

# Transparent Latecomer Support for Web-Based Collaborative Learning Environments

**Torsten Illmann**

Department of Multimedia Computing,  
University of Ulm, Germany  
[torsten.illmann@informatik.uni-ulm.de](mailto:torsten.illmann@informatik.uni-ulm.de)

**Rene Thol**

Department of Multimedia Computing,  
University of Ulm, Germany  
[rene.thol@gmx.de](mailto:rene.thol@gmx.de)

**Michael Weber**

Department of Multimedia Computing,  
University of Ulm, Germany  
[weber@informatik.uni-ulm.de](mailto:weber@informatik.uni-ulm.de)

## ABSTRACT

In this paper we examine the problems of synchronous collaboration of users in web-based learning environments. It is a strong challenge to develop efficient synchronous groupware systems which provide transparent collaboration of existing applications whereas participants may start at different points in time. Existing collaboration systems either provide transparency or the accommodation of latecomers. We developed a transparent support for accommodating latecomers in collaborative environments which may be integrated in any Java-based system on the web.

## Keywords

CSCW, CSCL, Synchronous Groupware, Latecomer Support, State Migration, Serializable Java User Interfaces

## INTRODUCTION

Virtual and especially web-based learning is popular. Within the project "Docs 'n Drugs - The Virtual Policlinic" (Illmann, et al., 2000) we are developing a web-based and case-oriented training systems for medical students. Students learn to come to case-based decisions by answering questions or interpreting/examining findings displayed as multimedia elements. Since the system is already embedded in the curriculum of medical students at the University of Ulm, it is often used and many cases are currently in development. Web-based applications realized as Java applets enable to process and create cases.

A big challenge of such systems is to support shared learning and authoring in location-independent groups. One distinguishes between synchronous and asynchronous collaborative learning.

When evaluating existent synchronous collaboration frameworks for our system, we noticed a system called JASMINE (El Saddik, et al., 2000) which provides transparent synchronous collaboration of Java applets and applications. Unfortunately it does not support latecomers. That means that all participants of a collaborative session must start the program at exactly the same time.

In this paper we present a transparent support for latecomers of UI-based applications in Java. We outline how synchronous collaboration in Java can be achieved and describe how transparent latecoming can be supported. We integrated our implementation in the JASMINE system. Next, we describe a quantitative analysis of this implementation and discuss problems and limitations. We further mention related and future work and close with a conclusion.

## JAVA AND COLLABORATION

To realize synchronous collaboration in Java, applications or applets may be used. Since applets are special-designed Java applications for the WWW, they are a good choice for implementing collaborative applications in Java. Applets reside on a web server. On request, they are transferred to and executed on the client computer with the permission to communicate with the web server host. These are ideal conditions for implementing a collaboration framework in Java with applets.

A sophisticated collaboration framework should be able to support collaboration for any existing applet. If there are fix interfaces to meet, applet programmers have to know the interfaces and the applets do not run without the collaboration framework any more. To achieve this goal of transparency, the framework must integrate collaboration in an applet without the applet's knowledge. The main issue is to transparently forward UI events to all other participants within the current session. Forwarding events happens in three steps: Catching events, distributing them and triggering them to the user interface(s) on the other location(s).

1. To catch all UI events, one can traverse the total UI component tree (starting from the root pane) and subscribe to each component for all possible events. This mechanism is quite time and data expensive. Another possibility is the registration of a general callback at the default UI toolkit of Java for all events:

```
Toolkit.getDefaultToolkit().addAWTEventListener(  
    new AWTEventListener() {  
        public void eventDispatched(AWTEvent e) {  
            ...  
        }  
    }, Integer.MAX_VALUE);  
}
```

The listener `eventDispatched` is called for every UI event of the application. The passed `AWTEvent` contains the event type, data and component it has been released on.

2. Events which have been caught have to be distributed to all other participants. This includes their transformation into a serializable structure which additionally contains the indices of the components the events have been released on. The distribution may be performed by a central dispatch server where participants of this session are registered or by using a multicast-capable publish/subscribe communication infrastructure.
3. When a remote event is received, it has to be triggered to the corresponding component on which it has originally been released. The index included in the serialized event structure identifies this component. It is triggered by the following AWT method of the component:

```
component.dispatchEvent(event);
```

JASMINE (El Saddik, 2000) is an existing collaboration framework which provides an implementation of these ideas. JASMINE does not include a support for latecomers. Based on these ideas, we develop a transparent support for latecomers by transferring the application's state (including UI) to the latecomer.

In the following sections, we use the following terms:

- **Collaborative application** is any application or applet that is synchronously and collaboratively used by at least two users.
- **Collaborator** means a person that currently uses a collaborative application.
- **Latecomer** is a person that wants to attend a collaborative application that is already executing by other collaborators.

## LATECOMING

Based on the ideas described above, we develop a transparent support for latecomers. All applications that are commonly used by several users must be grouped to a collaborative session. If a latecomer is willing to accommodate a collaborative application that is already running at one or several other locations, two tasks have to be ensured by the collaboration framework:

- The current state of the application (which is identical for all collaborators) has to be transferred to the latecomer.
- During the transmission of the state (which lasts some time), none of the collaborators applications may change their state.

### Transferring Application's State

According to the MVC design pattern (Bushman, et al., 1996), an interactive application can be divided in three separate parts, the model, the view and the control. In order to transmit the state of an interactive application, we look at these three parts separately.

To transfer the model (core functionality) of an application, we in fact would have to capture the current state of the program (Fünfroeken, 1998) (Tyren, et al., 2000) (Illmann, et al., 2000) including program counter and stack frames. In the case of interactive applications where all operations are initiated by the user, we suppose that the application resides in the main loop and is waiting for user events when latecoming is requested. Consequently, it is sufficient to transmit the member data of all objects of the model part. These are automatically transmitted if we use the serialization mechanism (Sun-1, 2001) of Java. The only requirement is that all classes of the model part are serializable.

The view and control parts of a Swing or AWT-based application (Sun-2, 2001) are realized using UI components and their event processing callbacks (*event listeners*). Therefore our task is to transmit the state of all currently used UI components. An simple way of realizing this is to transmit the main UI container of the application and in consequence all recursively embedded components. Fortunately, elements of the Swing and AWT object hierarchy are already serializable. But unfortunately, event listeners which are subscribed to UI components are not serializable and therefore get lost or produce undesired exceptions during transmission. The only possibility to fix this problem transparently is to patch the base interface of all event listeners, the `java.util.EventListener`:

```
public interface EventListener extends java.io.Serializable {
    ...
}
```

Since AWT event programming contains the programming convention that all listeners must be inherited from this interface, all other event listeners (standard and custom ones) automatically get serializable by inheritance. Using this small patch, we achieve the goal to transmit the interactive application's state completely to the latecomer's location. There, the application is deserialized, prepared for collaborative use and shown to the latecoming user.

Unfortunately, this patch lead to some unwanted side effects when transmitting applications with Java's standard look & feel. We discovered two bugs within Java's Swing API causing these problems. These bugs and the according patches are described in detail in section *Bugs in Swing API*.

### Locking Applications

The second task for latecomer support is to lock all collaborative applications simultaneously in order to avoid state-changing events during transmission. Since a simultaneous invocation of these operations without a synchronized common physical time among all participants is not possible, all applications are requested to disable asynchronously. User events that occur before all application have acknowledged their locking are buffered in a message queue and have to be sent to all applications (except the initiating one) after the latecoming process.

### Main Algorithm

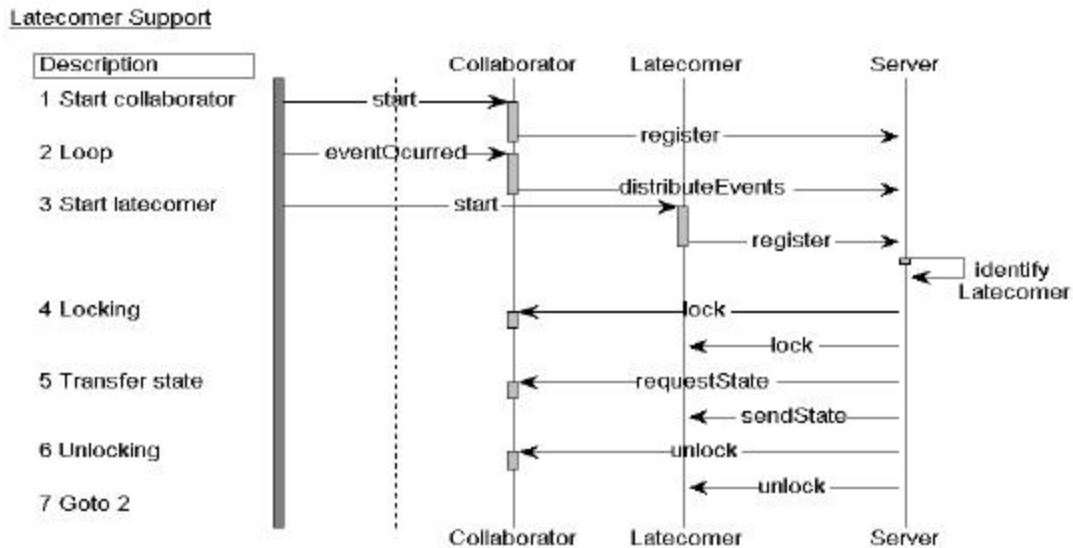
The main steps of the latecoming algorithm are illustrated in Figure 1.

The process of latecoming starts at step 3. A latecomer starts an application while another instance of it is already running (has been executed by a collaborator, step 1 and 2). The application registers at the server and is identified as latecomer. All currently collaborating and latecoming applications are requested to lock themselves (step 4) as described above. Events that occur during the locking phase are buffered. Next, the server requests the current state of the collaborative application (step 5). The server chooses one collaborator as the one that transmits its current

state. Further possible latecomers may integrate in the current latecoming process until the state is not completely transferred to the sender.

As soon as the state is received, the server distributes it to all registered latecomers. In step 6, all applications are unlocked. This includes that they are made visible again and get possible buffered events.

Finally latecoming is finished and all applications continue in their main event processing loop (step 2).



**Figure 1: Main algorithm of latecoming**

### Bugs in Swing API

The Swing API contains a pluggable look & feel concept (Wilson, 2001). The standard distribution JSDK1.3.1 (Sun-2, 2001) includes three different look & feel implementations: Metal, Windows and Motif. These implementations are all derived from the so called Basic look & feel implementation (`javax.swing.plaf.basic.BasicLookAndFeel`), an abstract implementation which provides general functionality for every concrete implementation.

The above mentioned approach works perfectly for AWT applications and Swing applications using the Windows or Motif look & feel. Whenever the standard look & feel (Metal) is used, a thrown exception collapses the state transmission because the Java serialization mechanism tries to marshal unserializable classes. After performing varies of intensive tests and detailed code inspections of the source code of the Swing API, we found two bugs in the implementation of the scroll pane component.

The Metal look & feel implementation consists of its base class `MetalLookAndFeel` and a set of UI classes derived from `javax.swing.plaf.ComponentUI` (e.g. `MetalScrollPaneUI`). Each UI class is attached to a corresponding Swing component to define its specific look & feel. When serializing a swing component, one of its base class (`javax.swing.JComponent`) initiates that its corresponding look & feel definition is uninstalled (unattached) before serializing the component.

Unfortunately the implementation of the scroll pane (`javax.swing.JScrollPane`) poses two problems when uninstalling is done. One problem resides in the abstract Basic implementation and another in the specific Metal one.

These two classes have some inconsistent method declarations for installing/uninstalling the particular UI component correctly. In general, each UI implementation has corresponding methods `installUI` and `uninstallUI` for installing and uninstalling the look & feel. In case of the UI component of `javax.swing.JScrollPane` these methods call further methods to install and uninstall event handlers (`un/installListeners`). The abstract look & feel implementation of the scroll pane (`BasicScrollPaneUI`) has different signatures for installing and uninstalling listeners:

```

public class BasicScrollPaneUI extends JScrollPaneUI {
    ....
    protected void installListener(JScrollPane c) ...
    protected void uninstallListener(JComponent c) ...
}

```

In comparison, the metal scroll pane look & feel has the same signature (`JScrollPane`) but a different visibility (`public`) as the base class before:

```

public class MetalScrollPaneUI extends BasicScrollPaneUI {
    ....
    public void installListener(JScrollPane c) ...
    public void uninstallListener(JScrollPane c) ...
}

```

Therefore, the protected method `uninstallListener` of the Basic look & feel has a different visibility and signature as the corresponding one in the derived Metal look & feel.

In case of uninstalling the metal scroll pane look & feel, this leads to the problem that the `uninstallListener` method is not invoked. During serialization, a still connected listener which is defined as inner class in the non-serializable `MetalScrollPaneUI` is tried to be marshaled and subsequent serialization fails. An equal declaration in visibility and signature avoids this behavior:

```

protected void installListener(JScrollPane c) ...
protected void uninstallListener(JScrollPane c) ...

```

We report these problems of wrong method declarations to the Swing Development Group in order to get them fixed in further versions of the JSDK.

## LIMITATIONS

In this section we discuss the limitations of our approach. Problems occur, if the collaborative application has open dialogs, has open connections to resources or is multi-threaded when latecoming is requested.

### Open Dialogs

If the collaborative application has open dialogs, especially dialogs without specifying a parent frame (e.g. in applets), the dialogs are not part of the component tree starting from the application's root window. For that reason, the collaboration framework performing the latecoming process does not find them.

This problem could be solved by explicitly searching for dialogs in the main component tree filtering for the current collaborative application and additionally transmitting them to the latecomer.

### Open Connections

If the collaborative application has open connections to resources like TCP/IP servers or databases, these connections are not transferred. We call this problem *resource migration* (Illmann, et al., 2001). It can be solved by either patching the `java.io` and `java.net` class hierarchy or using proxy/forwarding concepts. A more simple but less transparent solution is that applications with open connections to resources implement a well-known interface:

```

public interface LatecomerSupport {
    public void init() throws Exception;
    public void destroy() throws Exception;
}

```

The convention for using this interface is that all resources have to be opened in the `init` method and all open resources closed in the `destroy` method. In consequence, the collaboration framework initializes all connections at start time, destroys them before transmission and (re-)initializes them after transmission on both locations (latecomer and collaborator).

Unfortunately, this approach is not totally transparent since applications must be changed to manage their resources using this interface. Further problems of consistency still arise if connections to resources reside in a stateful session when latecoming is requested or incremental modifications are performed on the resource. .

### Multiple Threads

Java Threads are not serializable and therefore can not be transferred to other locations. Therefore, if the collaborative application consists of multiple threads or even dynamically creates and destroys threads, latecoming does not work correctly. Current research (Tyren ,et al., 2001) and (Illmann ,et al., 2000) is trying to cover this problem which we call *multi-threaded migration*.

## QUANTITATIVE ANALYSIS

The following section documents the average transmission time (in milliseconds) and the number of transmitted bytes needed by sender, server and receiver to send and/or receive the application's state.

All tests were performed using a 100 MBit LAN and three Athlon 800 MHz PCs equipped with 256MB RAM and Windows NT 4.0. Each computer took over another part of the framework: sender, server and receiver.

The application size varied by increasing the amount of GUI components in the test application. First, an empty Swing application was measured to get a baseline value. Then, we varied the amount of five representative components (JButton, 2 JScrollPane, JTextArea and JTable) from one times five (5) to eight times five (40). We measured the transmission time and the total amount of transferred data. The transmission time was measured at all three locations: the sender, the dispatch server and the receiver. The resulting times are average values of several measurements:

Number of components	Sender [ms]	Server [ms]	Receiver [ms]	Bytes [kB]
None	154	80	353	6
1 x 5	404	497	684	38
2 x 5	447	504	578	58
4 x 5	587	558	658	98
8 x 5	748	671	811	179

**Table 1: Average transmission time and data**

The transferred amount of data is nearly linear when the baseline value (just one empty root component) is subtracted. Since some components are not initialized with random content but identical one, the serialization mechanism probably does optimizations and therefore does not show a direct proportional result. An average of 5 kB per component documents that the Swing API is quite complex and components consist of many further objects and contain further references. Consequently, latecoming of Swing-based applications is an expensive task.

The transmission times at sender, server and receiver locations only increase weakly with the size of application's state. Probably for a small number of components, the connection and disconnection times between communication partners are much more important than the amount of sent data. For higher values this effect becomes obviously less significant.

Comparing the times at the three locations, the server generally produced the lowest values. Whereas sender and receiver are traversing object hierarchies and sending/receiving object streams, the server is only forwarding byte arrays without knowing exactly what information (or objects) are behind. The further advantage of this optimization is that the server does not need to know the specific class information of the transferred application's classes. Otherwise, all classes would have to be accessible in the server's classpath (which cannot be assured in general).

The receiver times are about 200ms higher than the ones of sender or receiver. This fix delay is caused at the server to forward the data.

## **RELATED WORK**

In this section we describe other work that relates to this approach.

### **JASMINE**

JASMINE (El Saddik ,et al., 2000) was developed at the Darmstadt University of Technology in cooperation with the University of Ottawa. It is a tool for tight and transparent synchronous collaboration using Java applets and/or applications without latecomer support. JASMINE works session-based and the applets/applications used for collaboration need no source code modifications (transparency). Therefore it allows the collaborative use of applets brought online into a session. Furthermore JASMINE uses moderation to coordinate session-members adequately. It also uses floor control in an intuitive way, so there is no need for users to care for it.

JASMINE provides its functionality using a server and clients for the collaborative sessions. Applets and applications which should be used for collaboration must be registered in a configuration file (where applets can also be brought live into a session using a URL). The collaborative aspect for collaboration unaware applets/applications is achieved by listening for events fired by user interactions and distributing them to other session members.

### **Habanero**

The Habanero (Jackson, 1997) framework which was developed at the University of Illinois uses Hablets (these are converted/newly created applets using a special API) for its synchronous/asynchronous collaboration. Due to some additional services which can be used along with Habanero, the user is able to participate asynchronously in sessions and work with some Hablets (those must possess asynchronous functionality). Habanero also keeps track of latecomers, which are provided with all running Hablets if they join a Habanero session. Similar to JASMINE the Habanero framework also uses server and clients in order to provide a collaborative environment. It addresses developers as well as users by providing an API. Hence, developers can convert applets and applications into collaborative tools or even implement complete new collaborative applications. Users finally are able to use these tools within their collaborative sessions. One of the eminent abilities Habanero provides is the possibility to record and replay sessions (or parts of them).

### **Mushroom**

Mushroom (Kindberg, 1996) is a platform for collaboration and groupware over the Internet and was developed at the Queen Mary University of London. It allows collaborative working within shared workspaces using tools called Mushlets. To integrate applets into Mushroom their source code has to be modified. Mushroom is able to handle external helper applications for documents worked on within the above mentioned shared workspaces. These helpers use mime types and file extensions to recognize their related document types. Mushroom users are able to form groups for a more scalable collaborative working. The platform consists of one or more server(s) and several clients which allows their users to work collaboratively within the common used workspaces. Besides the use of Mushlets and documents the Mushroom environment allows the import and common synchronously use of entire directories. Another ability for Mushroom clients is the possibility of integrating new Mushlets while they are connected with the server(s) and used for collaboration.

### **Latecomersupport for X**

(Chung, 1993) realized a solution for supporting latecomers in X-Windows-based applications using the XTV protocol (Abdel-Wahab, 1991), a suggested extension of the X-protocol for collaboration.

## **CONCLUSION**

In this paper, we describe the realization of a transparent latecomer support for synchronous collaboration in Java. Applications or applets developed for standalone operation do not have to be modified for collaboration or latecoming. One patch of the Swing API is necessary to enable the implementation of this idea. We will discuss and

suggest this patch to the Swing Development Group. During development, we found two bugs in the standard look & feel implementation of Swing which inhibited the implementation.

The work has been developed in cooperation with JASMINE and already integrated in their collaboration framework. Nevertheless, the concept is general and could be used in any other Java based collaboration framework to support latecomers.

We further show that a transparent support for latecomers raises analog problems as in load-balancing and mobile agents systems by transferring the application's state to another location.

## FUTURE WORK

As future work we will integrate the collaboration framework including latecomer support within the case-based training system DND. This allows learners to process training cases in a location-independent group without starting at the same time. Since mouse locations and movements of concurrently working learners are important issues to provide a realistic presentation of synchronous groupware, we are working on a transparent integration of multiple telepointers for existing applications.

We want to reduce the mentioned problems of transmitting open dialogs. To support a simple kind of resource and thread migration, we suggest to integrate a latecomer interface. Nevertheless, we concentrate our research in finding a better and more transparent solutions of handling open resources and multiple threads.

## ACKNOWLEDGMENTS

We thank the members of the JASMINE project (El Saddik, 2000) for providing the source code of the JASMINE system. Since collaboration already was implemented there, we could totally concentrate on the latecoming support.

## REFERENCES

- Abdel-Wahab, H., Feit, M. A. (1991) XTV: A Framework for Sharing X Window Clients in Remote Synchronous Collaboration, *Proceedings of IEEE Conference on Communications Software: Communications for Distributed Applications & Systems* (Chapel Hill, NC, April 1991), pp. 159-167.
- Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M. (1996) *Pattern-Oriented Software Architecture. A System of Patterns*, John Wiley & Sons.
- Chung, G., Jeffay, K. and Abdel-Wahab, H. (1993), Accommodating Latecomers in Shared Window Systems, *IEEE Computers* (January 1993), pp. 72-74, Vol. 26, No. 1.
- Dorohonceanu B., Sletterink B., Marsic I. (2000) A Novel User Interface for Group Collaboration, *Proceedings of the 33rd Hawaii International Conference on System Sciences* (Maui, Hawaii, USA, January 2000), 10 pages/CD-ROM.
- El Saddik, A., Shirmohammadi, S., Georganas, N., Steinmetz, R. (2000) JASMINE: Java Application Sharing in Multiuser Interactive Environments. *Proceedings of IDMS '2000* (Enschede, Netherlands, 2000), Springer, 214-226.
- Fünfroeken, S. (1998) Transparent Migration of Java-Based Mobile Agents. *Proceedings of MA'98* (Stuttgart, Germany, 1998), Springer, 26--37.
- Illmann, T., Kargl, F., Krüger, T., Weber, M. (2000) Migration in Java: Problems, Classifications and Solutions, *Proceedings of MAMA '2000* (Wollongong, Australia, December 2000).
- Illmann, T., Weber, M., Martens, A., Seitz, A. (2000) A Pattern-Oriented Design of a Web-Based and Case-Oriented Multimedia Training System in Medicine. *Proceedings of IDPT '2000* (Dallas, USA, June 2000), Social Science Computing Review.
- Jackson, L. (1997) NCSA Habanero Java Object-Sharing Collaboration Framework, Cooperative Research Centre for Distributed Systems Technology, University of Queensland, *Proceedings of JavAus'97* (Queensland, Australia, February 1997).
- Kindberg, T., Coulouris, G., Dollimore, J. and Heikkinen, J. (1996) Sharing Objects over the Internet: the Mushroom Approach, *Proceedings of IEEE Global Internet* (London, UK, November 1996), pp. 67-71.
- Sun Microsystems Inc. (2001) Java Object Serialization Specification, <ftp://ftp.java.sun.com/docs/j2se1.3/serial-spec.pdf>, visited 10.10.01.

- Sun Microsystems Inc. (2001) Java2 Platform Standard Edition V1.3 Homepage, <http://www.javasoft.com/j2se/1.3/>, visited 12.05.01.
- Truyen, E., Robben, B., Vanhaute, B., Coninx, T., Joosen, W., Verbaeten, P. (2000) Portable Support for Transparent Thread Migration in Java, *Proceedings of ASAMA'2000* (Zürich, Germany, September 2000), Springer, 29--43.
- Wison, S., The LookandFeel Class Reference - A PLAF Lookup Guide for Swing Programmers, <http://java.sun.com/products/jfc/tsc/articles/lookandfeel/reference/>, visited 03.07.01.