Simplifying Component Development in an Integrated Groupware Environment

Mark Roseman and Saul Greenberg

Department of Computer Science, University of Calgary
Calgary, Alberta, Canada T2N 1N4
Tel: +1-403-220-3532
E-mail: roseman,saul@cpsc.ucalgary.ca

ABSTRACT

This paper describes our experiences implementing a component architecture for TeamWave Workplace, an integrated groupware environment using a rooms metaphor. The problem we faced was how to design the architecture to support rapid development of new embedded components. Our solution, based on Tcl/Tk and GroupKit, uses multiple interpreters and a shared window hierarchy. This proved effective in easing development complexity in TeamWave. We discuss some of the strategies we used, and identify the types of interactions between system components. The lessons learned in developing this component model should be generally applicable to future integrated groupware systems in different environments.

KEYWORDS: Groupware, CSCW, Tcl/Tk, GroupKit, component architecture

INTRODUCTION

This paper details the component model built for TeamWave Workplace (formerly TeamRooms [6]). TeamWave is an Internet groupware environment that uses a rooms metaphor to integrate a team's tools and tasks. The component model supports the development of new groupware tools that can be integrated into the main application.

One of the strengths of TeamWave is that individual rooms can be customized by adding new components, called tools, with each being a small groupware application in its own right. To encourage the development of these custom components, we needed to make their development as easy as possible.

Cite as:
Roseman, M. and Greenberg, S. (1997). Simplifying Component Development in an Integrated Groupware Environment. Proceedings of the ACM UIST'97 Symposium on User Interface Software and Technology, p65-72, October 14-17, Banff, Alberta. ACM Press.

Historically, developing components with frameworks like OLE or OpenDoc [3] has been very complex, and therefore unsuitable for our needs. We decided instead to create our own model that would be oriented towards the requirements of building fairly small, highly interactive groupware tools.

The paper describes how component development within TeamWave is seen by developers, as well as the implementation framework that supports our component architecture. We then examine how the techniques we used simplified the development of new components.

The techniques we developed are based on our earlier work with GroupKit [5], a groupware toolkit developed in Tcl/Tk. GroupKit has been shown to be useful in developing stand-alone groupware applications. Here we extend this work to components, describing our use of multiple Tcl interpreters and a shared window hierarchy as the basis of our component model.

An interesting aspect of our architecture is how different components of the system — the overall application, the current room, and the individual tools in the room — interact with each other. In structuring the system, we used several different interaction strategies. In the following sections, we will see how the application used various levels of knowledge of the individual tools or room to call into them. We will also see how the application explicitly provides high-level interfaces to the components so as to limit the information each component needs to maintain. Finally, we will show how we took advantage of our decision to share the entire window hierarchy throughout the application. All of these had interesting implications for simplifying the development of new groupware components in TeamWave.

We expect to see more groupware environments in the future that integrate a number of tools rather than merely providing stand-alone applications. We believe many of the techniques

we employed in TeamWave and the lessons we learned will be applicable for new component based groupware systems in a variety of environments.

TEAMWAVE WORKPLACE DESCRIPTION

In this section we describe the TeamWave system, to illustrate the types of components we are interested in and the environment in which they run.

TeamWave Workplace is a groupware environment that supports a wide variety of collaborative activities. It integrates into a single environment shared whiteboards, chat facilities, and custom groupware components such as sticky notes, databases, and calendars. The system offers a persistent work environment supporting both synchronous and asynchronous work.

TeamWave is structured around a rooms-based metaphor. As in real-life rooms, a workgroup or community of users maintains one or more rooms that serve as a focal point for their collaborations. If several people occupy a room at the same time, they can engage in a synchronous collaboration. If a room is occupied by a single person, they can work alone, and possibly leave information for others when they in turn enter the room, thereby supporting asynchronous work. Users can bring arbitrary tools to the room, customizing it to suit their needs. When people leave a room, its contents persist, remaining in place for the next user who enters the room.

The electronic rooms in TeamWave Workplace support users who are not physically co-located. A workgroup's rooms are hosted on an Internet or Intranet server, and group members use the TeamWave client to connect over the network to this server. There they can interact with the rooms and other users who are present. The rooms contain standard tools such as shared whiteboards and chat, and allow adding custom tools for particular group needs; each of these are fully interactive groupware applications. Both clients and servers are supported on Macintosh, Windows and several Unix platforms, and communication is over TCP/IP, whether via LAN/WAN or modem connection. A persistence repository holds the state of the rooms, and allows retrieval of older versions of the room states. This is illustrated in Figure 1.

User Interface

Figure 2 shows the user interface of the system. In this example, the current user (Mark) is located in a room called the Foyer. Also present in the room are two other users (Carl and Saul). As we can see, there are no other users currently connected to the server (Logged in Users window), though there are other rooms besides the Foyer available (Rooms on this Server window). We also can obtain more information about particular users (business card window).

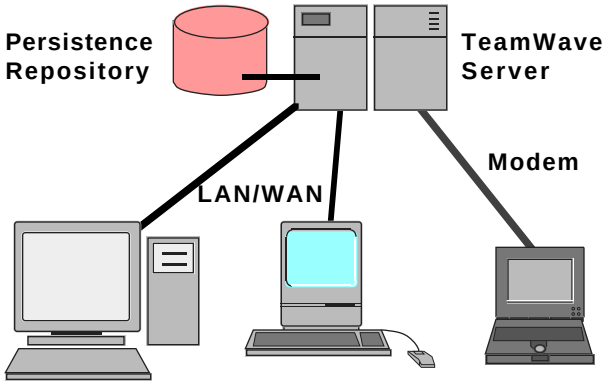

Figure 1. TeamWave System Architecture

The large window shows the Foyer room. The backdrop of the room is a shared whiteboard, permitting freehand sketching and text annotations, using a set of pens located immediately below the whiteboard surface. All actions on the whiteboard and throughout the system are WYSIWIS, and support telepointers. Below the pen tray is a chat area allowing text-based communication between users within the room. Along the left of the window are (top to bottom): a radar view (to stay aware of activity in other parts of large rooms), a set of doors (to indicate privacy), a bell (to draw attention), and a list of others in the room (showing idle times, telepointer colors, and pictures, either static or periodic video snaps).

Tools

On the Foyer's whiteboard surface we also see five tools or components, each miniature groupware applications in their own right. In particular, we have:

- an image tool, allowing images to be added to the room, either by uploading to the TeamWave server (for static images) or retrieved from a Web server (for images that are change periodically). Images can be annotated with the same tools used by the background whiteboard
- a post-it note allowing entry of short textual messages; as with all TeamWave tools, changes to the note are immediately visible to all users in the room
- a simple web browser, where all users in the room always view the same web page
- a shared calendar tool, allowing group members to add or view appointments
- a URL reference tool to leave a pointer to a web page that others in the group can easily view

These and a number of other components come standard with the system, including: a lightweight shared database (e.g. for address books or todo lists), an idea organizer, a concept map tools, a textual file viewer (to discuss a document), a file holder (to upload arbitrary files into the room), games, a

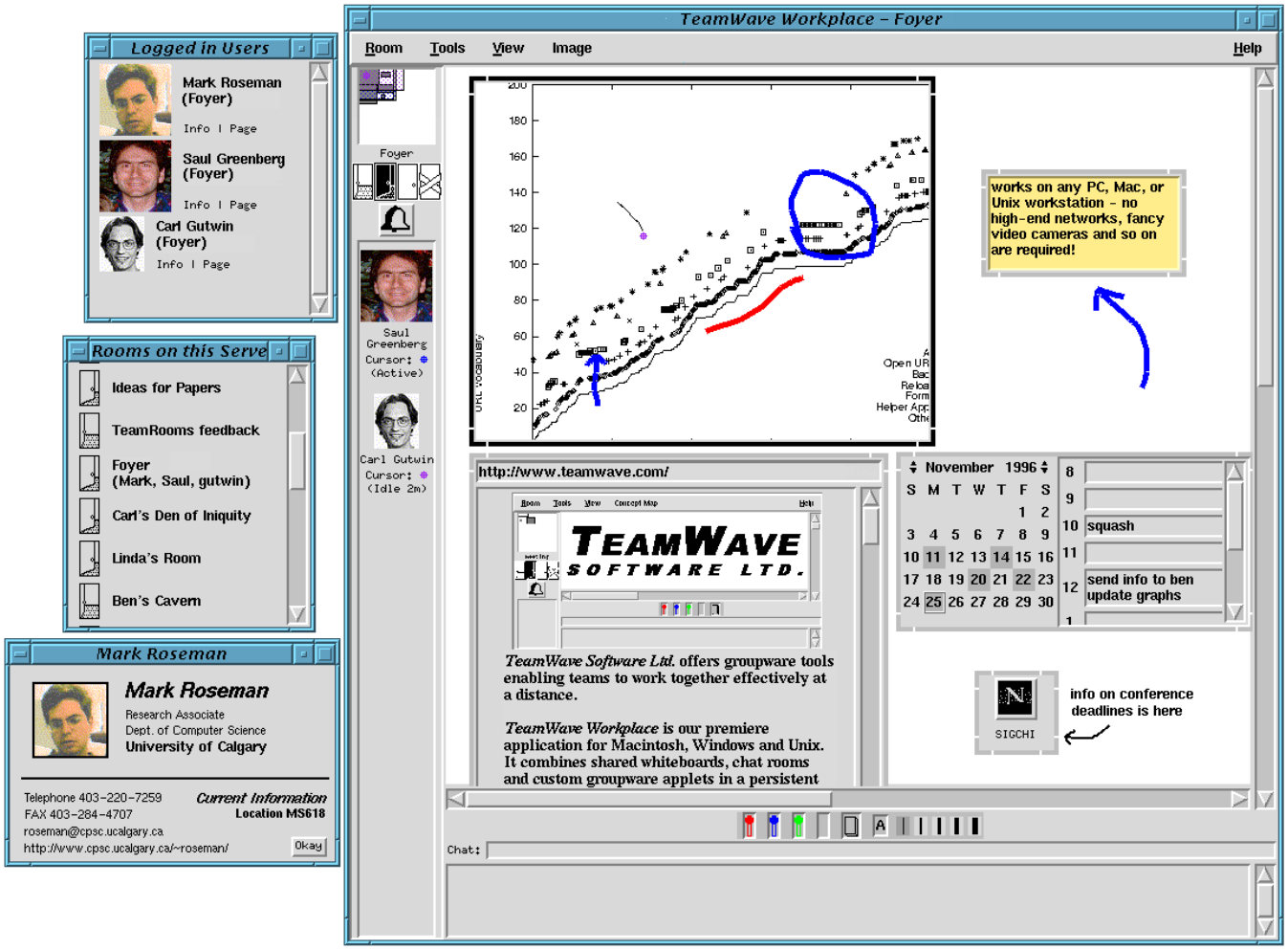

Figure 2. TeamWave Workplace User Interface

doorway to switch between rooms, and so on. There is also a facility whereby newly developed components can be added to the system.

Each component is enclosed by a frame that allows it to be moved around the room or resized. A single component can be made active at a particular time by clicking on it. This draws a black frame, as seen around the Image component in the Figure 2. The menubar changes to add a menu for the active component, containing both system supplied commands (e.g. delete) as well as component-specific menu commands.

BUILDING NEW COMPONENTS

This section describes how new components can be developed and added to the system. We focus here on our goals of making both development and use as straightforward as possible, and then detail how Tcl/Tk and our GroupKit API are used by developers in constructing these components. Later sections describe how TeamWave internally implements this support, and examines more closely some of the issues involved.

Goals

There were two sets of goals for component development. First, we wanted to make it easy for developers to construct new components so as to encourage third party development (as well as easing development of our own components). Second, we wanted to provide a fairly intuitive and straightforward user experience, reflecting our belief that in groupware systems, users should be able to focus on communicating with each other, and not with fighting to understand the tools.

Our first and primary goal was to make it very easy to develop new components. A big advantage of TeamWave Workplace over many other groupware systems is the ability to customize the environment to suit particular needs of a group. That required developing firstly a suite of fairly generic components (e.g. notes, editors, databases) applicable

to a wide variety of tasks. We also wanted to encourage development of tools for particular groups, such as a component that models a specific group process, or provides links to external information already in use by a group, such as in an external database.

A secondary goal was to provide a simplified user experience to end users, emphasizing interactions that could be most visibly communicated to others in the room. We therefore tended towards tools with minimal complexity, favouring direct manipulation interfaces over more intricate but indirect interfaces. We de-emphasized the use of menus (though each component did have access to a menu that was available when it was active). To promote the metaphor of components being contained in the room, we discouraged the use of new toplevel windows, though dialog boxes or property sheets were often used.

User Interface Basics

The basic infrastructure available to the component developer is the Tcl/Tk scripting language [4], originally from UC Berkeley and currently being developed by Sun. We had a number of reasons for selecting Tcl/Tk:

- it is a mature, well-supported development environment that was available on our target platforms
- because of its simplicity, it has a very short learning curve for new developers
- given we were trying to support development of fairly small self-contained tools, we did not anticipate Tcl's poor structure would be an issue
- it contains a number of high-level interface widgets, such as the canvas and text widgets, which provide good support for the highly interactive tools we envisioned
- we had considerable experience with it in our previous user interface and groupware development work, as well as a fairly sizeable code base

From the component developer's perspective, it appeared as if they were developing a standard Tcl/Tk application. They were not explicitly concerned that they were actually running as a component embedded within a larger application.

For example, a developer would construct their interface underneath the "." window (the topmost window of Tk's widget hierarchy), as in a standard application. No other parts of the application or other components are visible to them. Handling resizing of the topmost window (in actuality the frame embedded in the room) was handled the same as a resize by the window system of the toplevel in standard Tk.

At the Tcl scripting level, there was no concern about other components or other portions of the TeamWave application. Component developers could access internal variables, procedures etc. without worrying about namespace collisions with other parts of the application or other components.

There were some exceptions. We provided a different interface to the menubar (a single command to add new items to the component's menu). This restricted the developer to only the single menu. As well, components could access specific functionality from the main application or the room itself. This was accomplished by providing components with a new Tcl command ("workplace") which provided an interface to these various features. We will return to this point later.

In summary, a developer created a component as if it were a standard Tcl/Tk application. They did not have to do any extra programming or housekeeping. This significantly eased the burden of creating new components.

Groupware Functionality

The groupware functionality for the components was based on our GroupKit API [5]. GroupKit is a Tcl/Tk extension that adds a number of features that support real-time groupware development. It has been used fairly extensively, and we were convinced of its ease of use in terms of supporting rapid development of groupware.

GroupKit provides a number of facilities to TeamWave component developers:

- *Remote procedure calls* allow Tcl commands to be sent between TeamWave user processes. Tcl's "all the world's a string" paradigm seamlessly supports transport, data conversion and registration of target commands (though facilities are provided to prevent arbitrary commands from being executed).
- *Conference events* provide notification to let developers respond to such things as new users arriving, users leaving, and recording the state of the component, either to be sent to update newcomers or to be saved in the persistence repository.
- *Environments* provide a shared data abstraction that simplifies maintaining state across client processes. Combined with events for notification, environments can support a Model-View-Controller paradigm with a shared model. This has proven more useful than remote procedure calls in complex groupware applications [2].
- A number of *groupware widgets* provide interfaces to particular groupware functionality, such as displays to maintain awareness of other users' actions. Some standard widgets in GroupKit, such as telepointers, are implemented in TeamWave globally, so individual components do not need to add them explicitly.

In summary, developers use GroupKit's proven API and widget set to make their components group-aware. In practice, this means that groupware development is only slightly more difficult than developing equivalent single user applications.

IMPLEMENTATION

While the previous section dealt with the component developer's view of the system, we now turn to how this view was actually realized in the TeamWave client infrastructure. We look at the overall system architecture, which uses multiple Tcl interpreters to separate various system components. We then look at how the window system is shared between these interpreters, as well as how other resources such as communications channels are shared.

System Architecture

Tcl provides a mechanism to instantiate several interpreters within an application. While mostly independent of each other, the creating (master) interpreter has knowledge of the created (child) interpreters. Through an "alias" feature, the master can cause particular Tcl commands invoked in a child to actually invoke a different Tcl command in the master interpreter. The command runs in the namespace and dataspace of the master, and returns the result of the command to the child. This is commonly used to provide access to a restricted set of functionality; for example it has been used to support safe use of files and sockets from downloaded and untrusted code.

We partitioned the TeamWave client into several interpreters as illustrated in Figure 3. Note that each component runs in its own interpreter, as does the room as a whole (e.g. controlling the whiteboard and chat tool). Both of these are children of the application interpreter, which controls things like navigation between rooms, and is responsible for creating both room and tool interpreters. This means each component is coded like a separate Tcl application.

Window Embedding

We then needed to give each interpreter access to specific parts of the user interface. We chose a strategy whereby a single instance of Tk was run in the application interpreter. A rather elaborate set of aliases (adapted from some work done in a Tcl web browser called SurfIt!) was then used to partition the window hierarchy. This meant that the topmost window (".") of any particular interpreter might in fact correspond to a different window name in another interpreter.

Figure 4 illustrates how the topmost frame of a component might be known as "." to the component's interpreter, ".wb.tool325" to the interpreter for the room, and ".room312.wb.tool325" to the application interpreter. Note though that a component's interpreter would have no way to access windows in any other part of the application unless explicitly permitted to do so.

For example, when the interpreter managing the current room wants to create a new component, it first creates a new frame widget for the contents. It then inserts that frame into its own window hierarchy, within the whiteboard. The room then requests the application interpreter — using an alias provided to the room by the application interpreter — to create a component of a given type, rooted at the newly created frame. The application interpreter creates the new interpreter for the component, sets up all the appropriate aliases so its "." window is mapped to the frame created by the room, and then loads the program for the requested type into the new interpreter.

The frame surrounding each component (which allows movement and resizing) is managed by the room interpreter, since the position and size of tools is within the domain of the room. As the room resizes the frame widget holding the tool, changes in geometry management naturally propagate to the child, which reacts to them exactly as if it were running in a toplevel window which was being resized by the window system.

Switching between active components (and adjusting the menubar) is handled by the application interpreter. Because it has access to the entire window hierarchy, it can detect mouse clicks throughout the application. If the mouse is clicked inside a component other than the current active one,

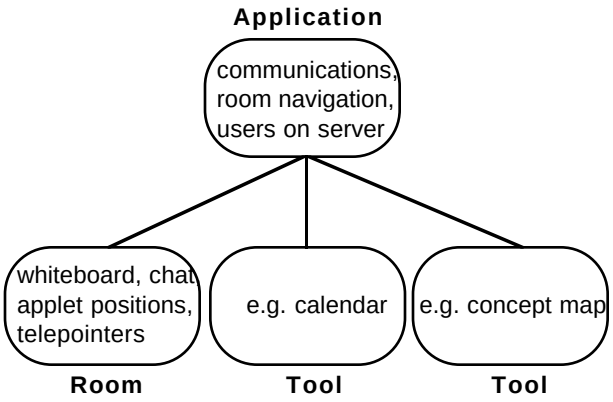

Figure 3. Use of Multiple Interpreters

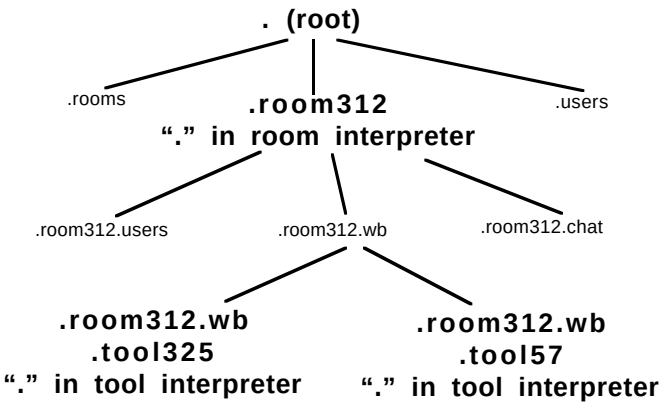

Figure 4. Partitioning the Window Hierarchy

it can adjust the menubar accordingly. It can then notify the room of the change, for example allowing it to change the color of the component's frame.

Thus, sharing a widget hierarchy and using aliases to partition each component's views of the hierarchy is a simple yet powerful way to manage widget sharing. Widget name clashes between interpreters could be controlled by partitioning. Parent interpreters responsible for managing sub-components could access and control widgets in their children's space. Parents could provide explicit interfaces to children for accessing certain 'external' widgets.

Resource Sharing

Besides the window hierarchy, several other resources were shared among the interpreters. For example, the modified version of GroupKit used in TeamWave redirected all communication requests from child interpreters through the master application interpreter. In this way, a single socket connection to the server supported all interpreters running in the client application. Again, this was accomplished through the use of aliases set up in the master interpreter. Each child was not aware that the resource was being shared, which greatly simplifies the coding of components.

EVALUATING COMPONENT COMPLEXITY

This section describes some of our own experiences in building TeamWave components using this framework. Though we are aware of a few third party tools constructed to date, we have not yet examined them in sufficient detail to draw any conclusions from external developers. We expect to examine this more as TeamWave is further deployed and more third parties begin to develop their own components.

The first observation is that we were successful in removing the need for components to know specifically about components. They did not know they were running inside a child interpreter and within other windows. None of the code in our components deals with this issue at all, and were coded as if they were running as standalone applications.

Calendar	197
Concept Map	870
Database	574
Image Annotator	172
Doorway	40
File Viewer	101
File Holder	160
Note Organizer	146
Postit	23
Tetrominoes	193
URL Reference	58
Web Browser	1499

Table 1. Lines of code of standard components.

For example, we successfully adapted several standalone GroupKit applications to run as TeamWave components. Most of the changes were minimal, related to changes in the security model and the menubar API. None of the applications we adapted relied on multiple windows (except for dialog boxes, which were supported), so there were no issues there. We were also able to simplify some of the code, as features like telepointers were provided in the main TeamWave application.

To give a very rough measure of the complexity of the components, Table 1 shows the lines of code count for each of the components supplied in the system. These range from a fairly simple text note to a groupware web browser, to a complex shared graph editor with user-defined types for nodes and edges. As with our earlier GroupKit experiences, most of the code dealt with maintaining the single-user aspects of the interface. The overall low code size we attribute to Tcl/Tk and GroupKit. The point is that creating an embedded component was almost equivalent to creating standalone GroupKit applications.

INTERACTIONS BETWEEN COMPONENTS

One thing we found very interesting was the different types of interactions between the different components in the system — the overall application interpreter, the room interpreter, and the interpreters for each tool.

Our design choices — use of Tcl/Tk, multiple interpreters, and a shared window hierarchy — provided us with a number of different strategies for structuring the application to simplify the development of individual components. Below we detail four of these interactions.

Note that reducing the complexity in the tool components comes with the cost of increased complexity in the application and to some degree the room component. For our application, this was the correct tradeoff, because while we expected many different tool components to be created, there would only be one application and one room created.

Application Calling into Tool Components

The first common interaction between components was where the application component would call into individual tool components. For example, the overall application would notify the tools when users entered and left the room, when the room's contents should be written out to the persistence repository, and so on. We would expect this sort of delegation in any component architecture. Because most of the work here was at the level of the underlying GroupKit layer shared by all tools, this strategy had minimal impact on the complexity of development.

Application Calling into the Room

A related interaction was the application calling into the room interpreter. The difference here was in the level of

knowledge the application had of the components vs. the room. For the tools, the only assumption the application had was they were standard GroupKit programs. For the room, we made a conscious decision to assume the application knew everything about the room's internals. Because of that, the application was free to call procedures, query variables (such as the active tool component), and so on. This type of component interaction was arguably far more useful in the early development stages, helping us to rapidly evolve all the complex interactions between the application and room. If we later chose to allow third party rooms to be added (e.g. different functionality for the shared whiteboard), this design will surely be revisited, trading the flexibility for a more conventional structured interface.

Components Using Application Services

In this type of interaction, the application would explicitly make functionality available to the components. The mechanism here was for the application to provide a new command via an alias, so that the requested command would be run in the application even when invoked from the child. This greatly simplified component development by making a simple high-level interface available to functionality that was either complex or required access to state not directly available to the component developer.

A "workplace" Tcl command provided front ends to many utilities. For example the image component could invoke "workplace pencolor" to get the current pen selected in the pen tray (which the application in turn looked up from the room interpreter), or the doorway could invoke "workplace roomchooser" to popup a dialog letting the user select a room from the available rooms on this server (data which resides only in the application interpreter).

Sharing the Window Hierarchy

The fact that a single instance of the Tk window hierarchy was shared throughout the application offered several benefits that we did not anticipate.

We saw previously how the room interpreter was able to handle resizing of individual components without intervention by the component itself. This was a direct consequence of the component's frame being accessible both from the room and the component interpreters. Similarly, because the application could receive events across the entire window hierarchy, it was able to detect mouse clicks in tools and set the active component appropriately.

Two examples particular to groupware are support of telepointers and idle times. The room interpreter could detect mouse movement throughout its window, including portions of the window assigned to individual components. This let telepointers be implemented in the room, so that each component did not have to implement its own. There was also an application-wide idle indicator used to provide awareness information on other users. Again, this was implemented by watching for mouse moves or keystrokes across the entire window hierarchy.

RELATED APPROACHES

In this section we will briefly compare our approach against other technologies that could be used for supporting the development of add-on components.

OpenDoc/OLE/ActiveX

These technologies are designed explicitly to provide component models such as we have implemented (in fact, many of the interaction techniques in TeamWave were based directly on those in OpenDoc), but none were reasonable candidates. None offered the cross-platform support we required. Further, high-level development tools are still not available, leaving potential developers to struggle directly with the complexities of part development.

Tools as Separate Objects

Another approach we had experimented with was designing the system as a single process (or if using one of the object frameworks in Tcl, as a single interpreter), where each tool was a separate object running in the same namespace as all others, and the application as a whole. This became quite complex, and we continually ran into the case where we had to be careful of namespace collisions. Using objects running in a shared address space would also preclude the use of downloaded tools, which would require some form of restricted access.

Java

Java [1] had just been released when we first began development of TeamWave, and only recently have the language and development environments matured enough to make serious development effort possible. However, there are many obvious comparisons that can be drawn between TeamWave's components and Java applets, and the approaches have much in common.

For our particular needs, we can point to several advantages of our approach over Java. Despite most of the uses to date in trivial web toys, Java truly shines in larger applications, where its strong typing, rich data structures, and explicit structure are great aids to development. In TeamWave, we envisioned individual components to be fairly lightweight. Consequently, we argue the overhead of typing, structuring etc. is less necessary, and an unstructured scripting language like Tcl may be more appropriate. Our GroupKit experience has shown that users not trained in formal computer science or OO techniques can still develop applications in Tcl/Tk.

There are two weaknesses in Java that are currently being addressed, the widget set and object marshalling. The earliest versions of AWT are notoriously simplistic, and don't have

the rich widgets provided in Tk. This makes some applications more difficult to implement, as well as complicating groupware features like telepointers. Later versions of AWT, or third party interface libraries are sure to address these limitations. Secondly, when developing groupware in early versions of Java, developers had to explicitly do the data conversions and message parsing on sending and receiving remote procedure calls, which results in much extra overhead. Tcl's everything-is-a-string "feature" turns out to be surprisingly useful for avoiding this problem in groupware. Again, this will be ammeliorated somewhat now that Remote Method Invocation and serialization have become available in recent Java implementations.

Java-based groupware today runs as isolated applets. With the introduction of the technologies mentioned above, as well as the new JavaBeans component architecture, we would soon expect to see more tightly integrated groupware along the lines of TeamWave. It will be useful to compare the development effort and the strategies used for interacting between the components in those environments. We see Java not as a competitor, but as a less mature groupware platform that could learn from our own development experiences.

LESSONS LEARNED

We began this work by looking at how we could simplify the development of modestly sized interactive groupware components for the TeamWave Workplace environment. In implementing an alternative to generic component architectures, we were able to take advantage of specific characteristics in our application.

We found that letting tools run in their own execution contexts (in this case, Tcl interpreters) separate from other parts of the application was a very useful and effective strategy for minimizing the information the tool developers needed to be concerned with. We were careful to design the support infrastructure so that developers viewed the world as if they were running stand-alone as opposed to running as a component within a larger application.

The different interactions between system components in TeamWave provided us with many opportunities to reduce the complexity of developing individual tools, though often at the expense of increased complexity in the main application component. Again, for our application (and we expect many others), this is an appropriate tradeoff to make. Providing a range of well-defined and ad-hoc interfaces between components resulted in several benefits. As well, our use of a single widget hierarchy shared across the application provided a number of ways to reduce the responsibilities of the individual tools.

Given the relatively small size of the components we wanted to support, a scripting language like Tcl/Tk and our GroupKit API was an appropriate solution. Had we been looking to support larger components, a more structured language such as C++ or Java would likely be a better fit. Similarly, Tk's widget set provided good support for what we needed, and Tcl's string based model was useful for network communications.

Again, many of the design decisions were particular to our own application needs. As groupware applications and development environments grow more sophisticated in the future, we expect to see more groupware applications that collect together a number of components, rather than running stand-alone as is generally the case today. We expect that many of the strategies and component interactions we described here will help to inform the design and implementation of these new groupware systems.

Note: Information on TeamWave can be obtained from:
http://www.teamwave.com.
GroupKit information is available from:
http://www.cpsc.ucalgary.ca/projects/grouplab/groupkit.

REFERENCES

1. Arnold, K. and Gosling, J.. *The Java Programming Language.* Addison-Wesley. 1996.

2. Hill, R. The Abstraction-Link-View Paradigm: Using Constraints to Connect User Interfaces to Applications. *Proc. of CHI '92*. May, 1992.

3. Orfali, R., Harkey, D. and Edwards, J. *The Essential Distributed Objects Survival Guide.* John Wiley and Sons. 1996.

4. Ousterhout, J. Tcl and the Tk Toolkit. Addison-Wesley. 1994.

5. Roseman, M. and Greenberg, S. Building Real Time Groupware with GroupKit, a Groupware Toolkit. *ACM TOCHI.* March, 1996.

6. Roseman, M. and Greenberg, S. TeamRooms: Network Places for Collaboration. *Proc. of ACM CSCW '96.* October, 1996.