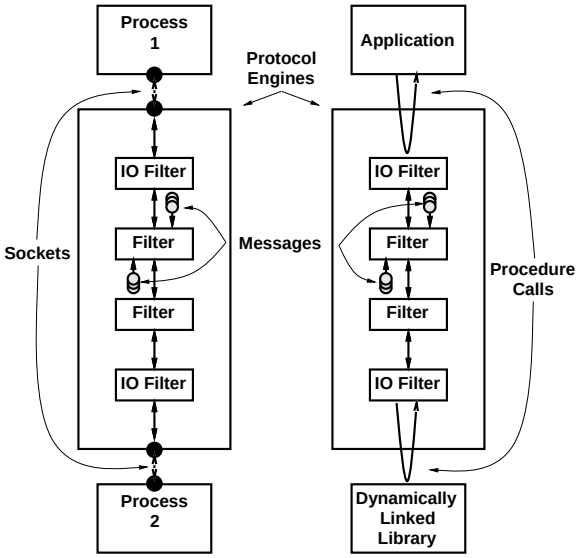


Figure 7: PEAL Protocol Engine Overview

Using the PEAL framework to implement Concur has a number of significant advantages. First, PEAL enables the implementation to be done at a high level of abstraction. For example, polymorphism is utilized with message objects, enabling filters in Concur to perform generic operations on messages without having to be concerned with the particular dynamic type of the message and how the operation would be implemented for each type. Second, PEAL enables solutions to separate problems to be implemented as separate filters, which in turn facilitates parallel code development by different programmers and minimizes code interdependencies that can become maintenance nightmares. Third, PEAL encourages code reuse of various forms (e.g., subclassing, templates, and component libraries), and many coding details are handled by the framework itself. Finally, PEAL has proved to be efficient enough for the implementation of a production window system.

8 Current Status

The Concur Window System prototype currently supports basic window sharing of X applications from zero to any number of viewers. Window migration is supported. Users may drag and drop windows into and out of shared virtual screens using ConMan, the Concur window manager. Access controls are partially implemented. During the next few months we intend

to focus on window view customization, window naming, and window locks supporting floor control mechanisms. We have also begun to do a performance analysis of Concur to prove the feasibility of the architecture and to identify areas for further research in mechanisms for improving perfomance.

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

[3] 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.

[4] Daniel Garfinkel, Bruce C. Welti, and Thomas W. Yip. HP SharedX: A Tool for Real-Time Collaboration. *Hewlett-Packard Journal*, 45(2):23–36, April 1994.

[5] Saul Greenberg and Mark Roseman. GroupKit: A Groupware Toolkit for Building Real-Time Conferencing Applications. In *CSCW '92 Proceedings*, 1992.

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

[7] John Menges. The X Engine Library: A C++ Library for Constructing X Pseudo-Servers. In *Proceedings of the Seventh Annual X Technical Conference*, pages 129–141, 103 Morris Street, Sebastopol, CA 95472, January 1993. O'Reilly & Associates, Inc.

[8] John Menges and Kevin Jeffay. Inverting X: An Architecture for a Shared Distributed Window System. In *Proceedings, Third Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises*, pages 53–64, Los Alamitos, California, April 1994. IEEE Computer Society Press.

[9] Robert Scheifler and James Gettys. *X Window System*. Digital Press, Bedford, MA, 2nd edition, 1990.