Prasun Dewan[1]

4. Remote Method Invocation

In the previous chapter, we saw how programming patterns and notifications based on them support fine-grained collaboration functions and communication efficiency. We must now address the problem of how distributed controllers, models, and views invoke operations in each other. We will look in depth at the general concept of remote method invocation, also called remote procedure call (RPC), and will illustrate it using the concrete example of Java Remote Method Invocation (RMI). Not all ideas presented here are actually implemented in Java RMI – therefore it is important to separate the general concept from its particular implementations. There are several other popular examples of remote method invocation including SOAP/WSDL and Google's GWT remote procedure call. We will, in fact, incrementally derive the notion of remote method invocation from first principles.

Layers of Distributed Communication

When an object invokes an operation in another object, the former is called the client object, and the latter is called the server object. A single object may play the role of both a client and server object. For example, when a view reads the state of a model, the view and model become client and server objects, respectively. Their roles are reversed when the model sends a notification to the view.

A variety of mechanisms exist today to support distributed communication, which can be grouped into three main layers, as shown in the figure.

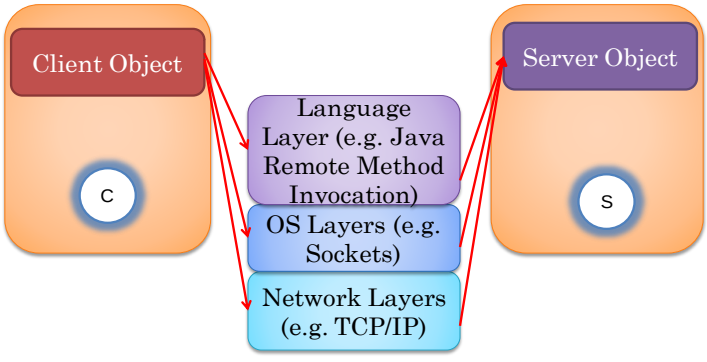


Figure 1 Communication layers

[1] © Copyright Prasun Dewan, 2009.

We could use one of the layers such as IP, UDP, or TCP/IP provided by networking software. Built on top of these layers are OS communication layers such as sockets. Even higher-level mechanisms are provided by some distributed programming languages, which tie the communication to programming language types. The most popular of these mechanisms is the concept of remote method invocation, which is implemented by Java RMI (Remote Method Invocation). In general, the higher the layer (a) the easier it is to use, and (b) the more the OS and Language interoperability supported by it. The networking layers can be used to communicate between, for example, an operating system supporting sockets and one that does not. They do not rely on the exact communication abstraction supported by the operating system. The OS layers, on the other hand, can be used to communicate between different languages. Language layers are tied to a particular programming language.

We will focus on remote method invocation for two reasons. First, low-level communication details are not the subject of this course, and are covered in other courses. Second, the notion of remote method invocation is the basis to some of the collaboration-specific abstractions such as broadcast methods, and by studying it we will get a better understanding of these abstractions. Here we will use the term remote method invocation for both the abstract idea of invoking methods in remote server objects and the concrete implementation of this idea in Java RMI (remote method invocation).

The reason why remote method invocation is high-level is that it has the same syntax as local method invocation. For example, a client object that wishes to invoke the parameterless method, magnify(), in the server object, remoteModel, uses the following syntax:

```
remoteModel.magnify();
```

From this call, we cannot even tell that remoteModel is on another machine. This is an important feature, as it is possible to substitute a local object with a remote one at program writing time.

However, as we see below, this does not mean that the application programmers and even the client and server objects can be oblivious to distribution issues.

Registering and Looking Up Remote References

When we wish to invoke a method on an object in the same process, the reference to it exists in that process, as some component of the process instantiated the object. This is not the case in remote method invocation as an object in one process wishes to reference an object in another process. How then is the remote reference obtained?

Remote method invocation is a special case of distributed inter-process communication. In general, when two processes wish to share some information, they give it a textual name, which is then mapped to internal objects. For example, processes sharing a file name do it by using a textual name. This same principle is used in remote method invocation. As the client and server processes may not share a common file system, a special name server process, called a *name server* or *registry*, rather than the file system are used to translate between internal addresses and external text names.

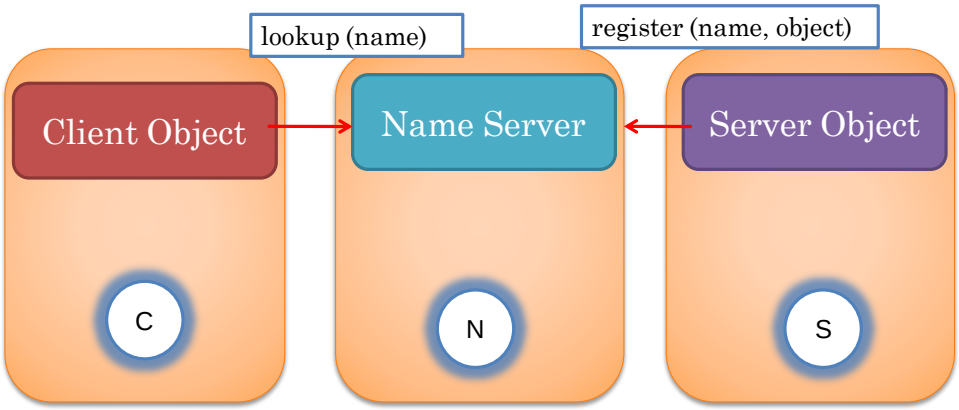


Figure 2 Name server registers and looks up remote objects

The name server allows (a) a server object to register an object under some text name, that is, map an internal address to an external reference, and (b) lookup a registered reference, that is, convert the external reference to an internal address. These two operations correspond to OS calls that convert between internal file descriptors and external file names. Just as two processes use different file descriptors to refer to the same file, the client and remote object use different internal addresses to reference the same object. In fact, as we will see later, the client and server object references will (directly) point to different kinds of objects.

A file system handles files of predefined types such as regular files, devices and directories. A name server, on the other hand, manages objects of programmer-defined types such as ConcertExpense. Therefore, the name server must have access to these types. In Java, that means putting these types in the class path of the name server.

The following Windows commands make the notion of a name server and setting is class path concrete.

```
set javabin=D:\"Program Files"\Java\jre1.6.0_03\bin
set CLASSPATH=D:/dewan_backup/java/concert/bin
%javabin%\rmiregistry 1099
```

startRMI.bat

Figure 3 Starting a name server with the right environment

Here rmiregistry is the name of the Java name server. javabin is the name of a command variable that is set to the location where the name server executable resides. Before we start the executable, we set the value of the environment variable, CLASSPATH, to the directory (directories) in which the classes of the objects that will be registered reside. If a class is in a package, the CLASSPATH should refer to the folder containing the root package. Java allows multiple name servers to execute at on the same computer. A port number, passed as an argument to rmiregistry, distinguishes among these name servers. 1099 is the default port number for rmiregistry.

Rather than repeat the steps above each time we start the name server, we can store them in a file and simply execute it. In Windows, executable files must have the suffix ".bat."

After a name server has been registered, client and server objects can invoke the lookup() and register() operations on it. However, this raises a recursive "meta" question, how do these objects get references to it and invoke operations on it? A "meta" question asks how one invokes some function on the agent that implements the function. There are several meta questions we face in other fields and in computer science: How is an OS bootloader itself booted? How is the police itself policed? The resolution to this "meta circularity" is to provide a special implementation of the function that is different and typically smaller from the implementation of the agent. In our case, we will rely on the fact that the name server implements predefined operations. As a result, we can use special mechanisms to reference it and invoke these operations.

In Java, these mechanisms are provided by a library, as shown in the examples below.

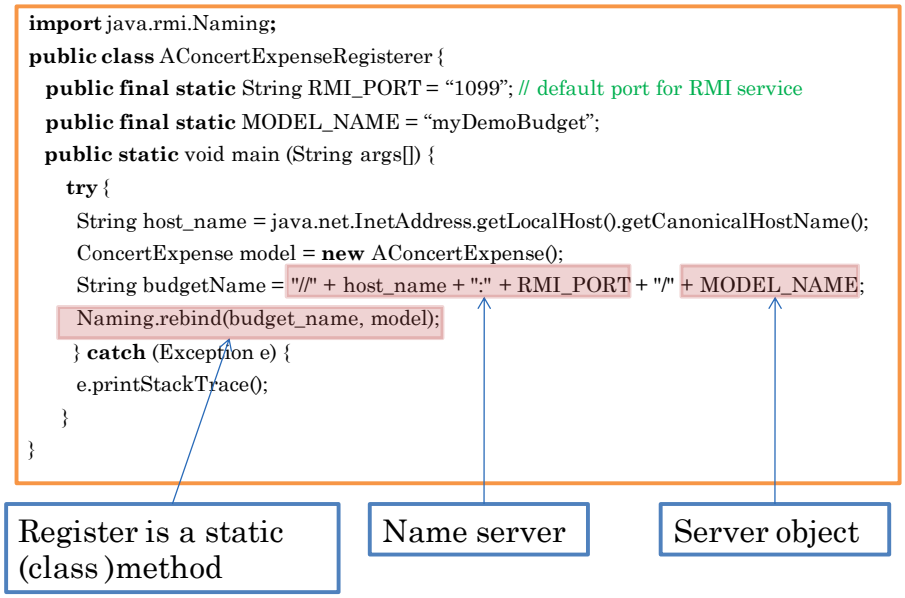


Figure 4 Registering an object

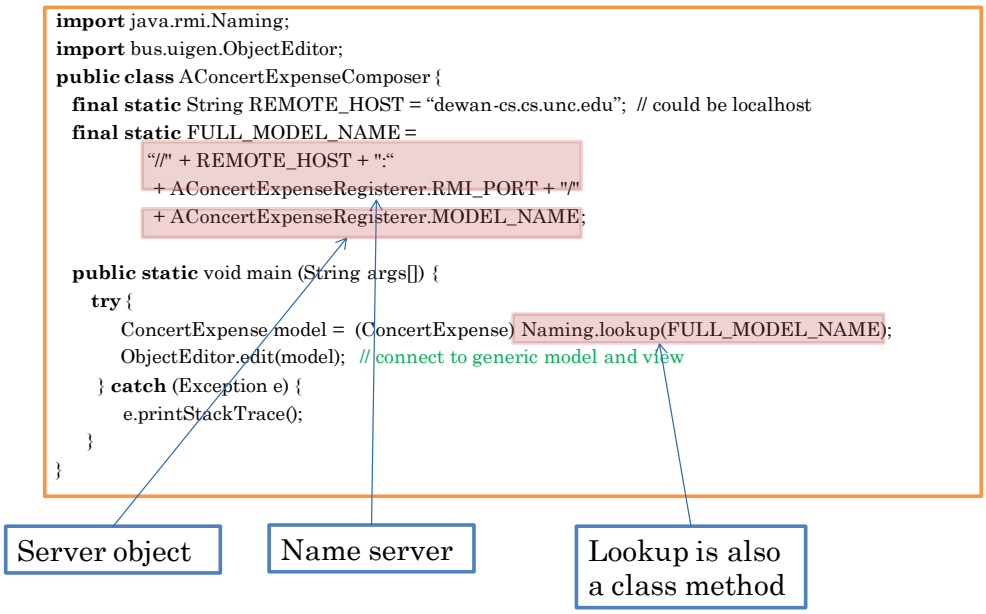


Figure 5 Looking up a registered object

Here Naming is the library class that provides the register() and lookup() methods. These two methods are static or class methods of the library – they are invoked directly on the library rather than on its instances. The former is called rebind() since it may be used to both give the initial name and change the name of the object. A name server is referenced by concatenating its domain (DNS) name and port number. Adding the text name of a registered object to the name server reference identifies the object.

Distinguishing between local and remote method calls

The following example illustrates the difference between local and remote method calls to an object of some type T. In the local case, the object was instantiated by the same process. In the remote case, it is received from the name server. However, in both cases, the method invocation itself is identical, and the object on which the method is called is typed the same way. As the compiler does not explicitly specify the destination of a model, the implementation must do so.

```
ConcertExpense model = (ConcertExpense) Naming.lookup(FULL_MODEL_NAME);
…
System.out.println( model.getNumberOfAttendees());
```

One approach to destination identification is to store, as part of the object header, the location of an object. Every object has a header that identifies meta information about the object such as its class and hashcode. The location would be another component in the header.

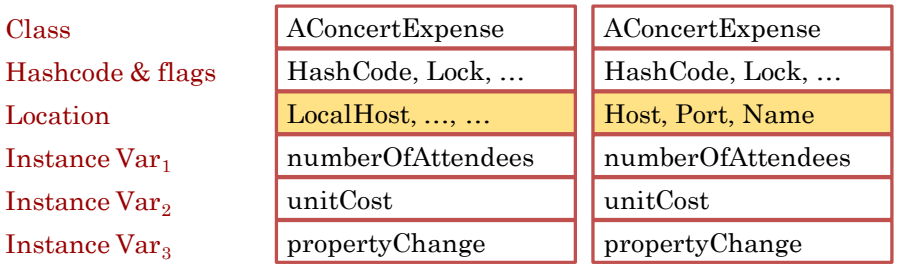


Figure 6 Storing location in the header

A problem with this approach is that all objects must store this information, even those that will never be accessed remotely. In particular every instance of Integer, Boolean, and other small objects must be allocated space needed to store location information. The overhead of an object header is a major issue in any object-oriented programming language. The reason why some languages such as Java distinguish between primitive values and objects is that the former are not associated with the object header.

One way to address this issue is store this information only for instances of those classes whose instances may be accessed remotely. We would require that such classes be made subclasses of some predefined remote classes such as the Java UnicastRemoteObject, whose instance variables store location information, as shown below. This means location information is not kept with classes such as Integer and Character that are not subclasses of these predefined remote classes. It is now kept in the instance variables of the predefined classes.

```
public class AConcertExpense extends  UnicastRemoteObject implements ConcertExpense
      float unitCost = 0;
      int numberOfAttendees = 0;
      …..

}
```

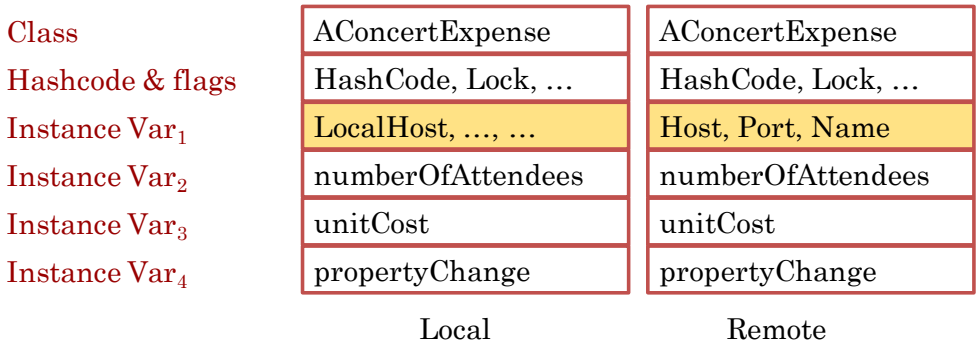


Figure 7 Storing location information in instance variables

If we use this approach, then the compiler could generate the following pseudo code to process a method invocation:

```
// code generated by compiler for each call
if  (object.getClass() IS_A Predefined Remote Class) {
            if ((UnicastRemoteObject) object).getLocation().isLocal())
                        call local method
            else
                        do network communication
} else {
        call local method
}
```

Figure 8 Determining the destination

There are several problems with this approach:

1. Time cost: On every invocation of a method on an object, an extra check is needed to determine if the class of the object is a subclass of a predefined remote class.

2. Size of object code: The compilation of every method call must have this code, which increases the size of the object code.

3. Distribution-aware code generator: The compiler is aware of distribution issues – in particular which classes are remote classes.

Size of local object: Even those objects of a subclass of a remote class that are not accessed remotely have the location information.

Generating server-side and client-side proxies

Our study of shared window systems leads us to a solution of the first four problems. A shared window system is implemented by attaching collaboration-aware proxies to existing collaboration-unaware applications. We can similarly put distribution-awareness, not in the compiler and the code it generates, but in proxies for objects that are accessed remotely. In the case of a shared window system, the API or interface was fixed, so it was possible to provide predefined proxies. In the case of RMI, the interface of a remote object is programmer-defined. Thus, the classes of the proxy objects must be generated by a special tool from the class of the server object, which will be called the base class. Such a tool is called an RMI "compiler." The term compiler indicates that it processes programmer-defined code. It is in quotes because it is not truly a compiler in that it does not generate code. It simply creates proxy classes, which are then compiled by the true compiler.

The following figure illustrates the architecture.

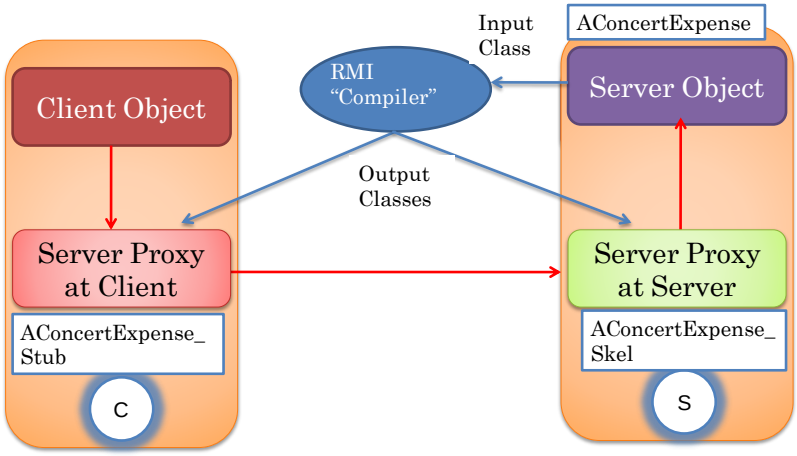


Figure 9 Proxy classes and instances

A client object indirectly invokes a method on a server object by invoking the method on a client-side proxy of the server object. This proxy object, in turn, uses low-level communication facilities such as TCP/IP or UDP to invoke the method on a server-side proxy of the server object. This proxy object invokes the method on the actual server object. Return values require communication in the reverse direction. The classes of the client-side and server-side proxies are generated by an RMI compiler from the base class. In Java these two classes are called stubs and skeletons, respectively. Thus, given the base class AConcertExpense, the RMI compiler invokes the classes AConcertExpense_Stub and AConcertExpense_Skel, respectively, which are linked to (the classes of) the client and server objects, respectively.

As we saw above, one of our goals is to allow, at runtime, a remote object to be substituted with a local one, and vice versa. This means a remote object need never be typed using a proxy class – it can be typed by the base class – the class which the proxy class was generated.

```
AConcertExpense model = (ConcertExpense) Naming.lookup(FULL_MODEL_NAME)

System.out.println(model.getNumberOfAttendees());
```

This in, in turn, implies that the stub class a subclass of the base class. In our example, this means

AConcertExpense_Stub IS_A AConcertExpense

The figure below shows the difference between the representation of an instance of a base remote class that is created locally and fetched from a remote process.

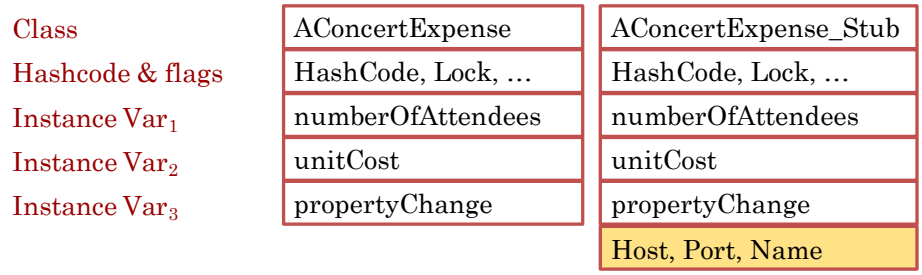


Figure 10 Representation of a local and remote instance of a base class

As the figure shows, the local proxy object is allocated space for instance variables of the base class even though they are never accessed by its methods. As long as the stub class is an extension of the base class, and language does not make a special case for stub classes, we cannot overcome this problem.

The figure below uses the setTicketPrice(float) method of AConcertExpense to illustrate remote method invocation. The client invokes an override of this method in the stub class. This method, in turn, calls a corresponding method in the skeleton which, finally, invokes the method in server object. Thus, there are several levels of indirection in the processing of a remote method invocation. This overhead is acceptable because it is insignificant compared to the cost of remote communication. The most important property of this implementation is that processing the more common local method calls does not impose extra overhead. Thus, it satisfies the more general principle in systems design: *Make the common case efficient and the rare one possible but not necessarily efficient.*

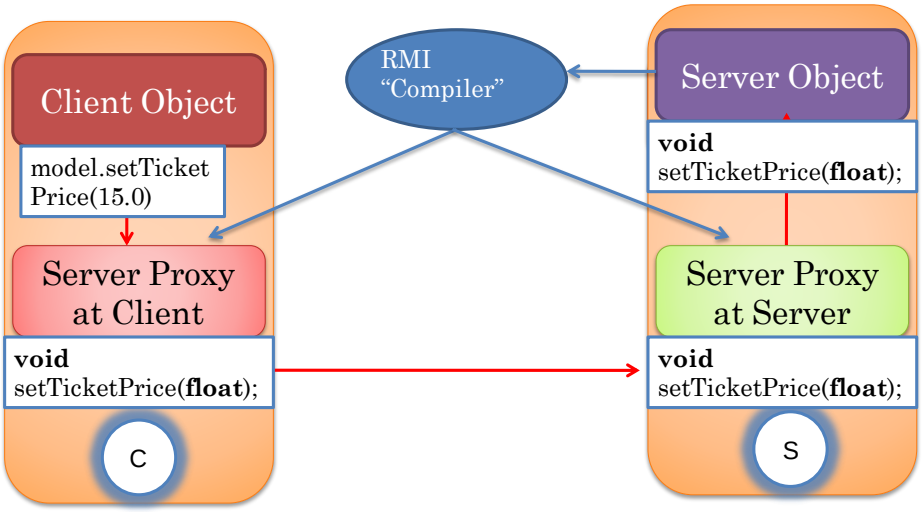


Figure 11 Remote method invocation via proxy methods

As the discussion above also shows, each time we add a method to the base class that can be called remotely, we must regenerate the proxy classes using the RMI compiler.

The following script makes the use of this compiler concrete.

```
set javabin=D:\"Program Files"\Java\jre1.6.0_03\bin
cd D:/dewan_backup/java/concert/bin
%javabin%\rmic  budget.AConcertExpense
```

In Java, the RMI compiler is not in the default path – the list of directories the operating system searches for executable files. Therefore, using a command variable, we give the full name of the executable of the RMI compiler. When we invoke the compiler, the current directory must be the root project directory – and not the directory corresponding to the package of the base class. The compiler is passed the full name of the class, including the name of its package. The proxy classes are put in the package of the base class.

Remote Exceptions

As we have seen above, a client process, the process in which a client object resides, is not entirely distribution-unaware. Some part of it must fetch a proxy of the server object from the name server. However, given our goal of allowing run time substitution of remote and local objects, it seems that code that invokes a remote method is distribution unaware. This would be possible if the semantics of remote and local method invocations are identical. However, a remote method invocation can fail because the server process is unavailable, which may happen for several reasons, because, for instance it has crashed or network connectivity has been lost. In this situation, the method invocation fails. If we wish to give the client object a way to recover from this failure, we should report this exception to the caller (the figure below), which is called RemoteException in Java.

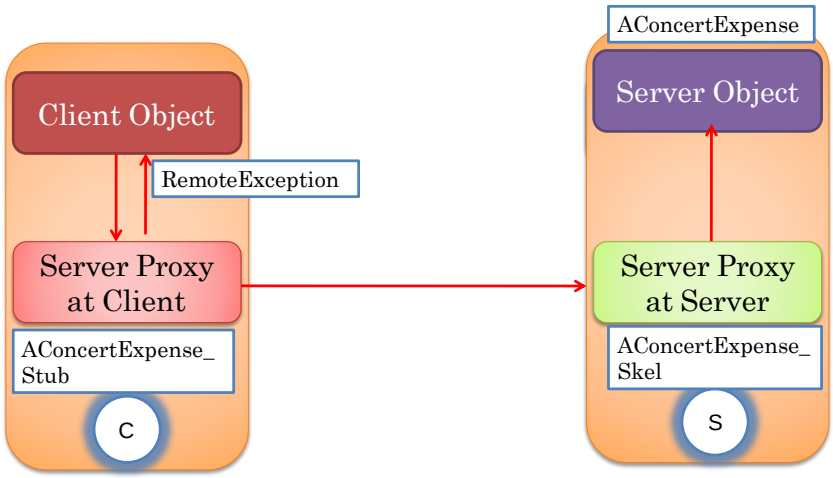


Figure 12 Remote Exceptions

The following Java code illustrates this exception.

```
ConcertExpense model =  (ConcertExpense) Naming.lookup(FULL_MODEL_NAME);
try {
    System.out.println( model.getNumberOfAttendees());
} catch (RemoteException e) {
    e.printStackTrace();
}
```

In Java, some exceptions must be caught. Java RemoteException is an example of such an exception. This means that even local invocations of the methods a remote class must catch this exception.

```
ConcertExpense model =  (ConcertExpense) Naming.lookup(FULL_MODEL_NAME);
try {
    System.out.println( model.getNumberOfAttendees());
} catch (RemoteException e) {
    e.printStackTrace();
}
```

It also implies that, in Java, the header of a method that can be invoked remotely must explicitly indicate that it throws this exception. Thus, a remote class is distribution aware, as illustrated in the following distribution-aware version of AConcertExpense.

```
public class AConcertExpense extends  UnicastRemoteObject implements ConcertExpense
    float unitCost = 0;
    int numberOfAttendees = 0;
    PropertyChangeSupport propertyChange = new PropertyChangeSupport(this);
    RemotePropertyChangeSupport remotePropertyChange = new RemotePropertyChangeSupport(this) ;
    public float getTicketPrice()  throws RemoteException { /* code omitted */ }
    public void setTicketPrice(float newVal) throws RemoteException { /* code omitted */ }
    public int getNumberOfAttendees() throws RemoteException { /* code omitted */  }
    public void setNumberOfAttendees (int newVal) throws RemoteException { /* code omitted */ }
    public float getTotal() throws RemoteException { /* code omitted */ }
    public void addPropertyChangeListener(PropertyChangeListener l) {
        propertyChange.addPropertyChangeListener(l);
    }
    public void addRemotePropertyChangeListener(RemotePropertyChangeListener l)
                throws RemoteException {
        remotePropertyChange.addRemotePropertyChangeListener(l);
    }
}
```

Figure 13 Distribution-Aware Remote Class

The model accepts both local and remote listeners, which call the addPropertyChangeListener() and addRemotePropertyChangeListener() methods, respectively. Local listeners are distribution unaware, while remote listeners define a notify method that throws a remote exception. Allowing local listeners accommodates existing views and other listeners that were not designed for distribution. The method addPropertyChangeListener() does not throw RemoteException as a distribution-unaware listener would in general not handle such an exception. All other public methods throw this exception.

Distinguishing between local and remote methods

The above example, thus, has both remote and local methods. The former can be called remotely, while the latter can never be called remotely. It is important for both the real compiler and RMI compiler to be able to distinguish between these two kinds of methods. The real compiler must enforce that the header of a remote method indicates that it throws a RemoteException. The RMI compiler must generate proxy methods only for the remote methods of a class. We can imagine tagging each method as local or remote, but a more efficient approach is to tag a whole group of methods defined in a class or interface. The exact tagging mechanism depends on the implementation.

Today, Java has annotation to tag methods, classes and interfaces. At one time, interfaces were the only mechanisms to do so – that is each tag was associated with a special tag interface. An interface that extended or a class that implemented a tag interface was labeled by the associated tag. This is the approach used in Java to declare remote methods. Java defines a special tag interface called Remote. All public methods declared in a class that implements the interface or an interface that extends this interface are considered remote.

We can use this interface to mark all methods other than addPropertyChangeListener() as remote. We can put only the remote methods in the interface, ConcertExpense, and make it extend Remote. As a result, addPropertyChangeListener() is not considered a remote method.

```
public interface ConcertExpense extends Remote {
    public float getTicketPrice() throws RemoteException;
    public void setTicketPrice(float newVal) throws Remot
    public int getNumberOfAttendees() throws RemoteException;
    public void setNumberOfAttendees(int newVal) throws RemoteException;
    public float getTotal() throws RemoteException;
    public void addRemotePropertyChangeListener(PropertyChangeListener l)
    throws RemoteException;
}
```

Empty interface

```
public class AConcertExpense extends UnicastRemoteObject implements ConcertExpense {
  public void addPropertyChangeListener(PropertyChangeListener l) {…}
  public void addRemotePropertyChangeListener(PropertyChangeListener l) throws
       RemoteException{…}
  ….
}
```

Figure 14 The header of the local method is not in the remote interface

Suppose we tagged the model class instead. In this case, all public methods of the class would be considered as remote, and the Java compiler would complain that the header of addPropertyChangeListener() does not acknowledge RemoteException.

```
public interface ConcertExpense {
    public float getTicketPrice();
    public void setTicketPrice(float newVal);
    public int getNumberOfAttendees();
    public void setNumberOfAttendees(int newVal);
    public float getTotal();
    public float add addRemotePropertyChangeListener(PropertyChangeListener
    l) throws RemoteException;
}
```

```
public class AConcertExpense extends UnicastRemoteObject implements ConcertExpense,
  Remote {
  public void addPropertyChangeListener(PropertyChangeListener l) {…}
  public void addRemotePropertyChangeListener(PropertyChangeListener l) throws
      RemoteException{…}
  ….
}
```

All public methods of class labeled remote

Compile error that remote method does not throw RemoteException

Figure 15 Making a class remote

The concept of remote methods helps us better understand why we could not use PropertyChangeListener as the type of a remote listener. This predefined Java interface does not extend Remote. As a result, its notification method, propertyChange() (change the font), cannot be invoked remotely. Therefore, the util package of ObjectEditor contains a remote version of this interface.

```
public interface PropertyChangeListener {
    public void propertyChange(PropertyChangeEvent evt) ;
}
```

```
package util;
import java.beans.PropertyChangeEvent;
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface RemotePropertyChangeListener extends Remote {
    public void remotePropertyChange(PropertyChangeEvent evt) throws RemoteException;
}
```

Figure 16 Local vs Remote Listener Interface

Remote Parameter Passing Semantics

As we saw above, the caller of a remote method is distribution aware in that it must handle RemoteException. If the caller already handles a superclass of this exception, Exception, then it can, syntactically, be completely oblivious to distribution. However, when we consider parameter passing, we see that the caller may, semantically, be very much aware of distribution.

Different languages differ in how they handle parameters of local methods. Similarly, different RMI schemes differ in how they pass parameters of remote methods. In fact, local and remote parameter passing can be very different from each other, which is what makes remote method invocation semantically aware of distribution.

The following figure illustrates the issue before us. Given a method with parameters, consider the relationship between the values assigned to the formal parameters and actual parameter values passed in the method call. This relationship, in turn determines what is sent over the network to describe the actual parameters.

```
Model model =  (Model) Naming.lookup(FULL_MODEL_NAME);
try {
    model.method1(param1, param2);
} catch (RemoteException e) {
    e.printStackTrace();
}
```

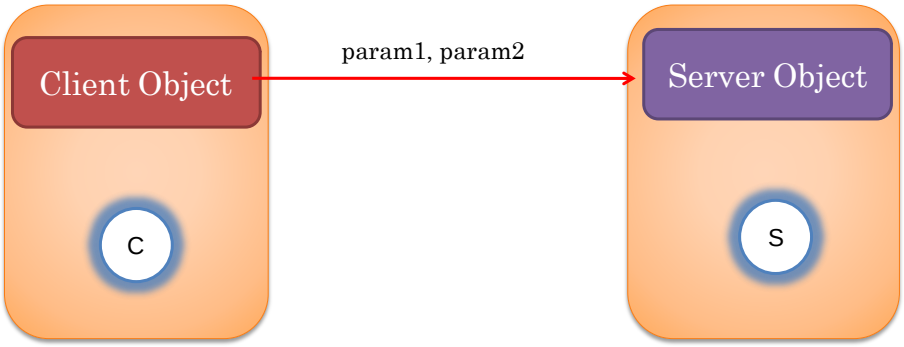


Figure 17 Parameter passing in remote method calls

Primitive Values and Network Representation

For primitive values such as Java int and boolean values, the answer seems straightforward. Pass a copy of the actual parameter on the network, and assign a copy of this network copy to the formal parameter. The network copy must have a standard machine-independent representation to accommodate heterogeneous computer architectures. This means that both the server and client computer must translate between their machine-dependent representations and the standard network representations, also called eXternal Data Representation (XDR).

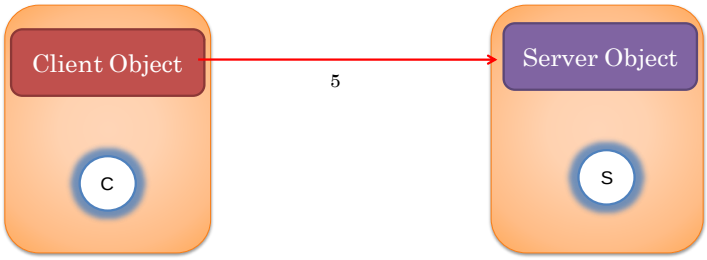


Figure 18 Parameter passing by copy

Of course, the server and client objects are oblivious to these translations. Thus, for them, the parameter passing is “by deep copy.” Even if the programming language supports parameter passing by reference for local methods, it does not make sense to pass addresses to the remote machine, as they would be meaningless at the remote machine.

Object parameters make the RMI semantics and implementation much more complicated. For this reason, some of the early RMI mechanisms supported only primitive arguments. However, this reduces the nature of distributed objects we can create. Some have argued this is not a practical restriction because programs that have used such mechanisms have never needed to pass programmer-defined values. However, it is likely that certain kinds of programs were never implemented because of this restriction.

Serializing Object Values

We can treat object parameters like primitive parameters by assigning deep copies of them to the method. A deep copy recursively copies the physical structure of the object. This approach is illustrated in the figure below. Here, a list of two elements is passed as a parameter. One of these elements is a String and the other one is itself a list of two String values. When the parent list is passed to the remote method, setList, the client proxy serializes or marshals it by creating a machine-independent deep network copy of the object. This network copy is sent to the server proxy, which unmarshals or de-serializes it by creating a machine-dependent deep memory copy of the object. It then passes this memory copy to the method in the client object.

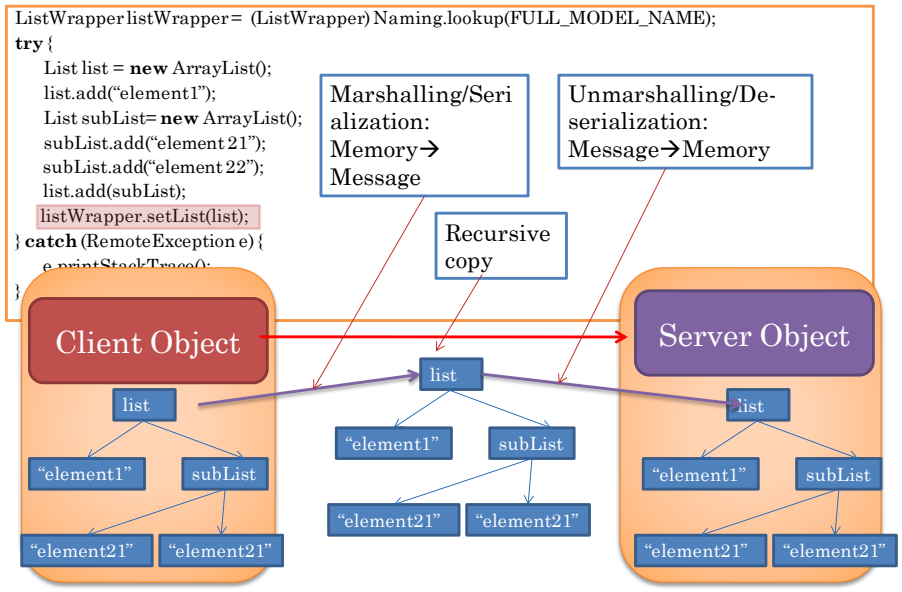


Figure 19 Marshalling and Unmarshalling of Object Parameters

Serializing Cyclic Structures

In a language such as Java that has pointers, it is possible to send a cyclic physical structure as a parameter, that is, a structure that is a graph rather than a tree or a DAG. The figure below shows an example of such a structure. Here, the parameter, list, has an element that points to it.

```
ListWrapper listWrapper =  (ListWrapper) Naming.lookup(FULL_MODEL_NAME);
try {
     List list = new ArrayList();
     list.add(list);
} catch (RemoteException e) {
    e.printStackTrace();
}
```

Figure 20 Cyclic Physical Data Structures

An implementation that recursively serializes a physical structure without checking for cycles can go into an infinite recursion and possibly even write arbitrary parts of memory. One that does perform this check can, in fact, create an isomorphic copy of the structure at the remote end. Assuming that recursive structures are uncommon, such an implementation makes the common case inefficient. Perhaps for this reason, Java RMI does not perform the check, and can cause the application program to behave erroneously. The example above is perhaps not a realistic one. A more common example of a cyclic structure is an object with a pointer to its parent. As we see later, it is possible to tell Java not to serialize such pointers, which can then be recreated by the server object.

Passing remote parameters by reference

Consider now the serialization of our example concert expense object. So far, we have been invoking remote methods on it. Suppose we pass it as a parameter to a remote method in some other server object, as shown below.

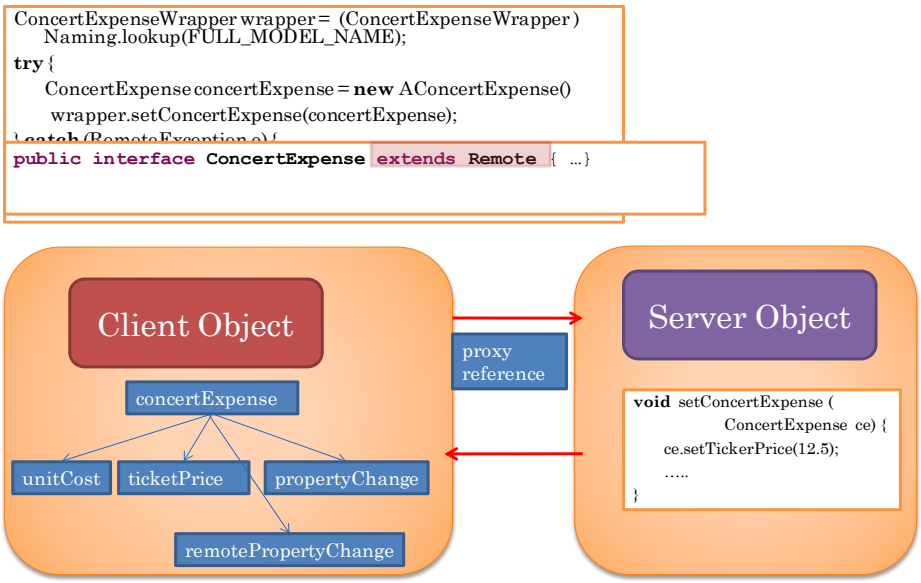
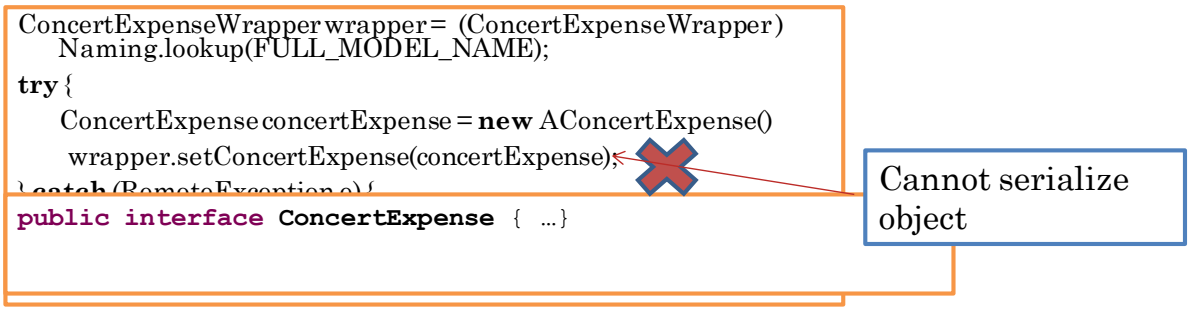


Figure 21 Passing a remote reference

As the parameter is a remote object, it is, in fact, possible to pass a remote reference to the server object. That is, the serializer can create a server-proxy for it and pass the id and type of the proxy over the network. The de-serializer can then create a client proxy for it, connect the client proxy to the server proxy, and pass a reference to the client proxy to the method. When the server object invokes methods in this client proxy, they are executed in the client process. Thus, the client and server swap roles when these methods are invoked, as shown in the figure. This is the approach taken in Java. Remote objects are not serialized when they are passed as parameters of a remote method. Instead remote references to them are assigned to the corresponding formal parameters.

Serializable vs. non serializable types

Invoking methods on remote objects results in wasteful communication if these objects do not change. In this case, it is better to serialize the objects. This means that we must make sure that the class of the object does not implement the Remote interface. This is what we have done below: ConcertExpense is no longer an extension of Remote.

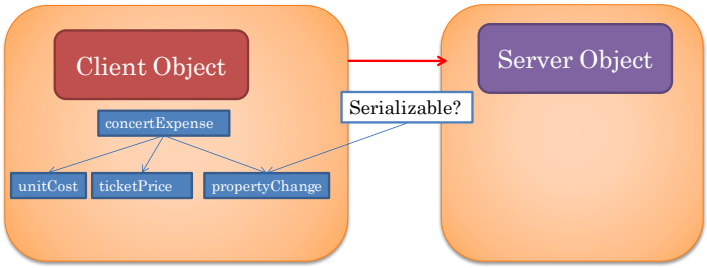


Java will, however, not let us pass an instance of AConcertExpense to a remote method, complaining that the object is not Serializable. To make it serializable, we must tag its class or interface as an implementation or extension of the tag interface Serializable, as shown below.

```
public interface ConcertExpense extends Serializable { …}
```

In general, an object can be passed as a parameter to a remote method if every component of its physical structure is either an instance of Serializable or Remote. The nested list example worked because List IS-A Serializable and String IS-A Serializable.

Why not simply assume that any object that is not remote is serializable? Our concert example provides the motivation for explicitly tagging objects as Serializable. Typically, it would not be possible to serialize the list of view observers referenced by propertyChange.



The reason is that a view typically contains widgets such as TextField and Button which directly or indirectly contain host-specific identifiers of local windows. These identifiers have no meaning at the remote site. One can imagine recreating corresponding windows at the remote host. However, this means that the RMI system is aware of the window systems of the various machines and is able to translate between these windows. In general, it knows only about the language types, and is unaware of specific programmer-defined types such as windows. Some languages such as Java allow a class to define its own serializer and deserializer methods, which could be used to provide the necessary translation. However, if a class contains references to non-serializable objects and does not define these methods, it should not be tagged as Serializable. The widget classes are not subtypes of Serializable. As a result, a class that has references to widgets cannot be declared as Serializable, as shown below.

```
public class AConcertExpenseView implements Serializable {
  TextField textField;
  Button button;
  ....
}
```

Partial Serialization

The rules above indicate that our example model class cannot be tagged as Serializable as it references one or more view classes that are not Serializable. This is because we assumed that the physical structure is either completely serialized/de-serialized or not at all. A more flexible approach is to allow only some of the components to be serialized/de-serialized. In our example, this would allow us to pass

the model to a server machine without its views and bind it to different views at that machine. Such flexibility requires a mechanism to divide the physical components of each class into those that are sent over the network and those that are not. In Java, this mechanism consists of using the **transient** keyword to label instance variables. Objects assigned to these variables are not sent serialized when the containing object is serialized. The following example illustrates the use of this keyword to make the propertyChange variable transient. The object assigned to the variable is not sent over the network when it is serialized.

```
public class AConcertExpense implements ConcertExpense, Serializable {
    float unitCost = 0;
    int numberOfAttendees = 0;
    transient PropertyChangeSupport propertyChange = new PropertyChangeSupport(this);
    public float getTicketPrice() {  return unitCost;  }
    public void setTicketPrice(float newVal) {
            if (newVal == unitCost) return;
            float oldVal = unitCost; int oldTotal = getTotal();
            unitCost = newVal;
            propertyChange.firePropertyChange("ticketPrice", oldVal,  newVal);
            propertyChange.firePropertyChange("total", oldTotal,   getTotal());
```

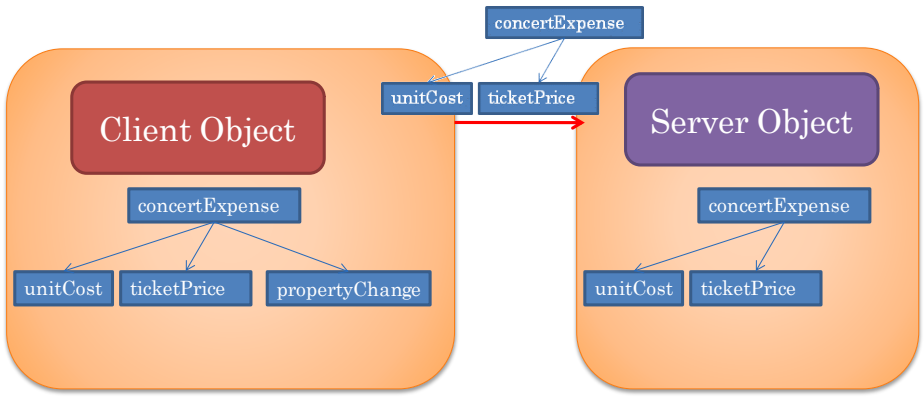


Figure 22 Serializing part of the physical structure

Serialization and files/sockets

Object serialization is used to not only send it over the network, but also to write it to a persistent file. Transient variables are called so because the objects are not persisted into a file. In most operating systems, sockets – bidirectional streams of data – have the same interface as files. Therefore, several languages such as Java allow objects to read and write serialized objects to a socket.

Converting a partial physical structure to a full structure

When a partially serialized object is read from a file or the network, its transient variables are uninitialized, that is, have null values. Any method that accesses these variables, such as the setter methods and the addPropertyChangeListener() method of the example, will fail if these variables are not initialized. How these variables are initialized is application dependent – hence the RMI mechanism cannot automatically initialize them. Thus, the programmer must specify how these are initialized. For

an object explicitly instantiated by the programmer, the constructor used for the instantiation can initialize variables. When the RMI mechanism creates an object as a result of de-serialization, it does not call any constructors, as these may re-initialize non transient variables also.

Therefore, the programmer must use ad-hoc mechanisms to do so. In our example, each setter method could check if propertyChange is null. If it is, it could assign to it a new instance of PropertyChangeSupport. This makes the code longer, messier, and slower. An alternative is to define in each class a special method that is called by application programs whenever it receives a de-serialized object. In particular, a remote method would call this method on each de-serialized object parameter it receives. For example, we could define the method, initSerializedObject(), in our example application, and have setConcertExpense() call it.

```
public class AConcertExpense implements ConcertExpense, Serializable {
    float unitCost = 0;
    int numberOfAttendees = 0;
    transient PropertyChangeSupport propertyChange = new PropertyChangeSupport(this);
    public float getTicketPrice() {  return unitCost;  }
    public void setTicketPrice(float newVal) {
            if (newVal == unitCost) return;
            float oldVal = unitCost; int oldTotal = getTotal();
            unitCost = newVal;
            propertyChange.firePropertyChange("ticketPrice", oldVal,  newVal);
            propertyChange.firePropertyChange("total", oldTotal,   getTotal());
    }
    public int getNumberOfAttendees() {  return numberOfAttendees;   }
    public void setNumberOfAttendees(int newVal) {
            if (numberOfAtendees == newVal) return;
            int oldVal = numberOfAttendees; int oldTotal = getTotal();
            numberOfAttendees = newVal;
            propertyChange.firePropertyChange("numberOfAttendees", oldVal, newVal);
            propertyChange.firePropertyChange("total", oldTotal, getTotal());
    }
    public float getTotal() {return unitCost*numberOfAttendees;   }
    public void addPropertyChangeListener(PropertyChangeListener l) {
       propertyChange.addPropertyChangeListener(l);
    }
    public void initSerializedObject() {propertyChange = new PropertyChangeSupport(this) ; }
}
```

```
public void  setConcertExpense (ConcertExpense  ce) {

      ce.initSerializedObject(); // explicit call

      ce.setTicketPrice(12.5);

       …..
```

```
}
```

This method could have to be called on each component of the de-serialized physical structure. Of course, it is easy to forget to call this method. If the method is standardized, then the RMI could automatically call this method for the programmers. One way to do so in Java is to make Serialized not a tagging interface but a real interface that contains a header for this method.

```
public interface  Serializable {
   public void initSerializedObject();
}
```

The standard method is like a standard constructor in that both are called by the implementation when an object is created. While a constructor initializes all variables, this method initializes only transient variables.

A disadvantage of this idea is that the method must be defined even if an object has no transient variables. If this is a real problem, it can be overcome with language support. Just as Java defines a null constructor if the programmer does not do so, it can similarly define a null initSerializedObject() if the programmer does not do so.

In some languages such as Java, it is possible to define one's own de-serialization method which can be used to initialize its transient variables and call the default serialization method. However, this approach forces the programmer to be aware of parameters of this method that are not relevant to initializing transient variables. When we study Sync later, we will see that it supports such a method.

Partial Serialization and cyclic structures

A variable should be made transient whenever the object assigned to it cannot be correctly serialized. If the type of the object is not serializable, then, of course, as we just saw, the object is not serializable. Even if its type is serializable, then the object may not be correctly serialized in Java if it is a back pointer that creates a cyclic data structure, as we saw earlier. Thus, transient variables can be used to transmit graphs to a remote site as long as the back pointers can be recreated at the remote end from the forward pointers.

Serializable and inheritance

To better understand the Serializable tag interface, consider ACommentedConcertExpense of the previous chapter, which is reproduced below.

```
public class ACommentedConcertExpense extends ATypedConcertExpense implements
VectorListener, CommentedConcertExpense {
   AListenableString comment = new AListenableString();
   static int LONG_COMMENT_SIZE = 10;
   boolean longComment = false;
   public ACommentedConcertExpense() {
      comment.addVectorListener(this);
   }
   public boolean getLongComment(){return comment.size() > LONG_COMMENT_SIZE;}
   public AListenableString getComment() {return comment;}
   public void setComment(AListenableString newVal) {
      comment = newVal;
      comment.clear();
   }
   boolean oldLongComment = false;
   public void updateVector(VectorChangeEvent evt) {
      boolean newLongComment = getLongComment();
      propertyChange.firePropertyChange("longCommentType", oldLongComment,
                    newLongComment);
      oldLongComment = newLongComment;
   }
}
```

Even though its interfaces, VectorListener and CommentedConcertExpense are not explicitly declared as extensions of Serializable, this class is serialzable because of the following relationships:

```
ACommentedConcertExpense IS-A CommentedConcerExpense

CommentedConcertExpense IS-A TypedConcertExpense

TypedConcertExpense IS-A ConcertExpense

ConcertExpense IS-A Serializable
```

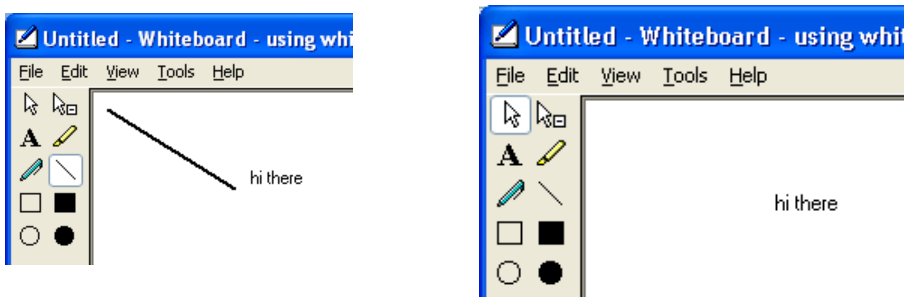
In other words, all subtypes of a type tagged as Serializable are also Serializable. This is somewhat counter intuitive because when we make a class Serializable, we do not know the kind of instance variables its subclasses will declare. Put another way, the Serializable tag, constrains the subclasses to make sure that the types of their instance variables are serializable.

Identifying the caller

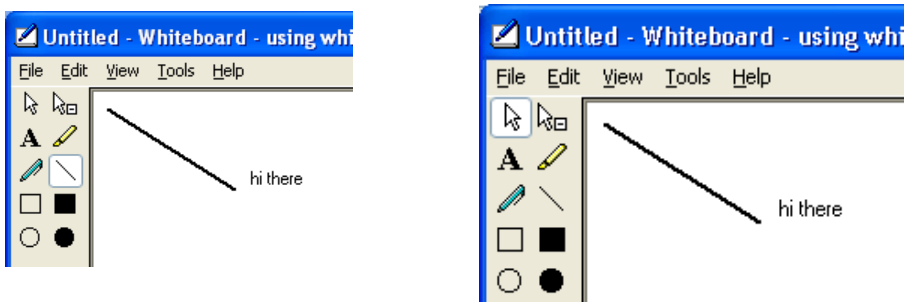
We have seen above how we can extend single-user MVC to distributed MVC. This requires us to make our views, controllers, and models distribution aware to some degree. However, these components are completely collaboration unaware. In particular, the model does not know if the controllers and views invoking methods are interacting with a single or multiple users.

A consequence of this implementation scheme is that the model cannot implement certain collaboration functions. For example, it cannot make sure that local and remote users get different feedback. This means that it cannot be used to automate the implementation of commercial multiuser whiteboards. In these systems, when users extend the size of a new or existing line by dragging the

mouse, they but not their collaborators can see the incremental changes as the mouse moves, as shown in the figure below.



(a) Local user (left) but not the remote user (right) sees new line extend as the mouse is dragged



(b) Remote user sees the new line when the mouse is released

Figure 23 Different users get different feedback

Similarly, in our example, it is not possible for the model to provide (1) concurrency control by disallowing two users to simultaneously edit the same property, and (2) access control by allowing only certain users to edit certain properties.

One way to overcome this problem is to provide to each to each write and read method the identity of the caller. Each invoker of this method could pass this value, as shown below.

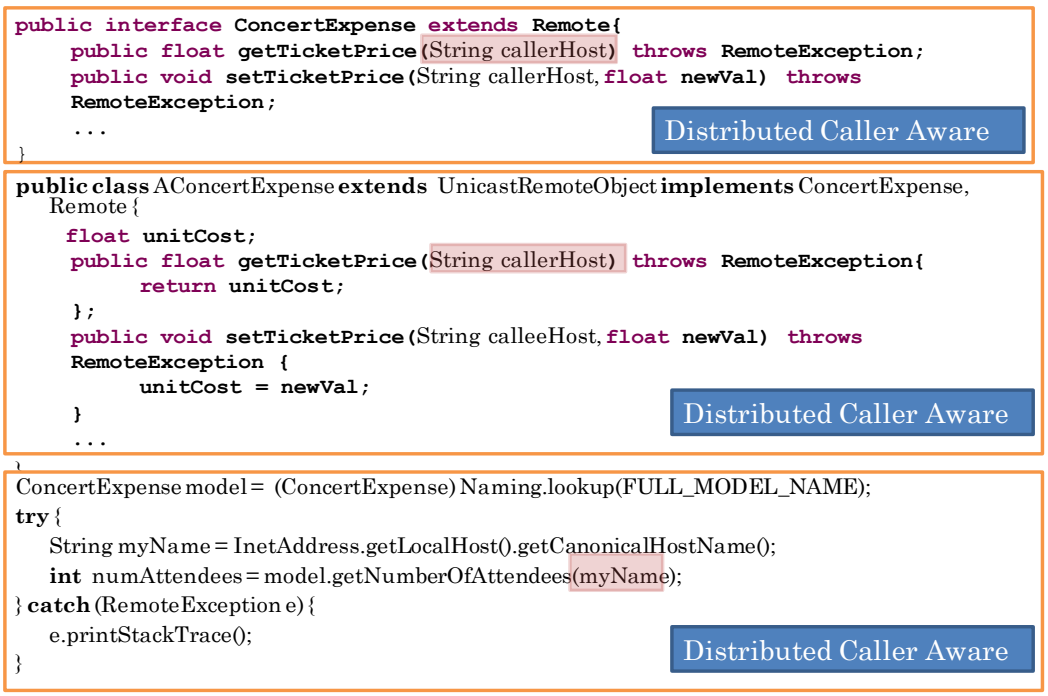


Figure 24 Distributed Caller Awareness

As we see above, the interface of the model, the server class implementing this interface, and the client code using this interface must now all be aware that the caller of a remote method is distributed. Moreover, the caller must be trusted to correctly provide its identity or go through the overhead of creating an unforgeable identity. Because a remote method is invoked by a proxy, it is possible to put this awareness only in the server class. In particular, the interface can declare the remote methods without the caller parameter. The client object calls these methods without this parameter in the client-side proxy. This proxy, which could be trusted as it is generated, supplies the value of this parameter, which is then transmitted via the server-side proxy to the called method. The following figure illustrates this approach using our example.

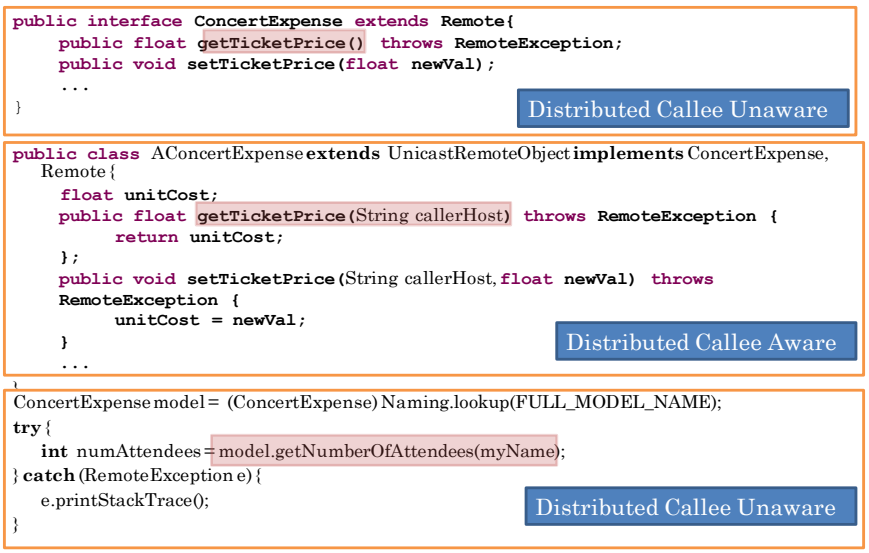


Figure 25 Client proxy supplies caller identity

The following code illustrates the use of the caller.

```
public void setTicketPrice(String callerHost, float newVal) throws
RemoteException {
      if (ac.authorized(callerHost))
            unitCost = newVal;

}
```

This approach is implemented by the LiveMeeting API. It has the problem that the class does not implement the interface methods exactly. A consequence is that local invocations of the remote methods must also go through proxies. Perhaps a better approach is to allow a remote method to access the name of its caller using a special keyword such as **caller**.

```
public void setTicketPrice(float newVal) throws RemoteException {

      if (ac.authorized(caller))

            unitCost = newVal;

}
```

However, this approach requires the compiler to implement a special mechanism to push and pop the value of the caller on the stack when a remote method starts and stops execution, respectively. When we study Sync, we will see how the keyword is replaced by a library call that keeps caller information in global variables.

Programmer-Defined MVC Proxies

The information about the caller is necessary for the implementation of several existing/practical collaborative MVC-based user-interfaces. However, it is not clear that model should be collaboration-aware and directly process this information. In fact, ideally, the models, views, and controllers should

not even be distribution aware, which would (a) allow reuse of existing single-user models, views and controllers, and (b) allow local and distribution/collaboration semantics to be kept in separate objects, allowing the implementation of them to change independently. As we saw in the design of shared window systems, it is possible to extend a single-user window system by adding collaboration-aware proxies that appear to the application as the single-user window systems and to the single-user window systems as applications. The figure below shows how this idea can be adapted to a distributed MVC system.

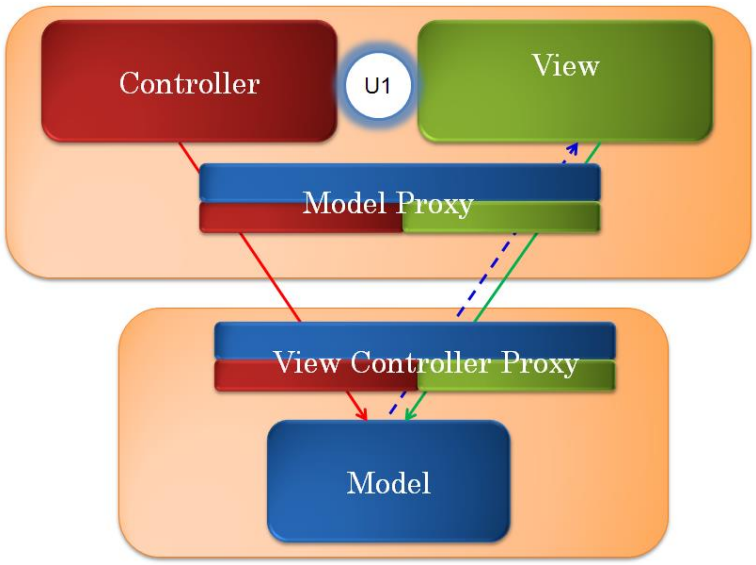


Figure 26 Proxy-based Distributed MVC

At the site of the model, we create a proxy that looks to all the remote views and controllers of the object as the model, and to the local model as local views and controllers. Similarly, at each site with one or more controllers and views of the model, we create a model proxy, that appears to the controller and views as a local model, and to the actual remote model as the controllers and views at that site. These MVC proxies use RMI or some other mechanisms to communicate with each other. In addition, they implement programmer-defined collaboration functions. For example, the proxy for the concert example could implement the following method to implement access control:

```
public void setTicketPrice(String caller, float newVal) throws
RemoteException {
      if (ac.authorized(caller))
            localModel.setTicketPrice(newVal);
}
```

Like the generated server and client RMI proxies, the programmer-defined MVC proxies intercept the communication between the models, view and controllers. In fact, if the MVC proxies can implement some of the functionality of the RMI proxies if they choose a mechanism such as sockets that is lower-

level than RMI to communicate with each other. The MVC proxies free the models, view and controllers from the nature of the communication mechanism.

Summary

- RMI allows invocation of methods in remote objects.
- To support dynamic substitution between remote and local objects, remote and local method invocation should have the same syntax.
- Both kinds of calls must be able to process remote exceptions.
- Client-side and server-side proxy classes are generated for remote classes.
- A proxy class includes proxy methods for all remote methods of a remote class.
- In Java, a class can be made remote by (a) making at least one method of it remote, and (b) making it a subclass of a predefined remote class such as UnicastRemoteObject that chooses the nature of the underlying communication mechanism.
- In Java, a method is remote if it is declared in an interface or class that is a subtype of the tagging interface, Remote.
- Clients can get references to remote objects by looking up objects registered with a name server.
- They can also get remote references as parameters of remote methods.
- Non-remote but serializable parameters are passed by deep copy.
- Primitive (components of) actual parameters are communicated to a remote machine using standard network or eXternal Data Representations (XDR's) of them.
- Object parameters can be passed by reference or value based on whether the classes of these parameters are serializable or remote.
- In Java, a class is serializable if it is a subtype of the tagging interface, Serializable, and the types of all non transient instance variables of the class are also serializable.
- Serialization of an object involves recursive serialization of objects assigned to all non-transient variables of the object.
- In Java this recursive algorithm does not check for cycles and thus can lead to subtle problems.
- When a serialized object is received, all of its non-transient variables should be initialized.
- Several collaborative user interfaces require identification of the remote caller of a method.

- This information can be passed automatically by the RMI system.
- It is possible to make distributed models, views, and controllers distribution and collaboration unaware by using MVC proxies.