WAM 1.7

APPLICATION SHARING FRAMEWORKS ON JAVA[1]

Shiuh-Sheng Yu, Chee-Wen Shiah, Leuo-Hong Wang, and Wen-Chin Chen
Communications & Multimedia Laboratory
Department of Computer Science and Information Engineering
National Taiwan University
Taipei, Taiwan, R.O.C.

ABSTRACT

Application sharing systems are important components of a CSCW environment. We present in this paper different architectures for application sharing systems and how to implement them on various versions of JDK (Java Development Kits). An application sharing system can be developed on either *event-sharing* model or *request-sharing* model. We discuss in this paper the implementation of an event-sharing system and also propose the architecture of a request-sharing system on the next generation JDK.

1. INTRODUCTION

A CSCW environment can be built on a *distributed-objects* model[1] or on an *application sharing* system. There are two models for the application sharing systems[2], namely, the event-sharing and request-sharing models. The request-sharing model starts an application on a server and multiplexes its graphical outputs to all the participants. This model can share any application, but do not support many users due to the performance limit. The event-sharing model starts the same application with every participant, then broadcasts the input events. As it applies only to the deterministic programs running in the same environment, it is often integrated with a distributed-objects model to extend its functionality.

The popular newly developed Java language enjoys the property of "write once, run everywhere"[3]. The combination of an application sharing system and Java allows us to "write once, run everywhere and cooperatively." In this paper, the implementation of a hybrid system of the event-sharing and the distributed-objects model on JDK 1.0 is discussed. Moreover, a request-sharing system for the next generation of JDK is proposed.

2. EVENT- SHARING ON JDK 1.0

JDK 1.0 uses a peer model to delegate the look-and-feel of the AWT (Abstract Window Toolkit) components to a set of underlying "peer" classes. We can not intercept graphical commands in pure Java to implement a request-sharing system. Our implementation of an event-sharing system includes three modules. The *SessionManager* shows the available sessions on a *RegisterServer*. The *SessionManager* calls *ShareManager* to create or join a session. The scenario is illustrated in Figures 1 and 2.

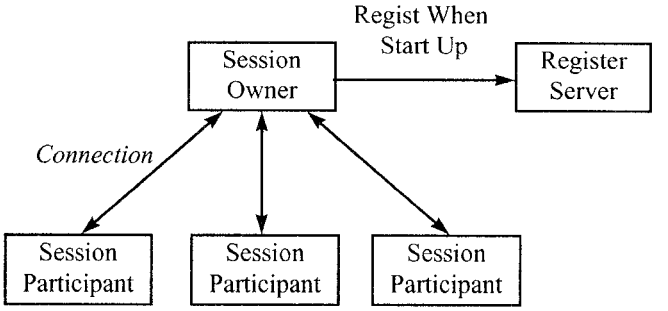

Fig.1 The relationship between session owner, RegisterServer and participants.

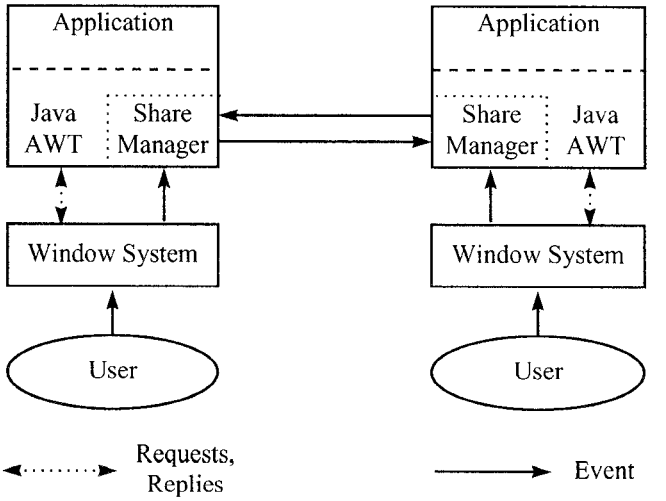

Fig.2 Messages between the session owner and a participant.

Only token holders can generate input events. A session owner can pop up a menu to kick out any participant, to revoke the token, or to close the session. A session participant can request the token from the session owner.

A JDK 1.0 program must inherit GUI components and override the *handleEvent()* method to handle events. When an event happens, the *postEvent()* method of the target component propagates sequentially up the GUI hierarchy to call the

[1] Research supported in part by National Science Council of R.O.C. under grant No: NSC85-2622-E002-015

0-7803-3734-4/97 $5.00 © 1997 IEEE

handleEvent() method on the propagation path, until either one *handelEvent()* returns "true" to consume the event or the root of the hierarchy is reached.

We modify the *postEvent()* method to intercept input events. The modified *postEvent()* returns "false" immediately if it is not the token owner; otherwise, it behaves as usual. If any component consumes the event, the *ShareManager* broadcasts the event to all the session participants.

The target component has the type of "reference," and is meaningful only in the same address space. We view the GUI hierarchy as a tree, and encode the component as the path from the root window to the target component.

When the *ShareManager* receives an event, it tries to post the event until the event is consumed. All input events are passed to and recorded in the session owner. When a connection is established, the session owner dumps the recorded input events to the connection to synchronize the late comer.

3. DISTRIBUTED-OBJECTS ON JDK 1.0

A distributed-objects framework has been implemented also. In this framework, the *ShareAction* interface must be implemented for a *ShareAction* component. A distributed-objects application calls the *ShareManager* to broadcast its actions. When the *ShareManager* receives a share action command, it calls the *doAction()* method defined in the *ShareAction* interface of the target component to manipulate the command.

If a component is contained by a *ShareAction* component, its *postEvent()* method behaves exactly as in Section 2, but without event broadcasting. Such an event passing mechanism allows a single-user application to embed multiple *ShareAction* components.

4. IMPLEMENTATION AND EVALUATION

The implementation of Sections 2 and 3 is part of the project "Multimedia Digital Classroom" currently under development in the Communications & Multimedia Laboratory of National Taiwan University. The project aims at providing a number of tools to support a distributed interactive educational environment. The tools include a shared browser and a shared whiteboard. The browser does not possess share capability in itself. Its share capability indeed results from the underlying event-sharing system. The shared whiteboard was implemented on top of the above distributed-objects model. The shared whiteboard was first developed for single-user mode. It was then converted to the distributed-objects model with less than 100 lines of new/modified code.

We tested the system by creating a session with participants from Microsoft Windows 95 and SUN Solaris systems. The browser can browse most of the document formats including HTML 2.0, VRML 1.0, MPEG I, and JPEG. It can also execute Java applets on Internet.

5. APPLICATION SHARING ON JDK 1.1 AND THE NEXT GENERATION

JDK1.1 introduces a new delegation-based event model [4]. A GUI component uses the new method *dispatchEvent()* to intercept input events. The implementation of an event-sharing system is similar to that in Section 2 on JDK 1.1 except we have to modify the *dispatchEvent()* method instead of the *postEvent()* method.

JDK 1.1 still uses the peer model to implement the AWT components. The next generation JDK promises to provide a "peerless" version, such that the AWT components are implemented in pure Java code[5]. To implement a request-sharing system on the next generation JDK, we propose an architecture shown in Figure 3. We will modify AWT components to intercept graphical commands and then implement a X-window like server for the session participants.

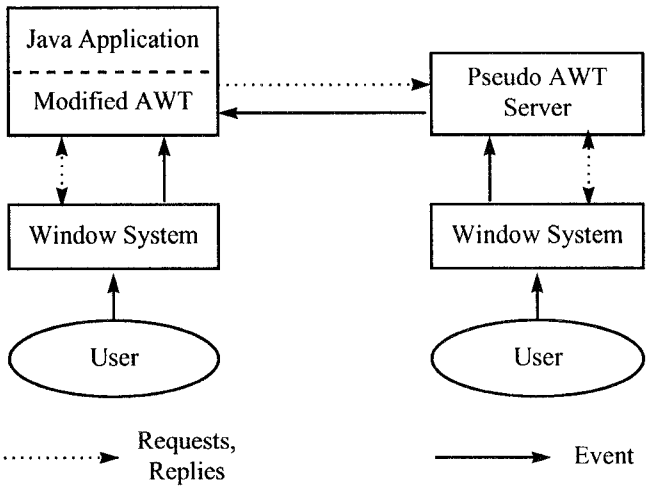

Fig.3 The proposed architecture of a request-sharing system for the next generation of JDK.

6. CONCLUSIONS

An event-sharing system is easy to be implemented but has limited scope of applications, whereas a request-sharing system can share any application. To take advantage of both models, we thus try to implement a hybrid system of the event-sharing and the distributed-objects model on JDK 1.0, such that the system can be applied to a shared browser application. The current versions of JDK are not adequate to the implementation of a pure Java request-sharing system. We are waiting for the next generation of JDK to build a request-sharing system.

7. REFERENCES

1. ITU-T Recommendation T.120 : "Data Protocols for Multimedia Conferencing," Draft, Oct. 1995.
2. W. Minenko, "The Application Sharing Technology," *The X Advisor*, Vol. 1, No. 1, June 1995.
3. "The Java Language: An Overview," *http://www.javasoft.com/doc/Overviews/java/java-overview-1.html.*
4. "Java AWT: Delegation Event Model," *http://www.javasoft.com/products/JDK/1.1/docs/guide/awt/designspec/events.html.*
5. "AWT: The Next Generation," *http://www.javasoft.com/products/JDK/1.1/docs/guide/awt/designspec/nextAWT.html.*