An Application Framework for Synchronous Collaboration using Java Beans

Ivan Marsic  
Department of Electrical and Computer Engineering and the CAIP Center  
Rutgers University, Piscataway, NJ 08854-8088  
http://www.caip.rutgers.edu/multimedia/groupware/

Bogdan Dorohonceanu  
Department of Computer Science and the CAIP Center  
Rutgers University, Piscataway, NJ 08854-8088

Abstract

This paper presents a JavaBeans framework that supports real-time synchronous collaborative applications. We contribute a generic collaboration bus as an enabling virtual “channel” that spans network fabrics and integrates collaborating clients. The bus provides a component-based plug-and-play environment that enables collaboration with applications that may or may not be collaboration aware. Any (including single-user) applications can be plugged in as is and made collaborative with no modifications to the application or to the collaboration bus. One of the activities supported by the framework is multi-user visual programming using JavaBeans: the users at geographically separate locations can collaboratively compose complex applications using component Beans. The framework has been implemented and tested on a variety of applications.

1. Introduction

This work presents a collaboration-enabling framework that simplifies the development of multi-user collaborative applications, in particular real-time synchronous groupware. Development of single-user applications is generally an expensive and challenging effort. Development of multi-user applications introduces many more degrees of complexity. The problem is well recognized and there is a large body of related work, for example [3,5,11,18-20]. But previous work has generally been constrained to monolithic systems that do not easily generalize to different applications, or to general toolkits for building shared applications. Most of the toolkits either do not support collaboration-transparent single-user applications or require porting process. Once an application is developed using a given framework, the application classes have some relationship to the framework classes, e.g., inheritance, thus effectively locking the application to the framework. Instead of being able to leverage existing single user applications, developers end up duplicating the effort expended on those applications. As a result, multiuser applications lag behind in features or compatibility with single-user applications.

The approach taken here is to dissociate to the maximum the communication and group aspects from the application task. This reduces the complexity of shared applications developing almost to the level of that for single-user applications. It also enables rapid porting of single-user applications to the multi-user domain. To develop such a framework without any constraints would be a daunting if not impossible task. We have thus constrained our work to a particular class of applications—applications known as JavaBeans. There is currently a strong software industry trend towards standardizing software development through reusable software components. Two major component architectures are JavaBeans from Javasoft [20] and ActiveX controls from Microsoft [7]. Components enable rapid development using third party software: independent components are used without modifications as building blocks to form composite applications.

The framework is conceptualized as a collaboration bus (Figure 1). The user plugs the applets into the bus and thus makes them shared with the other users that participate in a collaborative session. An applet is a small application program that provides a graphics user interface and accomplishes the task that the user is interested in, e.g., extracting image features or computing and visualizing a spreadsheet table. A user would typically use multiple applets, independently or linked in a more complex applet, to collaborate with other users. Since multiple users can simultaneously interact with their respective applications, the interactions need to be coordinated. The bus therefore contains the software components that manage synchronous group work, e.g.,

---

1 A framework is a reusable, “semi-complete” application that can be tailored to produce custom applications [8].
Concurrency control of simultaneous activities, degree of sharing the application (coupling), and degree of awareness about the originators of the activities.

The requirements that we have set for a collaboration-enabling framework are as follows:

R1 simplification of the development of multi-user applications by delegating the group and communication aspects to the collaboration-enabling framework

R2 ability to work with collaboration-transparent applications without modifications to the application code

R3 intact preservation of the underlying Java platform (graphics user interface (GUI) toolkit and other system packages)

R4 no requirement for explicit relationships to the framework classes for collaboration-aware applications, thus allowing the applications to work independently of the framework

R5 flexibility to employ complex cooperative algorithms, such as multiple concurrency control algorithms, varying degrees of coupling and sharing, group awareness features, etc.

An additional desirable feature is support for design-time collaboration, where users can at run-time customize the application or compose more complex applications from simpler ones. All of the above should be accomplished with a minimum impact on performance, since the framework is intended for real-time groupware.

We present a collaboration-enabling framework that satisfies all of the above requirements. It offers a single solution to what is currently viewed as two disparate problems in developing multi-user applications: the development of special purpose applications that are “collaboration aware,” and the adaptation of existing single-user applications to provide collaboration-transparent shared applications. The framework has been implemented and tested on a variety of JavaBeans applications. This work is part of the larger effort on the DISCIPLE system (DIstributed System for Collaborative Information Processing and Learning) [15].

The paper is organized as follows. We first give a brief overview of the key features of JavaBeans that form the basis for the framework. Section 2 defines the architecture for the collaboration bus—the main component of the framework. Section 3 presents the implementation of the collaboration bus and the entire framework. Section 4 deals with the issues that arise due to the flexibility of the framework. Finally, Section 5 compares the presented framework to other frameworks that support synchronous collaboration and Section 6 concludes the paper.

### 1.1. Java Beans

The collaboration-enabling framework is based on the JavaBeans component model, which is a part of the Java Development Kit version 1.1 or higher [20]. A Bean is a reusable platform-neutral software component that can be visually manipulated in a software development tool. It has a well-defined input and output interfaces, specified by the events that the Bean accepts or generates. An event is something of importance that happens at a specific point in time. An event can take place due to a user action such as a mouse click—when the user clicks a mouse button, an event takes place. Events can also be initiated by other means, e.g., input from a monitoring instrument. The delegation-based event model was introduced with JavaBeans. In this model, there is no central dispatcher of events; every component that generates events dispatches its own events as they occur (see Figure 2).

**EventSource** Any object can declare itself as a source of certain types of events. An event source has to either follow standard design patterns when giving names to the methods or use the BeanInfo class to declare itself as a source of certain Events [20]. The source should provide methods to register and remove listeners of the declared events.

**EventListener** An object can register itself as a listener of a certain type of events originating from an event source. A listener object should implement the generic java.util.EventListerner interface.

According to the delegation model, whenever an event for which an object declared itself as a source gets generated, the event is multicast to all the registered event listeners.
LISTENERS. The source object propagates the events to the
listeners by invoking a method on the listeners and
passing the corresponding event object. Delegation event
model is in the essence of the JavaBeans specification and
it is precisely the feature that makes the collaboration
framework possible.

2. Collaboration bus

The collaboration framework is based on a replicated
architecture for groupware [12]. Each user runs a copy of
collaboration client, and each client contains a copy of the
applet that is to be collaborated on. For each object
within the applet, there will be a counterpart in all the
other users’ applets. Collaboration in this type of
architecture essentially translates into intercepting the
state changes occurring in a user’s applet and replicating
the state changes in all the peer users’ applets.

The purpose of the collaboration bus (cBus) is to
provide a virtual interconnect for network distributed applications that
participate in a collaborative session. It replicates (in an
object-oriented way) the state changes to maintain consistent views and data
across the shared workspaces as well as signaling messages to handle special situations or exchange control
information. The overall architecture of the collaboration bus resembles the classic Broker pattern
[4], but differs in many details. A general object request broker would not be sufficient since the
collaboration-specific tasks would need to be handled by the application—the

situation we would like to avoid. Here we define the
requirements on the collaboration bus by considering the
process of event replication.

2.1. Event replication

The overall process of network distributed event replication follows the Java delegation event model. Figure 3 illustrates the process. From the viewpoint of the awareness about the existence of the bus, the applications can be classified as collaboration-transparent (unaware) and collaboration-aware. Event replication works in the same way for both.

**Event adapters** convert events of an arbitrary type generated by the application into unified events that can be processed by the cBus. Adapters are needed since the bus cannot have methods for arbitrary events that an application programmer may come up with. Event adapters are equivalent to object proxies (stubs, skeletons), but there are differences—a major being that the event adapters need to be registered as listeners of events generated by an application so that the bus gets notified about the application’s state changes. Event adapters for collaboration-transparent applications are predefined since the events are known in advance (i.e., Java AWT events). For aware applications (in which the developer can define their own events with specific semantics), the developer needs to generate the adapters using the IDL-like compiler supplied with the cBus.

**Communicator** is the layer in the cBus whose main purpose is to provide demultiplexing of requests (multithreading), as well as cooperative features, such as

---

**Figure 3. Event source (e.g., Java AWT Button) with event listeners.**

The source object propagates the events to the listeners by invoking a method on the listeners and passing the corresponding event object. Delegation event model is in the essence of the JavaBeans specification and it is precisely the feature that makes the collaboration framework possible.

**Figure 2. Event interception and replication in DISCIPLE.** Local user events get intercepted by the collaboration bus which multicasts them to the remote peers. The architecture of the collaboration bus follows the Broker design pattern [4].
concurrency, coupling, etc. Communicator multicasts the application's state changes to the remote peers. It also receives state change notifications from the remote peers and dispatches them to the local application.

A concurrency controller processes the events before they are delivered to the application components and does conflict resolution when multiple users simultaneously interact with the shared workspace. Each proxy (EventAdapter) may have its own concurrency controller associated with it, meaning that each component in an application may use different concurrency control algorithms. However, it is enforced that all copies of the same component in remote applications use the same concurrency control algorithm.

Protocol can be any multicast protocol that, however, needs to be reliable, rather than a plain datagram-based multicast. All remote calls are one way and no reply is expected. Pseudo-multicast can also be used, in which case the underlying protocol can be CORBA IIOP [22] or Java RMI [23].

2.2. cBus and object request brokers

cBus resembles an ordinary object request broker, such as CORBA [22], Java RMI [23], or DCOM [10]. However, it differs in that it is customized for real-time collaborative applications. The following features distinguish the cBus:

- simultaneous support for multiple communication modes (multicast for event replication and unicast for signaling)
- definition of remote object references (see below)
- semantics and instantiation of proxies for remote objects (see below)
- event echoing prevention
- concurrency control for conflict resolution of simultaneous state changes

The semantics and instantiation of proxies for remote components in the cBus are the result of the need to avoid modifications to collaboration transparent applications. Unlike traditional object request brokers, cBus automatically creates co-located both stub and skeleton proxies. In CORBA, stubs are created (automatically) based on the receipt of a remote object reference as part of the reply to a remote call [22]. Unlike this, event adapters are created based on the information on what events the local application can generate. The corresponding event adapter is instantiated for all events of each of the application's components. It is assumed (but not verified) that the remote peer applications have the counterpart components that will accept the multicast events.

3. Collaboration-enabling framework

Collaboration bus functionality as presented in the previous section can be implemented as a single module which subsumes both object communication and group aspects. It can also consist of two layered modules, where one layer deals with the object communication aspect of the cBus whereas the other deals with the group aspects and event adapters. The latter solution provides greater flexibility, since the first layer can use a general object communication environment, in particular CORBA IIOP [22]. However, IIOP introduces overhead, cannot be implemented with true multicast, and is not needed in a closed system such as the one where all collaborating applications use the cBus for communication. The former case provides optimized performance, but the general communication protocols (e.g., IIOP, RMI) cannot be easily plugged-in. We chose this since performance is of great importance in real-time synchronous groupware.

The rest of the collaboration-enabling framework automatically interconnects the cBus and the applets by doing the following:

1. determines the constituent components of the applet and the events each component generates
2. creates corresponding EventAdapters and register them as listeners to the components

3.1. Collaboration-enabling framework components

The main components of the collaboration-enabling framework are shown in Figure 4.

Workspace is the central part of the Collaboration-Enabling Framework. It can contain JavaBeans-compliant applets and applications that are to be shared. Workspace can be public or private. A public Workspace corresponds to a collaborative session and each session participant obtains a copy of the Workspace upon joining to a session. The Workspace works in a relaxed WYSIWIS (What You See Is What I See) mode: each conferencee can position the Beans at different locations in the Workspace. A Bean can be displayed in a Workspace window or it can launch an independent window.

Collaboration Bus multicasts the state changes taking place in an application contained within the Workspace to the remote Workspaces. The bus also receives events from remote Workspaces and dispatches them to the local application. Each Workspace has its own cBus (see

---

4 cBus has no similarity to InfoBus [6]. InfoBus-connected applets must all reside in the same Java Virtual Machine, although InfoBus can participate in distributed applications. In such a case, the application must use some other communication system, such as CORBA, RMI, etc. Unlike InfoBus, cBus doesn't address data interoperability since it replicates data between the copies of the same application, i.e., in a homogeneous environment. There are many dissimilarities, but the basic one is that InfoBus is not an Object Request Broker, neither it has any special features to support synchronous collaboration.
Figure 4), similar to how the traditional object request brokers (ORBs) operate: each application has its own ORB. We could have had only one cBus per application Bean, but this would introduce additional complexity to the cBus. On the other hand, bus-per-workspace implies that every time an application Bean needs to join a different session, a new Workspace needs to be created.

**JavaBean Application** The application refers to a third party single-user applet that is to be made collaborative. To become collaborative, the applet should be developed as a JavaBean in accordance with JavaBeans specifications [19]. The framework supports both collaboration-aware and transparent Beans.

**SessionManager** keeps track of the ongoing conferencing sessions and all the Workspaces involved in each session. It keeps a record of all user actions and generates a session history. When a client joins a session, the SessionManager downloads all Beans in the session to the client and updates their state, thus providing support for latecomers. The SessionManager may also participate in concurrency control policies of the collaborative system, e.g., granting the token in a locking-based concurrency control scheme. Each public Workspace is required to register with this object.

The current version of the bus and the shared workspace is shown in Figure 5. The group awareness is supported through telepointers (display multiple cursors for the session participants), radar views (show zoomed-out the entire workspace, while each user may be working on a different part of the workspace), and filtered user actions.

### 3.2. Collaboration-aware applications

A collaboration-aware application is an application developed with the intention of being used with the collaboration-enabling framework. Such an application is provided with fine-grained, less restrictive mechanisms that mediate the collaborative session. The benefits of developing this kind of applications include:

- Grouping of several related low-level events into a single high-level event. For example, in a given context the sequence of events (*mouse pressed, mouse dragged, mouse released*) may result in rotating a graphic figure in the workspace. The sequence, then, may be compressed to a single event (*rotate figure*) before being multicast to the other users. This feature is useful when collaborating over low-bandwidth networks.

- Access to advanced concurrency algorithms. For example, it is very difficult to implement group undo features for collaboration-transparent applications. This is very easy in case of aware applications, since the developer supplies the undo method for each input.

- Coupling in a more meaningful manner. In case of transparent applications, coupling is possible only for the AWT components, whereas for the aware applications, it is possible for any component of a Bean.

- Replication of all types of state changes. Replication of certain types of state changes (e.g., inputs from network ports, files, or acquisition devices) may be very difficult in transparent applications [12].

---

5 Notice, however, that this feature compromises the user's group awareness, since the observing users don't see the intermediate results of the user's activity.
The specifications for the development of collaboration-aware applications along with examples are available on the Web site listed at the end of the paper. Here is a brief summary of the steps that the developer needs to follow:

1. define the high-level semantic events that each application can generate and receive; each event should extend java.util.EventObject
2. provide listener methods for all events that the application receives so that the events can be delivered
3. provide registration methods for all events that the application generates so that interested parties can register as listeners
4. each Bean has to have the method dispatchEvent(); Java AWT components come with this method, and for collaboration-aware Beans, the developer needs to provide the method.
5. provide the undo methods for all events so the user can work with different concurrency controllers; due to the undo mechanism, the application is allowed to execute the event internally before delivering it externally.

3.3. Bean loading and event interception

Application event interception is based on the property of Java Development Kit (JDK) version 1.1 event model that any object can register as an event listener to the event sources and all listeners get notified upon the event occurrence (see Section 1.1). The framework registers intermediary event adapters that deliver the intercepted events to the Communicator component of the cBus.

Initially, when the user opens a workspace the Workspace registers an EventAdapter as a ContainerListener with Workspace (see Figure 6). As a result, the Communicator (through EventAdapter) gets notified at any point during the session when a Bean gets loaded into the Workspace. The Communicator then notifies the remote Communicators to load the same Bean in their Workspaces.

In the case of collaboration-transparent applications, the collaboration bus discovers the constituent GUI components of the Bean and registers as a listener for all the events that those components generate. This way the...
bus gets notified about the events and replicates them remotely as described in Section 2.

Instead of detecting the AWT components of the application as in the case of transparent applications, the Workspace reads the BeanInfo contained in the aware application to determine the application events. In this case, Workspace registers only the listeners for the events listed in the BeanInfo object. If the developer wishes to expose some of the application’s AWT events, they need to be explicitly listed in the BeanInfo object.

4. Session management

The flexibility of the collaboration framework introduces certain complexities in the collaboration model that are not present in other collaboration frameworks. Here we consider an example.

Users join to collaborative sessions using the interface shown in Figure 7. The session manager keeps track of the ongoing sessions and the user selects the session to join from a list of active sessions. Traditional groupware does not encounter the problem of interoperability between the collaborating applications, since all collaborating users are forced to use the same application.

However, in the framework presented here, session management is complicated due to the following framework capabilities:

- the user can assemble an arbitrary JavaBean application and try to join to an existing session where the other users already use a slightly different application
- the users may have two applications, participating in different sessions and, in a design-time collaboration, establish a link between these applications

The problem in the first case is of interoperability: the new application may attribute different meaning to the event messages coming from the peers. Also, the application may have additional components/events, not having counterparts in the peer applications. Thus, it is possible that some of the events generated by this application cannot be properly replicated.

In the second case, it is not clear to which session the new application belongs. Does it belong to one or both of the old sessions or perhaps a new session should be created? In the current implementation of the framework, only the first user to create a session can drop an application Bean in the Workspace. All other users get the Bean downloaded from the SessionManager. Each

Figure 6. Sequence diagram for Bean loading. The example shows two collaborative users, where the user on the left loads the Bean and the user on the right gets a replicated Bean. (The diagram uses the Unified Modeling Language notation, where vertical lines represent time axis and horizontal lines represent the method calls.)

Figure 7. User interface for joining to a session.
Workspace with its contained applets belongs to a single session. This solution is similar to the one used in [13], where the applets are downloaded from the session manager to all but the first user. A more flexible solution would be that the user loads a local application Bean in a Workspace, selects the Workspace, and invokes the Join menu entry (see Figure 4). This would allow the users to use their own (possibly slightly different) applications to collaborate. However, the problem of interoperability needs to be solved. We are currently investigating the use of Java reflection and introspection features [20] to verify the interoperability.

4.1. Workspace management

As a result of the above problems with session management, we are currently refining the role of workspaces in collaborative sessions. We introduce a new component of the framework—Workspace Manager. The manager keeps track of the user's workspaces and cooperates with the SessionManager. Every time a user creates a Workspace, WorkspaceManager gets notified. The new Workspace is initially private. When the user wants to make a Workspace public, he/she opens the WorkspaceConnector and moves the Workspace from the private to the public “folder,” see Figure 8. By moving the Workspace to a particular folder, the user effectively joins the Workspace to the corresponding session. The individual applets within a Workspace can be moved/copied between the Workspaces using the cut-and-paste feature. The user is also able to set-up the coupling/sharing preferences for the Workspace or for individual applets within the Workspace. The system displays all the input and output events and through a series of check-boxes the user chooses which inputs should be replicated to the other conferees.

The user can move a Workspace back and forth between the private and public folders. However, when the Workspace gets moved from a private to a public folder, the user is asked to synchronize with the current state of the session.

5. Related work

The existing approaches to toolkits or frameworks for building collaborative applications violate at least one of the requirements listed in the Introduction. Most other approaches to building collaborative applications are intrusive, and this can be categorized into two types:

- replace the underlying GUI toolkit, see e.g., [1,2]
- modify the application classes to become related (e.g., via inheritance) to the collaboration framework classes, see for example [3,5,18,19]

Java Collaborative Environment [1] provides a collaborative abstract windowing toolkit (Collawt) for use in place of the java.awt graphics user interface toolkit. Each java.awt component is replaced with a double that intercepts the user events and multicasts them to the collaborating peers. The Java applications can be developed as single-user applications without any knowledge of collaboration. The main disadvantage with this architecture is that the applications have to be built using the Collawt windowing toolkit. This rules out the existing single-user applications that have been built using ordinary java.awt.

In a similar but less restrictive approach [2], java.awt is replaced by a new GUI toolkit which also intercepts the events and multicasts them to the peer applications. Although the single-user applications can be used as is, the approach works only if the underlying GUI toolkit is replaced. An additional problem with this approach is that the substitute toolkit needs to be re-built every time a new version of the original toolkit is released.

GroupKit [18], DistView [17], and more recent similar toolkits [3,5,20] provide as set of classes and interfaces that the developer uses to develop a shared application.

The Habanero framework [5] provides two strategies to develop collaborative applications using Java: (i) guidelines for developing applications with awareness about Habanero, and (ii) guidelines for porting single-user applications into collaborative applications that work within the Habanero framework. Porting includes implementing additional interfaces and modifying method signatures, and even though it is fairly straightforward, it represents a great obstacle for non-Java programmers. Once the modifications are done, the application is no longer an independent entity since it is type-cast for the particular collaborative framework.
Collaboratory Builder's Environment (CBE) [13] supports shared workspaces using the concepts of rooms, user roles in rooms, applets (slightly different from Java applets), and applet groups. The architecture is centralized around a Web-browser-centric room manager, which coordinates and multicasts user actions. CBE also provides a set of interfaces to be inherited by the applets and the resulting applets have to be run within CBE.

The Java Shared Data Toolkit (JSDT) [3] defines a multipoint data delivery service for collaboration-aware Java applications. JSDT provides a set of APIs 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.

Microsoft NetMeeting [16] provides for sharing of collaboration-transparent COM-based (Component Object Model [7]) applications. However, it has limited capabilities for state replication as well as very limited cooperative features (concurrency, coupling, awareness, etc.). It also supports the development of collaboration-aware applications, but the application's classes need to inherit or use certain object from the development kit.

The framework presented here is the only (to our knowledge) non-intrusive and self-contained framework in the sense that neither the underlying graphics toolkit nor the application require modifications. For example, most of the existing approaches do not support collaboration-transparent applications or require porting process. Once an application is developed using a given framework, the application classes have some relationship, e.g., inheritance, to the framework classes, thus effectively locking the application to the framework. Unlike this, in our framework single-user applications can be used as is, without a porting process, and collaboration-aware applications can be used independently of the framework. The functionality provided by the framework can be viewed as an additional layer, not present in the other collaboration frameworks. It can be extracted and implemented in a separate layer that runs on top of and object communication system (in particular CORBA ORB or Java RMI) or on top of JSDA, where some group-related features already exist. Even though this solution would offer greater flexibility in choosing the object communication middleware, due to the performance concerns the collaboration bus is currently implemented as a single module which subsumes both group and communication aspects.

6. Conclusions

We present a novel framework for replicating state changes across the shared workspaces. The framework enables separation of the semantics of applications from the semantics of collaboration and of distributed computing. It provides a mechanism to control the cooperative features of the system in an application-independent manner. The main strength of the framework is that it does not require modifications to the underlying graphics toolkit or application.

We offer a unified treatment for both collaboration-transparent and collaboration-aware applications. The difference is in the degree of exposed internal structure of the application. There are no explicit relationships established between the application's and the framework's classes. As a result, being collaboration-aware does not prevent an application to run outside of the framework, in a single-user mode or in some other framework.

The framework has been implemented [21] and tested on a number of Beans, both collaboration-transparent (available on the World Wide Web) as well as collaboration-aware, developed internally. The applications include whiteboarding, collaborative mapping, speech signal acquisition and processing, and image analysis. The concept succeeds in replicating most of the state changes while satisfying all of the requirements stated in Section 1, although there remain some state changes that need to be addressed.

Since the framework is based on the replicated architecture, it requires only a minimum network bandwidth for event replication. Due to its simple design, the collaboration bus introduces a minimum overhead compared to the Java sockets, so the performance is very good. We haven't yet systematically measured the latency nor done any comparative studies to examine its performance against other real-time collaborative environments. However, the experiments between two sites over a wide-area Internet connection as well as over modem lines demonstrate a near real-time performance.

Collaboration-enabling framework source code, documentation, and example applications are freely available at the following URL:

http://www.caip.rutgers.edu/multimedia/groupware/

Acknowledgments

The authors had many inspiring discussions with Stephen Juth, Cristian Francu and Weicong Wang. The research reported here is supported by DARPA Contract No. N66001-96-C-8510 and by the Rutgers Center for Computer Aids for Industrial Productivity (CAIP). CAIP is supported by the Center's Corporate Members and by the New Jersey Commission on Science and Technology.

References


