

Design of the DISCIPLER Synchronous Collaboration Framework

Weicong Wang, Bogdan Dorohonceanu, and Ivan Marsic
{weicong, dbogdan, marsic}@caip.rutgers.edu
Center for Advanced Information Processing (CAIP)
Rutgers — The State University of New Jersey
Piscataway, NJ 08854-8088

Abstract

This paper presents the DISCIPLER framework, a novel architecture for synchronous groupware, which enables sharing of software applications by conferees that participate in collaborative knowledge work. The particular class of applications dealt with is Java components (Applets and Beans). The central part of DISCIPLER is conceptualized as a collaboration bus. It spans network fabrics and provides a virtual interconnect for geographically distributed clients. Collaborating users import Java components by drag-and-drop manipulation into their virtual workspaces. The imported component becomes a part of a multi-user application and all conferees can collaboratively interact with it. The bus achieves synchronous collaboration through real-time event delivery, event ordering and concurrency control. In addition, importing various Beans also allows user tailoring of the human-computer interface. Interface customization can be achieved with multimodal human-machine interfaces and the collaboration components (such as group awareness widgets, concurrency controllers, etc.). The system has been implemented and tested on a variety of Java applications.

Keywords: CSCW, synchronous groupware, shared workspaces, JavaBeans.

1 Introduction

The rapid growth of the Internet and the Web has fostered great interest in interactivity between remote users and cooperative knowledge work. While chat rooms and multi-player games enjoy great popularity, classical productivity applications have not found their way to geographically distributed teams. Early commercial efforts, such as NetMeeting [1], provide very limited sharing of general applications and lack support for key groupware principles: concurrent work, relaxed WYSIWIS (What You See Is What I See), and group awareness. The few systems that do deal with general application sharing are available only as research tools in university laboratories. In addition, they require duplication of effort already expended on single-user applications when extending to multi-user domain, which often results in a failure to keep up with the latest features available in the single-user counterparts.

This paper presents the DISCIPLER (*D*istributed *S*ystem for Collaborative Information Processing and *L*earning) collaboration-enabling framework design for synchronous telecollaboration [2]. DISCIPLER is a *medium* rather than an application—it enables sharing of other applications. An important goal of this work is to enable easy sharing of third-party single-user applications, since the majority of applications continue to be developed for a single user. This is achieved by relieving, to the largest degree possible, the application programs from performing collaboration tasks, that is, those tasks that must be performed to allow more than one user to interact with these programs.

We have targeted a particular class of applications—Java components (Beans [3] and Applets). A component is a software module capable of publishing or registering its interfaces under certain contexts. Components like Java Beans support *persistence*, *visual manipulation*, *introspection*, *events* and *customization*. Java Applets provide a GUIs running under the applet context support. By integrating components into complex applications via assembly environments, such as DISCIPLER, one does more assembling and less constructing.

Since multiple users can simultaneously interact with their respective assembled Java components, the interactions need to be coordinated. DISCIPLER provides software components that manage synchronous group work, such as concurrency control of simultaneous activities, degree of sharing the application (coupling) and degree of awareness about the originators of the activities. The user-provided components and these system components are all first-class components.

This paper is organized as follows. We discuss the related work on existing collaborative frameworks in Section 2. The conceptual model of collaboration is reviewed in Section 3. We present our collaboration architecture in Section 4 and our approach for event management in Section 5. We continue with collaboration space visualization and with the collaboration artifacts and tools we provide (Section 6). Finally we conclude with the contributions of our project in its current state of development (Section 7).

2 Related Work

The model of collaboration used in DISCIPLÉ has certain similarities to the *locale* concept and its *Orbit* implementation [4]. *Orbit* also uses publish-subscribe communication implemented via Elvin toolkit [4]. *Orbit* focuses on visualizing the content of locales, with some information about the artifacts' states, but it is not concerned with the viewers and editors for interaction with the artifacts. Unlike this, DISCIPLÉ focuses on the application sharing, i.e., viewers for Java Beans that allow interaction with the Beans, distribute the Bean events, and provide customization of the user interfaces.

Corona group communication service [5] provides for a middleware communication layer consisting of a set of "common services" that can be used to support data dissemination and tools for collaborating using this data. The services are used in their *Builder's Environment* (CBE) toolkit for creating collaboration environments. *Corona* provides for reliable delivery with different levels of reliability and awareness information but relies on other systems for event ordering. CBE uses *rooms* to partition the shared workspace consisting of multiple applets and data sources represented by URLs. CBE relies on the Web browser in order to display the session manager, the rooms, and the applications (applets) in separate windows.

The *Habanero* framework [6] provides support for *sessions*, similar to rooms in CBE, which can contain URLs of data sources and collaboration-aware applications. A session requires predefining the set of applications to be shared and does not allow dynamic adding of applications. Unlike this, DISCIPLÉ allows for moving Beans or active applets among places and organizations.

The *Java Shared Data Toolkit* (JSDT) [7] defines a multipoint data delivery service for collaboration-aware Java applications. JSDT is data-centric in the sense that the applications share data elements. Unlike this, DISCIPLÉ is application-centric, and the users share Java components. JSDT provides a set of API's for the application programmers to design run-time collaborative applications. The resulting applications are tightly coupled with the toolkit and cannot be run without it.

JavaGroups [8] provides a toolkit for reliable group communication that can be used to build collaborative applications, but does not provide a collaboration framework like DISCIPLÉ. It implements an abstraction layer of communication middleware and adds reliable state replication.

Flexible JAMM [9] supports sharing of single-user Java applets in synchronous collaboration. It relies on a custom-modified version of JDK 1.1, which makes it non-portable. JAMM primarily targets *unanticipated sharing* or spontaneous collaboration where a user is able to initiate sharing at any time during the execution of the

application, not only before application is started. Due to its different collaboration model, the JAMM user interface does not represent the collaboration space. The workspace accepts only one applet at a time, but does offer a radar view and different types of telepointers.

A key principle in DISCIPLÉ design is to allow flexible composition. The user can visually compose the shared applications but also the user interface. Although some of the components of the design presented here exist in other systems, the run-time composition and simultaneous support for collaboration-aware and collaboration-transparent applications are, to our best knowledge, unique to the DISCIPLÉ framework.

3 Collaboration Model

The type of team collaboration targeted here is synchronous collaboration of geographically distributed teams, the so-called *same-time-different-place* collaboration. The users are sharing multiple applications and multimedia data. The system presented here focuses on sharing and assumes that group communication channels will be provided, using currently available audio/video conferencing tools.

From the user's viewpoint, the act of synchronous collaboration may emphasize either the meeting itself or the place where the meeting occurs. Human collaboration includes both meeting places with specialized resources to support the meetings, as well as spontaneous discussions that occur regardless of location and often do not need special meeting support from the environment. Places are persistent and may get more elaborate and complex with time.

A *place* roughly corresponds to the concept of *session*, where several users work on a topic. The notion of session, as often used in synchronous groupware, does not incorporate asynchronous events, such as disconnected users and failure recovery after a network shutdown. Places are persistent, meaning that users can suspend meetings and resume them later. This provides a sense of continuity and enables asynchronous work. The model is similar to the *room* model (e.g., in [10]) and to the *locale* model in [4]. A place, however, in our model is linked to the *topic* of collaboration rather than to the physical *location* of the place. Figure 1 shows a highly simplified conceptual model of collaboration. An appropriate theoretical model of collaborative work and workgroups is of critical importance and we mostly draw upon the above-cited references.

Although this model does not preclude spontaneous collaboration, it is more difficult to initiate a spontaneous meeting since the model is not tailored for such a type of collaboration. In fact, the concept of places conflicts with spontaneous collaboration since it requires the conferees to meet at a *specific place*. The place either already exists populated with resources or it has to be created and populated. The lack of spontaneity is in the sense that

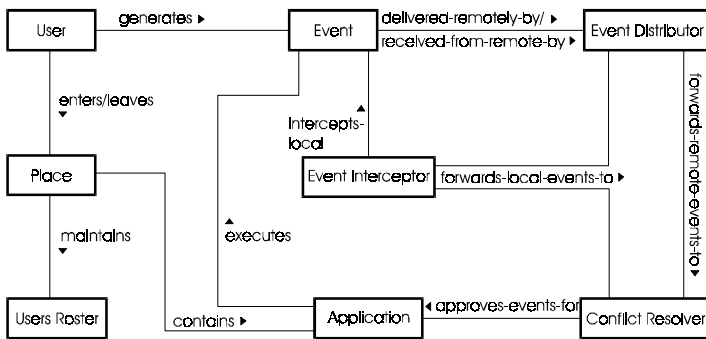


Figure 1: Conceptual model of synchronous telecollaboration. The arrows indicate the direction to read the association names, as in “User generates Event.”

users cannot start sharing an arbitrary application on the screen. Rather, the user needs to enter a place and use the applications therein, or initiate new ones. This kind of spontaneity is different from that in collaboration with a set of formal rules (workflow) vs. informal meetings. The model does, however, support spontaneity in the sense that users can enter and leave places dynamically without affecting the ongoing activities in the place.

One of the reasons for inadequate support for spontaneous meetings in DISCIPLÉ (as well as other Java-based approaches) is due to the way the Java Virtual Machine (JVM) is currently implemented. JVM runs as a user mode process and affects neither the window manager nor other applications. In contrast, collaboration systems based on shared window systems (e.g., [11]) get interposed between the user and the operating system—the entire user interaction goes through the window system. This provides for sharing the entire user’s desktop, and any application can become collaborative at any moment. An equivalent to a shared window system would be a shared JVM running on a Java Operating System. DISCIPLÉ is only an application running in a separate JVM process. Spontaneous collaboration is hindered by the fact that only the applications that run in one of the DISCIPLÉ workspaces can be shared.

The users collaborate by sharing artifacts, tools and resources. In DISCIPLÉ these are Java Components. DISCIPLÉ supports both the applications that conform to certain specifications, the so called *Collaboration-Aware Beans* (CAB), as well as those developed with no knowledge of the DISCIPLÉ framework called *Collaboration-Unaware Beans* (CUAB) which include any Java component available on the Web. The former know about their peers’ existence, while the latter operate as if used by a single-user. CABs can take advantage of the advanced features of the DISCIPLÉ framework, such as concurrency control and coupling.

4 System Architecture

DISCIPLÉ is a mixture of client/server and peer-to-peer architecture. It is based on a replicated architecture

for groupware. Each user runs a copy of the collaboration client, and each client contains a copy of the applications (Java components) that are to be collaborated upon. All copies of replicated applications are kept in synchrony and activities occurring on any one of them are reflected on the other copies. The coarse architecture of the DISCIPLÉ system corresponds to the three-tiered architecture:

- Presentation*—graphics user interface;
- Application Logic*—conceptual model of the system;
- Storage*—persistent storage mechanism.

The current system comprises two software applications: the *Virtual Desktop* that is run by each client and *Place Server* that runs on a server host and corresponds to “Storage” part. Design patterns [12] are widely used in order to design for reusability and to run parallel team development efforts. In particular, the Publisher-Subscriber Pattern is used to support indirect communication from lower to upper tiers. In order to support spontaneity at the communication level, we first introduce the concept of *communication channels*, which are crucial to dynamically changing and asynchronous collaboration.

4.1 Communication Channels

A communication channel designates a communication medium to exchange information. There are two roles in a channel: *publisher* (talker) posts messages to the channel and *subscriber* (listener) receives messages from the channel. The publishers and subscribers are autonomous—they arbitrarily join the group and do not need to know each other’s identity. A user can register to a channel as a talker, a listener, or both. All registered listeners simultaneously receive posted messages. Multiple channels might share one multicast address in which case they are multiplexed onto a logical channel. In this case, a physical multicast group with software de-multiplexing contains multiple logical channels. Due to multicast, the network traffic does not depend on the number of channel users.

The users can join and leave the channels dynamically, thus allowing the system to exhibit spontaneous conferencing characteristics. Neither registries nor naming service are needed for locating places and users.

In addition to publish-subscribe (multicast) channels, DISCIPLÉ provides peer-group (named) channels. The latter allows distributing events to selected user(s) rather than to everyone in the place.

Channels in DISCIPLÉ are implemented using the iBus object communication middleware [13]. We use reliable communication for all events. Notice, however, that certain events could be transported over an unreliable channel. An example of such events is mouse movement (between depress and release).

4.2 Consistency and Event Distribution Model

A key requirement in replicated architecture of real-time groupware is the maintenance of consistent state across all copies of the shared application. The user events in multi-user applications should not be processed before all participating applications can consistently process the event. Otherwise, the users will end up with different conditions of their applications and collaboration is rendered impossible. For example, imagine two users simultaneously drawing straight lines. Without consistency enforcement, the result would be a zigzag line instead of two straight lines. This is the role for the Conflict Resolver in Figure 1. The consistency issue is closely related to the event distribution model. The three most frequently used models are shown in Figure 2 [14]. In cases (b) and (c), events are ordered at a central point and thus appear to the application as if only one user interacts with it. Case (a) is more complicated and requires concurrency control algorithms.

DISCIPLE uses the model shown in Figure 2b, where one client assumes the role of serialization point. The main reason for choosing model (b) over centralized server (c) is to support mobile teams. In practice we need a directory to store system information (e.g., the roster of current users), but our solution is location-independent in the sense that the clients retrieve information without knowing the directory address. Thus, users can easily assemble an ad-hoc collaboration team and start collaborating without the need for a centralized server. To support this model, we create two types of communication channels: *poster* and *merger* channels.

Each client can have a role of *event poster* and/or *event merger*. By default, every client assumes the poster role and only one client per place assumes the merger role. The first client in the place automatically assumes the merger role¹. At certain instances, such as failures, other clients negotiate out the merger role. The event merger receives the events occurring at the event posters, manipulates them and distributes them in the same order to every client in the place. The manipulations include: event reordering; event concurrency control; discarding the events when users prevent unwanted remote events from executing.

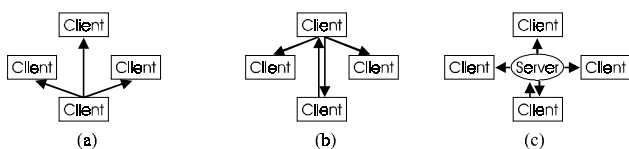


Figure 2: Common approaches to event distribution: (a) each client distributes its own events; (b) one client acts as serialization point that distributes events for all other clients; (c) server acts as serialization point [14].

¹ A more intelligent procedure could include load-balancing techniques. For example, the merge role may always be assigned to the client with the highest computing power.

Accordingly, each place has *poster* and *merger channels*. They are used to exchange the messages regarding the user interaction inside the Place. Each client in a place registers as a talker to the poster channel and as a listener to the merger channel. The merger picks up the messages on the poster channel, manipulates them and posts the manipulated messages to the merger channel. Since all clients listen to the merger channel, they will get the manipulated messages and process them.

An example of event traffic is given in Figure 3, where there are two users in a place. User A assumes the merger role. As any user posts messages ①, the merger receives the messages ②, and after manipulation posts them onto the merger channel ③. All the users in the place, including the merger, receive the messages ④.

The Place Server also registers as a listener to the merger channel of every place for monitoring purposes. It monitors whether a place has been shutdown, changes in currently active users, users' or places' crashes, etc. Additionally, by listening to the merger channel, the Place Server can archive the Place activity for a future retrieval and replay.

The approach with the centralized event merger is called *virtual synchronization* [10]. An alternative design is to let all clients assume the merger role. That is, every client multicasts its own events (Figure 2a). Although this yields faster response, concurrency control has to be implemented at every user site. It is also difficult to maintain consistency across such concurrency controllers. Thus a special protocol has to enforce the consistency, which increases network traffic.

There are two major advantages to the current design. One is simplicity. Since there is a single point of event merging, we can simply use the event queue to synchronize the events without affecting the client sides. Another major advantage is that events are distributed with no information about their origin. Rather than directly dispatching the local events, they are first sent to the merger, manipulated, and distributed to every client in the place, including the client who originated the event.

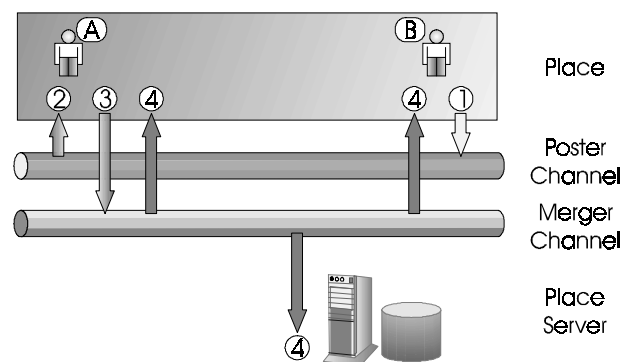


Figure 3: Example of event traffic over the poster and merger channels. Arrows from objects to channels refer to talking whereas arrows from channels to objects refer to listening.

4.3 Collaboration Bus Design

The collaboration bus implements the conceptual model shown in Figure 1 and corresponds to the application logic tier of the three-tiered architecture. The collaboration bus can be implemented on top of many communication middlewares such as OMG CORBA. But unlike an object communication bus, it has certain features specific for real-time application sharing.

As can be seen in Figure 1, the collaboration bus essentially deals with distributing the user-generated events to maintain a consistent state across all collaborators. There are two basic types of events generated by the user: (i) interaction with the other users and (ii) interaction with the Java components. An example of the former is when a user announces his/her presence to all other users in the place. An example of the latter is when the user draws a figure using a drawing editor Java component.

Software design for the collaboration bus is presented in Figure 4. The top row represents the user interface. Notice that shared Java components are indicated to belong to the presentation, rather than application logic tier of the DISCIPLER system, although the system provides some application logic related components. This reflects the fact that users share the view of the applications, and the system makes no distinction between the application's view and its logic.

We use the technical term *node* to refer to an organization. Each node has also one *announcer channel* (Figure 4) used by all users in the node to communicate the node-level information. The node users register as publishers and subscribers to this channel. The messages posted by the node users are mainly about management requests, such as a node image request. The *node image* is necessary for the clients to learn about each other. It contains minimum information about the current state of the node, such as the active nodes (which contain some active Places or users), node user profiles, and active places. This information is also used to avoid user name conflicts, as each client must have a unique user name within the given node. At runtime, the node image is dynamically updated and distributed to the clients.

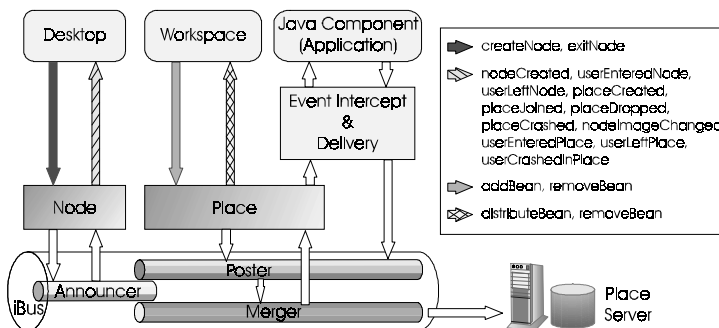


Figure 4: DISCIPLER collaboration bus design. Filled arrows represent direct calls, whereas patterned arrows represent publisher-subscriber upcalls.

The Place Server (Section 4.4) picks up the messages on the announcer channel, interprets them and replies back with administrative messages. The administrative messages carry information about the current status of the system, e.g., a user entering or leaving a Place, or administrative actions, e.g., forcing the shutdown of a Place.

4.4 Place Server

The Place Server in essence maintains the “phone book” of all collaborators and historical record of their collaborative meetings. It is a desirable but not essential component of collaboration in DISCIPLER.

But, a newcomer needs to get the current status of the peers, at least the status of its own organization. A solution is similar to diskless clients. The Place Server keeps track of the node image, which contains the roster of the users that are currently active in the node and the places the users have entered.

The Place Server runs permanently and maintains the information about collaborative activities. It monitors the nodes by listening to all channels and archives the Place activities. At present, the Server functions are:

1. Provides the current node image to bootstrap new/late-comers so they can engage in meetings. At the time of client bootstrap, the client requests the node image from the Place Server. The request is pull-based and thus autonomous.
2. Monitors the node membership. In real world, users dynamically join and leave the meetings, while some users may crash. There are two kinds of membership—a user can belong to a node or to a place. The membership is established by connecting to the node or place channels. In the current implementation, a user can enter only one node at a time, but a user can enter any number of places across different nodes at a time. The server notifies existing users about newcomers or newly created places.
3. Provides failure notification. There are many failure points. For example, the user may shutdown the client, a part of the network may be down, or the user shuts down some Places. The server needs to monitor such client crashes and notify the other users. The monitoring is accomplished by using heartbeat messages.

The Place Server under the current design can be easily extended with the following features:

- Provide node administration—the administrator can force malicious users to leave a place or the node. Also, the Access Control List (ACL) could regulate the users' access rights.
- Link nodes to create a hierarchical tree to exchange the node information. Thus, the users from different nodes could create or join intra-node places.
- Provide place archiving and virtual synchronization.

4.5 Class Loader and Resource Server

DISCIPLE loads multiple Java archive (JAR) files dynamically at run time. There are certain restrictions in the JVM concerning the ClassLoader in that it can only access the classes loaded by the parent class loader(s) and not vice versa. Classes loaded by different ClassLoaders could not access each other. To work around these restrictions, we load the system and applications (Java components) via one ClassLoader.

Resource servers provide the resources such as images, audio, and JAR files of the beans to be used in collaboration. A resource server could be a regular Web server (e.g., the HTTP or FTP servers). The users only need the URLs and the loader loads the requested resource. DISCIPLE clients could also play the role of resource servers. For example, the place P_i has two users A and B and the bean BA loaded in. Only user A has a local image file I_A and the bean BA on A 's desktop loads the image. The system needs to provide for user B to also access the I_A . There are two solutions. First is embedding the I_A in an event object using object serialization and transmitting it over. But then the bean needs to understand the semantics of the event. As a result, this approach only applies to the CABs.

Second solution is for client A to become a resource server. Such resource server can be implemented by embedding a simple HTTP server. The server transfers the localized resources such as class bytecode, images, etc. to the requesters. In the above example, when the user initiates the image loading, the system will distribute to the peers the event containing only the URL of I_A . Another major advantage here is that the system can easily solve the problem of versioning and become an extremely thin client. The whole DISCIPLE client becomes a resource (JAR file). The user then initially only needs the system class loader. With the new released system, the users simply load the system from that server. Loading the bean from one location to all place users simplifies the system administration, since the system is certain that they are using the same version. At present, we use this approach only to load application beans. Suppose user A loads the bean archive BA.jar located one resource server such as a Web server. Other users receive the event containing the URL pointing to the JAR and their loaders automatically acquire it.

5 Event Management

Collaboration in the replicated type of groupware architecture essentially translates into intercepting the state changes occurring in a user's Java components and replicating the state changes in all the peer users' Java components.

We can consider event interception from a *top-down* or *bottom-up* approach. The bottom-up approach adds event listeners to each component of the bean to be shared. The listeners are notified once the events are fired

on the components. In this way, we can capture both low-level and semantic events. The top-down approach puts a transparent GUI component at the highest Z-order. It intercepts all input events occurring in the area that the component covers.

As already pointed out, there are two types of beans in the system, CABs and CUABs. Comparing the two approaches, we find that the top-down approach is more suitable for the low-level events while the bottom-up approach better intercepts the semantic events. The bottom-up approach in CUABs cannot intercept the user events *before* they get executed locally and thus cannot provide any type of concurrency control. Conversely, since CABs are aware of peers and voluntarily subject to event ordering, bottom-up solution is simpler.

5.1 Event Types

There are four main types of events used for communication, as indicated in the legend of Figure 4: DiscipleEvent at node level, PlaceEvent at place level, BeanEvent for CABs, and BeanAWTEvent for CUABs, all of which are derived from java.util.EventObject, whose event source is null. Each type has a corresponding event listener.

The DiscipleEvent events deal with places (create/join/drop/crash), node image request/reply and main user events (enter/leave/crash in a node/place). The announcer channel distributes them. The PlaceEvent events carry updates regarding bean manipulation (add/distribute/remove/resize bean). The place channels distribute these events to the remote peers.

5.2 Collaboration-Unaware Beans

To intercept all the events generated by the user inputs (mouse, keyboard, input focus events), we use a transparent component, called GlassPane, which is available in the Java Swing toolkit. It is at the topmost Z-order to cover the bean's GUI area. This topmost component intercepts all the user's events, but does not occlude the underlying Java component. The GlassPane could also be dynamically shrunk or expanded, thus allowing easier management of public and private areas of the same workspace. An additional benefit is the accompanying visual effects (e.g., mouse click causes depressed button) on the remote sites as well.

We chose to exclusively use the Swing toolkit due to the problems with heavyweight Java AWT toolkit. In a lightweight GUI framework, such as Java Swing, the lightweight components directly extend the java.awt.Component and java.awt.Container classes. Thus they do not have native opaque windows associated with them, and require no native data structures or peer classes. This leads to a complete consistency across different platforms.

To dispatch the intercepted events to the underlying component we do the following:

1. For mouse events, we need to locate the deepest component in the underlying layer whose bounding box contains the mouse position and set the event's source attribute value to that component.
2. For keyboard events, we need to remember who actually has the input focus, rather than the component over which the event happened.

In order to avoid all users interact with the dialog boxes in CUABs, we also modify the Java component's bytecode and replace certain classes with DISCIPLE-specific code. One example is replacing the Swing's file chooser dialog with our own version.

5.3 Collaboration-Aware Beans

For CABs, it is necessary that the bean comply with design guidelines prescribed by the DISCIPLE framework. Essentially, the bean should know about the existence of remote peers. It should be able to receive peer events besides sending them. Since beans are loaded dynamically, the system needs to construct bridges between different beans in the same JVM as well as between the beans and the communication middleware. Such bridges are automatically generated and loaded at run time by the framework. The information necessary to generate the bytecode is obtained by examining the bean properties using the introspection mechanism of the Java language. The bridge code is following a predefined generic skeleton (which for example contains methods to relay the events or properties) and is generated by a generic bridge generator module.

Event interception uses the bottom-up approach. For example, by dynamically configuring and de-configuring the listeners, the system enables connecting multiple beans. An event in one bean triggers an event in another bridged bean.

One important aspect is that aware beans could be treated just as unaware beans without any modification since they do not have direct relationship to the DISCIPLE code.

6 User Interface and Desktop

A key function of the user interface is to effectively visualize the collaboration space. The *collaboration space* is essentially a "phone book" of all people that a user can collaborate with. It is structured to reflect various people groupings as in everyday life. It may be visualized in different ways, for example, using an abstract model or using a physical model. The simplest abstract representation shows a plain list of collaborative places. A more complex representation structures the places into a tree or a graph, where the nodes at higher hierarchical layers correspond to buildings, cities, etc. Even more complex representation positions similar places proximally according to certain distance measure. On the other hand, in a physical representation the space is represented as a 3D virtual world, where the user walks

through streets and corridors to reach a collaboration place. Since the places in our model are characterized by the topic of collaboration, rather than by their physical location, an abstract representation is more appropriate than a physical one. Our approach to collaboration space visualization is presented in [15]. This section presents a brief overview.

6.1 Hierarchical Space Visualization

The main window of the client part of the DISCIPLE system is shown in Figure 5. It displays the hierarchical organization of the collaboration space structured according to the available collaboration servers and the current groupings of all collaborators. The hierarchy is as follows: place servers > organizations > (meeting) places > users and is represented with a tree, shown on the left side of Figure 5 (similar to a file system representation). In Figure 5 one can see the communication node (*disciple*) with one organization (*CAIP*) that contains two meeting places (*Military* and *Medical*). The users that participate in meetings (*boi*, *dbogdan*, *latha*, and *juth*) are shown in places, and there is one user (*francu*) that has connected but presently is not participating in any meeting. The current snapshot shows a description of or information about the selected item (user *boi*). The description is provided for only one item at a time. Using the main window users can navigate the collaboration space, create new meeting places, or enter existing ones.

In the right window of Figure 5 there is a stack of pages representing different utilities available to the user. Besides the information item currently selected, there are also other common tools such as chat window, video/audio conferencing startup, e-mail, query window for finding users, etc.



Figure 5: Client GUI for the hierarchy of collaboration places and users.

6.2 Collaboration Workspaces

A workspace is a client visualization of a meeting place and provides an individual view of the place (Figure 4). The workspace window gets automatically launched as the user enters a place. The window shows the artifacts that are currently in the place and their relationships (Figure 6). Sharing a Bean in collaboration is as easy as pointing to its URL. A Bean can be loaded from a local file system or, given the Bean's URL, it can be loaded from a Web server. The toolbar on top provides

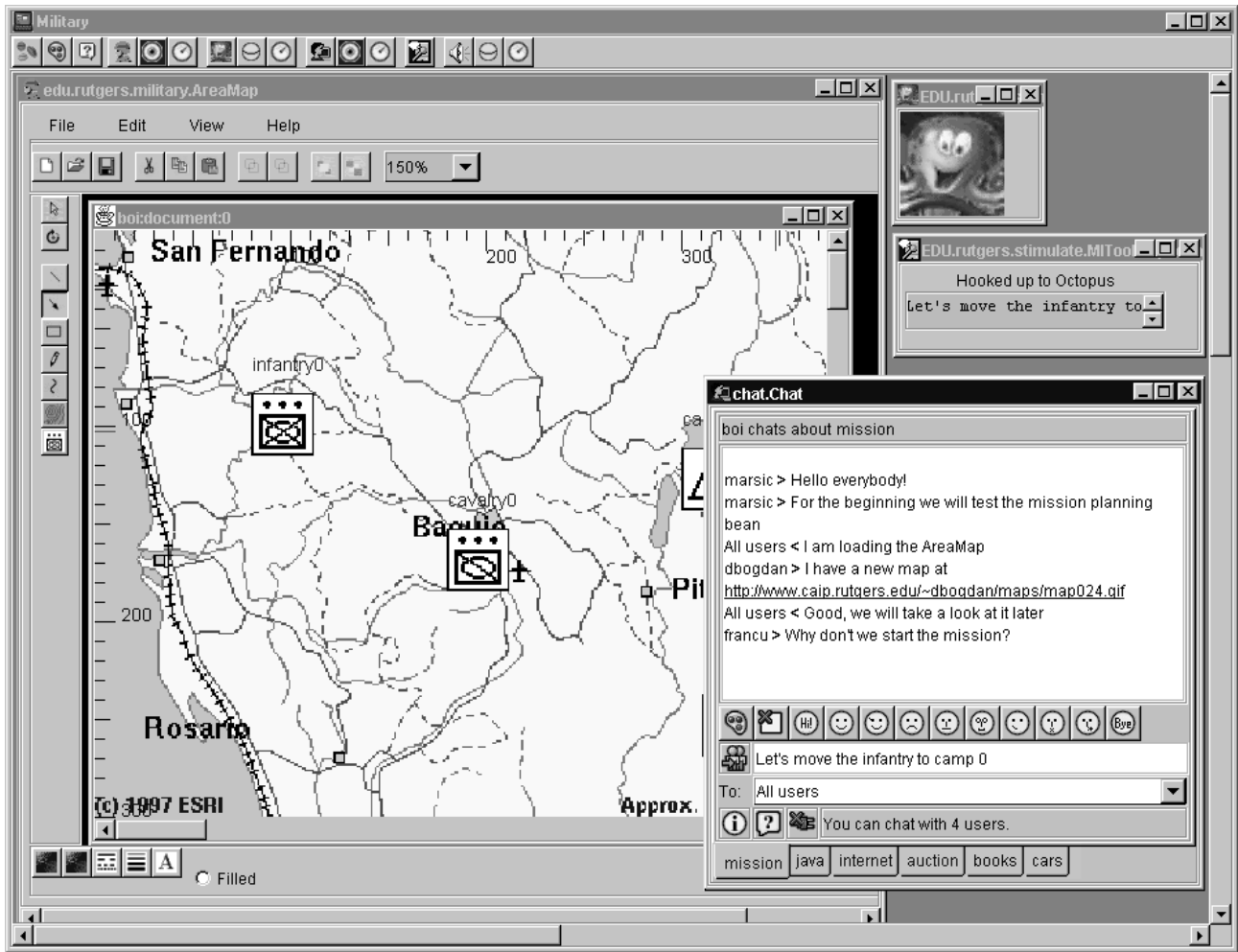


Figure 6: Snapshot of a user view of a collaboration place.

for opening the Beans browser, changing the telepointer color and getting help, respectively. In addition, the toolbar has three buttons per each bean in the workspace: show minimized/hidden, activate/deactivate telepointers, and activate/deactivate radar view.

Workspaces provide a single seamless UI environment with high level of consistency. As is the case with the overall desktop view, the Workspace also does not require strictly the same view across all clients, in the sense that each conferee can position the Beans at different locations within the Workspace. However, Beans themselves are presently limited to consistent views and actions for all the conferees, because it is not possible to uncover the entire semantics of the collaboration-transparent Beans. The telepointers and radar views are provided at the bean level because users are free to adjust the size and position of each bean in the workspace.

6.3 Collaboration Components

The idea of collaboration components is that, even in specialized collaborative applications, a common set of

functions exists that can be extracted and implemented as self-contained software components. These features include such things as concurrency control, group awareness, degree-of-coupling control, etc.

It would be ideal to enable an application developer to focus on the application itself and not the above. By providing a set of reusable software components that encapsulate such multi-user features, the collaboration components allow the user to easily configure the shared workspace [16].

DISCIPLINE currently provides awareness through telepointers and radar views that are included in the set of Beans loaded by default at start-up. A *telepointer* represents the position of a remote user's mouse cursor, providing location awareness. It has unique color and a name, and both identify the user. A *radar view* displays a miniaturized image of the workspace with highlighted regions that the participants are currently viewing. Collaboration components also include several concurrency control algorithms (non-optimistic locking, optimistic locking, and undo/redo) [16].

6.4 Interface Customization

Componentization of the collaboration framework introduces the possibility for interface customization. Figure 6 shows an example where the workspace is augmented by multimodal human/machine interaction Beans. By loading and activating different multi-modal Beans, the user can dynamically choose the modality (e.g., speech, keyboard, eye gaze pointer, etc.) for interacting with the workspace. Similarly, by loading different collaboration component Beans, the user can vary the degree of awareness about the other conference participants or select the concurrency control algorithm that applies to a particular Bean.

The users can also edit their profile (picture, personal information, URL address of the Web page, etc.) and the properties of the awareness widgets representing the remote peers (such as telepointers and radar views) at any time during collaboration

7 Conclusions

This paper presents a design and implementation of a framework for synchronous collaboration that allows interaction with shared Java components and provides mechanisms to control its cooperative features in an application-independent manner. Its interface is open and customizable with respect to the context in which it is placed and particular user needs. Other unique features of the DISCIPLE framework are simultaneous support for collaboration-aware and collaboration-transparent applications and ability to support *ad hoc* collaboration groups.

The DISCIPLE framework developed to date has been implemented and tested on both third party CUABs and our CABs. The applications include whiteboard, collaborative mapping, speech signal acquisition and processing, and image analysis tools. For updated information or software download, check the following web site:

<http://www.caip.rutgers.edu/disciple>

7.1 Acknowledgments

We wish to thank to Cristian Francu, Stephen Juth, and Wen Li who have contributed to the implementation of DISCIPLE. The research reported here is supported by DARPA Contract No. N66001-96-C-8510, NSF KDI Contract No. IIS-98-72995 and by the Rutgers Center for Advanced Information Processing (CAIP).

8 References

1. NetMeeting 2.1 Resource, Microsoft Corporation. <http://www.microsoft.com/netmeeting>.
2. I. Marsic, DISCIPLE: A framework for multimodal collaboration in heterogeneous environments. To appear in *ACM Computing Surveys*, 1999.

3. JavaBeans API Specification, Sun Microsystems, Inc., <http://www.javasoft.com/beans>.
4. T. Mansfield, S. Kaplan, G. Fitzpatrick, T. Phelps, M. Fitzpatrick and R. Taylor, Toward locales: supporting collaboration with Orbit, To appear in *Journal of Information and Software Technology*, 1999.
5. R.W. Hall, A. Mathur, F. Jahanian, A. Prakash, and C. Rasmussen, Corona: A communication service for scalable, reliable group collaboration systems, *Proceedings of ACM 1996 Conference on Computer-Supported Cooperative Work (CSCW'96)*, 1996, 140-149.
6. Chabert, E. Grossman, L. Jackson, S. Pietrowicz, C. Seguin, Java object-sharing in Habanero, *Communications of the ACM*, 41(6), 1998, 69-76.
7. R. Burrige, Java Shared Data Toolkit (JSdT), Sun Microsystems, <http://www.javasoft.com/people/richb/jsdt>.
8. B. Ban, Design and implementation of a reliable group communication toolkit for Java. <http://www.cs.cornell.edu/home/bba/javagroups.html>
9. J. B. Begole, M. B. Rosson, and C.A. Shaffer, Supporting worker independence in collaboration transparency. *Proc. 1998 ACM Symposium on User Interface Software and Technology (UIST'98)*, 1998, 133-142.
10. M. Roseman and S. Greenberg, TeamRooms: network places for collaboration, *Proceedings of ACM 1996 Conference on Computer-Supported Cooperative Work (CSCW'96)*, 1996, 325-333.
11. C. Lauwers, Collaboration transparency in desktop teleconferencing environments, *Ph.D. Thesis, Technical Report CSL-TR-90-435*, (Stanford, CA: Computer Systems Laboratory, Stanford University, 1990).
12. F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern-Oriented Software Architecture: A System of Patterns* (New York: John Wiley & Sons, Inc., 1996).
13. iBus Programmer's Manual, SoftWired AG, <http://www.softwired-inc.com>
14. J. F. Patterson, M. Day, and J. Kucan, Notification servers for synchronous groupware, *Proceedings of ACM 1996 Conference on Computer-Supported Cooperative Work (CSCW'96)*, 1996, 122-129.
15. B. Dorohonceanu and I. Marsic, A desktop design for synchronous collaboration. *Proceedings of Graphics Interface '99 (GI'99)*, 1999, 27-35.
16. S. Juth, Collaboration Components for programming real-time synchronous groupware applications. *Master's Thesis* (Piscataway, NJ: ECE Department and the CAIP Center, Rutgers University, 1998).