On the Partitioning of Function in Distributed Synchronous Collaboration Systems*

John Menges *Kevin Jeffay*
Department of Computer Science
University of North Carolina
Chapel Hill, NC 27599
{menges,jeffay}@cs.unc.edu

Abstract: A large number of experimental (and a few commercial) distributed synchronous collaboration support systems have been developed to date for the UNIX[1]/X environment. These systems typically fall into one of two categories: toolkits for collaborative application development, and shared window systems. Collaboration toolkits usually focus on supporting the development of collaboration-aware applications, and shared window systems are typically intended to make existing single-user (collaboration-unaware) applications available to multiple users concurrently. The two types of systems are usually developed and used independently; that is, integrated systems with the capabilities of both types of systems are rare.

This field of research is now mature enough that we can identify the major components of such systems and the functions they typically provide. In this paper we analyze a typical partitioning of function among the usual set of components, and suggest changes that can be made to this partitioning to improve the characteristics of future collaborative support systems. In the process, we find that our re-distribution of function makes it easier to develop integrated systems supporting both collaboration-aware and collaboration-unaware applications.

1. Introduction

Support for distributed synchronous collaboration typically comes in one of two forms: that of a collaboration toolkit supporting the development of collaboration-aware applications [3, 4, 10, 11], or that of a shared window system that makes it possible for existing single-user applications to be shared [1, 2, 5, 6, 9]. Many experimental support systems of these types have been developed to date, and though they differ widely in their capabilities and designs, at a high level of abstraction the components of these support systems and the functions performed by each are fairly consistent.

Collaboration toolkits and shared window systems are typically developed and used independently; that is, the two types of systems are not normally integrated. As such, as we consider a typical collaboration support system, we will actually be viewing it as the union of two separate support systems, one of each type. Certain types of components are used by both, but they are not shared in any meaningful way.

The components and associated functions of a typical distributed synchronous collaborative support system are as follows. Each component has its own view of its execution environment and its own paradigms of interaction with the objects it maintains and its interfaces with neighboring components.

- **Application**. An application is typically developed using a particular programming paradigm and a programming language supporting that paradigm. For example, an application might be built using an object-oriented, event-driven paradigm, and be

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

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

written in the C++ programming language. It is primarily responsible for maintaining the state of the application and implementing operations on that state.

A typical application is also responsible for using its neighbor, the toolkit (below) to set up a user interface that makes portions of the application state visible to a user (or set of users) and enables users to invoke operations on the application's state. Most programming paradigms require the application to keep the user interface view up-to-date and to define the bindings between user input and application operations.

The application may also use the toolkit to set up interprocess communication in the form of shared objects, procedure calls, or messages. This use of a toolkit is particularly relevant to our discussion if the toolkit implements some form of application replication or general object sharing among applications (as opposed to just user-interface sharing among users). Depending on the toolkit's communications paradigm, it may or may not be the application's responsibility to explicitly update multiple replicas of shared data.

- **Toolkit**. We use the term "toolkit" to refer to an independently-developed library of software that an application can use to perform high-level user-interface (and perhaps communication) functions. A Toolkit implements a particular user-interface model (and possibly a communication model), which it presents to the application via an interface suitable for a particular programming paradigm and language. In today's world, many toolkits are experimental and there is little standardization among them. They build, however, on existing standardized low-level window and communications systems (below). At least a portion of a toolkit is usually linked with an application, so it runs in the same memory space as the application.

  Toolkits implement a mapping between a neighboring window system's window model and the toolkit's own user-interface model. As such they are responsible for setting up the user-interface and keeping it up-to-date in response to application requests. In the other direction, they perform a mapping between input events and application operations, as specified previously by the application. The state of the user interface is maintained by the toolkit.

  In addition to their user-interface functions, collaboration toolkits may implement some high-level communication model (*e.g.*, shared objects) on top of a low-level communications system. They also implement naming, access controls and concurrency controls on user-interfaces, applications, and shared objects. Collaboration toolkits usually also implement their own user interface for session management.

  Since Toolkits and shared window systems are usually not integrated, collaboration toolkits often provide the window sharing, annotation, and gesturing capabilities also found in shared window systems.

- **Window System**. The window system provides a particular, low-level, standard windowing model on top of which higher-level user-interfaces can be built. The X Window System[2] [12], in particular, is implemented by server processes that maintain a window database manipulable by clients (toolkits, in this discussion). The server is also responsible for driving the I/O devices of a workstation — a bitmapped display, keyboard, and mouse — and mapping their interfaces into the windowing model presented to its clients.

---

[2] The X Window System is a trademark of the Massachusetts Institute of Technology.

- **Window Manager**. The window manager is responsible for maintaining the extra-application aspects of the overall desktop user-interface. Since applications are ultimately responsible for the contents of top-level application windows, current window managers treat top-level windows as black boxes and only concern themselves with aspects of the desktop above this level, *e.g.*, the size, shape, position, and iconification status of top-level windows.

- **Communications System**. Like the window manager, the communications system provides low-level, standardized services on which higher-level services can be built. The most common standard in this community is a socket interface to TCP/IP, but other communications systems provide similar services. Remote Procedure Call services are nearly as standardized and widely available as socket services, and they provide communication at a higher level of abstraction.

- **Pseudo-Server**. The pseudo-server is responsible for implementing any window system functions that are lacking in the standard window system. The main function currently provided by pseudo-servers is wysiwis sharing of top-level windows. In window-sharing environments, separate applications are typically used to provide a user-interface for session management.

- **User**. In addition to interacting with applications, the users of a collaborative system specify customizations for both the applications' and the window manager's user interfaces. These customizations include both the appearance (view) of the user interface and the bindings between user inputs and the operations exported by applications.

2. Partitioning Analysis and Proposals

The partitioning of function described in the previous section is typical for existing collaborative support systems. In this section, we will suggest improvements to this partitioning that we believe will result in collaborative support systems that are more flexible and uniform, and have a wider range of capabilities than existing systems. In addition, these modifications to the typical partitioning will facilitate the integration of collaboration toolkits and shared window systems.

- **Window System Enhancements** [7]. Many of the common low-level functions of collaboration toolkits can and should be subsumed by the window system. In particular, we propose that windows should become first class sharable objects similar to files in a distributed file system. The window system would then directly support wysiwis window sharing, access and concurrency controls on windows, and a hierarchical name space for windows. Moving these functions from the toolkit to the window system enhances inter-toolkit consistency and simplifies toolkit development. (New window systems can be built on top of old ones using the pseudo-server approach as necessary. In any case, new window systems must also be responsible for maintaining backward compatibility.)

- **Client Device Drivers** [7]. Current window system servers (*e.g.*, X) directly drive I/O devices such as bitmapped displays, keyboards, and mice. We propose that these functions should be moved to a separate device driver application which is a client of a window server. The window server's responsibility is reduced to that of maintaining the window database and make it available to its clients. The window server should present an abstract terminal to its application clients; that is, it should not make the characteristics of particular terminals visible. This arrangement facilitates the implementation of window sharing, as multiple device driver applications can express an interest in the same windows, and the server or device driver client, rather than the application, can deal with heterogeneity in terminal characteristics.

- **Input Bindings**. Input events are typically bound directly to application operations by the toolkit, as specified by the application (which, in turn, often gets these specifications from the user). This makes customization of bindings on a per-user basis extremely difficult in a shared window system which makes a collaboration-unaware application available to multiple users simultaneously. We propose that there should be two levels of binding. The first should be performed by the window system, and should consist of a binding between user input and a generic representation of the application operations exported by a particular application. A string-oriented scripting language such as Tcl [8] would make a good language for representing application operations generically. The second binding, between these generic operation representations and the application's programming paradigm/language should be performed by the toolkit itself. This scheme enables user input to application bindings to be specified by either the application (for defaults) or the user (for customizations) in the form of requests to the window system. It also enables multiple users to use different bindings.

- **Model/View Separation**. It has often been suggested that applications should only be responsible for maintaining application state, implementing operations on that state, and exporting portions of the state in a form accessible to other programs. Unfortunately, this model is rarely realized, and applications typically specify and maintain user interfaces directly. We believe this is mostly because good, general programming paradigms supporting change notification have not yet emerged. Should this difficulty be overcome, it would have an interesting effect on the partitioning of function described above. Whereas current window managers treat top-level application windows as "black boxes" and do not manage their contents, with a programming paradigm supporting model/view separation the window manager (or, more appropriately, a user-interface manager) could take on the responsibility of managing the entire user interface. Applications and/or users would be able to use the user-interface manager to select and compose toolkit components into a user-interface and connect them to applications as they desire. Model/view separation would also make it possible to separate the toolkit component into two separate components, one that performs bindings to a particular programming paradigm/language, and another that implements a particular style of user-interface. This would enable application designers and users to choose programming paradigms and user-interface styles independently, which is rarely possible in existing systems.

- **Communications System**. Undoubtedly, as experience is gained in toolkit implementations of high-level communications paradigms (such as shared objects), they will become part of the underlying, universal communications systems, relieving toolkit designers of these tasks and making communication among applications more uniform.

3. Summary

In this paper we have identified the components of typical distributed synchronous collaborative support systems, and have outlined the typical partitioning of function among these components. Suggestions were then made for moving certain functions from one component to another, resulting in more flexible and uniform support systems. In the process, shared window system functions become part of the underlying window system, on top of which toolkits for building collaboration-aware applications can be built. This, in turn, facilitates the implementation of integrated collaborative systems providing both collaboration-aware applications and window sharing for collaboration-unaware applications.

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

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

[4] 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.

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

[6] 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.

[7] 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*, Morgantown, WV, April 1994, IEEE Computer Society Press, pp. 53-64.

[8] John K. Ousterhout. *An Introduction to Tcl and Tk.* Addison-Wesley, 1993.

[9] 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.

[10] 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.

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

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