Enabling Synchronous Joint-Working In Java

Vladimir Minenko
International Computer Science Institute (ICSI), Berkeley, CA, U.S.A.
minenko@icsi.berkeley.edu
phone: +1 (510) 642 4274 ext. 306

ABSTRACT

This paper gives an outlook on technologies for joint-working with Java-based programs - applets and applications. Various approaches and APIs applied to the Java environment are discussed and compared. A new architecture for scalable Java application sharing is presented. Several suggestions on possible future features of JDK facing synchronous joint-working are presented.

Keywords: collaboration, joint-working, Swing, Java, JDK, conferencing, application sharing, CSCW.

INTRODUCTION

This article focuses on design approaches and technological possibilities of synchronous joint working with Java applications. In this case, “Joint working” means two or more individuals synchronously work in the same Java-based application environment in different locations. This environment can be either a shared single-user Java application or a special multi-user capable data processing application. In this article the term “application” does not refer to any chat or document exchange systems, but to an office suite, a CAD system, a software IDE, etc.

The idea of synchronous joint working is not new in the CSCW field. It mostly left research labs and turned into commercial products. Several application sharing engines are available for X Window [11], [12], Apple QuickView and Microsoft Windows [9] operating environments. Nevertheless, the growing role of Intranets in the business data processing and several peculiarities of the Java environment open new horizons for enabling technologies, especially for high-level collaborative frameworks.

The most important impact of Java on joint-working is that it solves one of CSCW’s key problems [4]: distribution and access to software and services for a group of users. There is no sense in possessing a telephone, if nobody else has one. Having a joint-working environment is useless, if your partners do not have one. A Java program resides on the network. Anybody who can access the network can use this Java program. The same is valid for collaboration systems realized in Java. This feature makes ad-hoc collaboration sessions and long-term team building easier and more flexible. In addition, object-oriented Java run-time and component-based GUI architecture enable the realization of new scalable joint-working and remote-access services.

This article discusses several approaches in the design of synchronous joint-working in Java. All approaches focus on the use of single-user Java applications[*] in multi-user environments. It also describes basic elements of a new approach which realizes presentation independent scalable joint-working services on a GUI object level.

This article also gives a brief overview over the two most powerful Java APIs for the design of collaborative environments and multi-user applications.

LEVELS OF AWARENESS AND TRANSPARENCY

Applications involved in joint-working can be placed on different levels of group-awareness. These levels reflect how to what extent an application and its execution environment know about being used by multiple users at same time:

1. *Native multi-user.* Applications with a data processing flow and GUI specifically designed for synchronous use by multiple individuals.
2. *Multi-user enabled*. Originally single-user applications with some minor internal modifications in the event processing and data synchronization.
3. *Multi-user view.* Originally single-user GUI elements of the execution environment modified to support the multi-user metaphor. Any single-user application with these GUI elements can be operated by multiple users.
4. *Shared Input*. Modified event processing of the run-time environment, which provides the mouse, keyboard and some system access synchronization of several instances

*. If not indicated differently, here the term “application” also includes here Java applets as well.

of one application executed by each user. Every single-user application can be operated by multiple users with some restrictions [3], [8].

5. *Shared Graphic*. Modified graphic rendering of the run-time environment providing multiple graphic outputs of one application instance. Every single-user application can be operated by multiple users. Better network and graphic performance is required.

The first two levels require either modifications in the application code or the development of new applications with build-in multi-user support. The last three approaches do not assume any changes in applications. The required processing takes place in the execution environment. These levels are also know as *application sharing*.

There are several works [8], [5] discussing the advantages and drawbacks of each level as well as different architectures and implementations. The most important observation is that the first level provides the best support of groupware aspects of joint-working, whereas the last one provides the best integration of any single-user application into a multi-user environment. All currently known commercial products use either the first or the last approach. Several run-time and synchronization problems prevent the availability of stable and flexible solutions for traditional platforms (X Window, Microsoft Windows, Apple QuickView) by using other levels. The network-centric, object-oriented Java architecture enables the design of new joint-working services on level 3, which would be impossible on traditional platforms. Therefore the advantage of level 3 is that it provides a high level of group-awareness and yet does not require any changes in applications.

BALANCING TRANSPARENCY AND AWARENESS

Talking about joint-working, a suitable proportion of transparency and awareness must be found for a given system in a given application area. The character of this proportion is very similar to the “memory”/“access time” problems in data structures and search algorithms. Finally, we want to put a maximum amount of multi-user knowledge into any single-user application with as little changes as possible in the application itself and in the execution environment.

Again, like with the “memory”/“access time” problems, some tricks can be applied to reach unreachable. Considering Java, the main idea is to use its object-oriented character and implement the group-awareness on the most abstract level of the execution core of the system used by all applications.

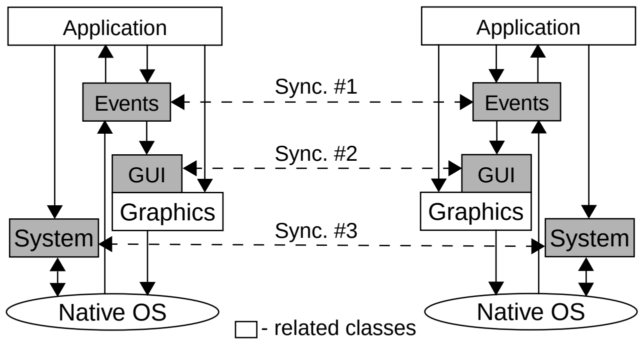
An excellent example of what is possible on levels 3 and 4 is the JAMM system designed by James Begole, Virginia Tech [3]. JAMM applies “relaxed” WYSIWIS (What You See Is What I See) metaphor by putting awareness into some GUI object (i.e. level 3 on the list). Basing on the Swing GUI from JDK 1.2, JAMM allows, for example, several users to synchronously work on different pieces of one text in a single-user `Notepad` application. A “radar view” displays the working areas of each user.

This work introduces a different approach that is based centralized processing with a kind of reflection of the Java GUI model over the network. It also provides a higher level of group-awareness than known systems on level 5. Core ideas and features of this approach are discussed in sections below.

SHARING JAVA

Commonly speaking, there are two run-time architectures for enabling synchronous joint-working with single-user applications: multiple and centralized execution of shared applications. Both architectures have their advantages and drawbacks discussed in detail in several publications [8], [5]. When applied to the Java environment, these architectures undergo some changes and become new features.

Figure. 1. Multiple Execution*



Multiple execution is known to have to provide sophisticated synchronization mechanisms while executing several instances of the same application for each user. Each instance receives input and control events from the other users via a synchronization channel #1 (Figure 1). A management module of channel #1 also has to keep the event flow plausible and consistent for each application instance.

Early implementations of this architecture for Java (Collaborator Toolset, from Old Dominion University, [1]) display significant problems with an awkward event processing of JDK 1.0.x; essential event information was consumed by native window peers and some events carried unportable data about its window peer. [3] gives a complete overview of these problems. The Collaborator Toolset partly solved these problems concerning event processing by replacing the entire AWT toolkit with its own specially modified implementation. Collaborator’s AWT classes provide some additional synchronization and token control on channel #2.

The new event model and peer-less AWT in JDK 1.1 significantly improved the situation. A version of JAMM used these features for more efficient event distribution. JAMM also solved another problem of the multiple execution architecture: joining latecomers to a running session. The problem is to create a new instance of an application on the remote host without playing back all the events of the session to a new instance of a shared application. By using the Java Seri-

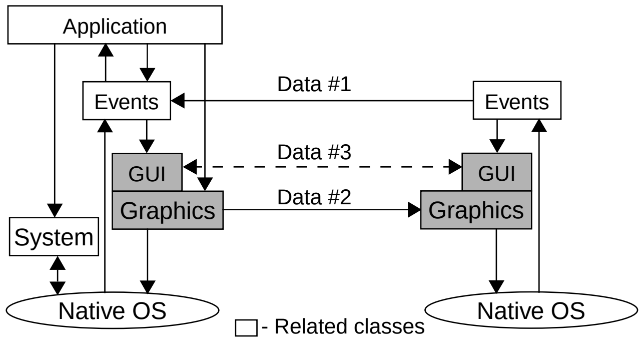
*. The structure of the Java run-time is simplified in this and all following figures. Only the most important elements are listed

alization API, JAMM transfers object information to a new participant and re-constructs the application state by creating new object instances.

Nevertheless, two serious problems remain in the multiple execution environment, if sharing needs to support complex Java programs. It is still not possible to synchronize the access to system resources (Figure 1, Sync. #3) without having to change core modules of the run-time environment. Many factors, such as data access times, quality of service (QoS) parameters of the network, CPU load, etc., can negatively influence the execution of an application and make multiple instances of a shared application inconsistent among participants. JAMM is announced to include special wrapping system classes in future versions in order to keep those factors under control and maintain application instances consistent during the whole session.

Another open issue is the interaction of applications with data. Sharing an application in a multiple execution architecture also means sharing its data in the session. While synchronizing the QoS parameters of data access can help to keep consistency, the required data replication among session participants may have negative impacts on data security. However, in numerous cases, such as teleconsulting and telelearning, the data replication is not required. Users simply need to have a joint view on the application and make some inputs into it. Frequently, the application data is relevant only for one session and can disappear after closing the session. If some participants have small or portable devices, data replication is not desired and can even be impossible.

Figure. 2. Centralized Execution



The centralized execution architecture involves only one instance of the shared application. Remote hosts receive information about visual presentations of an application on the screen (usually Data #2 on Figure 2) and forward input events back to the shared application (Figure 2, Data #1). Since the application is running in its "original" environment synchronization problems do not occur. Additionally, remote participants need access neither to application classes nor to its data. These are two main advantages of the centralized architecture. The drawbacks of implementations on traditional platforms include that a relatively high network throughput and a low network latency are required. The multi-user awareness of centralized execution is also very simple and provides only a token control with several passing policies.

It does not make much sense to realize this architecture with processing of Data #1 and Data #2 under Java. Joint-working on this level can be efficiently done with native sharing tools such as [11], [12] or [9].

Nevertheless, new types of joint-working services can be provided in Java, if a centralized sharing system will be based on the processing of Java GUI component objects. This can be done on the data channel #3 (Figure 2).

Since Java GUI components contain more information about the application context, the centralized execution architecture in Java integrates a higher level of group-awareness than on traditional platforms. In this event, the sharing system sends abstract data about the visual presentation of GUI components. The GUI is rendered locally on each remote host. This enables the system to send less data than on channel #2 and save network bandwidth.

This article presents the Component Model Reflector (CMR) approach which is designed using the suggestions made above. More details on this approach can be found under "Inside the Component Model Reflector".

Figure. 3. Centralized Java Joint-Working Scenarios

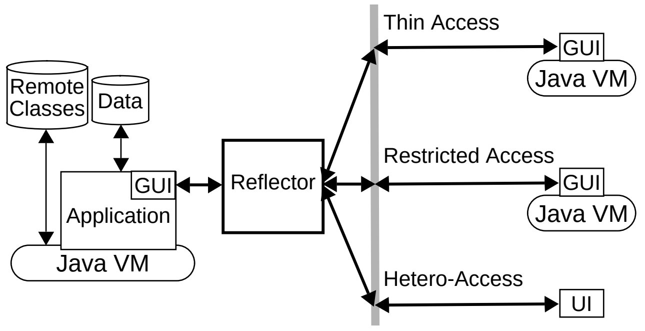


Figure 3 shows three application scenarios with a Java joint-working service based on the new approach. These scenarios are thought to show high scalability and flexibility of the provided joint-working infrastructure. The core of this system is a GUI reflector-storage. It collects data on the GUI of applications assigned to a joint-working session. From a functional point of view, the Reflector can be considered a network proxy which provides GUI data for external representations. A visualization of these data depends on a given usage scenario (see Figure 3) and on the capabilities of remote devices.

In the *Thin Access* scenario remote clients may be small or portable devices like PDA. Users of these devices usually need to access only some GUI elements of an application, for example, a text in their main office. The Reflector sends a pre-selected amount of low-bandwidth GUI data, thus enabling remote clients to save resources and render only the most important GUI elements. Since remote clients support a Java environment, the application GUI may even keep its original look-and-feel.

In numerous collaboration situations it is required to restrict manipulation of a shared application by remote session participants. In the *Restricted Access* scenario the initiator of a session can block several GUI elements for other session participants. Using information about component objects, the Reflector can apply special policies to all (or some) GUI elements. Users can jointly work on a joint-venture document but would not be able to load sensitive data, because, for example, the *Load* and *Save* menu items are blocked in the Reflector.

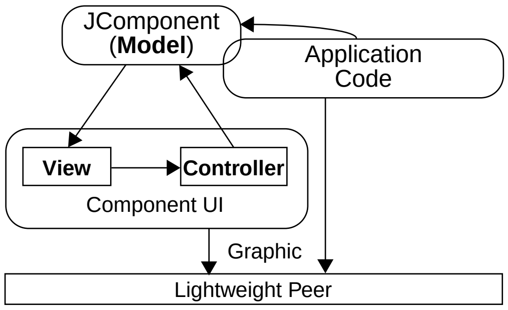
In the *Hetero-Access* scenario users can remotely control or collaborate in highly heterogeneous environments. Remote users can, for example, perform actions in an application by calling from a touch-phone or joining a session from another UI environment which does not support Java. This is possible because the Reflector does not contain any fixed visualization data. Remote sites are responsible for the visualization of GUI and for forwarding event data in the format defined by the Reflector. In this case, the actual implementation of the remote UI environment does not make any difference for the processing flow.

The next section discusses the main principles of CMR. It is followed by a section about the impacts of Java security on the realization of synchronous joint-working.

INSIDE THE COMPONENT MODEL REFLECTOR

Swings, the new GUI API in the JDK 1.2, implements the MCV (Model-Controller-View) approach [6] (see Figure 4). MVC separates the visual presentation (*view*) of the GUI from its logic (*model*) and event processing procedures (*control*). In Swings the combination of *view* and *control* is realized by Look-and-Feel (L&F), a set subclasses of the `ComponentUI` class. The *model* is realized in subclasses of the `JComponent` class. L&F can either be changed at run time by the application itself or specified in a configuration file. Swing provides a rich set of various basic components, which allow to build complex GUI.

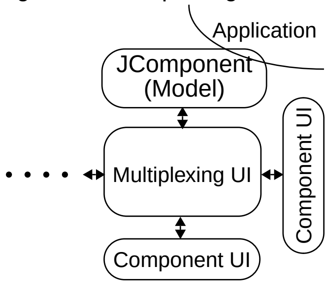
Figure. 4. Swing Look and Feel (JDK 1.2)



In addition to Java-, Motif- and Windows-L&F, a Multiple L&F (ML&F, also called “Multiplexing UI”) is provided (see Figure 5). ML&F is able to manage numerous simultaneous views of one GUI. This feature is thought for interfaces with special (e.g. speech) input devices and for people with disabilities[*].

*. see the JavaSoft web site (`www.javasoft.com`) for more information about Swing and JDK 1.2.

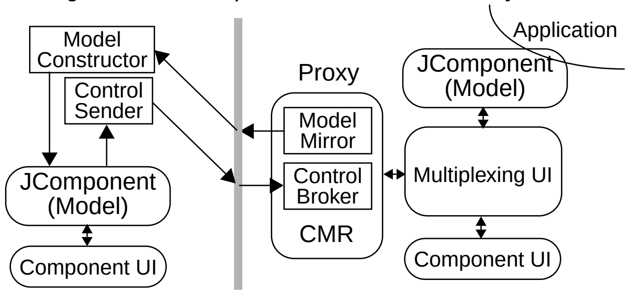
Figure. 5. Multiplexing Look-And-Feel



ML&F can be also used to build a CMR which can install a custom L&F, i.e a new Component UI, and thus gain access to the model data and GUI rendering of all Java programs running in the virtual machine. At present this seems to be the only suitable interface to realize the centralized execution architecture without having to change the core system classes.

The main parts of CMR are a *Model Mirror* module and a *Control Broker* module (see Figure 6), which can be implemented in a proxy server. The *Model Mirror* module collects information about models of locally running Swing components. When using the property interface of JavaBeans, *Model Mirror* is notified each time a parameter in a component has been changed. In addition to the data on the component model, *Model Mirror* also stores information about the layout and GUI hierarchy. Remote clients use *Model Constructor* to create a copy of a component and to feed its model with current data. *Model Constructors* receive this data from *Model Mirror* via network or inter-client exchange channels.

Figure. 6. A Component Model Reflector System



Once a remote instance of a component has been completely created and installed, *Control Sender* begins to monitor all events processed by a remote component. If an event on the remote site requires changes in the model, it will be sent to the *Control Broker* of CMR and forwarded to the local component. If the owner of this application has defined special token passing policies, the forwarding of input events can be blocked. *Control Broker* uses the new `EventQueueClass` interface to inject remote events into the application event queue. All other events which do not change the model, such as `PaintEvent`, are processed on the remote side without involvement of *Control Broker*, thus saving network bandwidth and increasing the rendering performance.

If a remote site does not use Java, it has to provide at least a *Model Constructor* and a *Control Sender*. In this case, a remote UI environment is responsible for (e.g. visual, audio, etc.) presentation of the model data and event processing.

Swing is a set of components. Most applications take advantage of this by composing complex meta-components or extending standard classes for special purposes. While CMR does not have any problems processing meta components, the processing of extended components may become a problem.

If a new class extends only a `JComponent` successor, no problems will occur since the new class must use the interface to a Component UI of the extended class. CMR will still be able to handle this correctly. If an application also extends a Component UI, CMR can only function, if these new components are JavaBeans-conform and the new extended classes are available on the remote site. Otherwise CMR will not be able to detect which data it has to store in *Model Mirror* and *Model Constructor* to load appropriate UI classes.

A serious limitation to the current CMR architecture is that it cannot process direct graphic operations. This occurs if an application uses a component surface to paint some custom graphic bypassing the Component UI methods (see Figure 4). The only problem to support this is that the `Graphics` object does not provide any hooks or other interfaces which would be suitable to monitor and replicate graphic rendering. CMR may provide a multi-user replacement to the `Graphics` class but this will require changes in the Java core classes. A simple monitor interface in instances of the `Graphics` class might help to provide a solution for this problem in the future. In this case, CMR will keep its functionality, and broadcast graphic calls only if they do not form part of the basic GUI rendering.

A simple custom ML&F to test the idea of CMR is realized in our lab. The implementation supports only the `JButton` class and intended to test the basic elements of the CMR architecture. Tests shown that the approach is feasible to support synchronous joint-working in Java. The methods of Swing component classes provide enough information to collect required data about the model of a component and GUI layout. There is still a lot of work to do until the first version of a complete system will be available.

JAVA SECURITY RESTRICTIONS

Java introduced heterogeneous execution of applications loaded from a network. The use of public networks (particularly, Internet) as main transport mediums for Java code moved JavaSoft to implement strong security restrictions in the execution of network code.

Sharing single user applications requires some additional processing at run-time. Current Java security policies *do not allow* to implement required processing *without making any changes* to the Java environment.

All available implementations require a combination of the following changes in JRE[*] on each participating host:

- installation of new classes;
- replacement of original classes with new “group-aware” classes;
- modifications in the local JRE configuration files.

Component Model Reflector requires modifications of AWT property files only. This must be done only on the host, where a shared application is running. Remote participants do not need to change local configuration or install any new classes; they can download processing applets on-demand from a CMR site. Nevertheless, in order to support graphic output processing on component surfaces, installation of new custom classes in JRE and access to core classes in JRE could be required in the future.

An ideal environment for synchronous joint-working in Java must support ad-hoc sessions: “load joint-working service classes and share any Java application”. This is possible if the following additional APIs were be integrated in JDK:

- all features below are available for signed Java classes only;
- an applet API to create a custom `Graphic` class;
- a set of hooks into rendering methods of the standard `Graphics` class and its subclasses.
- an applet API to reside custom classes in JRE and replace system classes, if required.
- an applet API to change JRE configuration files.

JAVA COLLABORATION TOOLKITS AND APIS

Unlike sharing systems, collaboration toolkits provide basic collaboration APIs. The main goal of these APIs is to hide the communication and management overhead from the developer and simplify the development of collaborative applications and integration of the collaboration paradigm into single user applications. Applications based on such APIs are on level one and two of the awareness list in section “Levels of awareness and transparency”. Sharing systems on other levels, such as CMR, can also use these APIs to implement multi-point data transport and basic session management. Following subsections give a brief description of the two most powerful APIs. Promondia (former COMO) [2] was also a candidate for review but is not presented here due to its weak API and insufficient documentation.

NCSA Habanero

Habanero (`www.ncsa.uiuc.edu/SDG/Software/Habanero`) is probably best know Java development environment for joint working applications. Along with an API, Habanero includes several applications for synchronous collaboration: video, audio player, chat, drawing tools, etc. Habanero’s unique feature is that it provides a set of classes and interfaces for porting single user applications to multi-user environments. This is done by using “wrapped objects”. Single user applications are thought to be executed simultaneously by all participants. If a class of an application imple-

*. JRE - Java Run-time Environment, contains all classes and tools required to execute Java code

ments the `Marshallable` interface, its objects can be sent across the network. Synchronization of GUI interactions is supported by the `Wrapped` interface. In addition, it is necessary to replace the event processing loop `handleEvent()` with the `doEvent()` and add the application to a special `MirrorFrame`. Unfortunately, the current release (1.0) supports only JDK 1.0.x event processing model. JDK 1.1.x will be supported in an upcoming release.

JavaSoft JSDT

JSDT - Java Shared Data Toolkit (former JSDA) - (`www.javasoft.com/people/richb/jsdt`), a new part of the JavaMedia API suite, JSDT implements a multipoint data delivery service for use to support highly interactive, collaborative applications.

The primary functionality provided by JSDT is a set of APIs for collaboration-aware Java code (*Client*) to send data (*Message*) to all (or a part) of the participants (also *Clients*) within a communication *Session*. This is accomplished by implementing the Observer-Observable model with a single send method. In addition, a *Token* abstraction provides synchronization and locking mechanisms. JSDT also includes a simple naming service that is required to locate a session. In comparison to Habanero, JSDT is a set of APIs only; it does not provide any ready-to-use applications. JSDT is very flexible and powerful, but unlike Habanero, it does not provide any guide-lines or APIs for porting single-user applications to multi-user environments.

For a long time JSDT in an unofficial public review and testing. Version 1.0 of JSDT has recently been moved to the Java Developer Connection pages on the JavaSoft web site. In addition to the JSDT API classes and documentation, the distribution includes some sample applications in source code: chat tool, sound player, stock viewer, white-board.

CONCLUSION

This article focused on technologies for synchronous joint-working with Java applications and applets. This part of CSCW tools is also known as “application sharing”. Application sharing systems can be divided in multiple and centralized execution systems. Multiple execution systems synchronize multiple instances of shared applications. Although very advanced prototypes [3] are available, no common solution for all synchronization problems has been found. Centralized execution distributes GUI presentation of one application instance to remote participants and, therefore, has no synchronization problems. Classic realization approaches provide very simple groupware support and require good network QoS parameters.

The Component Model Reflector (CMR) is a new approach to enable synchronous joint-working in Java. CMR has several features, which were impossible to implement in application sharing on traditional platforms. CMR requires only a few changes in JRE configuration files to be ready to share applications. Remote participants do not need to make any changes at all. Additionally, CMR is operational via slow network links and provides better group-support than traditional application sharing systems. Further improvements of CMR in connection with complex graphic applications and flexible joint-working network services require new APIs in JDK, if an implementation should require changes in system classes.

Single-user Java applications and applets can also be modified to support synchronous joint-working. Habanero provides a JDK 1.0.x-based environment for porting Java code to multi-user environments. JSDT by JavaSoft delivers a powerful API for multi-point communication. JSDT is very useful for implementing all classes joint-working services.

RELATED PUBLICATIONS AND REFERENCES

1. H. Abdel-Wahab, B. Kvande, *An Internet Collaborative environment for Sharing Java Applications*, 5th IEEE Workshop on Future Trends of Distributed Computing Systems (FTDCS’97), 1997
2. U. Gall, F-J. Hauck, *Promondia: A Java-Based Framework for Real-Time Group Communication in the Web*, Proceedings of WWW6, 1997
3. J. Begole, C. Struble, C. Shaffer, *Leveraging Java Applets: Toward Collaboration Transparency in Java*, IEEE Internet Computing, March-April 1997
4. IEEE Internet Computing, *Interview with Tom Malone: Free on the Range*, vol. 1, #3, May/June 1997
5. T. Graham, T. Urnes, R. Nejabi, *Efficient Distributed Implementation of Semi-Replicated Synchronous Groupware*, Proceedings of UIST 96, 1996
6. G. Krasner, S. Pope, *A cookbook for using the model-view-controller user interface paradigm in Smalltalk-80*, Journal of Object Oriented Programming 1(3), 1988
7. J. Lauwers and K. Lantz, *Collaboration Awareness in Support of Collaboration Transparency: Requirements for the Next Generation of Shared-Window Systems*, Proceedings of Human Factors in Computing Systems, ACM Press, New York, 1990
8. J. Lauwers, T. Joseph, K. Lantz and A. Romanow, *Replicated Architectures for SharedWindows Systems: A Critique*, Proceedings of Office Information Systems, SIGOIS Bulletin, (11)2,3, 1990
9. Microsoft, *NetMeeting*, www.microsoft.com/netmeeting
10. V. Minenko, J. Schweitzer, *An Advanced Application Sharing System for Synchronous Collaboration in Heterogeneous Environments*, SIGOIS Bulletin, (15)2, 1994
11. V. Minenko, *The Application Sharing Technology*, The X Advisor, 1995, re-published in MotifDeveloper, 1998, www.motifzone.com/tmd/articles/XpleXer/XpleXer.html
12. Sun Technology Enterprises, Inc, *The Complete Guide to ShowMe*, www.sun.com/products-n-solutions/sw/ShowMe/products/ShowMe_SharedApp.html