1998 International Computer Symposium
Workshop on Computer Networks, Internet, and Multimedia
December 17-19, 1998, N.C.K.U., Tainan, Taiwan, R.O.C.

USING RMI TO IMPLEMENT REMOTE CONTROLLER/VIEWER APPLETS FOR JAVA APPLICATIONS

Jr-Ren Tsai, Gwo-Cheng Chao, Daniel J. Buehrer

Institute of Computer Science and Information Engr.
National Chung Cheng University
dan@cs.ccu.edu.tw

ABSTRACT

It is well-known that X-windows has the capability to remotely display and control applications. This paper uses Java's Remote Method Invocation (RMI) technology to design a package that has a similar capability. By using this package, a server side Java application will be viewed and controlled remotely by an applet running on a Java-enabled browser. The Java programmer need not worry about the messages being passed between the client and server, since the message passing is hidden in the package. A Java application writer can first write a standard Java application, and then use our xawt package to replace the java.awt Abstract Window Toolkit (AWT) package. He need only make very minimal changes to his source code to create an RMI version of his application which can use a remote Java applet to display and control the server-side application.

1. INTRODUCTION

In recent years, World Wide Web (WWW) technology has been developing quickly. The Java programming language has become well-known, since its applets can run on the Web by using a Java-enabled browser. There are many advantages to the Java language, such as portability, pure object-orientation, support for multiple threads, reflective capabilities, serializability, interactive form and resource editors, ODBC and CORBA bridges, security cassettes and signature capabilities, internationalizability, etc.

Our goal is to allow a Java programmer to develop remote services by first coding a stand-alone application, and then replacing the standard **java.awt** window library by our **xawt** library, causing the program to be remotely displayed and controlled by a light-weight applet which is running on a Web browser. Such an arrangement has several advantages. First, the applet is very light-weight, and has low down-load time. The services provided by the server side can be much more powerful since there are no security constraints on Java stand-alone applications. The server can even use libraries implemented in other programming languages, such as C/C++. The Java stand-alone application can call these libraries via the Java Native Interface (JNI). The server can freely write to disk and can freely communicate with any other servers or agents on the network, unlike applets, which can only communicate with the host from which they were loaded.
The Java Development Kit 1.1.x, a Java language standard, includes Remote Method Invocation (RMI), Java's enhanced platform-independent, object-oriented remote procedure call methodology. The important contribution of this paper is to develop a methodology for transforming a standard Java application, which can run on a single machine, into a client-server application which uses RMI calls to a remote lightweight Java applet. This will be done by implementing an **xawt** package which will replace the standard **java.awt** package. Our goal is to minimize the changes in the source program that are necessary to transform it into a client-server application.
As well as displaying the window objects remotely, the low-level keyboard, mouse, and other events are handled on the client side, and the higher-level events are returned to the server, also via RMI calls. Thus, client-server programmers no longer need to be concerned with the complexities of the messages between the client and server applications. Moreover, clients can rely on the security of Java applets, since the applets theoretically cannot look up or destroy other information on the client machines.

2. BACKGROUND

It is important to first understand what Java applets can and can't do, as well as some details about Java's message passing and inheritance mechanisms. This section will briefly describe these details.

2-1 What Applets Can and Can't Do

There are many things that an applet cannot do, but which a stand-alone Java application can do. That's why we want to use applets, which can be safely executed anywhere on a browser, to control and view a stand-alone application that has no security restrictions. Even though browsers such as Internet Explorer™ (a registered trademark of Microsoft) and Netscape™(a registered trademark of Netscape) now permit signed applications to be downloaded and run on the client, this could prove to be very dangerous if these applications are widely used. Can you imagine what would happen if some day a disgruntled employee at some famous company would put a virus into their Java spreadsheet application? The other main advantage to our approach is that the client applets are all very lightweight, and are suitable for network computers. They can be downloaded very quickly, unlike applications.

Although each browser has its own implementation of security policies, basically Java applets are not allowed to access the client machine's disk or execute native methods on the client's machine[10]. They can only make network connections to the host from which they

were loaded. The applet windows have a special look so that the user can tell they are not completely trustworthy. Applets can, however, display HTML documents, and can call public methods of other applets on the same HTML page that they were loaded from.

2-2 Abstract Window Toolkit (AWT)

The Abstract Window Toolkit (AWT) is a Java package providing powerful GUI classes, such as GUI components (e.g. Buttons, Frames, TextFields, etc.), Event classes (e.g. ActionEvent, AdjustmentEvent, etc.), Event Listeners (e.g. FocusListener, ItemListener, KeyListener, MouseListener, etc.), and other helpful classes (e.g. Font, Image, Point).

2-2-1 AWT GUI Components, Peers and the Java Virtual Machine

Peer classes are interfaces to native code that the AWT GUI components can call. For example, java.awt.Button has a variable of class type java.awt.peer.ButtonPeer. When we call a method that depends on the GUI of a Button, such as setLabel("test"), the Button passes the request to the method setLabel("test") of its own ButtonPeer. Then this ButtonPeer passes the request to a machine-dependent code to change the label of the Button. Going the other direction, when a user clicks the mouse on this Button, an operating-system-dependent window message occurs. The Java VM receives the message and lets the peer call the method defined in the Button class [4]. As of JDK version 1.1, programs should not directly manipulate peers.

2-2-2 Packages and Inheritance

A package contains classes that are related to each other. A package is represented as a directory of an operating system. That is, the directory of a package named **xawt** must be put in the directories of the ***CLASSPATH*** environment variable. Java source code which uses the classes in the package must contain the statement ***import*** **xawt.*;** For instance, in windows95/NT systems the autoexec.bat file might contain the command:

set CLASSPATH=.;c:\mypackages;

Then, the class files of the **xawt** package must put into either a subdirectory named xawt of the current directory or the c:\mypackages\xawt directory. It is quite reasonable that an **xawt.event** package would be put into a subdirectory named **event** of the **xawt** directory [7].

.Name collisions

Suppose there are two classes named Button, one in the java.awt package, and one in a subdirectory of current directory named xawt. Suppose that a java source program has the following lines in it:

```
import java.awt.*;
import xawt.*;
....
/* Here is some code using Button */
Button b = new Button();
```

Then a name collision error will occur since the compiler does not know which **Button** class is to be chosen. Here are some solutions when we want to choose the **Button** in the **xawt** package:

```
(1)
import java.awt.*;
import xawt.*;
....
/* Here is some code using Button */
xawt.Button b = new xawt.Button();

(2)
import xawt.Button;   /* add this line */
import java.awt.*;
import xawt.*;        /* this line can be removed
if there are no other classes in the xawt package which
will be used except Button */
....
/* Here is some code using Button*/
Button b = new Button();
```

Figure 2-1 Alternative methods of importing xawt.Button

One goal of this paper is to change user source code as little as possible, so solution (2) is the best choice. In the example above, if many new Button objects are defined, solution (1) requires us to change much source code, while solution (2) does not.

Modifiers

For our concerns, the following is an important feature of the AWT GUI components:

> *In AWT, all **variables** are defined as **default**, **private** or **public static final**. In other words, it is impossible to change variables inherited from a superclass directly in other packages, except by using methods which are inherited from the superclass.*

2-3 Message Passing

Delegation-Based Event Handling Model

Each component in JDK 1.1.x has methods named ***addXXXListener*** and ***removeXXXListener*** (where XXX is one of a specified set of Events) to add or remove a Listener Object. Several listeners may be delegated to listen to the same events, and they will all be called when that event occurs.

Event Masks

When we call ***addActionListener()*** of a Button, the ***ACTION_EVENT_MASK*** of the Button will be enabled. Similarly, ***removeActionListener()*** will disable the mask. When a user clicks on a Button, if ***ACTION_EVENT_MASK*** of the Button is enabled, ***processActionEvent(ActionEvent e)*** of the Button will be called. If the ***ACTION_EVENT_MASK*** is disabled, the click will have no effect. Besides ***addXXXListener / removeXXXListener***, there are methods named ***enableEvents(long eventsToEnable) / disableEvents(long eventsToDisable)*** to enable or disable the event mask.

2-4 Remote Method Invocation

Remote Method Invocation (RMI) allows

programmers to create distributed Java-to-Java applications, in which methods of remote Java objects can be invoked from other virtual machines, even on different hosts and operating systems [2]. A special case is when a client and server mutually call each other's methods [2].

.RMI Architecture

RMI can dynamically call remote methods. For example, given a method name, RMI can dynamically create a stub with the appropriate argument types and call the corresponding skeleton on the other side. These arguments and values may even make use of Java's methods for type checking and type casting.

3. SYSTEM ARCHITECTURE

3-1 Structure Overview

First, let us look at the main ideas of the **Controller/Viewer Applets** and the **Application Server** as illustrated in ***Figure 3-1***. This Figure shows the Server Side **Application Server** providing launching services for user applications. The dotted arrows show that Applet 1, with **client id 1**, has launched two applications, **ApB** and **ApE**, and Applet 1 is controlling and viewing instances of these two applications.

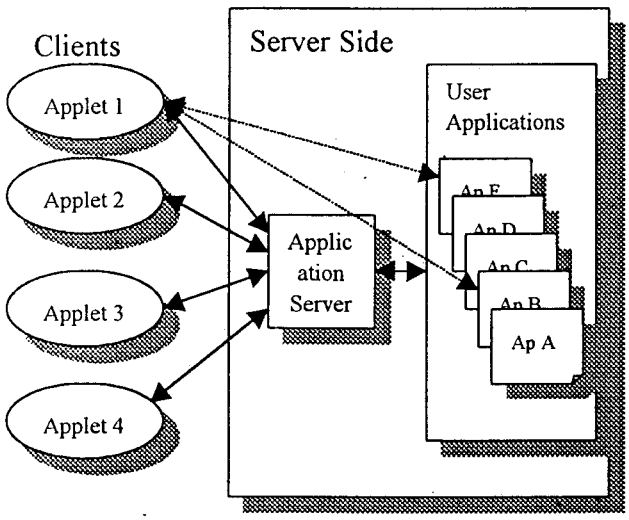

Figure 3-1 Controller/Viewer Applets and Application Server

Second, ***Figure 3-2*** gives a detailed view of the relationship between ***Controller/Viewer*** Applet 1 and ApB. For Every GUI component of ApB, Applet1 creates a similar component. These pairs of components must cooperate with each other.

Although the xawt package uses same set of names as components in the AWT GUI, we use the names SFrame, SButton, SLabel instead of Frame, Button, Label here in our explanations, to distinguish between the xawt package and the java.awt package. **C**Frame, **C**Button and **C**Label are the client side components that SFrame, SButton and SLabel will cooperate with. Both the server side and the client side components extend the components from the java.awt package. For instance, SFrame and **C**Frame are subclasses of the java.awt.Frame.

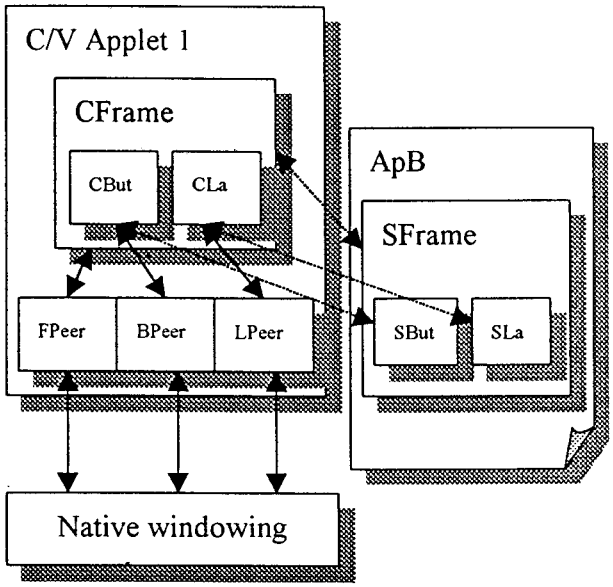

Figure 3-2 Detailed view of a controller/viewer applet and a launched application

Figure 3-3 shows the structure of the cooperation between client side and server side components. We give a simple explanation for this Figure here:

1. ***Default AWT method:*** call the original methods in the java.awt package.
2. ***Calling an overridden method:*** when a subclass extends our component, a method call in a client must be redirected to the overriding method of the subclass running on the server, not the original mehtod inherited from the superclass.
3. ***Listener calls:*** when a Listener object is added to a component, we must let the client know about the Event, since the control of that component is in the client.
4. ***Event processing calls:*** when the client side component receives an Event, it must let the server side component know about it so that it can notify the Listener Objects.

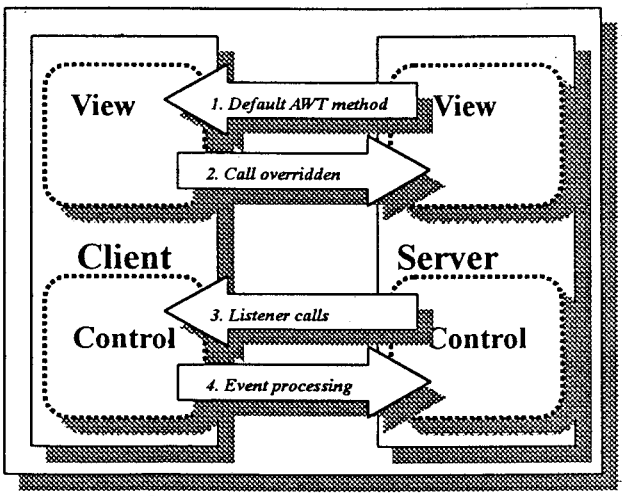

Figure 3-3 Structure of cooperation between client side and server side

3-2 Cooperation between the client side and server side components

In our model, we must separate each component **Comp** into **CComp** and **SComp** running on the client and the server, respectively. The relations between them are described in ***Figure 3-4***. The labels **1, 2, 3, 4** here have the same meaning as in ***Figure 3-3***.

1. Variables related to Event Masks:

We keep the event masks on the client side **CComp** to let Controller/Viewer Applet know which Events of **Comp** we care about and which we don't. The server need not know about the Mask behavior.

2. Variables related to Event handlers:

When Events occur, only the server side component knows which listener objects should be notified. So, we put the variables regarding the Listener objects in **SComp**.

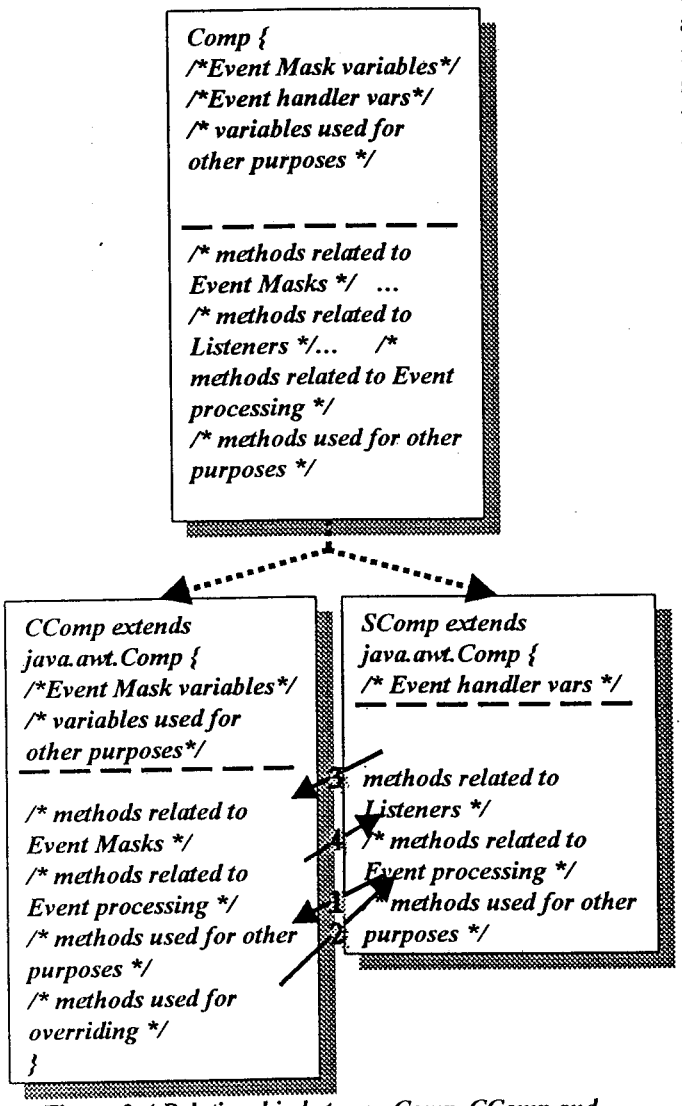

Figure 3-4 Relationship between Comp, CComp and Scomp

3. Variables used for other purposes:

For the variables not belonging to *1* or *2*, when showing the **CComp** in the Controller/Viewer Applet, some variables may be used by the methods doing the showing. If we were to keep the data in **SComp**, when the methods are called in **CComp**, they would use the unknown values of the variables, since the variables have not been assigned by the application. That's why we keep these variables in **CComp**.

4. Methods related to Event Masks:

When ***addXXXListener(l)/removeXXXListener(m)*** of **SComp** are called by an application, the ***XXX_EVENT_MASK*** of **CComp** must be enabled/disabled (for the same reasons as described in **1, 2**). Here, we mean the methods named ***enableEvents(long eventsToEnable) / disableEvents(long eventsToDisable)*** described in Section 2-3-2.

5. Methods related to the Listeners:

These methods are named

addActionListener(ActionListener m)

and ***removeActionListener(ActionListener m)***,

as described in Section 2-3-1. We keep these method calls in **SComp**. The meaning of **3** in ***Figure 3-4*** is that when some Listener Object is added to/removed from **SComp**, we must enable/disable the Event Mask in **CComp** to let the Controller/Viewer Applet handle/ignore the Events.

6. Methods related to Event processing:

Now, let's see the meaning of **4** in ***Figure 3-4***. When some Event occurs, **CComp** will check the Event Mask. If it had been enabled, **CComp** will redirect the method call from **CComp** to **SComp** to let **SComp** notify the Listener Objects.

7. Methods used for other purposes:

(A) Methods used for other purposes: **1** in ***Figure 3-4*** says that when an application calls SComp.compMethod(...); we redirect it to CComp.CcompMethod(...); where CcompMethod looks as follows:

```
CcompMethod(...) {
super.compMethod(...);      /* call the default method */
}
```

If this method changes some variable, it will affect variables in CComp instead of in SComp. (This will let **3** be executed in ***Figure 3-5*** on the next page.)

Step **4**: redirects the method call to CComp.

Step **5**: calls the default compMethod of Comp.

(B) Methods used for overriding: **2** in ***Figure 3-4***: If some subclass of SComp named UserComp overrides the compMethod(...), the calls between CComp and SComp and UserComp will be as shown in ***Figure 3-5***. (Here, the application creates a component of type UserComp, not SComp. CComp will cooperate with UserComp).

Step **2**: calls the overriding method

Step **3**: occurs when the overriding method calls the super method

Step **1**: occurs when other methods of CComp call compMethod

3-3 Inheritance Problem

Section 3-2 describes some, but not all of our system. A subclass can do things to its superclass that an instance can't, such as calling a protected method. That is, if a subclass of the component in our package can keep the same behavior as a subclass of the component in java.awt

package, an instance of that component can replace the jave.awt component. Here, in this section, we will focus on the override problem of applications using our package. We will use the symbols Comp, SComp, CComp, UserComp that we used in Section 3-2.

3-3-1 For Variables, Is It Safe to Create a Subclass of an xawt Component?

From Section 2-2-2, we know that for an AWT GUI component, all variables are defined as **default**, **private** or **public static final**, and it is impossible to change variables inherited from an AWT GUI superclass directly, except by using methods which are inherited from the GUI superclass. This property is consistent with the constraint that variables are kept on client side, as discussed in Section 3-2. When a subclass (UserComp) wants to change variables, it must call ***...super.method(...);***. The superclass will then redirect this request to client side component (CComp).

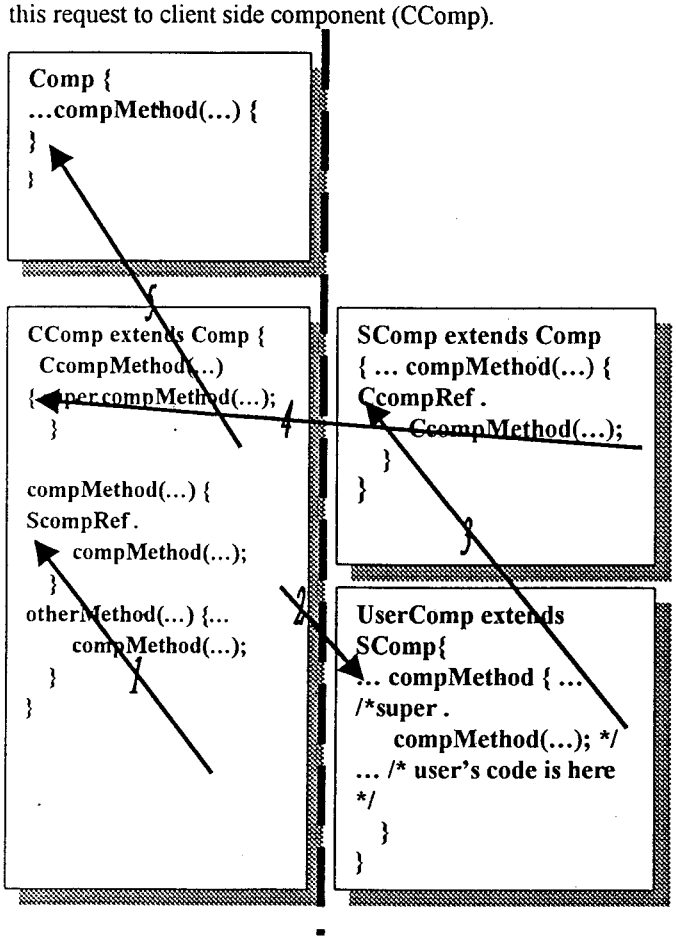

Figure 3-5 Solution for the overriding problem – complex case

A subclass can have its own variables, and these variables are kept on the server side. The client need not know about them. The client-side component only provides default methods and variables inherited from the java.awt package.

3-3-2 Is It Safe to Override Methods Inherited from an xawt Component?

Let's see ***Figure 3-5*** again. We will take a closer look at the problem of overriding methods:

Case 1:

See ***Figure 3-6***. In Comp, if methodA does not appear in any other methods of components in the java.awt package, we can say that all method calls of methodA must be made on the server side (we do not discuss the case of event handling here). When overriding occurs, if methodA of UserComp calls ***super***, the calls will be 1 --- 2 --- 3. If we do not call ***super*** or other inherited methods, the calls will stop at the server since no variables defined in Comp will be changed by the override method of UserComp. Of course, if some method calls **super.methodA(...)**, the calls 2 - 3 will occur and may change some variables, as for the default behavior of methodA in Comp.

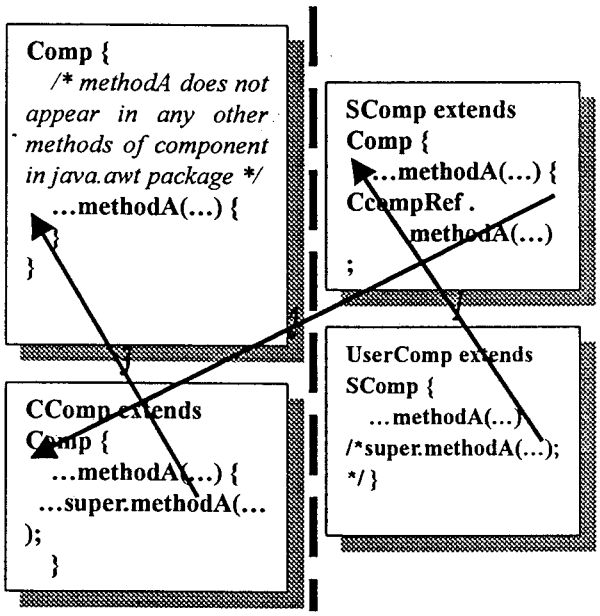

Figure 3-6 Solution for the overriding problem – simple case

Case 2:

See ***Figure 3-5***. In this case, compMethod(...) and otherMethod(...) are defined in Comp. Our CComp inherits Comp. In order to call the overridden code defined in UserComp.compMethod(...), we redirect the call as 1---2. A method named CComp.CcompMethod(...) keeps the default behavior that the Comp.compMethod(...) has. If ...UserComp.compMethod(...) calls super, the calls beginning at 1 will be 1 - 2 - 3 - 4 - 5.

3-3-3 Restrictions of RMI — a Solution for Protected Methods

In our model, the calls between the client and server are based on RMI technology. RMI methods must be defined as public methods. This is not sufficient for our use because of the existence of protected methods in components. We use a public method as a proxy for each protected method (see Figure 3-7).

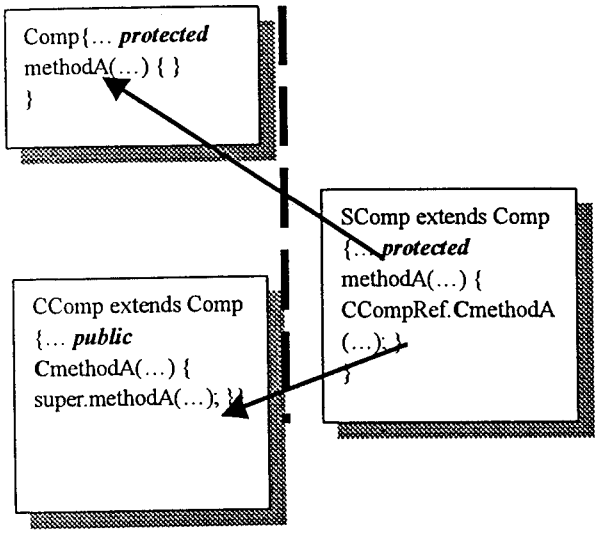

Figure 3-7 The solution for protected RMI method calls in our model

3-4 Add / Remove a Component in a Window

We will describe the add/remove problem by using the Frame and Button components. Other components can be treated in a similar manner. Label, TextField, etc. are like a Button and FileDialog is like a Frame. A Frame is shown on the screen by calling the **show()** method of the Frame, but a Button cannot be shown before it is added to a window such as a **Frame**.

In our model, we must let SFrame and SButton know where the client is so that they can cooperate with CFrame and CButton of the Applet running on the client. See the following code segment, where again, to distinguish our package and java.awt package, we use the symbols SFrame and SButton to denote Frame and Button provided in our package :

1. normal application:

```
import java.awt.Frame;
import java.awt.Button;
...
    Frame f = new Frame("Original Frame");
    Button b = new Button("Original Button");
f.add(b);
f.pack();
f.show();
...
```

2. application using our package:

```
    import xawt.Frame;
    import xawt.Button;
    ...
Frame f = new Frame("OurFrame");
Server.registerFrame(Client, f);  /* register a frame to a
specific client */
    Button b = new Button("OurButton");
f.add(b);    /* Server.registerButton(Client, b) hidden in
this method call */
f.pack();
f.show();  ...
```

After getting the **Client** to launch the application, besides importing our package components to substitute for those in the java.awt package, only one line must be added --- **Server.registerFrame(Client, f);**. The **show()** must be done on the Client. The **registerFrame** will create a CFrame object running on the client Applet and the cooperation between the SFrame object and the CFrame object begins. Besides, the SFrame object will save the **Client** object in a private variable. When adding the Button b, method **Server.registerButton(Client, b);** will be called by the add(...) method.

```
... SFrame.add(...)  ...
/* the Client is saved in a private variable of the SFrame
*/
Server.registerButton(Client, b);          ...}
```

A CButton object will be created on the client Applet, and the cooperation between SButton abd CButton begins.

Everything seems to be ok, except if we add lines that change the behavior of SButton b before it is added to the Frame f:

```
Button b = new Button("OurButton");
b.setLabel("New Label");  /* user may add lines here */
f.add(b);
```

The method setLabel(...) will change the label of SButton b to a new one. But there is no client side CButton object to cooperate with, since b does not know who the client is yet (In fact, this problem also occurs when constructing the CButton object, since we don't know which constructor and arguments to use when creating SButton). If we kept the label in SButton, the rule that variables must be kept in the client side component in our model will be broken (see Section 3-2). Two possible solutions to this problem are:

1. Tracking the methods that have been called before the add(...):

There are plenty of methods existing in a component. If many methods have been called, we must keep the correct order of these method calls and the arguments they used. After add(...) is called, do the same method calls to CComp. This solution is quite expensive in terms of overhead.

2. Copying variables after the add(...) takes place:

After adding b to f, variables that have been changed must be retrieved from SButton and sent to CButton. This is impossible since some variables are private and no method can retrieve their values.

Neither 1 nor 2 is a good solution. To understand the solution we provided, we must take a closer look at what **register** does:

```
CComp extends Comp{
  private SCompRef proxy;
  public void setProxy(SCompRef s){
    proxy = s;
  }
}
```

```
SComp extends Comp {
   private CCompRef proxy;
   public void setProxy(CCompRef c) {
      proxy = c;
   }
}
```

Server.registerComp(Client,SCompObject) will:

1. *new* a CCompObject in Client.
2. The cooperation between the client and server using RMI is created by writing:

CComp.setProxy(SCompObject);
SComp.setProxy(CCompObject);

That is, they get the reference to each other and use the reference as the object itself. All redirect method calls between the two objects is done by calling ***...proxy.method(...);***

We introduce our **Fake Proxy** idea that we use:

```
SComp extends Comp {
   private CComp fakeProxy;
   private CCompRef proxy = fakeProxy;
   public void setProxy(CCompRef c) {
      proxy = c;
   }
}
```

We initialized the proxy variable to a CComp object **fakeProxy** in **Server**. Before the f.add(b); takes place, method calls and variables that are changed will affect the fakeProxy. When f.add(b); occurs, we use the ***Object Serialization technology*** to send a **copy** of fakeProxy to the Client Applet, and store it into a running CButton object. Then, call the **setProxy** as specified above to make the connection between the SButton object and CButton object.

The case of removing a component from a window has a behavior which is similar to that of adding a component. Before removing, we must use ***Object Serialization technology*** to send the CComp object running in the client Applet to the server and store it into the **fakeproxy** variable of the SComp object. Then, call the **setProxy** to make the connection (start the redirect mechanism) between the SComp object and CComp objects running in the **Server**.

4. MAKING THE LIBRARY EASY TO USE

In this section, we focus on the restrictions that programmers must know about when using the package we provide. For convenience, we discuss a problem with two components --- Frame and Button.

4-1 The Problem of Importing

One goal of this paper is to change user source code as little as possible. We provide an alternate set of AWT GUI components. With this approach, one application can use the classes we provide by writing the following code:

import xawt.Button; /* xawt is the name of the package we provide */
import xawt.Frame;
...
import java.awt.*;
import java.awt.event.*; ...

Note that there exist many classes that we do not need to provide, such as **java.awt.Font, java.awt.event.*, java.awt.AWTEvent,** etc. This is because a client Applet only uses the default superclass behaviors, and applications which define new subclasses eventually just invoke these superclass behaviors, which are provided by the client's browser. Otherwise, instances are automatically serialized and sent as an argument to the client by RMI's message-passing mechanism. For example, a font is an object which can be serialized and sent to the client. This solves the problem in X-windows, where a client is unable to display some text because the client has a different set of fonts than the server.

We must import **java.awt.*** for the server application to use these classes which **xawt** does not provide. Because of the name collision problem discussed in section 2, we cannot just write:

import xawt.*; /* xawt is the package name we provide */
import java.awt.*;
import java.awt.event.*;
...

4-2 The Problem of Passing the Client to an Application

One important fact is that we must let an application using this package know where the client applet is. We must pass the client to the application when it is launched. A Java application starts with the method:

public static void main(String args[]) {...}

If we were to keep the **clientid** (in string format) in args[0], much code would need to be changed (i.e. we must change args[0] to args[1], args[1] to args[2], and so on). Hence, we pass **clientid** to an application in the last element of the **args[]** array in our system. Programmers only need to skip the last element of this array. That is, **args.length–1** is the actual count of the number of arguments passed to the application and **args[args.length-2]** is the last argument which the application programmer is concerned about. After passing the **clientid**, the following code is needed:

ControlViewInterface Client =
Server.getControlViewInterface(args);

Here, **Client** is the reference to the client launching it. **getControlViewInterface** is a static method of the application **Server** to be used to translate the **clientid** into a client *reference*.

4-3 The Problem of Showing a Frame, Dialog, etc.

One main difference between a Frame and a Button is the "show" method. A Frame can be shown on the screen by calling the **show()** method of the Frame. A Button cannot be shown before it is added to a window such as a **Frame**. When we want to show a Frame in an application using our package, we must let the client know about it, since the showing must done on the client. So, after

constructing a Frame, we must register it with the **Client**:
Frame serverSideFrame = new Frame("test");
Server.registerFrame(Client, serverSideFrame);
Why don't we combine these two lines into one line as follows?
Frame serverSideFrame = new Frame(Client, "test");
If we were to do it this way, every constructor of Frame would have to take an extra argument --- **Frame(ControlViewInterface Client, ...)**, and some special code would have to be added to each constructor to handle the client side behavior. Subclasses of the Frame would need to do this, too. This approach is not a good idea since much code would have to be changed.
In the case of Button, we can hide the similar code:
Server.registerButton(Client, serverSideButton);
We put such calls in the **add(...)** methods of the Frame. So, there is no need for the programmer to change his source code for these components. Note that for a source file, the lines to "**import**" and "**get Applet reference**" only need to be added one time. Most applications do not have many frames, so not very many "**register**" calls are necessary. Hence, for a typical application with thousands of lines, there may be only about twenty lines that need to be added (changed).

4-4 Performance Issues

(1) Our system will be implemented by using threads:

There may exist many clients running at the same time, and components may be created simultaneously. Assigning at least one thread to each client will improve the performance of the server [1].

(2) Downloading the xawt package in advance can speed up the applications:

For a user of our system who wants to launch applications, he can download the ***xawt*** package in advance, and set the CLASSPATH environment variable of his system to let the browser use the ***xawt*** package that he downloaded. When running applications, the classes defined in the ***xawt*** package then need not be passed through the network because of the already-existing downloaded ***xawt*** package.

(3) No user-defined classes need to be passed to client:

Remember that a client Applet only provides the default behaviors the GUI classes in the java.awt package have, and applications are actually run in a server. There is no need for an application to pass user-defined "classes" to client Applets ("objects" of default class types and their subclasses will be passed if they appear in arguments of some default method).

(4) In most situations, one application does not contain many GUI components to be shown:

A typical application does not usually contain many GUI components. In other words, the total cost for creating and running client-side GUI components is not expensive.

5. SUMMARY AND CONCLUSIONS

By using this package, a server side Java application can be viewed and controlled by an applet via a Java-enabled browser. The programmer need not worry about the messages being passed between the client and server, since the message passing is hidden in the package.

```
import xawt.Frame;
      /* solve the name collection problem */
import xawt.Button;
import xawt.*;
import java.awt.*;
public class testApp {
   public static void main(String args[]) {
         ControlViewInterface Client =
            /* get the Applet reference */
            Server.getControlViewInterface(args);
         Frame f = new Frame("test");
         Server.registerFrame(Client, f);
/* after this line, we can treat f as a normal Frame
   nothing special to do with a Button*/
            Button b = new Button("This is a Button");
            f..add(b);
            f.pack();
            f.show();   } }
```

For a Java programmer who wants to provide services, if he chooses to use an applet and put it in the Web, the security constraints of the applet may confine the development of the application. Furthermore, large programs may have terrible download time. By using this package, we can use lightweight components to control and view the applications instead of downloading them. Besides, there are many already existing Java applications in the world. By re-compiling the source files with our package, the applications can be controlled and viewed on the web browsers easily.

Our **xawt** package has the same class interfaces as the original **java.awt** package. This means that users can use our package as the default package without any difficulty.

REFERENCES

[1] Doug Lea, Concurrent Programming in Java, 2nd Ed. Addison-Wesley. ISBN: 0201310090, Dec 1997.

[2] Ann Wollrath, Jim Waldo, Roger Riggs, "Java-Centric Distributed Computing JavaSoft," IEEE Micro, May/June 1997, 0272-1732/97.

[3] Jamie Jaworski, Java Developer's Guide, Sams.net. ISBN: 157521069X, 1996.

[4] Nataraj Nagarathnam, et al, Java Networking & AWT API SuperBible, Waite. ISBN: 157169031X, 1997.

[5] Paul Tyma, et al, Java Primer Plus, Waite. ISBN; 157169062X, 1997.

[6] JDK 1.1 新境界, 黃昕暐著, 旗標,1997

[7] Sun Microsystems Java HomePage http://java.sun.com

[8] James Gosling, et al., The Java Application Programming Interface Vol. 1: Core Packages, Addison-Wesley. ISBN: 0201634538, 1996.

[9] James Gosling, et al., The Java Application Programming Interface Vol. 2: Window Toolkit and Applets., Addison-Wesley. ISBN: 0201634597, 1996.

[10] Mary Campione & Kathy Walrath, The Java Language Tutorial: Object-Oriented Programming for the Internet, 2nd Ed., Addison-Wesley,Jan 1998.

[11] Glenn Vanderburg, et al., Tricks of the Java Gurus Sams.net. ISBN: 1575211025, 1997.

[12] John Rodley, Writing Java Applets, Coriolis Group, ISBN: 1883577780, 1996.