Groupware Toolkits for Synchronous Work

SAUL GREENBERG AND MARK ROSEMAN *Department of Computer Science, University of Calgary*

ABSTRACT

Groupware toolkits let developers build applications for synchronous and distributed computer-based conferencing. This chapter describes four components that we believe toolkits must provide. A *run-time architecture* automatically manages the creation, interconnection, and communications of both centralized and distributed processes that comprise conference sessions. A set of *groupware programming abstractions* allows developers to control the behaviour of distributed processes, to take action on state changes, and to share relevant data. *Groupware widgets* let interface features of value to conference participants be added easily to groupware applications. *Session managers* let people create and manage their meetings and are built by developers to accommodate the group's working style. We illustrate the many ways these components can be designed by drawing on our own experiences with GroupKit, and by reviewing approaches taken by other toolkit developers.

1.1 INTRODUCTION

Building groupware for synchronous, distributed conferencing can be a frustrating experience. If only conventional single-user GUI toolkits are available, implementing even the simplest systems can be lengthy and error-prone. A programmer must spend much time on tedious but highly technical house-keeping tasks, and must recreate interface components to work in a multi-user setting. Aside from the normal load of developing a robust application, the programmer of groupware must also attend to the setup and management of distributed processes, inter-process communication, state management and process synchronization, design of groupware widgets, creation of session managers, concurrency control, security, and so on.

Consequently, a variety of researchers have been exploring groupware toolkits. Their purpose is to provide tools and infrastructures powerful enough to let a programmer develop

A version of this paper will appear in *Trends in CSCW*, Edited by Michel Beaudouin-Lafon

robust, high quality groupware with reasonable effort. Some in-roads have been made, but we are far from a complete solution. Realistically, most of today's groupware toolkits are best seen as breakthrough research systems used to either explore particular architectural features of groupware toolkits, or as platforms to build experimental groupware prototypes. While they have not reached the maturity of single-user GUI toolkits, these pioneering efforts have laid a foundation for the next generation of toolkit design.

This chapter examines the technical foundations of groupware toolkits. The toolkits we consider are those that construct real-time distributed multi-point groupware applications, where two or more people in different locations would be able to visually share and manipulate their on-line work. Typical applications produced by these systems would be electronic whiteboards, games, multi-user text and graphics editors, distributed presentation software, textual chat systems, and so on. The discussion is heavily influenced by our experiences with our own groupware toolkit called GroupKit [62, 66, 29] as well as the issues raised by other researchers doing similar work.

The chapter highlights four critical features that such toolkits should provide to reduce implementation complexity:

- *Run-time architectures* can automatically manage processes, their interconnections, and communications.
- *Groupware programming abstractions* can be used by a programmer to synchronize interaction events and the data model between processes as well as the views presented across displays.
- *Groupware widgets* can let programmers add generic groupware constructs of value to conference participants.
- *Session managers*, crafted by programmers, can let end-users create, join, leave and manage meetings.

An important omission from this list are the audio and video links necessary for the interpersonal communication channel between conference participants. This is a large area in of itself. For simplicity, we will assume that audio and video are handled out of band, where toolkits can include hooks to bring up other audio/video systems. However, we do point the reader to Hiroshi Ishii's chapter in this book, which provides an excellent example of an integrated audio/video/computational space. It should go without saying that future toolkits must incorporated audio and video as first-class building blocks.

1.2 RUN-TIME ARCHITECTURES

Real time distributed groupware systems are almost always composed of multiple processes communicating over a network. Because this can be complex to create, we feel strongly that toolkits should provide not only programming facilities for creating groupware, but also the run-time architecture for managing the run-time system. In this section, we will concentrate only on the tension between centralized *vs.* replicated architectures, and its impact on the design of toolkits. In his chapter in this book, Dewan will continue this theme by revisiting the issues and by explaining further architectural differences possible in collaborative applications.

1.2.1 Centralized vs. Replicated Architectures

Groupware researchers have long argued the merits of centralized vs. replicated architectures [1, 22, 45, 46, 56, 28, 79, 37, 55, 12, 20]. *Centralized* architectures use a single application program, residing on one central server machine, to control all input and output to the distributed participants. Client processes residing at each site are responsible only for passing requests to the central program, and for displaying any output sent to it from the central program. The advantage of a centralized scheme is that synchronization is easy—state information is consistent since it is all located in one place, and events are always handled from the client processes in the same order because it is serialized by the server. *Replicated* architectures, on the other hand, execute a copy of the program at every site. Thus each replica must coordinate explicitly both local and remote actions, and must attend to synchronizing all copies so they do not get out of step.

Because of its simplicity at handling concurrency and of maintaining a single state model, centralized architectures for groupware has had many advocates [1, 22, 45, 79, 37, 20], and one may wonder why a replicated approach would ever be considered. The main issues are latency, bottlenecks, and heterogeneous environments. First, a centralized scheme implies sequential processing, where user input is transmitted from the remote machine to the central application, which must handle it and update the displays (if necessary) before the next input request can be dealt with. If the system latency is low, this is not a problem. But if it is high, the entire system will become sluggish. While sluggishness is annoying when others' actions are delayed, it is devastating when the system is unresponsive to a person's own local actions, especially in highly interactive applications. Second, the central system can become a performance bottleneck. Highly interactive and graphical applications can push even the fastest CPUs to their limits when several screens must be updated. Similarly, the relaying of all activities to and from a single process can create a traffic jam in some environments. Third, centralized architectures will have problems dealing with heterogeneous environments, as it is unlikely that a single process can update properly remote clients running on (say) a Windows 95 and a Macintosh environment, as they a.ll have a different look and feel.

A replicated scheme, on the other hand, implies parallel processing, where the handling of interactions and screen updates can occur in parallel at each replication. If done properly, communication is efficient as replicas need only exchange critical state information to keep their models up to date. While remote activities may still be delayed, a person's local activities can be processed immediately. Process bottlenecks are less likely—each replica is responsible for drawing only the local view, unlike the central model which must update the graphics of all screens. Consequently, heterogeneous environments are easily handled, for the communication protocol can act as a device-independent graphics layer, and views can be drawn using the native look and feel.

The cost of replication is increased complexity. We are now programming and synchronizing a distributed system, and must handle issues such as concurrency control. Different replicated toolkits handle this in a variety of ways. For example, Share-Kit [40] has no direct concurrency control, and it must be programmed infrom scratch if a programmer requires it. Others do provide concurrency capabilities. DistEdit [43] uses atomic broadcasts. ObjectWorld's shareable objects have the ability to detect messages that have arrived out of order, and allow programmers to do non-optimistic locking [76]. GroupKit [62] can force serialization for some actions by funneling selected activities through one of the replicated processes.

Somewhere in-between are semi-replicated hybrid architectures that contain both central-

ized and replicated components. For example, Patterson [58] advocates a centralized *notification server,* whose sole job is to maintain shared state, to respond to state change requests by clients, and to notify others when state has changed. It would be up to the replicas to decide what the view should look like, and to update the display accordingly.

1.2.2 Impact on Toolkit Design

System designers often argue that a good toolkit will hide implementation and architectural concerns, leaving the programmer to concentrate on the semantics of the task. Yet architectures cannot be completely hidden in groupware toolkits, for the type of architecture may have profound impacts on the way programmers code their systems, and on the system performance. For example, centralized systems often have performance limitations that must be well understood, so that they can be mitigated by the application programmer. Similarly, replicated architectures are distributed systems, and programmers must be concerned with issues such as concurrency control, communications, and fault tolerance.

The runtime architecture also affects the programming paradigm style. For example, many toolkits separate the underlying *data abstraction* (i.e. the data model) from the way a graphical *view* of that data is generated on the display [44, 36] (discussed further in Section 1.3). Figure 1.1 illustrates this. The abstract data model here is an array with three numbers, and the view is generated separately from this abstraction. Views of the abstract model may differ. In this case, two participants view the data as a bar chart, and the third participant sees it as a pie chart. Whenever a value in the data model is changed, the views are regenerated to keep themselves consistent with it. In terms of the run-time architecture, the way the abstraction and views are dealt with depend upon how they are distributed across the system. For example, we could have the data abstraction and view generation done wholly by a central process. Alternatively, the abstraction may be centralized, and the mechanisms to create the views replicated. Or perhaps all components are replicated. Whichever variation is used, the abstract data model should be kept consistent across the entire groupware system, and synchronization must be maintained between the model and the individual views generated from it. This means that the infrastructure to support a separate abstraction and view, as well as the nature of the programming API provided by the toolkit, are highly dependent on the nuances of the runtime architecture.

A good toolkit will provide programmers with high level constructs to deal with all the issues mentioned above, but not mask them [11, 12, 20, 52]. To illustrate this point, the rest of this section will show why programmers need to know about concurrency control, synchronization of abstract models and views, communications, and fault tolerance.

1.2.2.1 Concurrency control

Greenberg and Marwood [28] argue that no generic concurrency control scheme can handle all groupware applications, simply because the user is an active part of the process. For example, conservative locking and serialization schemes that block processing until concurrency can be guaranteed can have deleterious effects on highly interactive user actions due to processing delays and latency, while optimistic schemes have problems when on-going events have to be undone. They also argue that some conflicting interactions are best left to the users to solve by social means, implying that some feedback of conflicting actions be shown within the interface.

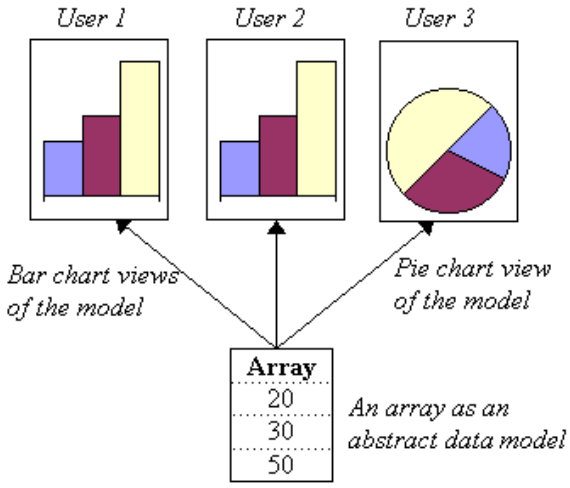

Figure 1.1 An example of an abstract data model, and views generated from the model.

Because of this, toolkits should provide a variety of concurrency control schemes and feedback mechanisms, and programmers must explicitly decide which of them to deploy when designing the application. Note that this argument becomes moot when latency is not perceivable, since the users would not notice any effects of concurrency control. In this case, either a centralized approach and its implicit serialization of events, or a replicated approach using hidden concurrency control would work well.

Many groupware researchers have investigated concurrency control. While it is beyond the scope of this chapter to do a comprehensive survey, readers are referred to the surveys by Greenberg and Marwood [28], and the earier work of Ellis and Gibbs [16]. Further discussions of consistency and concurrency control are found within three other chapters in this book, by Prakash, by Dourish, and by Dewan.

1.2.2.2 Synchronization

As mentioned earlier, specific architectures usually lend themselves to particular ways of separating the underlying abstract data model from the graphical views generated from it. A centralized system keeps both model and view in the same place, so synchronization is easy. In contrast, replicated architectures maintain copies of both the data state and the view at all sites. In-between is Patterson's [58] Notification Server, which keeps the abstract data model in a central server, with replicas deciding how to display the view of that information when state changes are transmitted to them. At the toolkit level, this division of model and view as well as its distribution across processes is usually visible to the programmer—the programming abstractions provided are used by them to update the abstract model or the view, and to synchronize replicas when needed. Similarly, the way the toolkit provides the abstractions to process user events and to synchronize models and views often depends upon the way the model and the views are distributed in the architecture. This topic will be taken up again in more detail in Section 1.3: Programming Abstractions.

1.2.2.3 Communication

Inter-process communication can be a complex task, especially when efficiency is a concern. Centralized models are particularly vulnerable to communications bottlenecks, as the server must not only handle input from the client, but update all displays as well. Replicated architectures can be more efficient, for the events sent across the network can be short messages containing semantic changes to state. At the toolkit level, the programmer would rarely want to deal with all the annoyances of setting up communications connections. However, they should have the means to decide what to communicate between processes for efficiency purposes, and also the means to decide priorities. For example, consider a drawing application containing telepointers, where the telepointers are not supplied as a widget. In terms of what to communicate, the complete telepointer graphics need not be shipped. Instead, a message can be sent specifying the pointer shape, with subsequent messages sending out a pointer id and its x-y coordinates. In terms of priority, when a pointer is moved the programmer should be able to specify that only that last pointer location need be transmitted if there is a communications bottleneck, and that this should have a lower priority than (say) a drawing message.

1.2.2.4 Fault tolerance

Because almost all groupware systems are distributed in one way or another, fault tolerance becomes a concern. At the toolkit level, the programmer should be able to determine the system's response to particular faults. These include degradation or complete loss of communications between processes, excessive delays, and so on. This implies that the toolkit must have a notification mechanism that indicates faults to the program. It also implies that the programmer is aware of the faults that are inherent in the particular architectural design.

1.2.3 Examples

1.2.3.1 WScrawl: a centralized architecture that leverages X Windows

WScrawl [79] is a multi-user sketchpad built using the X Window system. While WScrawl is not a toolkit, the author describes how his program leverages the communications and display capabilites of X Windows, as well as its client/server architecture [67]. X Windows allows a programmer to open several displays, to read input from each workstation, and to write graphics to the screen. Groupware such as WScrawl is created by tracking the display and input stream for each user, all within a single program. Each stream is monitored for input events. For every input event (such as a mouse move that initiates a draw line action), the event is processed, and all displays can be updated accordingly. For example, the pseudocode below handles a trivial conference of two users, each using separate displays named Display1 and Display2, where the conference just draws a point on each display [79]:

```
display[1] = XOpenDisplay ("Display1");
display[2] = XOpenDisplay ("Display2");
for (i=1; i<=2; i++) {
    XDrawPoint (display[i], 20, 100);
    XCloseDisplay(display[i]);
}
```

1.2.3.2 Rendezvous: a centralized architecture

The Rendezvous groupware toolkit [37, 57, 56, 36] is heavily modeled on the idea of maintaining a single abstract data model that is shared by everyone. As mentioned earlier, multiple views of that model can be drawn differently on each person's display. Rendezvous places both the single abstraction and the view models on a single processor. Its developers claim that the single abstraction always contains the correct state of the application. Consequently, all copies or view updates derived from this abstraction will be correct. The problem is that Rendezvous is slow, because all views run off the same processor. Its designers suggest, but have not implemented, a semi-replicated approach that keeps that shared abstraction at a central site, with views being replicated at other sites.

1.2.3.3 The Notification Server: a centralized component of a hybrid architecture

Patterson, one of the authors of Rendezvous, revisited the idea of a semi-replicated hybrid architecture [58]. He is now constructing a centralized "Notification Server" that could be provided as a toolkit component in an otherwise replicated architecture. Its job is to provide a central service for managing common state information, akin to the shared abstraction seen in Rendezvous. While the natural use of the server is to centralize the abstract data model, the choice of what state information to centralize is ultimately up to the designer.

The Notification Server contains two kinds of objects: Places and Things. *Places* identify what common states are accessed by which applications. Clients who enter a particular place are notified about any state changes in that place. *Things* are the actual objects that maintain state, and are essentially property-value pairs extended to contain attributes that specify access control and types of notifications triggered (e.g., on creation, change or deletion). What is important here is that the server has no understanding of application semantics. Virtually any state can be represented, as long as it can be described as a property-value pair. It is left up to the replica how to deal with state changes upon notification.

Patterson argues that this centralized Notification Server simplifies concurrency control because locking is done in one place through a Thing's attributes, and that serialization is a natural consequence of centralization. He also argues that the availability of a consistent, centralized state makes it easier to update newcomers—participants who have just entered a conference that is already in progress. Finally, this dedicated server model implies that attention can be devoted to making it efficient and robust—in Patterson's words, "a lean, mean notification machine."

1.2.3.4 GroupKit: a mostly replicated architecture

The GroupKit groupware toolkit [62, 29] includes a mostly replicated run-time infrastructure. It actively manages the creation, location, interconnection, and teardown of distributed processes; communications setup, such as socket handling and multicasting; and groupware-specific features such as providing the infrastructure for session management and persistent conferences. Its infrastructure consists of a variety of distributed processes arranged across a number of machines. Figure 1.2 illustrates an example of the processes running when two people are communicating to each other through two conferences 'A' and 'B'. The three large boxes represent three different workstations, the ovals are instances of processes running on each machine, and the directed lines joining them indicate communication paths. Three types

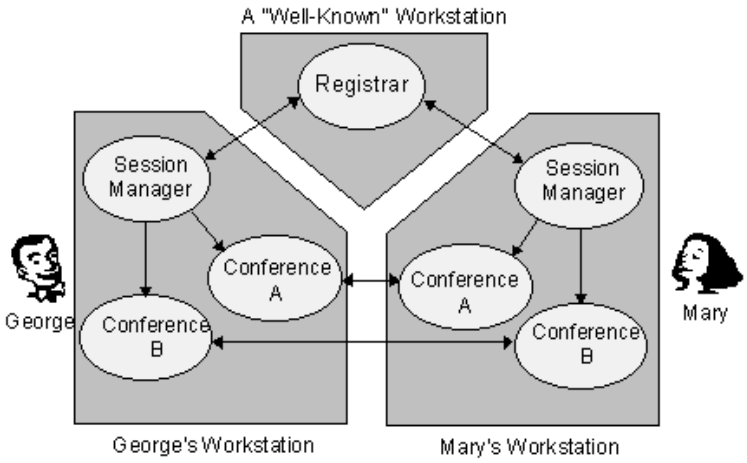

Figure 1.2 An example of GroupKit's run-time architecture and process model.

of GroupKit processes are shown: a single registrar, session managers, and conference applications.

The *registrar* (top box in Figure 1.2) is a centralized process that acts as a connection point for a community of conference users. Its address is "well-known" in that other processes know how to reach it. This is the only centralized process required by GroupKit's run-time infrastructure. The *session manager* is a replicated process, one per participant (side boxes). It provides both a user interface and a policy dictating how conferences are created or deleted, how users are to enter and leave conferences, and how conference status is presented (see Section 1.5: Session Management). When session manager processes are created, they connect to the registrar. The registrar maintains a list of all conferences and the users in each conference. It thus serves as an initial contact point to locate existing conference processes and their addresses. Finally, a *conference application* is a GroupKit program (e.g., shared editor, game) invoked by the user through the session manager. Conference applications typically work together as replicated processes, in that a copy of the program runs on each participant's workstation. They are connected via peer to peer communication channels. Two conferences, each with two distributed replicas, are shown in Figure 1.2.

GroupKit programmers build both session managers and conference applications, and the two are separate from one another. Programmers are aware that they are building distributed applications, and must attend to issues such as concurrency control, fault-tolerance, and synchronization. The programming abstractions let the programmer choose and mix several styles of coding: view synchronization through multicast RPCs, or state synchronization of a replicated abstract data model (see Section 1.3: Programming Abstractions). The toolkit provides a few simple concurrency control schemes for the programmer to choose from, mostly available within the shared data model. Communications are mostly hidden away; while it is possible to massage communication events for efficiency, this is mostly done by working around the system rather than with it. Fault tolerance is done by primitive events that notify a programmer when participants have "left" the conference and when a conference has "died". However, they are not notified nor can they easily handle performance degradation.

This run-time infrastructure is maintained entirely by GroupKit. The conference applica-

tion code does not need to take any explicit action in process creation or communication set-up. Instead, the application may just ask to be notified through an event when particular session activities occur. The conference processes that comprise a conference session can also co-ordinate with each other through the high-level programming abstractions provided by GroupKit, as discussed in Section 1.3.

1.2.3.5 Clock: a flexible architecture

The main goal of the Clock language and ClockWorks programming environment [18] is to support the development of groupware applications at a very high-level, hiding all details of the underlying implementation architecture. This high level has two consequences. First, programmers do not need to be concerned with the details of distribution, networking and concurrency control. Second, implementations of Clock are free to use any implementation architecture, as long as the semantics of the Clock language are preserved. (Unlike other languages for groupware development, Clock has precisely defined semantics, independent of any implementation [17].)

The *abstract architecture* of Clock programs is developed using the visual ClockWorks programming environment. This abstract architecture captures the structure of Clock programs, but does not specify how the program will be implemented in a distributed context. The abstract architecture language is based on separating the abstract model from its views, similar to Rendezvous [36] and the Model-View-Controller (MVC) paradigm used in SmallTalk [44]. Because of its high-level, the architecture language supports rapid development and easy modification of groupware programs [21].

Abstract architectures can be mapped into a variety of implementation architectures. By locating the complete architecture on a server machine and using the X window system to post windows on different client machines, a centralized architecture can be obtained. By locating the shared components of the Clock architecture on a server machine and replicating private components on client machines, a semi-replicated hybrid architecture is obtained. By replicating both shared and private components, a replicated architecture can be obtained. Currently, Clock programs can be implemented as either centralized or semi-replicated.

There are several advantages with the Clock approach to flexible implementation architectures. Since the run-time system is completely responsible for implementing network communication and concurrency control, complex optimizations may be built into the system that would be too hard to develop on a per-application basis [20]. Also, programmers can easily experiment with what kind of architecture is most appropriate for their application without having to extensively modify the program. The primary disadvantage of the Clock approach is that programmers give up control over precisely how different components are going to communicate. For example, the Clock semantics demand that concurrency control be pessimistic, which is not practical over networks with very bad latency.

1.2.4 Discussion

There is no real answer to whether a centralized or replicated scheme works best for groupware. Rather, it is a set of trade-offs that revolves around the way they handle latency, the ease of program startup and connection, programming complexity, synchronization requirements, processor speed, the number and location of participants expected, communication capacity and cost, and so on. For example, a centralized system would likely work just fine for a very

small group of users (e.g., pairs), given a high bandwidth, low latency network and an application that makes only modest demands of the processor. Replicated systems are probably better for larger groups, for slower networks, and for applications that demand local responsiveness.

Because these situations are neither static or universal, no single solution will suffice. Perhaps what is required is a "dynamic and reactive" groupware architecture, where the decision of what parts of the architecture should be replicated or centralized can be adjusted to fit the needs of particular applications and site configurations. We have already seen that Clock components can be configured to run as centralized or semi-replicated objects [20]. O'Grady [52] takes this one step further in his design of GEN, a prototype groupware toolkit based upon distributed objects that allows a high degree of run-time configuration. GEN not allows application designers to chose whether individual objects are centralized or replicated, but also allows designers to create their own strategies for data distribution and concurrency control. For example, GEN was altered to allow for object migration, where centralized objects are automatically moved to the site that uses them the most frequently. In parallel work, Dourish's chapter in this book presents his design of Prospero, a groupware toolkit that also allows decisions on data distribution and other aspects to be made on the fly [11, 12]. Essentially, toolkits such as GEN and Prosepero are designed to be highly flexible. Not only can developers choose between a variety of strategies, but they can also extend the toolkit to cover situations not envisaged by the original toolkit creators.

1.3 PROGRAMMING ABSTRACTIONS

Groupware toolkits must provide programmers with abstractions for coordinating multiple threads and distributed processes, for updating a common abstract data model, and for controlling the view derived from that model. The actual abstractions supplied usually depend upon the run-time architecture (as described in Section 1.2), as well as the schemes used to share state information.

Patterson [55] argues that the degree to which abstract data models are separated from the views generated from them lead to several different shared state architectures, with consequences to the programming abstractions provided.

1. In an unshared system, neither data nor view model are shared. It is up to the programmer to maintain the underlying data models (if any exists), the graphical views, and the links between the views and the model (if any).
2. In a shared model, the data model is shared by the entire system. Programming abstractions allow one to access and change the shared model, and to specify how the (possibly different) unshared views are to be created from the shared model.
3. In a shared view, both views and models are shared. Programming abstractions are available to change the view or the model, with changes automatically propagating from one to the other.

This section describes several programming abstractions that are now common: multicast remote procedure calls, events and notifiers, shared data, and shared data and views. Each lends themselves to the three architectures mentioned above.

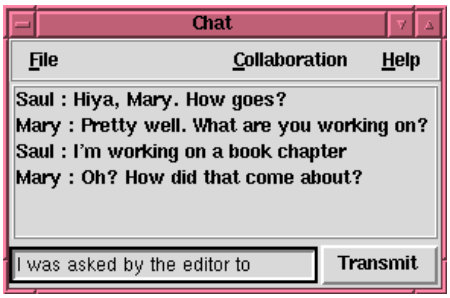

Figure 1.3 Using multicast RPCs for a simple chat application.

1.3.1 Multicast Remote Procedure Calls

With replicated processes, replicas can communicate, share information, and trigger common program execution through multicast *remote procedure calls* (RPCs). As with conventional RPCs, a programmer specifies the procedure and arguments that should be executed in remote processes. It is multicast because several processes can be designated in a single call.

Through this simple yet powerful abstraction, any unshared system can be synchronized. For example, traditional callbacks to a user's input can be replaced by a multicast RPC that causes the resulting action to be performed in all processes. The following pseudo-code illustrates this by showing how the simple text chat system shown in Figure 1.3 can be implemented. The main window contains the dialog transcript, and is common across all displays. Each participant types their text into their private text input field at the Figure's bottom. Whenever a person presses the "transmit" button, their name and the text they composed are sent to all others for insertion into the transcript.

```
Set this_user to the name of the local user
When transmit button is pressed
    Set message to the contents of the input field
    Multicast to everyone:
        Insert into your chat box "this_user: message"
    Clear the input field
```

In the above example, there is no data model. Only the view is synchronized by explicitly manipulating the widgets in the view. Data can be synchronized as well by multicast RPCs, although it is the programmer's responsibility to do all the house-keeping and to generate the view from the data model.

1.3.1.1 Examples

Several systems use multicast RPCs as its sole programming abstraction. Share-Kit [40] uses C and the Unix RPC mechanism to build its multicast layer; its programmers must register a procedure and its argument formats as an RPC and use special keywords to invoke them. The Conference Toolkit [6] uses a routing table to let developers specify the routing of data between application instances, that is, how commands from one replica are directed to other replicas. The Notification Server [58] provides a "back door" that allows programmers to channel multicast messages between clients; these messages could be constructed in a way that simulates multicast RPCs.

GroupKit [62, 29] simplifies multicast RPCs by allowing RPCs and arguments to be spec-

ified in the same way as normal procedure calls, and by hiding routing and communications details. To do this, GroupKit's run-time system tracks the addresses and existence of other application processes, and decides how to multicast the RPCs to some or all conference processes. This means that the programmer does not have to track details such as the file descriptors, socket management, and so on. GroupKit provides three forms of RPCs, and each differs in who the messages are sent to. The first, called `gk_toAll`, multicasts the procedure to all conference processes in the session, including the local user. This results in the same procedure being executed everywhere. The second, called `gk_toOthers`, multicasts a command execution to all other remote conference processes in the session except the local process that generated the call, which is useful when local actions differ somewhat from remote ones. The third form directs the command to a particular conference process. This is valuable for handling special cases, such as updating a new arrival to an on-going conference about the current state of the application. Additionally, GroupKit's RPCs are non-blocking. Once the request for an RPC invocation is made, the local program continues its execution without waiting for a reply from remote processes. This ensures that conference processes are not delayed or blocked in the event of network latency or crashes on remote machines.

As an example, we implemented the simple text chat system shown in Figure 1.3 in GroupKit (which extends the Tcl/Tk scripting language by John Ousterhaut [53]) using the `gk_toAll` RPC. The complete code is shown below, excluding a few minor bits that format the widgets on the display. What is important to realize is that only a few lines of code are required to make this program group-aware: `gk_initConf` initialize the runtime architecture for the conference; `gk_defaultMenu` includes GroupKit's menu widget, `[users local.username]` retrieves the name of the local user, and `gk_toAll` multicasts the RPC to insert the user's name and text into the chat box. All other lines are just the standard Tcl/Tk code necessary to create the interface.

```
gk_initConf $argv             # Initialize the conference

#== Create widgets
gk_defaultMenu .menubar       # Add the default groupkit pulldown menu bar
listbox .chat                 # The shared chat box is actually a listbox
entry .input                  # User type there text into this entry box
button .b -text Transmit \    # Create a button labelled 'Transmit'
    -command "broadcastLine"  #  and attach a callback to it

#== Not shown: code to format widgets on the display

#== This callback multicasts an RPC to all replicas (using gk_toAll)
#== along with this user's name and text
proc broadcastLine {} {
   gk_toAll doAddLine \           # Multicast the doAddLine RPC + arguments
        [users local.username] \  #   1st argument: the user's name
        [.input get]              #   2nd argument: the text
   .input delete 0 end            # Now clear the input field
}

#==This is executed as an RPC at all sites.
#==It inserts the name and text into the chat box
proc doAddLine {name text} {
    .chat insert end [concat $name ": " $text]
}
```

While simple, GroupKit's multicast RPC model provides a powerful yet flexible approach to distributed programming. The programmer does not have to know the addresses of other

conference processes or track process creation and destruction as people enter and leave the session. The calls work the same way whether one user or twenty users are in the conference session.

1.3.2 Events and Notifiers

A second programming abstraction allows a programmer to synchronize changes to either views or models by specifying interesting events and how others are notified when these events occur. Because events can be tied to anything, they can serve both unshared and shared systems.

An *event* provides a way for conference applications to track when various things happen. Events can be generated automatically by the run-time architecture, such as when participants join or leave the conference session, or from (say) communications failures. They can also be generated directly from the programmer in application-specific circumstances. Either way, the programmer can take action on a specific event by attaching a *notifier* to it, which typically executes a callback whenever the event occurs (notifiers are also known as handlers in some systems).

1.3.2.1 Examples

Patterson's Notification Server [58], described previously in Section 1.2, illustrated an architecture that supports notification. Here, events are simply changes in the state of the underlying data ("Things"). Notification is controlled by the attribute field of the Thing, and occurs automatically whenever a state changes.

GroupKit contains an event/notifier mechanism as well as events automatically maintained and generated by the run-time infrastructure [62]. Events are typically used to handle arriving and departing participants, updating latecomers, synchronizing distributed processes, and noticing changes to shared data. Events consist of an event type and a set of attribute/value pairs that provide information about the event. While in some ways similar to Patterson's Notification Server, state information is replicated rather than centralized. Programmers trap particular events by attaching a notifier, with desired actions specified through callbacks that are automatically executed when the event occurs.

GroupKit's run-time infrastructure automatically sends three different event types to conference processes. The first two event types are generated when users join and leave the session, as a conference process may want to take special action when this happens. For example, the code fragment below tells everyone that a new participant has arrived by printing a message on all screens. The first line attaches a notifier to a "newUserArrived" event, which is automatically generated by GroupKit when a new user joins the conference. This will trigger execution of the subsequent lines.

```
gk_bind newUserArrived {                        # Attach code to this event
  set new_user_name [users remote.%U.username]  # Get the new person's name
  puts "$new_user_name just arrived!"}          # Print message to the screen
```

The third event automatically generated by GroupKit is used to handle latecomers to conferences that are already in progress. When a latecomer arrives, its conference process is brought up to date by one of the other conference processes in the session, usually by sending it the existing state of the conference. Details of how to update the newcomer is left up to the programmer by having them create an appropriate callback.

Finally, application developers can generate their own custom events. This can be useful in more complex applications, where a change being handled in one part of the program can generate an event to notify other parts of the program (or other processes) of the change. For example, a programmer can create a shared data model and use events to generate views from it. Changes to the model's state can be attached to events, with notifiers created to update the view accordingly. Different views are handled by attaching different callbacks to the notifiers.

A variety of other toolkits use some type of event/notification scheme e.g., Rendezvous [36], Chiron-1 [74], and Weasel [19]. However, these are typically tied to directly linking the shared views with a data model, discussed next.

1.3.3 Shared Models and Views

While multicast RPCs and events can be used to co-ordinate conference replicas, they do demand more housekeeping as the application becomes complex. Consequently, several groupware toolkits provide programming abstractions to maintain and update a shared data model, and some means for attaching a view to the model.

The idea of separating a data model from its view originated in Smalltalk's model-view-controller [44], later extended to groupware [56, 36, 19, 76]. In most implementations, the system maintains a consistent shared data model (i.e., by handling concurrency and synchronization), and either notifies processes of changes to the data or automatically updates views whenever changes occur.

1.3.3.1 Examples

GroupKit provides a shared data model called an *environment*, a dictionary-style data structure containing *keys* and associated *values* [62]. While instances of environments run on different processes, the run-time system makes sure that changes to one instance are propagated to other instances. What makes GroupKit's environments powerful are that changes to an environment's state can be tracked as events that trigger notifiers (as discussed previously). The programmer can bind callbacks to an environment, and receive notification when a new piece of information is added to it, when information is changed, or when information is removed. Corresponding actions are then triggered at all sites.

This scheme can generate different views from the same data abstraction. Events can be monitored by the interface code, and the view adjusted to reflect the state of the data model contained in the environment. For example, the code fragment below creates a shared environment called "data", which contains a field called "number". A groupware button is displayed that shows the current value of the number, incremented whenever any user presses the button.

```
gk_newenv -bind -share data             # "Data" is a shared environment
button .button -command \               # Create a button. Whenever it is
  [data number [expr [data number]+1]   #   pressed, increment "number", a
                                        #   key in the "data" environment
data bind changeEnvInfo  {              # Update the view of the number in the
  .button configure -text [data number]}#  button whenever its value changes
data number 0                           # Initialize "number" to 0
```

A programmer uses GroupKit's environments to implement synchronized views and models. In contrast, the Rendezvous toolkit treats views, models and the links between them as first class citizens [37]. The system encourages developers to create groupware applications using its powerful *abstraction-link-view* (ALV) model [36], whose constructs are:

- a shared underlying data abstraction,
- a view of the abstracted entity that may differ for each user,
- a constraint (called a link) that automatically adjusts the view when the data abstraction is changed.

Rendezvous differs architecturally from GroupKit, in that the data model and the propagation of constraints are centralized. As well, constraints are more powerful than the event/notifier scheme, because complex relationships are automatically maintained by the system through a one-way constraint solver. The Clock system [21] also uses constraints to link views with the underlying model.

A variety of other systems also have a strong notion of maintaining the relation between a model and a view. The Chiron-1 user interface system has abstract data types (abstractions), dispatchers (links) and views; however, a simpler event-based architecture rather than constraints are used to propagate changes [74]. While Chiron-1 was not explicitly designed to be a groupware toolkit, a multi-user Tetris game was developed to show the flexibility of its architecture. In Weasel [19], programmers use a special declarative language called RVL to specify the relations between abstractions and views, how views are customized, and the co-ordination required.

Populated virtual environments also use an abstract model/view paradigm. The model is the 3-d abstraction, while the rendered views of the model are perspectives generated from a particular (x,y,z) viewpoint into the model. The model is typically spatial. People enter the spatial environment, where they are represented as 'avatars' to others (icons or even video images of themselves). They can move through the space and manipulate artifacts within it. They are usually aware of the presence and (perhaps pseudo-) identity of others, can see where others are attending, and can begin text or voice based communications with them. Examples are Dive [7] and Moondo [38].

1.3.4 Discussion

Programming abstractions such as the ones described above ease considerably a programmer's task of building groupware. For example, since multicast RPCs are a natural extension of the way normal callbacks are used, novice GroupKit programmers were able to create simple groupware applications with minimal training. The event/notifier and shared data abstractions are more elegant, but demand that the programmer learn a new coding style, for it usually takes more planning and initial coding to separate the data model from its view.

However, groupware programming abstractions do not eliminate all coding complexity. The programmer must consider the interaction between the processes that are being coordinated by multicast RPCs, by events, and by shared data; unconsidered side effects can cause the unexpected to happen. There is also a craft to using the programming constructs effectively. For example, multicast RPCs usually demand that the programmer consider what local actions should be taken and what variables should be set before the procedure and arguments are multicast. The shared data abstractions have their own problems. When data model and views are separated, the programmer has to handle exceptions that often occur when most, but not all of the view is identical. When views intentionally differ (such as when one person sees an array as a bar chart and the other as a pie chart, as in Figure 1.1), the programmer has to make difficult interface design decisions that will allow people to interact over disjoint views.

In all cases, debugging can be hard when problems do occur, because the interaction between conference processes can be non-deterministic and difficult to envisage.

1.4 GROUPWARE WIDGETS

Perhaps the greatest benefit of today's graphical user interface toolkits is their provision of tried and tested interface widgets. Programmers can typically configure and position them in a few lines of code, perhaps with the help of an interface builder. When done properly, pre-packaged widget sets provide a consistent look and feel to the interface. Because widgets are often designed by interface experts, the everyday programmer can insert them into the application with some assurance that they are usable.

Because many groupware applications will be graphical, groupware programmers have the same need for widgets. The toolkit should therefore make it easy for programmers to add groupware features to applications that conference participants will find valuable. However, groupware widgets differ from normal widgets. They have different semantics; actions performed on them must be reflected across displays; and novel widgets have to be designed that address needs specific to groupware. In this section, we consider two classes of groupware widgets: groupware versions of single user widgets, and group-specific widgets that support activities found only in group work.

1.4.1 Groupware Versions of Single User Widgets

Some researchers have created multi-user analogues of conventional single-user widgets, such as buttons, menus and simple text editors, and investigated how to make the sharing of widgets between conference participants flexible enough to fit different applications and group situations.

To highlight several issues, let us consider the problems we face when redesigning a button widget to fit groupware. Buttons are simple devices in conventional interfaces. When a user presses the button, its look changes to reflect that it is being selected. Upon release, the button shape returns to normal and an action is executed. If the cursor is moved off the button during a mouse press, the button reverts back to its original appearance and the release will have no effect.

When the button is redesigned as groupware, several issues arise.

1. When should feedback of one user's actions be shown to their partners as *feedthrough*? Should feedthrough be shown for every interface action (e.g., highlighting that matches button presses and releases), or only for the final action (that the button press resulted in an action)? Should feedthrough appear graphically identical to the local user's feedback, or should it be stylized to communicate only the essence of the other's actions?
2. How does the button handle multiple and simultaneous access? Does it contain an idea of ownership, so only one person is allowed to press it? If so, how is access control handled? Or does the button implement turn-taking so that only one person can press it at a time, and if so, how does it show another that they cannot press the button? If simultaneous access is allowed, what are the semantics of simultaneous presses, and how is feedthrough displayed?
3. How are resulting actions handled? For example, are attached callbacks automatically in-

voked in all replicas on one person's button press, or must the programmer distribute its effects explicitly?
4. What happens when people are viewing different parts of the display? If one person cannot see the button because they have scrolled to another area, is feedthrough shown in a different manner, and if so, how?
5. If different representations are used (e.g., two differing native look and feels because groupware is running across two different platforms), how can the interface syntax of one button be translated to the perhaps different syntax of the other button?

These issues become much more problematic when we move to multi-user equivalents of complex widgets that have a high interaction component, such as list boxes, text entry fields, graphical canvases and so on. None of these problems have trivial solutions, and designers of groupware toolkits have to make hard decisions on what to do in each case. Part of the design space includes how much flexibility they can provide the programmer to allow them to make their own application-specific decisions.

A few researchers have begun to address these issues by creating generic programming attributes for groupware widgets. Several have concentrated on a widget's coupling level and access control. Others have tried to redesign conventional widgets to make them more appropriate to groupware settings.

1.4.1.1 Coupling

Dewan [9, 10] defines coupling as the means by which interface components share interaction state across different users. In tight coupling, state is shared by all aspects of the interface component, and a person's actions in one display results in immediate update on another display. In loose coupling, one person's actions propagate over to another display only when a critical event is performed; the final state is the same, but intermediate states are not seen. For example, a tightly-coupled button would appear identical on all displays as it was being pressed, moved across, and released. A loosely-coupled button would only show the release action, with intermediate feedthrough eliminated.

Dewan and Choudhary [10] argue that flexible coupling is important for a variety of reasons. First, groupware programs range from fully synchronous, to nearly synchronous, to asynchronous; coupling is just another way of setting synchronicity. For example, we can argue that the only difference between a real time text program that shows characters as they are being typed (text chat), *vs.* complete messages (email) is their coupling level! Second, tightly coupled actions showing intermediary steps may be annoying to users in situations where they are pursuing their own individual work. Alternatively, tightly coupled systems are critical during highly-interactive exchanges between people [73]. Third, loosely coupled systems exchange state less frequently, which means there are less performance demands on the system. Finally, coupling can control the degree that people work in private spaces, and how and when they wish to make that space public.

Dewan and Choudhary [10] implemented coupling in their Suite groupware toolkit by allowing programmers and users to set *coupling attributes* that are associated with individual interaction entities (although these can be arranged in a multiple inheritance structure). Attributes indicate the level of coupling, as well as how they should be applied selectively to members of a group. Suite also divides interaction entities into disjoint coupling sets. For example, the data state, the view state, and a format state can be coupled independently (the later

allows the view of the data to be formatted in different ways across displays). Furthermore, action coupling can be set to determine how the commands (or callbacks) attached to user actions are executed at other sites.

Reconsider the button example. The coupling levels can define: the way button presses are tied to underlying data models by coupling data state; the level of feedthrough desired in the view by coupling views; and how callbacks are invoked by coupling actions. Ideally, the groupware programmer would consider coupling levels to be just another set of attributes that can be configured when creating the button. The same idea can be applied to more complex widgets, and Suite has several examples of how coupling can be applied to complex editing and form-filling systems.

Coupling is available in other toolkits as well. The Rendezvous toolkit [37] allows flexible coupling because of the way views are separated from data. Because the links in Rendezvous' ALV model specify how views and models are synchronized [36], different levels of coupling can be specified by the programmer. The difference is that the programmer has to code the way coupling is achieved, rather than simply set the attributes of a widget.

1.4.1.2 Access control

Access control determines who can access a widget and when. Access control may be required for several reasons. First, people may wish to have their own "private" widgets, where only they can manipulate (or even view) them. An example is a text field in an groupware outliner, where the person editing the field wishes to maintain ownership of it, perhaps just for the duration of the edit or for the length of the session. Second, it may not make sense for users to simultaneously manipulate some widgets. Perhaps only one person at a time should be able to press a button, manipulate a scroll bar (to prevent "scroll wars"), or insert text into a field. As with coupling, the demands for access control may be highly dependent on the particular interface being constructed, and groupware programmers need to be able to control this.

Few groupware toolkits let programmers manipulate access control in a light-weight, fine-grained fashion. If anything, they group it into concurrency control, with access being mediated by locks and other tedious mechanisms. The notable exception is again Suite. In it, Shen and Dewan [68] associate the fine-grained data displayed by a groupware application with a set of "collaboration rights", where the rights are specified by either programmer or user through a multi-dimensional, inheritance-based structure. Collaboration rights include read and write privileges, viewing privileges, and coupling. Through the inheritance structure, access control can be specified at both a group and individual level. Sets of objects can be clustered together, with specific access definitions overriding general ones.

Smith and Rodden's [70] "shared interface object layer" SOL, an architectural layer rather than a toolkit, considers how shareable versions of single-user widgets such as buttons and text entry fields can be created. They provide a set of generic access control mechanisms that determine what people could do with these shareable objects. Settable options include who can see the widget, who can use it, who can move it, and so on. The same group has created a more generalized shared object service called Cola [77].

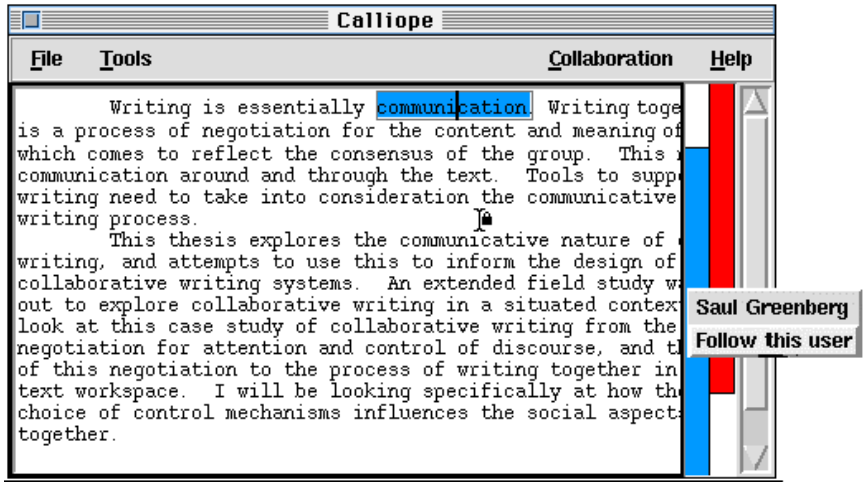

Figure 1.4 The Calliope multi-user editor, with permission from Alex Mitchell.

1.4.1.3 Widget redesigns

Most single user widgets should be completely redesigned to fit their groupware settings, because they would otherwise be too limiting. While there is no recipe for doing this, we can illustrate by example several groupware redesigns of single user widgets.

Our first example is the *multi-user scrollbar*, first seen in the SASSE text editor [2]. It differed from conventional scrollbars in that two thumbs (the selectable box) are displayed. Participants are allowed to scroll independently, and the thumbs' positions would reflect each person's relative position in the document. While SASSE's scrollbar was hard-wired into the editor, GroupKit developers turned it into a real multi-user widget that can be attached to any scrollable object in one or two lines of code [62, 29]. As shown on the right side of Figure 1.4, the right half of the scrollbar is a normal single-user scrollbar, allowing the user to move within the document. To its left are vertical bars showing the relative location of each conference user, identified by a unique colour. The bar's position and size is continuously updated as participants scroll through the document or change their window size. Additionally, the name of the bar's owner is displayed as a popup by mousing over it, and a "Follow this user" option allows participants to toggle the coupling status from independent scrolling to linked scrolling.

Our second example is a multi-user text widget. Single user text widgets are simple text editors, while a true multi-user text editor should have features that allow simultaneous editing. Mitchell [51] used GroupKit [62] to create Calliope, a multi-user text editor. While not packaged as a widget, Calliope does indicate how such a widget could behave. As seen in Figure 1.4, Calliope provides a window displaying a shared text editor, and people can scroll independently through the text through GroupKit's multi-user scrollbars. Access control is user-selectable via a "sharing" menu option, and can range from the selection, word, line, paragraph or document level. As a region is selected, the lock request is automatically made. When another person attempts to select a locked region, the cursor changes to show conflict (the lock icon in Figure 1.4). Calliope also has extra tools, such as the ability to attach external notes to text for commentary that can be seen by others, to create private text which is added to the shared view only when desired, and access to a shared whiteboard for brain-

Figure 1.5 GroupKit's Participants widget.

storming activities. Text can also be queried to find who wrote it and when it was written, and colour-coded to show authorship.

1.4.2 Group-specific Widgets

While group-aware versions of single-user widgets should be a part of any groupware toolkit, they are not enough. Toolkits should strive to provide novel widgets that support particular aspects of group work. In this section, we show several examples of group-specific widgets that are implemented or prototyped in GroupKit [62, 29]. These include widgets for participant status, telepointers, and awareness.

1.4.2.1 Participant status

As people enter and leave a conference, other participants should be able to see their comings and goings, much in the same way that we can see people arrive into a room. Because these people may be strangers, it can be useful to find out some information about them. GroupKit provides a rudimentary *participants widget*, illustrated in Figure 1.5, that can be included in any application. It lists all participants in the current conference session (left side), and the list is automatically updated as people enter and leave. When a participant is selected, a "business card" containing further information about them is displayed. This could include contact information (as shown), a picture of the person, and any other material that person wished to pass on about themselves.

An experimental variation of this widget displays participants in several ways, dependent on the information available about them: charicatures, still photos, and (if available) video snapshots whose images are updated every ten to twenty seconds. The video snapshots implement our version of the Portholes system [14]. These widgets also include the ability to monitor the activity of participants, such as whether they are actively using their computer. This is useful for facilitating contact between partners [8, 30].

1.4.2.2 Telepointers

Studies of small face to face groups working together over a shared work surface reveal that gesturing comprises about 35% of the group's activities [72]. Gestures are a rich communication mechanism. Through them, participants indicate relations between the artifacts on the display, draw attention to particular artifacts, show intentions about what they are about to do, suggest emotional reactions, and so on. Many groupware systems now use telepointers (also known as multiple cursors) to provide a simple but reasonably effective mechanism for communicating gestures [35]. Unfortunately, modern window systems are tied to the notion of a single cursor, and application developers must go to great lengths (and suffer performance penalties) to implement multiple cursors. By supplying telepointers as widgets that can be attached to a view with a few lines of code, a programmer's burden is decreased significantly, and they are more likely to include this important feature within their application. For example, GroupKit programmers can add telepointers to an application with two lines of code:

```
gk_initializeTelepointers
gk_specializeWidgetTreeTelepointer .canvas
```

GroupKit's telepointers can partially handle displays where people may not see exactly the same thing because widgets laid out in different locations. Instead of tying a telepointer to a window, a programmer can attach it to particular widgets and their children (this is the purpose of line 2, which adds telepointers only to the "canvas" widget). The telepointer is always drawn relative to the widget, rather than the application window. Similarly, we have applied telepointers to groupware text widgets that may format their contents differently on different displays. The telepointer in this case is tied to the position of the underlying text, rather than the Cartesian coordinates of the window. To illustrate the value of this approach, we applied these techniques to GroupWeb, a groupware web browser [27]. Because people have different sized windows, the HTML text and images can be laid out quite differently across participant's displays. However, their telepointers are always on top of the correct character or image.

An experimental version of GroupKit's telepointers allows them to be overloaded with semantic information to provide participants a stronger sense of awareness of what is going on, with little consumption of screen real estate. Because telepointers tend to focus participants' attention, any information attached to them is probably noticed quickly. For example, we allow programmers to overload telepointers to indicate identity information (such as people's names), state information (such as what mode each participant is in), and action information (such as what action a person is taking). Figure 1.6 illustrates an example of how a telepointer can be overloaded with both action and identity information. The left window shows participant Carl's display, where he is navigating through a pop-up menu. We see a second cursor on the bottom of the display, which identifies its owner 'Saul'. The right window shows Saul's display. Showing the complete menu that Carl has popped up on Saul's display could be annoying, especially if Saul were working in the area immediately underneath it. Instead, Carl's telepointer image and labels are altered to indicate a menu selection is being made (the mode), and what item is being selected (the action). In this case, the same semantic information of a menu action is shown on other displays concisely and with little loss of meaning.

1.4.2.3 Workspace awareness

In real life working situations, we are kept aware of what others are doing, sometimes by speech, and sometimes by seeing what others are working on through our peripheral vision

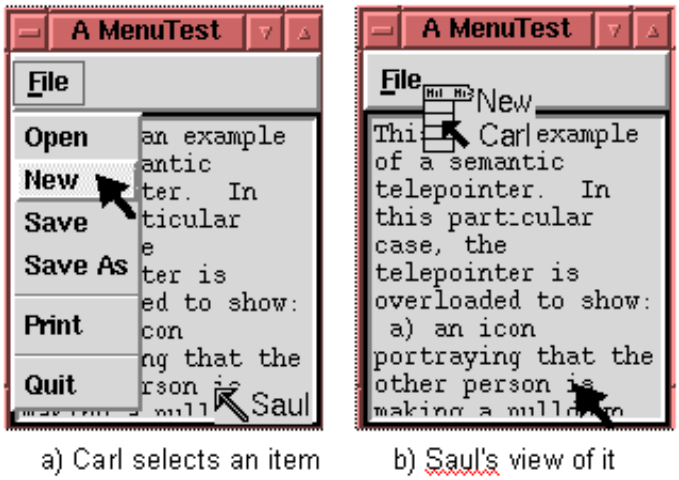

Figure 1.6 Overloaded telepointers, showing both action and identity information.

and through glances. This helps us co-ordinate our work. These cues may not be available in the groupware channel, especially when people are allowed to have different viewports into a large workspace. Consequently, *workspace awareness* widgets must be provided that inform a participant about where other people are working in the shared work-surface and what they are doing [13, 30, 34, 31]. We should mention that workspace awareness does not have the same meaning as collaboration awareness (mentioned in other chapters by Dewan and by Prakash): workspace awareness concentrates on how a person's up to the moment awareness of what others are doing can be supported by representations and extensions of the actual shared workspace, which is a more restrictive definition.

An example of awareness widgets are radar overviews [71, 2]. These displays present a miniature overview of the document overlaid by coloured areas that show the actual viewport of each participant in the session. GroupKit contains several widget prototypes based on this idea [30, 34, 31]. The radar overview shown in Figure 1.7 is one example. It includes an overview of a large shared workspace containing a concept map (a graph of ideas). Viewport outlines, one for each participant, contain portraits identifying their owners, and indicates what each can see. In addition, telepointers are displayed. The overview is tightly coupled to the main view of the document (not shown), and any changes are immediately reflected. A usability study has shown radar overviews to be an effective way for people to maintain awareness of others in a spatial layout task [33]. They see changes as they occur, they know where others are working, and telepointers in the overview are used for deictic references.

We have developed a variety of other prototype widgets supporting workspace awareness. Detail views are miniatures showing exactly what another can see [34, 31]. The Headup Lens combines an overview with a person's main viewing area as transparent layers, one on top of the other [25]. The Fisheye Lens uses a fisheye view with multiple focal points to show where others are in the global context, and to magnify their area of work on all displays [24]. These and other awareness widgets are illustrated in two videos [23, 32].

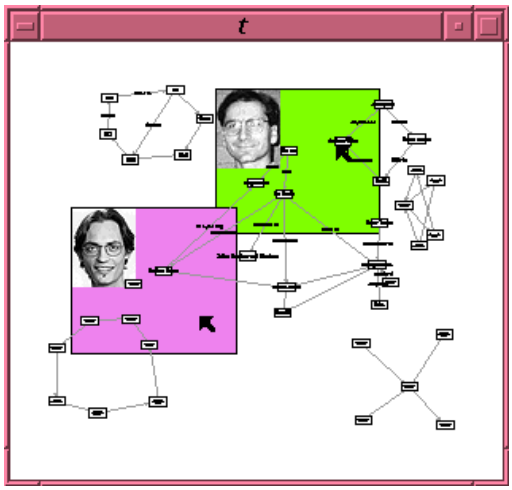

Figure 1.7 A miniature overview of a concept mapping system built in GroupKit, showing other's viewports, portraits, and telepointers. System created by Carl Gutwin, with permission.

1.4.3 Discussion

The design of groupware widgets is still a young area. While many interface components exist in groupware applications that have potential as widgets, much work remains to be done in generalizing and packaging them as self contained widgets that are easily added to any application. We need strong programming abstractions, such as the notions of coupling and access control, to provide a programmer with the flexibility to specify a widget's behaviour in different groupware settings. We need to redesign today's single-user widgets into reasonable yet powerful groupware counterparts. Finally, we need to create the next generation of groupware widgets, which includes refining their design and testing their worth through usability testing.

On the technical side, there is the issue of how widgets can be created by toolkit developers. Current tools are poor or non-existent. Rendezvous and Clock creators, for example, had to build all their widgets from scratch from graphical primitives [37, 18]. GroupKit creators constructed a rudimentary "class builder" and were thus able to use and extend the existing GUI widgets supplied by the Tcl/Tk toolkit [62]. However, the class builder is awkward to use, and suffers run-time efficiency problems which can affect the performance of highly interactive widgets.

Finally, programmers of groupware could still benefit from interface builders as found in conventional GUI toolkits, which greatly eases the task of widget placement and attribute setting. Unfortunately, most groupware toolkits now available do not provide interface builders, with the exception of Visual Obliq [4]. Similar to most modern conventional toolkits, groupware applications in Visual Obliq are created by designing the interface with an interface builder and then embedding callback code in an interpreted language. The resulting application can be run from within the interface builder for rapid turnaround time.

1.5 SESSION MANAGEMENT

Groupware developers often concentrate on building applications, such as multi-user sketchpads, games, and text editors. While it is important for developers to provide good groupware once people are connected and working together, it is just as important to provide a community with "session managers" for actually establishing their groupware connections. We firmly believe that toolkits must allow developers to construct or select from a large library of session management interfaces in a flexible enough fashion to accommodate the diverse requirements of different communities. Unfortunately, most of today's toolkits force a single, often rudimentary, session management interface onto its applications.

A session manager typically controls and presents an interface to the following tasks [64]:

- creating new conferences,
- naming conferences,
- deleting conferences,
- locating existing conferences,
- finding out who is in a conference,
- joining people to conferences,
- access control to conferences,
- allowing latecomers,
- allowing people to leave them, and
- deciding whether conferences persist when all users exit.

For example, the interface of the session manager could present these as explicit steps that a user takes to begin and maintain the collaboration. These could also be implicit actions, where (say) the act of jointly editing an artifact automatically initiates the collaboration [15].

Being able to provide different interfaces for session management is an important aspect of supporting the working patterns of a group. We believe that one of the obstacles to groupware use is the difficulty of starting up a groupware session [8]. The obstacle may be in terms of usability (e.g., the system is difficult to initiate) or social (e.g., the policy the system imposes is not acceptable to the group). Session management must be more than an afterthought added to the applications, and should be tuned to the needs and collaboration patterns of the target user group.

1.5.1 Policies and Metaphors for Session Management

Session managers can implement and provide a broad variety of policies to users, as illustrated by the examples in this section.

1.5.1.1 Rudimentary Policy

When session managers are not attended to, users are forced to handle session manager aspects themselves. That is, it is entirely up to the user to decide who to connect to, often by specifying low-level addressing such as Internet host names and TCP/IP port numbers. An example of this is the session manager for early versions of the NCSA Collage groupware system, which presents a form asking the user to supply one's login name, the IP address of the collage server, and the server port number.

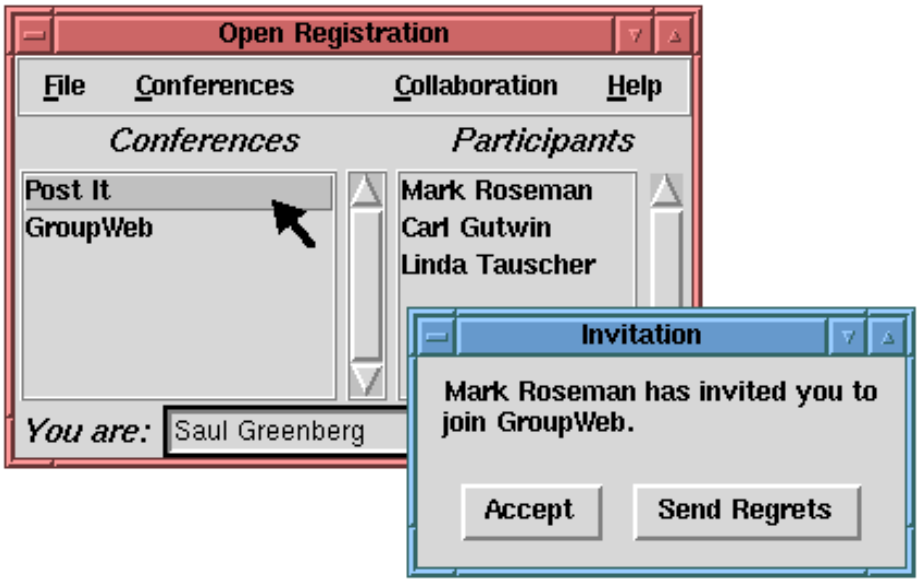

Figure 1.8 The Open Door session manager. Two conference sessions are shown, with three participants present in the "Post It" conference.

1.5.1.2 Open Door

The basic session manager provided by GroupKit [62, 29] offers an "Open Door" permissive policy of creating and joining conferences, where people think in terms of conferences and participants instead of IP addresses. Figure 1.8 shows an example. Each conference contains a single groupware application (the application windows are not shown in the figure). In the "Conferences" pane, the local person (Saul Greenberg) sees that two conferences are in progress: "PostIt" and "Design Session". By selecting one of them, he can then see who is in a particular conference (the list in the "Participants" pane).

Conferences are entered in several ways: joins, invitations, and creation. First, Saul can *join* a conference by double clicking any conference name. This adds him to the list of participants and causes the particular application to appear on the display. Second, a person already in a conference can *invite* Saul into the session via a menu option, and a dialog will appear on the screen asking him if he wishes to join in. An example of this is shown in the figure. Third, Saul can *create* a new conference via the "Conference" menu: when he selects from a list of applications, a window running the application appears on the display and others are informed of its availability through the Conferences pane. This session manager also handles departure, and exiting attendees disappear from the Participants pane. When the last one leaves, that person is asked if the conference application should persist i.e., that its state should be saved so it can be re-entered later with its contents intact.

1.5.1.3 Rendezvous Points

A quite different policy provides common rendezvous points. People go to a "place", and are automatically connected to all others in that place. The best known example of these are the popular Multi-User Dungeons (MUDs). When a person connects to a MUD via a

well-known Internet address, they enter one of several rooms where they can engage in a text-based chat dialog with all others in the room. TeamRooms [63] carries the ideas of MUDs to graphical groupware by a rooms-based metaphor. Users of a community can create virtual meeting rooms, and stock them with groupware meeting tools. To create opportunities for collaboration, anyone can see what rooms are available, who is around, what rooms they are in, and how active they are. People can freely move between rooms. When they enter a room they are joined to all the conferencing tools located in the room; when they leave the room, any tools used in the room are left behind. If only one person is in a room, then it behaves as a single-user system. If no one is in a room, the tools and groupware artifacts remain available as they are treated as persistent conference sessions. This system could serve the needs of collaborators working on many tasks over a period of time, allowing them to easily move between tasks. It also serves as a meeting place, where people can see who is around in what room, and converse with them after entering the room. We expect place-based systems such as TeamRooms to have wide appeal, and other researchers are also pursuing this policy [75]. For example, Lee, Prakash and Jaeger [47] are developing a general software architecture and API to such systems.

1.5.1.4 Other Policies

Many other session managers are possible. For example, a facilitated meeting session manager has been implemented in GroupKit, where a chairperson has complete control over what applications are part of a meeting, and who can participate. Other policy examples follow the model of telephone calls, or the way conference calls are established through a central switching point. A session manager can also be document-centric. For example, if a person opens a file that is currently being edited by someone else, the groupware connection can be made automatically. The point is that a developer requires the tools to modify packaged session managers or create new ones that fit the community.

1.5.2 Building Blocks for Session Managers

Most toolkits provide only rudimentary and hard-wired session management facilities. ShareKit, for example, provides only basic connection facilities, although it does allow information about participants and about the session to be transmitted to others upon connection. Similarly, Rendezvous has a built-in session manager which they call a startup architecture [57]. There have been a few investigations into architectures for flexible session management e.g., Intermezzo [15, 47] but these are not really toolkits. Excepting GroupKit, most toolkits do not let programmers build both applications and session managers, or do not separate the two concepts.

Because few toolkits support session management as a first class entity, we are a long way from knowing exactly what primitives and API should be provided to the developer. In our own experiences with GroupKit, we have developed flexible session management facilities around the idea of *open protocols* [65]. Briefly, the Registrar central server (Figure 1.2) provides a replicated data structure that tracks meetings and attendees, but specifies no policy for how the data structure is to be used. Session managers are clients to the Registrar, and specify the policy by the selection of operations they perform. Maximum flexibility is achieved by providing open access to the Registrar's data structure via a protocol or interface of small but

powerful operations (e.g., add or delete conference). Clients may be different, as long as they are well behaved with respect to each other and to the policy.

In terms of programming session managers, programmers can trap session manager events and take actions upon them via callbacks. Different session managers will use these in different ways to create their policy. To ease the programmer's chore, GroupKit also provides default callbacks to handle routine operations. The programmer can over-ride these when necessary. Using these events, the programmer can create different access control mechanisms, start new applications or end existing ones, and build the interface in a way that shows the user what is going on. Examples of some of the events are described below.

- *userRequestNewConf:* the user has requested that a new conference be created
- *newConfApproved* and *deleteConfApproved*: the request for a new conference or termination of an existing one has been approved
- *foundNewConf* and *foundDeletedConf:* a new conference has been created, or an existing one has been removed.
- *foundNewUser* and *foundDeletedUser:* a user has entered or left a conference.
- *newUserApproved*: the user's admittance into the conference has been approved
- *lastUserLeftConf*: the last user in a conference has left
- *conferenceDied:* a conference process we created has terminated

1.5.3 Discussion

Both good groupware applications and good session managers are needed for groupware to succeed. Without good session managers, it is hard to make electronic contact and get groupware started; many opportunities for collaboration will likely fall by the wayside. We believe that next generation toolkits will, like GroupKit, include session management as an important building block. At the very least, the toolkit should provide a reasonable set of stock session managers that implement a broad range of policies. If adequate primitives are provided, the programmer should be able to modify existing session managers and create new ones to fit the particular needs of a work community.

It is even possible that session management toolkits can be developed that are completeley independent from the application component and its run-time architecture. As evidence, the GroupKit session manager was recently repackaged as a stand-alone toolkit. Since then, it has been adapted to work with the Clock groupware development tool [18] to manage both centralized and semi-replicated sessions. While minor code changes were required, it works well in spite of the radical differences between the run-time system and underlying language of Clock and GroupKit.

1.6 CONCLUSION

This chapter has presented four components that we believe toolkits must provide to groupware programmers. A run-time infrastructure automatically manages the creation, interconnection, and communications of the distributed processes that comprise conference sessions, greatly simplifying a programmer's job of managing a distributed system. Groupware programming abstractions allow developers to control the behaviour of distributed processes, to take action on state changes, to share relevant data, and to generate views. Groupware widgets let a programmer quickly add interface features of value to conference participants. Session

managers that let users create and manage their meetings are built by developers to accommodate the group's working style. Examples were taken from a variety of different toolkits to illustrate how these components can be provided in practice.

The class of groupware toolkits considered in this chapter consider only real time distributed applications. This is just a subset of groupware, and many groupware toolkits address disparate application domains. For example, ConversationBuilder [41] and Strudel [69] are used for constructing speech act protocols. Oval is used to build semi-structured messaging and information management systems [49]. Lotus Notes, although not a programming toolkit, lets people develop and tailor a wide variety of asynchronous applications (Lotus Inc.). Even toolkits within the domain of real-time interaction handle different niche problems. Dewan and Choudhary's Suite toolkit [10] applies only to highly structured text objects and investigates how flexible access control mechanisms are incorporated into them. Knister and Prakash's [43] DistEdit provides groupware primitives that could be added to existing single user text editors to make them group aware. DistView, produced by the same group, is oriented towards a fairly strict view-sharing approach to sharing window components and underlying data via an object replication scheme [61]. Smith and Rodden's SOL considers design features for making single user widgets shareable [70].

The chapter also limited its discussion to four components. While we believe these are fundamental building blocks, there are certainly other components that must be included in a commercial, robust groupware toolkit. A few examples follow (see Urnes and Nejabi [78] for a further list of features).

- *Security and privacy.* Groupware could be a large security hole unless great care is taken in determining that only the right people are allowed in a meeting, and that permissions to execute actions at sites other than their own does not compromise the system. Similarly, communication channels should be encrypted in case the conference deals with sensitive information. These should all be supplied as part of the stock toolkit.
- *Audio and video support.* Most of the toolkits mentioned do not directly support audio and video. Yet almost all real time groupware requires at least audio. These can be provided out of band, through telephones, video conferencing systems, and media spaces. Still, there is a trend in application design to integrate audio, video, and computational groupware. The ClearBoard system described in Ishii's chapter, for example, allows participants to see through their computational space to a video image that portrays correct eye gaze position and hand gestures relative to the surface (see also [39]). There is also the problem of synchronizing audio/video with actions in the computational space, for even a few seconds of delay between the two can be disconcerting to the group members. A further discussion on multimedia in groupware can be found in Dourish's chapter.
- *Communication channel and networks.* All groupware systems depend upon communication channels. Ideally, the underlying network will be tuned to support the performance demands of groupware, and the API should reflect the programmer's needs. Example extensions to standard networks are MBONE [48], an Internet multicast backbone that lets one send multimedia on wide area networks such as the Internet, and Isis [5], which guarantees correct serialization of events over the network.
- *Fault tolerance.* As network loads increase and connections become less reliable, fault tolerance becomes increasingly important. Groupware toolkits must include facilities to allow the application to degrade gracefully, to checkpoint failed conferences for later resumption,

and to seek alternate communication paths when a channel fails. Dourish also addresses some of these issues in his chapter.

- *Versioning and downloading.* In replicated architectures, problems arise when one site is missing software or has a different version of it. The system should be able to check versions, and download software when necessary.
- *Session capture and replay.* Records of meetings are sometimes crucial. While capturing video is straightforward, capturing computational actions is more difficult [50]. The challenge remains on how to capture automatically the highlights of lengthy meetings in a concise manner.
- *Multi-user undo.* Many single user systems contain undo facilities. Yet undo in groupware is a hard problem. While a few researchers have been working in this area [60, 3], we still have a long way to go before we can package undo facilities so that groupware programmers can include it easily within their application. The chapter by Prakash contains a detailed discussion of the role of undo in a group editor.
- *Concurrency control.* While mentioned as part of the run-time architecture, concurrency control in groupware is a sub-field in its own right. Much work remains to be done crafting appropriate tools, architectures, and abstractions that make concurrency control easy for the programmer, while minimizing its impact on the end-user's interface.
- *Application domains.* In all probability, some groupware toolkits will have to be specialized to handle the nuances of particular real-time applications domains. DistEdit, for example, concerns itself only with text editing [43]. Others will deal with the structured meetings found in group support systems [59], or with extending capabilities of existing single-user systems e.g., primitives to make the *emacs* text editor group-aware [54].
- *Alternate models.* The separation of model and view is only one of the many ways that groupware can be configured. For example, Karsenty and Beaudouin-Lafon [42] have defined the seven-layer SLICE model. Some of these layers are: an abstract document (the model), a document layer (the displayed view), a direct manipulation layer (the means to interact with the view); a view representation layer (to control how views are displayed); and a cursor layer that tracks the mouse and shows telepointers. Dewan's chapter in this book considers other architectural models as well.
- *Development environments.* All the toolkits mentioned have inadequate development environments. For example, debugging groupware is hard because it is a distributed system, and we need appropriate debuggers. Interface builders are lacking. Appropriate tools for testing are non-existent.
- *The Web.* The recent popularity of the World Wide Web, as well as the network and multi-platform properties of the Java programming language, implies that the Web could become *the* delivery vehicle for real time groupware. While the Web, Java and the Internet itself have particular features that lend themselves towards groupware (e.g., its ubiquity, its client/server model, its telecommunications constructs), it also includes constraints that may challenge the design of groupware toolkits (e.g., security, performance, session management styles). While the Web does provide incredible opportunities for groupware (some are surveyed in the chapter by Dourish), we may find ourselves compromised by its technical constraints and by the way it is commonly used.

While the next generation of toolkits are now being built, groupware systems still have a long way to go to catch up to their single-user counterparts. We look forward to the day when

all toolkits will incorporate multi-user features. When that day comes, the artificial distinction between constructing single and collaborative systems will disappear.

Acknowledgments.

Carl Gutwin and Ted O'Grady participated in many discussions about what is required for groupware toolkits, and helped influence the contents of this paper. Prasun Dewan, Nicholas Graham, and John Patterson reviewed versions of this manuscript. They contributed both constructive comments and further system description. Comments by anonymous referees helped improve this document. Funding by the National Science and Engineering Research Council of Canada, and by Intel Corporation are gratefully appreciated.

References

[1] Ahuja, S.R., Ensor, J.R. and Lucco, S.E. (1990) "A comparison of applications sharing mechanisms in real-time desktop conferencing systems." In *Proceedings of the ACM COIS Conference on Office Information Systems*, p238–248, Boston, April 25–27.

[2] Baecker, R., Nastos, D., Posner, I. and Mawby, K. (1993) "The user-centered iterative design of collaborative writing software." in *Proceedings of ACM InterCHI'93 Conference on Human Factors in Computing Systems*, p399–405, Amsterdam, The Netherlands, April 24–29.

[3] Berlage, T. (1994) "A selective undo mechanism for graphical user interfaces based on command objects." *ACM Transactions on Computer-Human Interaction*, 1(3), p269–294, September.

[4] Bharat, K. and Brown, M. (1994) "Building distributed, multi-user applications by direct manipulation." in *Proceedings of the ACM UIST'94 Symposium on User Interface Software and Technology*, p71–80, Marina del Rey, California, November 2–4.

[5] Birman, K.P. (1993) "The process group approach to reliable distributed computing." *Communications of the ACM*, 36(12), p37–53, December.

[6] Bonfiglio, A., Malatesta, G. and Tisato, F. (1989) "Conference Toolkit: A framework for real-time conferencing." in *Proceedings of the EC-CSCW '89 First European Conference on Computer Supported Cooperative Work*, p303–316, Gatwick, London, UK, September 13–15.

[7] Carlsson, C. and Hagsand, O. (1993) "DIVE – A Platform for Multi-User Virtual Environments." *Computers and Graphics*, 17(6).

[8] Cockburn, A. and Greenberg, S. (1993) "Making contact: Getting the group communicating with groupware." in *Proceedings of the ACM COOCS'93 Conference on Organizational Computing Systems, p31–41,* Milpitas, California, November 1–4.

[9] Dewan, P. (1991) "Flexible user interface coupling in collaborative systems." in *Proceedings of the ACM CHI'91 Conference on Human Factors in Computing Systems*, p41–48, New Orleans, Louisiana, April 28–May2.

[10] Dewan, P. and Choudhary, R. (1992) "A high-level and flexible framework for implementing multi-user user interfaces." *ACM Transaction on Information Systems.* 10(4), p345–380.

[11] Dourish, P. (1995) "Developing a reflective model of collaborative systems." *ACM Transactions on Computer-Human Interaction.* 2(1), p40-63, March.

[12] Dourish, P. (1996) "Consistency guarantees: Exploiting application semantics for consistency management in a collaboration toolkit." in *Proceedings of the ACM CSCW'96 Conference on Computer Supported Cooperative Work*, Boston, Mass., November 16–20, in press.

[13] Dourish, P. and Bellotti, V. (1992) "Awareness and coordination in shared workspaces." in *Proceedings of the ACM CSCW'92 Conference on Computer Supported Cooperative Work*, p107–114, Toronto, Canada, October 31–November 4.

[14] Dourish, P. and Bly, S. (1992). "Portholes: Supporting awareness in a distributed work group." in *Proceedings of the ACM CHI'92 Conference on Human Factors in Computing Systems*, p541–547, Monterey, California, May 3–7

[15] Edwards, W.K. (1994) "Session management for collaborative applications." in *Proceedings of the ACM CSCW'94 Conference on Computer Supported Cooperative Work*, p323–330, Chapel Hill,

North Carolina, October 22–26.
[16] Ellis, C.A. and Gibbs, S.J. (1989) "Concurrency control in groupware systems." In *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, p399–407, Seattle, Washington, USA.
[17] Graham, T.C.N. (1995) *Declarative development of interactive systems.* Volume 243 of Berichte der GMD, R. Oldenbourg Verlag, Munich, July.
[18] Graham, T.C.N., Morton, C.A. and Urnes, T. (1996). "ClockWorks: Visual programming of component-based software architectures." *Journal of Visual Languages and Computing*, July, Academic Press.
[19] Graham, T.C.N. and Urnes, T. (1992) "Relational views as a model for automatic distributed implementation of multi-user applications." in *Proceedings of the ACM CSCW'92 Conference on Computer Supported Cooperative Work*, p59–66, Toronto, Canada, October 31–November4.
[20] Graham, T.C.N., Urnes, T. and Nejabi, R. (1996) "Efficient distributed implementation of semi-replicated synchronous groupware." in *Proceedings of the ACM UIST '96 User Interface Software and Technology*, Seattle, Washington, November 6-8, In press.
[21] Graham, T.C.N. and Urnes, T. (1996). "Linguistic support for the evolutionary design of software architectures." in *Proceedings of the ICSE'18 Eighteenth International Conference on Software Engineering*, p418–427, March, IEEE Press.
[22] Greenberg, S. (1990) "Sharing views and interactions with single-user applications." In *Proceedings of the ACM COIS Conference on Office Information Systems*, p227–237, Boston, Mass., April 25-27.
[23] Greenberg, S. and Gutwin, C. (1996) "Applying distortion-oriented displays to groupware." in *Video Proceedings of the ACM CSCW'96 Conference on Computer Supported Cooperative Work*, Boston, Mass., November 16-20. Videotape available from ACM Press.
[24] Greenberg, S., Gutwin, C. and Cockburn, A. (1996) "Awareness through fisheye views in relaxed-WYSIWIS groupware." In *Proceedings of Graphics Interface'96*, p28–38, Toronto, Ontario, May. Distributed by Morgan-Kaufmann.
[25] Greenberg, S., Gutwin, C. and Cockburn, A. (1996) "Using distortion-oriented displays to support workspace awareness." In A. Sasse, R.J. Cunningham, and R. Winder, (Editors), *People and Computers XI (Proceedings of the HCI'96)*, p299–314, Springer-Verlag. Conference held at Imperial College, London, August 20–23.
[26] Greenberg, S. Gutwin, C. and Roseman, M. (1996) "Semantic telepointers for groupware." in *Proceedings of OZCHI '96: The Sixth Australian Conference on Computer-Human Interaction*, Hamilton, New Zealand, November 24–27. In press.
[27] Greenberg, S. and Roseman, M. (1996) "GroupWeb: A WWW Browser as Real Time Groupware". in *ACM SIGCHI'96 Conference on Human Factors in Computing System, Companion Proceedings*, p271–272, Vancouver, Canada, April 13-18.
[28] Greenberg, S. and Marwood, D. (1994) "Real time groupware as a distributed system: Concurrency control and its effect on the interface." in *Proceedings of the ACM CSCW'94 Conference on Computer Supported Cooperative Work*, p207–217, Chapel Hill, North Carolina, October 22–26.
[29] Greenberg, S. and Roseman, M. (1994) "GroupKit." in *ACM SIGGRAPH Video Review*, Issue 108, Videotape available from ACM Press.
[30] Gutwin, C., Greenberg, S. and Roseman, R. (1996). "Supporting awareness of others in groupware." in *ACM SIGCHI'96 Conference on Human Factors in Computing System, Companion Proceedings*, p205–215, Vancouver, Canada, April 13-18.
[31] Gutwin, C., Greenberg, S. and Roseman, M. (1996) "Workspace awareness in real-time distributed groupware: Framework, widgets, and evaluation." in A. Sasse, R.J. Cunningham, and R. Winder, (Editors), *People and Computers XI (Proceedings of the HCI'96)*, p281–298, Springer-Verlag. Conference held at Imperial College, London, August 20–23.
[32] Gutwin, C., Greenberg, S. and Roseman, M. (1996) " Staying aware in groupware workspaces." in *Video Proceedings of the ACM CSCW'96 Conference on Computer Supported Cooperative Work*, Boston, Mass., November 16–20, Videotape available from ACM Press.
[33] Gutwin, C., Roseman, M., and Greenberg, S. (1996) "A usability study of awareness widgets in a shared workspace groupware system." In *Proceedings of the ACM CSCW'96 Conference on Computer Supported Cooperative Work*, Boston, Mass., November 16–20, in press.
[34] Gutwin, C., Stark, G. and Greenberg, S. (1995) "Support for workspace awareness in educational

groupware." in *Proceedings of the CSCL'95 Conference on Computer Supported Collaborative Learning*, p147–156, Bloomington, Indiana, October 17–20. Distributed by Lawrence Erlbaum Associates.

[35] Hayne, S., Pendergast, M. and Greenberg, S. (1994) "Implementing gesturing with cursors in Group Support Systems." *Journal of Management Information Systems*, 10(3), p43–61.

[36] Hill, R.D. (1992) "The Abstraction-Link-View paradigm: Using constraints to connect user interfaces to applications." in *Proceedings of the ACM SIGCHI'92 Conference on Human Factors in Computing Systems*, p335–342, Monteray, California, May 3–7.

[37] Hill, R.D., Brinck, T., Rohall, S.L., Patterson, J.F. and Wilner, W. (1994) "The Rendezvous architecture and language for constructing multi-user applications." *ACM Transactions on Computer-Human Interaction*, 1(2), p81–125, June.

[38] Intel Corporation. Software available through the world wide web, http://www.intel.com/iaweb/moondo/index.html.

[39] Ishii, H. and Kobayashi, M. (1992) "ClearBoard: A seamless medium for shared drawing and conversation with eye contact." in *Proceedings of the ACM CHI'92 Conference on Human Factors in Computing Systems*, p525–532, Monterey, California, May 3–7.

[40] Jahn, P. (1995) "Getting started with Share-Kit." Tutorial manual distributed with Share-Kit version 2.0. Communications and Operating Systems Research Group, Department of Computer Science, Technische Universitat, Berlin, Germany. Available via anonymous ftp from ftp.inf.fu-berlin.de:/pub/misc/share-kit.

[41] Kaplan, S.M., Tolone, W.J., Bogia, D.P. and Bignoli, C. (1992) "Flexible, active support for collaborative work with conversation builder." In *Proceedings of the ACM CSCW'92 Conference on Computer Supported Cooperative Work*, p378–385, Toronto, Canada, October 31–November 4.

[42] Karsenty, A. and Beaudouin-Lafon, M. (1995) "Slice: A logical model for shared editors." In *Groupware for Real Time Drawing, A Designer's Guide*, Edited by S. Greenberg, S. Hayne and R. Rada, p156–173, McGraw-Hill Europe.

[43] Knister, M.J. and Prakash, A. (1990) "DistEdit: A distributed toolkit for supporting multiple group editors." in *Proceedings of ACM CSCW'90 Conference on Computer Supported Cooperative Work*, p343–355, Los Angeles, California, October 7–10.

[44] Krasner, G.E. and Pope, S.T. (1988) "A cookbook for using the model-view-controller user interface paradigm in Smalltalk-80." *Journal of Object Oriented Programming* 1(3), p26–49, August/September.

[45] Lauwers, J.C. and Lantz, K.A. (1990) "Collaboration awareness in support of collaboration transparency." in *Proceedings of the ACM SIGCHI'90 Conference on Human Factors in Computing Systems*, p303-211, Seattle, Washington, April 1–5.

[46] Lauwers, J.C., Joseph, T.A., Lantz, K.A. and Romanow, A.L. (1990) "Replicated architectures for shared window systems: A critique." In *Proceedings of the ACM COIS'90 Conference on Office Information Systems*, p249–260, Boston, Mass., April 25–27.

[47] Lee, J.H., Prakash, A., Jaeger, T. and Wu, G. (1996) "Supporting multi-user, multi-applet workspaces in CBE." in *Proceedings of the ACM CSCW'96 Conference on Computer Supported Cooperative Work*, Boston, Mass., November 16–20, in press.

[48] Macedonia, M.R., Brutzman, D.P., (1994). "MBone provides audio and video across the Internet."*IEEE Computer*, 27(4), p30–36, IEEE Press.

[49] Malone, T.W., Lai, K.Y. and Fry, C. (1992) "Experiments with Oval: A radically tailorable tool for cooperative work." *Proceedings of the ACM CSCW'92 Conference on Computer Supported Cooperative Work*, p289–297, Toronto, Canada, October 31–November 4.

[50] Manohar, N.R. and Prakash, A. (1995) "The session capture and replay paradigm for asynchronous collaboration." in *Proceedings of the ECSCW'95 Fourth European Conference on Computer Supported Cooperative Work*, p149–164, September.

[51] Mitchell, A.(1996) "Communications and shared understanding in collaborative writing" M.Sc. Thesis, Department of Computer Science, University of Toronto, Canada.

[52] O'Grady, T. (1996) "Flexible data sharing in a groupware toolkit." M.Sc. Thesis, Department of Computer Science, University of Calgary, Calgary, Alberta, Canada. November.

[53] Ousterhout, J. (1994) "Tcl and the Tk Toolkit." Addison Wesley.

[54] Patel, D. and Kalter, S.D. (1995) "Commercializing a real-time collaborative toolkit." In S. Greenberg, S. Hayne and R. Rada, Editors, *Groupware for Real Time Drawing, A Designer's Guide*,

p198–208, McGraw-Hill Europe.

[55] Patterson, J.F. (1994) "A taxonomy of architectures for synchronous groupware applications." Paper presented to the *Workshop on Software Architectures for Cooperative Systems*, held as part of the ACM CSCW'94 Conference on Computer Supported Cooperative Work.

[56] Patterson, J.F. (1991) "Comparing the programming demands of single-user and multi-user applications." in *Proceedings of the UIST'92 Symposium on User Interface Software and Technology*, p87–94, Hilton Head, South Carolina, November 11–13.

[57] Patterson, J. F., Hill, R. D., Rohall, S. L., and Meeks, W. S. (1990). "Rendezvous: An architecture for synchronous multi-user applications". in *Proceedings of the CSCW'90 Conference on Computer Supported Cooperative Work*, Los Angeles, California, October 7–10.

[58] Patterson, J.F., Day, M. and Kucan, J. (1996) "Notification servers for synchronous groupware." in *Proceedings of the ACM CSCW'96 Conference on Computer Supported Cooperative Work*, Boston, Mass., November 16–20, in press.

[59] Pendergast, M. (1995) "GroupGraphics: Prototype to product." In S. Greenberg, S. Hayne and R. Rada, Editors, *Groupware for Real Time Drawing, A Designer's Guide*, p209–227, McGraw Hill Europe.

[60] Prakash, A. and Knister, M.J. (1992) "Undoing actions in collaborative Work." in *Proceedings of the ACM CSCW'92 Conference on Computer-Supported Cooperative Work*, p273–280, Toronto, Canada, October 31–November 4.

[61] Prakash, A. and Shim, H.S. (1994) "DistView: Support for building efficient collaborative applications using replicated objects." in *Proceedings of the ACM CSCW'94 Conference on Computer-Supported Cooperative Work*, p153–164, Chapel Hill, North Carolina, October 22–26.

[62] Roseman, M. and Greenberg, S. (1996) "Building real time groupware with GroupKit, a groupware toolkit." *ACM Transactions on Computer-Human Interaction*, 3(1), p66–106, March.

[63] Roseman, M. and Greenberg, S. (1996). "TeamRooms: Network places for collaboration." In *Proceedings of the ACM CSCW'96 Conference on Computer Supported Cooperative Work*, Boston, Mass., November 16–20, in press.

[64] Roseman, M. and Greenberg, S. (1994) "Registration for real time groupware." Research Report 94/533/02, Department of Computer Science, University of Calgary, Alberta, Canada.

[65] Roseman, M. and Greenberg, S. (1993) "Building flexible groupware through open protocols." in *Proceedings of the ACM COOCS'93 Conference on Organizational Computing Systems*, p279–288, Milpitas, California, November 1–4.

[66] Roseman, M. and Greenberg, S. (1992). "GroupKit: A groupware toolkit for building real-time conferencing applications." *Proceedings of the ACM CSCW'92 Conference on Computer Supported Cooperative Work*, p43–50, Toronto, Canada, October 31–November 4.

[67] Schieffler, R.W. and Gettys, J. (1986) "The X-Windows system." *ACM Transactions on Computer Graphics, 5*, p79–109.

[68] Shen, H. and Dewan, P. (1992) "Access control for collaborative environments." in *Proceedings of the ACM CSCW'92 Conference on Computer Supported Cooperative Work*, p51–58, Toronto, Canada, October 31–November 4.

[69] Shepherd, A., Mayer, N. and Kuchinsky, A (1990) "Strudel — an extensible electronic conversation toolkit." in*Proceedings of ACM CSCW'90 Conference on Computer-Supported Cooperative Work*, p93–104, Los Angeles, California, October 7–10.

[70] Smith, G. and Rodden T. (1993) "Using an access model to configure multi-user interfaces." in *Proceedings of the ACM COOCS '93 Conference on Organizational Computing System*, p289–298, Milpitas, California, November 1–4.

[71] Smith R. B., O'Shea T., O'Malley C., Scanlon E., and Taylor J. (1989). "Preliminary experiences with a distributed, multi-media,problem environment." In *Proceedings of the EC-CSCW '89 1st European Conference on Computer Supported Cooperative Work*, Gatwick, U.K., September 13–15.

[72] Tang, J.C. (1991) "Findings from observational studies of collaborative work." International Journal of Man Machine Studies, 34(2), p143–160. Republished under the same title in Saul Greenberg, editor, "Computer Supported Cooperative Work and Groupware," Academic Press.

[73] Tatar D. G., Foster G., and Bobrow D. G. (1991). "Design for conversation: Lessons from Cognoter." *International Journal of Man Machine Studies*, 34(2), p185–210, February. Republished under the same title in Saul Greenberg, editor, "Computer Supported Cooperative Work and

Groupware," Academic Press.
[74] Taylor, R.N., Nies, K.A., Bolcer, G.A., MacFarlane, C.A., Anderson, K.M. and Johnson, G.F. (1995) "Chiron-1: A software architecture for user interface development, maintenance, and run-time support." *ACM Transactions on Computer-Human Interaction*, 2(2), p105-144, June.
[75] Tolone, W., Kaplan, S. and Fitzpatrick, G. (1995) "Specifying dynamic support for collaborative work within wOrlds." In *Proceedings of the ACM COOCS '95 Conference on Organizational Computing System, ,* p55–67, Mipitas, California, August 13–16.
[76] Tou, I., Berson, S., Estrin, G., Eterovic, Y. and Wu, E. (1994) "Prototyping synchronous group applications." *IEEE Computer*, 27(5), p48–56, May.
[77] Trevor, J., Rodden, T. and Mariani, J. (1994) "The use of adaptors to support cooperative sharing." in *Proceedings of the ACM CSCW'94 Conference on Computer Supported Cooperative Work*, p219–230, Chapel Hill, North Carolina, October 22–26.
[78] Urnes, T. and Nejabi, R. (1994) "Tools for implementing groupware: A survey and evaluation." Technical report CS-94-03, Department of Computer Science, York University, Toronto, Canada.
[79] Wilson, B. (1995) "WSCRAWL 2.0: A shared whiteboard based on X-Windows." In S. Greenberg, S. Hayne and R. Rada, Editors, *Groupware for Real Time Drawing, A Designer's Guide*, p129–141, McGraw Hill Europe.