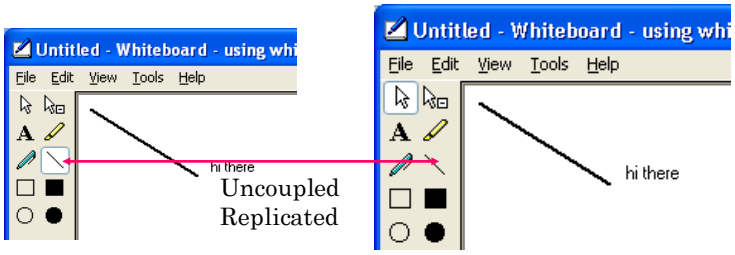
Prasun Dewan[1]

5. Replicated Objects

In the previous chapter, we studied the MVC architecture and saw how it could be distributed by creating a central model that communicates with distributed models and views. Here we will see an alternative approach for distributing MVC, which replicate models. This approach can, in fact, be used to replicate any object. We will study three variations of this approach, which bind replication semantics to objects at different times.

Functional Motivation: Partial Coupling and Disconnection

As motivation for replicated objects, consider the whiteboard implementation below. Here, the two whiteboards displayed to different users are partially coupled. The drawing palette, which allows users to determine the tool selection, is not coupled. On the other hand the drawing canvases are coupled. Thus, as shown below, if the left user selects the line-drawing tool, and draws a line, then the line is shown in the canvas of the right user, but the right user's tool selection is unaffected.



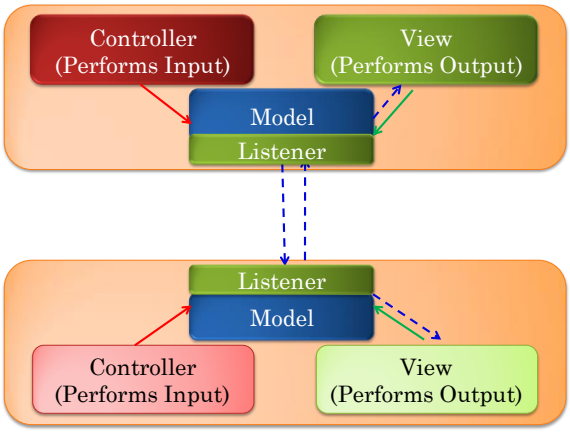
Assume that the whiteboard is implemented using MVC, and the current tool selection is part of the model. If a single centralized model is created, then this model would manage multiple tool-selections, one for each user. Thus, the model would have to be collaboration-aware. As we see below, if we replicate the model, it is possible to use an existing single-user collaboration-unaware model storing a single tool selection to create such an application.

Perhaps an even more compelling reason for replicated MVC is disconnected collaboration, that is collaboration sessions in which are connected to each other all the time. Centralized MVC requires the local views and controllers of the users to be continuously connected to the central model. Replicated MVC, on the other hand, allows these UI components to access a local model.

Replicated Architecture

Replicated MVC with Centralized Semantics and no Consistency

In replicated MVC a separate model is created for each user, and the models communicate with each other, directly or indirectly, to couple the users. The most straightforward replicated architecture is shown below.

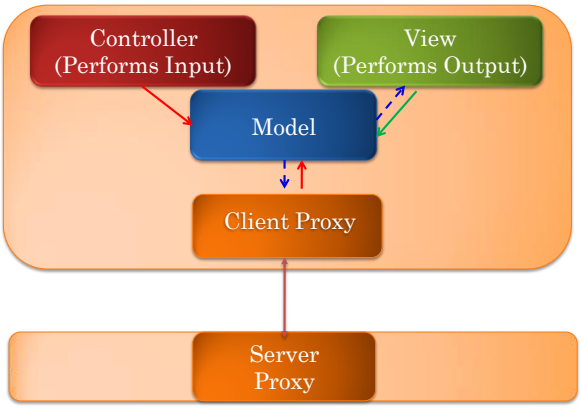


The inter-model communication protocol is the same as the model-view communication protocol. In other words, each model becomes an observer or listener of all other models. In response to a state-change notification received from a remote model, the local model updates its state accordingly. For example, if a whiteboard model receives a notification about a new shape added by a remote canvas, it adds that shape to its local canvas object.

This is a useful architecture in that it offers different performance from the centralized MVC architecture we saw earlier. In the next chapter, we will look more in depth at the performance issues. However, it does not meet the requirements of the motivating whiteboard example. As in the centralized case, we cannot directly use an existing single-user model to implement this architecture. The model is notification and distribution aware in that it now receives distributed notifications from remote models. Moreover, implementing partial coupling requires that the model also be coupling aware. If multiple users can edit concurrently, the models would also need to implement special consistency protocols. Nonetheless, this architecture provides the basis for replication solutions without these drawbacks.

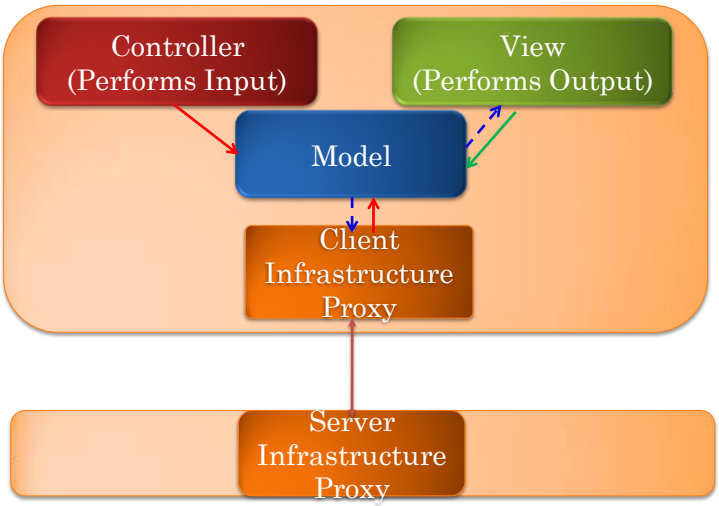
General Replicated MVC Architecture

In our discussion of centralized MVC and centralized and replicated shared window, we saw that collaboration-aware proxies allow shared application objects to be collaboration-transparent. We can apply this insight also to replicated MVC architectures, as shown in the figure below.



Now the model is no longer an observer. It is only an observable, as in classical single-user MVC. A client-side proxy, residing on each computer, (a) listens to notifications from a collaboration-unaware model, and (b) invokes corresponding write methods on remote replicas through remote client proxies. An optional server proxy can be used to implement some centralized functions such as session management, as we see later. These collaboration proxies can provide selective sharing, locking, access control, disconnection and other collaboration functions for the collaboration-unaware models, views and controllers.

To reduce programming effort, a better idea is to provide general infrastructure client and server proxies, as shown below.

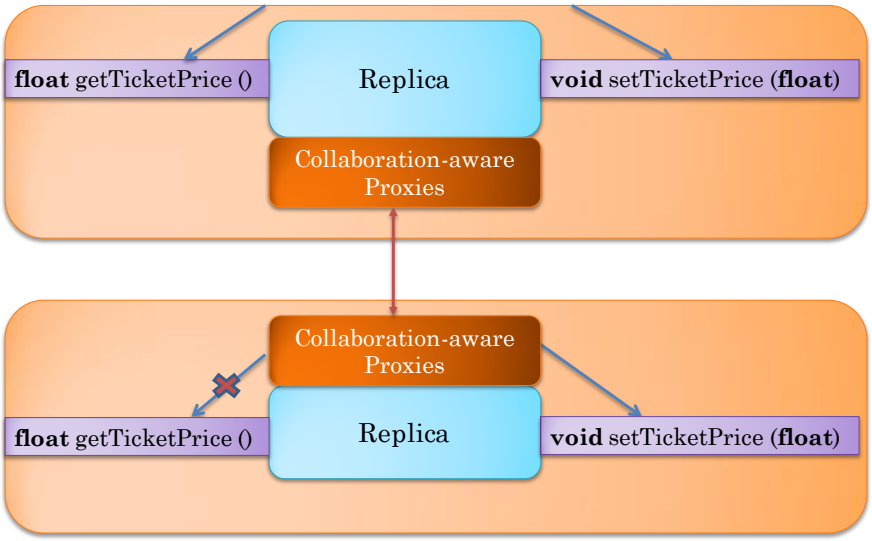


Programmer-defined Replicated Types

One way to provide such an infrastructure is to support replicated types in the manner remote types were supported in RMI. Inspired by the Java Remote interface, we can define a Replicated interface. If an object in an application implements this interface, then it is automatically replicated on the computers of all users sharing the application. Thus, if the concert expense class of the previous chapters implements this interface, its instances are automatically replicated.

```
import java.beans.PropertyChangeListener;
public interface ConcertExpense extends Replicated {
     public float getTicketPrice();
     public broadcast void setTicketPrice(float newVal);
     public int getNumberOfAttendees();
     public broadcast setNumberOfAttendees(int newVal);
     public float getTotal();
     public void addPropertyChangeListener(PropertyChangeListener l);
}
```

The Remote interface was implemented by supporting invocation of methods on some remote instance through infrastructure proxies. The Replicated interface can be implemented by supporting invocation of methods on *all* replicas of an object through infrastructure proxies. Thus, if the setTicketPrice() method is invoked by some controller on its local model, it can be automatically invoked on all replicas of the model. As a result, the replicas can remain synchronized.



Just as all methods of an object should not be invoked remotely, all methods of an object should not be broadcast to all replicas. For instance the getTicketPrice() method, which does not change the state of an object, should not be broadcast. We could use the approach used in RMI to distinguish between broadcast and local methods.

The use of a Replicated type has not actually been implemented in any system. So let us look at a variation of it that was actually implemented in Xerox's Colab programming environment

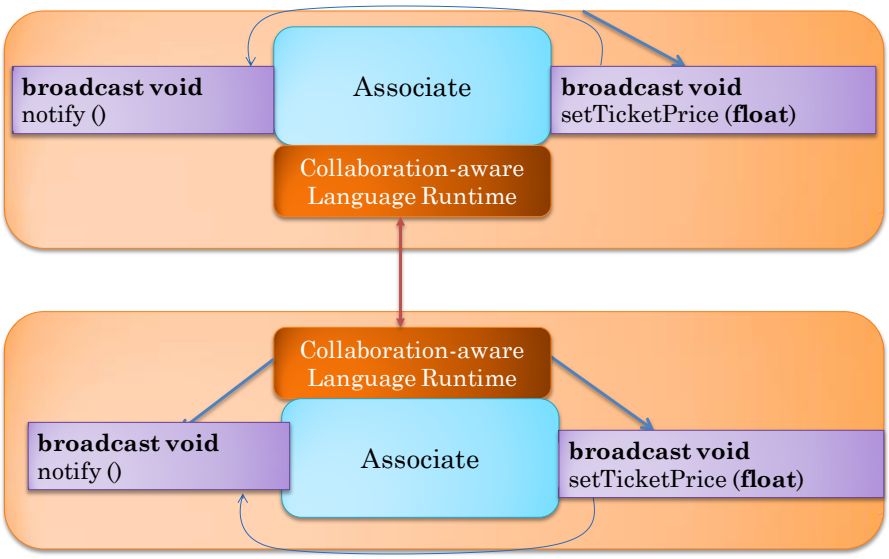
Broadcast Method Declarations

Colab extended the programming language with the keyword **broadcast** to mark methods that are broadcast. Methods not marked with this keyword are executed only on the local replicas. We see the use of this keyword in our example below.

```
import java.beans.PropertyChangeListener;
public interface ConcertExpense {
     public float getTicketPrice();
     public broadcast void setTicketPrice(float newVal);
     public int getNumberOfAttendees();
     public broadcastsetNumberOfAttendees(int newVal);
     public float getTotal();
     public void addPropertyChangeListener(PropertyChangeListener l);
}
```

Here we have been careful to make only state-changing methods broadcast. Moreover, we have not made all of these methods broadcast. In particular, the addPropertyChangeListener() method is local, because this object allows "replicas" to have local listeners. The quotes around the term "replicas" indicate that these objects are not true replicas, because only part of their state can be coupled. Colab used the term *associates* to refer to such partially coupled objects, and the term *association* to a set of partially coupled associates.

The following figure shows an association of two associates. As shown in the figure, a call to a broadcast method in an associate results in the method being called on all associates in the association. A call to a local method in an associate results in the method being called only in that associate.



Associates may not only be models, but also views, controllers, and other objects. To illustrate, consider a controller object that listens to text widget events. Such an object may have the method textChanged(evt) called whenever new text is entered by the user in the widget. This method can be made broadcast to provide controller-level coupling.

Spurious Broadcasts

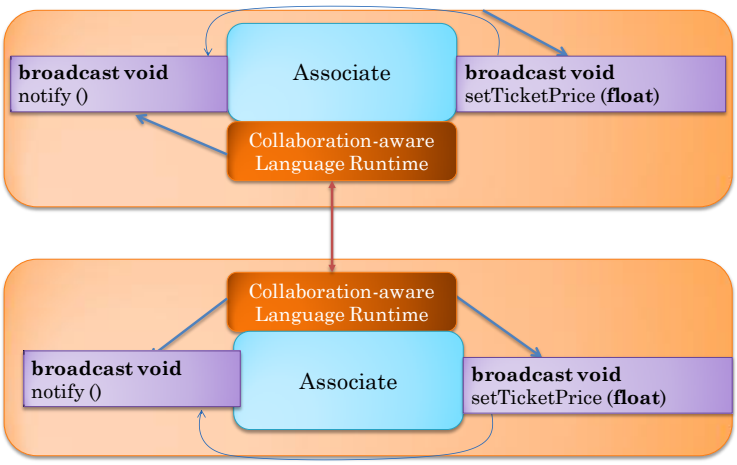
The implementation of broadcast methods is handled by the infrastructure, as shown in the figure. Thus, the programmer is responsible only for determining which state-changing methods of a class are broadcast. This may be tricky.

In the concert expense example, we looked only at the interface to decide which methods should be broadcast. We may also need to look at the class for two reasons. First, we may wish to make private methods broadcast. For example, if a set of public methods change some state through some a private method, then the programmer could make the private method broadcast. Second, we have to look at the implementation to ensure we do not have needless broadcasts. Spurious broadcasts occur when a broadcast method (b1) calls another broadcast method (b2) because the called broadcast method is called more than once in the local and remote associates: once in response to the call to b1 in the local associate, which in turn makes a local call to b2; once in response to the local call to b2 in each peer associate.

Spurious broadcasts are illustrated in the example below, where the broadcast method setTicketPrice() calls the broadcast method notify().

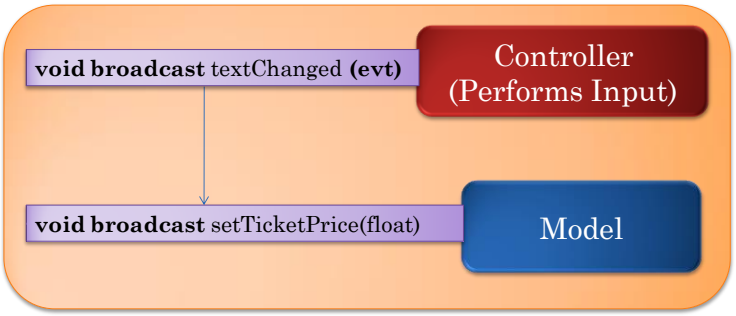
```
public class AConcertExpense implements ConcertExpense{
    float unitCost = 0;
    int numberOfAttendees = 0;
    Vector<Observer> observers= new Vector() ;
    public float getTicketPrice() { return unitCost; }
    public broadcast void setTicketPrice(float newVal) {
        unitCost = newVal;
        notify();
    }
    public int getNumberOfAttendees() { return numberOfAttendees; }
    public broadcast void setNumberOfAttendees(int newVal) {
        numberOfAttendees = newVal;
        notify();
    }
    public float getTotal() {return unitCost*numberOfAttendees; }
    public void addObserver(Observer observer) {
        observers.add(observer);
    }
    broadcast void notify() {
        for (Observer observer: observers) {
            observer.update()
        }
    }
}
```

The notify() method is called twice in each associate, once by setTicketPrice() method in the local associate, and once in response to the call to the method in the peer associate.



Spurious broadcasts of a method not only unnecessarily use computing and communication resources but also lead to incorrect program behavior when the method is not idempotent.

In a single class, it is relatively easy to avoid spurious broadcasts. It is harder to do so when the classes interact with each other as in the MVC design pattern. The designers of communicating classes must carefully look at all of these classes to avoid spurious broadcasts. In the example below, the designers of the controller and model classes should coordinate with each other to ensure that the spurious broadcast of setTicketPrice() is avoided.



Controlling the What and When of Sharing

Broadcast methods allow application programmers to determine both (a) which parts of the state of an object are shared, and (b) when shared state is synchronized. This is illustrated in a version of our running example that does not clear the comment.

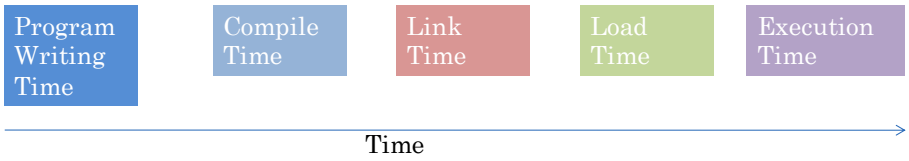
```
public class ACommentedConcertExpense extends ATypedConcertExpense implements
VectorListener, CommentedConcertExpense {
   ...
   public broadcast void setComment(AListenableString newVal) {
      comment = newVal;
      //comment.clear();
   }
   ...
   public broadcast void updateVector(VectorChangeEvent evt) {
      boolean newLongComment = getLongComment();
      propertyChange.firePropertyChange("longCommentType", oldLongComment,
                    newLongComment);
      oldLongComment = newLongComment;
   }
}
```

Here, the two broadcast methods ensure that incremental changes to the comment property are shared. If we do not want incremental changes to be shared, we can make the updateVector() method local. In this case, only the setComment() methods are coupled, which means remote users see the comment only when it is committed. If we do not want the comments in the associates to be coupled, we can make both methods local.

Thus, broadcast methods support coupling flexibility. To evaluate the extent of this flexibility, it is also important to determine when coupling decisions are made. This leads us to the general notion of binding times of entity attributes.

Binding Time

Often entities are associated with attributes. For example, variables are associated with types and memory addresses. The binding time of an entity attribute is the (latest) time at which the entity can be given a (new) value of the attribute. By a time, we do not mean the absolute clock time but some relative time based on the stages through which the entity passes. In the case of program entities such as variables, we can identify five successive stages: the program containing the entity is written, compiled, linked, loaded and executed, which lead to five increasing relative times: program-writing, compile, linking, loading and execution time.



An even earlier time than all of these stage-based times is language writing time.

The flexibility of a system binding an attribute to a program element is measured by the time at which it binds the attribute to the element – the later this time, the more flexible the system. On the other hand, sometimes a later binding results in a less efficient system Thus choosing a binding time involves a tradeoff.

To illustrate, consider the binding of various kinds of addresses to an instance variable in a class. The offset of the variable relative to the containing object is bound at compile time. This offset is added to additional offsets generated at link and load times to create the virtual address of the object. The virtual address is bound at run time to an actual physical address.

Binding the physical address of the variable at a time earlier than execution time would not allow the variable to be relocated in memory during execution time – that is, it would not allow the virtual page containing it to be moved in memory. On the other hand, binding the offset of the variable within its object as late as execution time would require this value to be searched or computed at run time. Thus, we see here that early binding can be more efficient and late binding leads to more flexibility.

In this course, we are interested in binding of collaboration attributes to sharable program entities. In this chapter, we are looking at the binding of coupling attributes to associate.

In a system that requires the use of the **broadcast** keyword to define coupling, at program writing time we bind which parts of the state of a replicated object are shared and when these parts are synchronized. Specifically, these attributes are bound when the class of the replicated object is written.

This early binding reduces the flexibility of application programs. In our example, different users of the program cannot determine if their comments are shared incrementally, on commits, or not at all. In GoogleWave IM, such early binding would not allow users to decide if their messages are seen incrementally by others.

Sometimes it is useful to share some state with a subgroup of users in a session. For example, a user may wish to share the comment with close friends in a session. Broadcast methods, as the name suggests, broadcast to all elements of an association. They do not multicast to a subgroup. Thus, which associates in an association execute non-local method is decided at language definition time.

Multicast calls

We can overcome these limitations by making the caller instead of the callee of a method determine if the state changed by the method is shared. The callee can specify, as part of the call, the destination of the call, that is, the set of associates on which the call should be invoked. The following example

illustrates its use to create a MUDs like user-interface, in which a message M entered by a user U is entered in his history as "You said: M" and in others history as "U said: M". The **others** keyword in a method call specifies that the method should be invoked on all associates in the association other than the local one. We will refer to such a call as a multicast call as it specifies a subgroup of the association. The call is destination-aware, as it explicitly indicates the associates in which it should be invoked. The absence of a destination argument indicates the call should be made only on the local associate.

```
public void addMessage(String prefix, String msg) {
      history.add(prefix + newVal);
}
public void addMessage(String msg) {
       addMessage("You said:", msg);
       addMessage(others, myName() + " said:", msg);
}
```

Caller provides extra argument specifying callee

A destination-aware multicast call is the dual of a source-aware method declaration we saw in the previous chapter, shown below.

```
model.addMessage(prefix, comment);
public void addMessage(String caller, String prefix, String msg) {
   if(ac.authorize(caller)) {
        history.add(prefix + msg);    }
}
```

Callee gets an extra argument specifying caller.

In the former case, the callee provides an extra argument specifying the caller(s). In the latter case, the caller receives an extra argument specifying the caller.

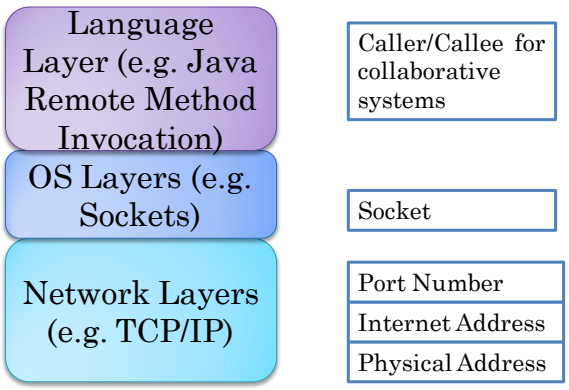
It is in fact possible to combine the two dual ideas, as shown below.

```
public void addMessage(String msg) {
       addMessage("You said:", msg);
       addMessage(others, myName() + " said:", msg);
}
public void addMessage(String caller, String prefix, String msg) {
   if(ac.authorize(caller)) {
        history.add(prefix + msg);    }
}
```

In the combined case, the caller provides an extra argument specifying the caller, and the caller receives an extra argument indicating the callee.

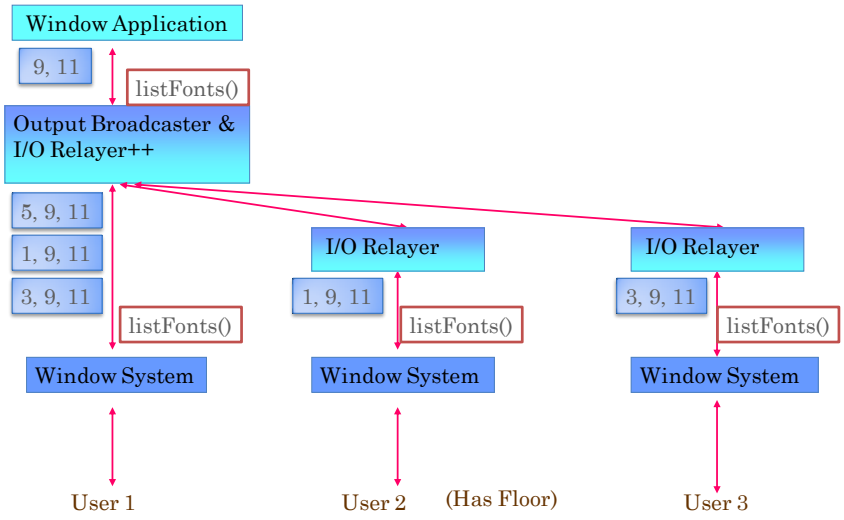
Syntactically, it seems there is no extra argument as both the call and declaration have the same signatures. However, semantically, the two arguments are unrelated. The caller indicates the destination and the callee receives the source. Thus the value of the caller formal parameter will not be **others** in the example above, it will be an identifier of the machine that made the call.

The idea of making application programs aware of the destination and source of a remote call is an abstraction of the idea of making each communication layer aware of the source and destination of the message abstraction supported by the lower-level layer.



The physical, IP, and TCP/UDP network layers support addresses that are physical, internet, and port numbers, respectively. The OS layer supports higher-level addresses such as sockets. The conventional wisdom in distributed RPC supported by the language layer has been that the caller and callee of a procedure need not be aware of each other. Our discussion above shows that when RPC is used to create some collaborative applications, such awareness is indeed useful. There is no standard defining the exact form of the identifier of caller and callee. If these identifiers are used for access control, they must be unforgeable certificates. In the examples above, we have assumed they are textual representations of these certificates identifying the caller/callee hosts and/or users.

The notion of multicast calls is related to the handling of calls that return values in a shared window system. Recall that when a window client made the call, listFonts(), the call could be directed at the local window system, the window system with the floor, any window system, or all window systems, as shown in the figure below.



As the call was destination-unaware, the window system made the choice based on the call. A destination-aware multicast call can allow a collaboration-aware window client to explicitly specify the subgroup to which the call is directed.

The example above illustrates a problem with broadcast method declarations and multicast calls. What is the return value of such calls? In general, it is the value returned by the local associate. If the return value is a list, such as in the case of listFonts(), then the return values of each call can be combined by an infrastructure that understand the semantics of the call.

Multicast calls are not currently supported by any RPC system. However, they are supported by some collaboration proxies created for collaboration-unaware tools. For each collaboration-unaware call supported by the tool, the proxy provides multiple calls, one for each multicast group defined by the proxy. This is illustrated using the listFonts() method provided by a window system.

listFontsOthers();	Every one other than caller of method
listFonts ();	Local
listFontsAll();	Everyone: Broadcast
listFontsSpecific({“joe”, “alice”});	Specific users
listFontsAny();	Any user (anycast)
listFontsInputter();	Host whose input triggered the call

Here we have defined six different multicast groups:

1. others: (The associates at) all hosts other than the host of the caller of the method.
2. local: The complement of the group above – it includes only the caller host.
3. all: The union of the two groups above – it allows the call to be broadcast.

4. specific: A set of hosts explicitly identified by the caller.
5. any: Any host chosen by the proxy – this group supports the recently popular “anycast”.
6. inputter: The host of the user whose input triggered the call, in case such a user can be identified. In a shared window system, this would be the user with the floor. In a fully replicated system such as Colab, the inputting and local hosts are the same. In a semi-replicated system such as a “centralized” window system shown above, in which the window system is replicated but the window client is centralized, the inputter may be different from the local user. In the figure above, the floor holder is User 2 rather than the local user, User 2.

As the discussion above applies, the physical windows on different sites to display a shared logical window form an association. The following example shows the application of the notion of a multicasting tool proxy in GroupKit.

```
proc insertIdea idea {
    insertColouredIdea blue $idea
    gk_toOthers "insertColouredIdea red $idea" }
```

The coupling binding time of multicast calls depends on when the target of the call is specified. If the group is passed as a runtime argument to a call, then coupling decisions are made at execution time. If the group name is built into a proxy call, as in gk_toOthers, the destination is specified at program writing time. However, even in this case, multicast calls are more flexible than broadcast declarations. It is possible to multicast to a subgroup of the association. In addition, the group is specified when the client of a replicated classes is written, not when the replicated class is written. As a result, different clients of the same replicated class can couple instances of the class differently.

Application and Association Binding

Each associate in an association runs on a different host as part of an independent program. How are the various instances of a replicated class created in different distributed applications bound into an association? We can assume that each program takes as an argument the name of a session.

```
java ConcertExpenseMain demoSession
```

This argument is processed by the collaboration infrastructure and used to group processes on different computers into sessions. It assumes a central session manager with which names are registered. A completed replicated approach is to pass each process locations of all other processes in the sessions.

Instances of replicated classes will be put together into an association only if the processes that created these instances are replicas, that is are in the same session. If a replicated class is a singleton, that is has only one instance in a process, then forming associations is straightforward - the singleton instances of all processes in a session form an association.

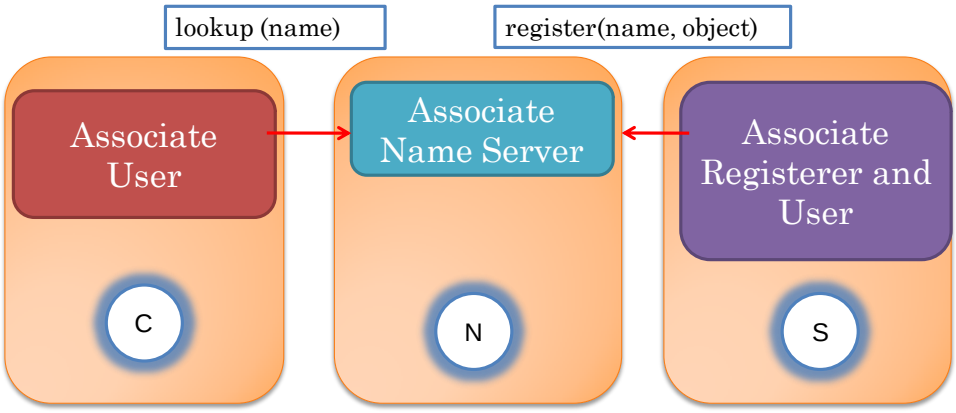
However, if a process creates more than one instance of a replicated class, then we must decide how we should find their associates in the replicas of the process. If we assume that each process creates the

same sequence of replicated objects, then the index of an instance in this sequence can be used to find its associates. That is, the i^{th} members of these sequences in the different replicated processes are put in the i^{th} association. This approach is, in fact, used in shared window systems to bind physical windows on different computers.

However, the approach has two related problems, especially when applied to shared models. It requires that processes in a session run the same program, and the program is non deterministic. This requirement does not allow collaborating users to use different devices, as the programs on these devices will generally contain different views and controllers.

Abstract Associate Name Server

These problems arise because associates are being implicitly grouped into associations without any input from the user. This is in contrast to explicit binding of processes into sessions, which is based on explicit session names provided by users. The solution then is to support explicit binding of associates. Imagine an RMI-like server that supports registration and lookups of named objects. The register operation does not store a reference to the registered object. Instead it creates a replica of the registered object, that is, a new associate in the association to which the registered object belongs. Similarly, the lookup call does not return a reference to the named object. Instead, it returns a replica of this object.



Thus, the process that registers the object and the name server, and the processes that lookup the object all have associates kept consistent using some synchronization scheme such as broadcast method declarations or multicast calls. These processes do not have to run the same program deterministically as in implicit binding. Instead, they must agree on the name of each association created in a session.

Association Name Server

This approach is implemented by the Sync server, which is built on top of the RMI server. The following Sync programs concretely illustrate the approach. Moreover, as some of the exercises are based on Sync, the examples give some of the details needed to do these exercises.

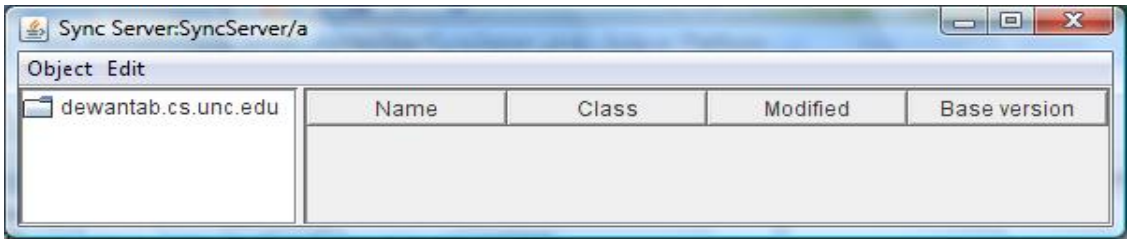
The code executed to start the Sync server is given below.

```
package sync_server_starter;
import edu.unc.sync.server.SyncServer;
public class StartSyncServer {
  public static void main(String[] args) {
    String[] myArgs = {"--ui", "--trace",
       "--server_id", "A"};
    SyncServer.instantiate(myArgs);
   }
}
```

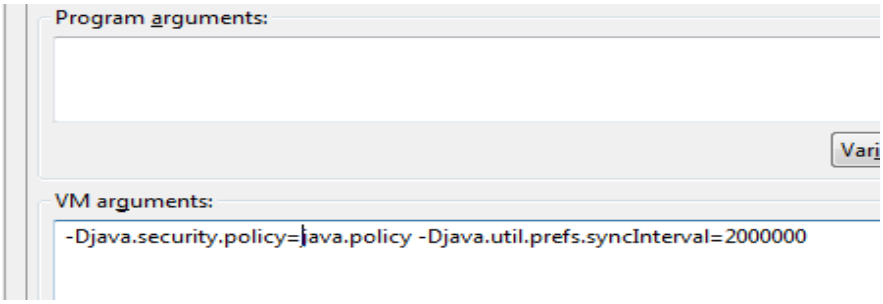
Trace client-server interaction

Takes place of port #

The program simply calls the SyncServer method, instantiate, to start the server program. This method takes a list of String options. The "server_id" option specifies the name of the Sync server on a host. Here we have named it "A". It takes the place of the port number in the case of an RMI server. The "trace" argument makes the server print the register, lookup and other operations on the console. Unlike the RMI server, the Sync server can display the objects registered with it. The "ui" option asks it to create this display.



To allow the Java Sync server to be accessed by other processes, we must pass the following VM arguments to the program launching it.



The argument, java.policy, is a file stored in the directory from which the program is launched, and has the following contents.

```
grant {
  permission java.security.AllPermission;
};
```

The following is a client program that registers an associate with the Sync server and uses ObjectEditor to interact with it.

```
package budget;
import bus.uigen.ObjectEditor;
import edu.unc.sync.Sync;
import edu.unc.sync.server.SyncClient;
public class SyncBudgetCreator {
static String SERVER_NAME = "localhost/A";
static String MODEL_NAME = "demoBudget";
static String CLIENT_ID = "alice";
    public static void main(String[] args) {
        String[] syncArgs = {"--oe"};
        Object model = new AConcertExpense();
        Sync.replicate (SERVER_NAME,
                MODEL_NAME,
                model,
                CLIENT_ID,
                syncArgs);
        ObjectEditor.edit(model);

    }
}
```

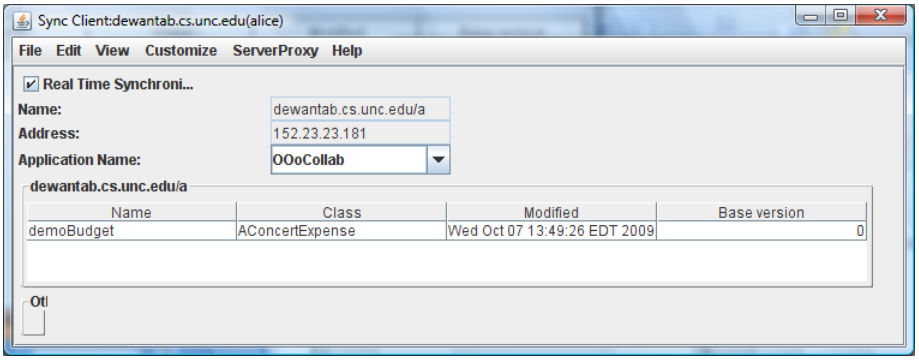
register (name, object)

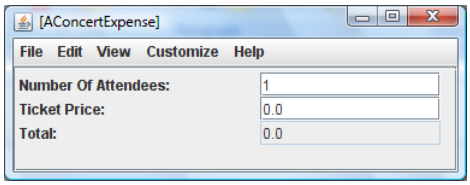
Takes place of port #

For session management, awareness and other user-specific functions.

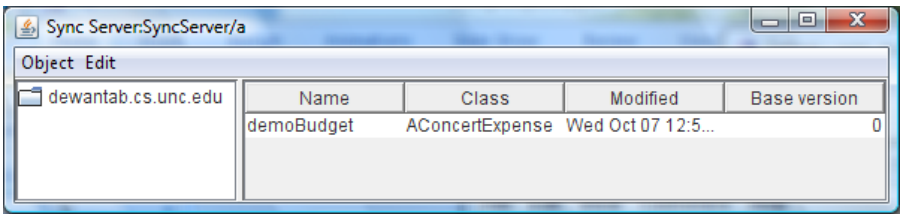
Client-side UI

Like the RMI server, the binding operation specifies both the name of the server and the object being registered. In addition, it gives the name of the replica process registering the object, which as we will see later is used in Sync to allow application programs to create collaboration-aware programs. The syncArgs argument gives optional arguments. The one used here asks Sync to create an ObjectEditor user interface to control synchronization and display session state. Thus, this program starts two ObjecEditor interfaces, one for interacting with the registered model, and one for interacting with the Sync infrastructure.





After the register operation, the server UI displays information about the registry.

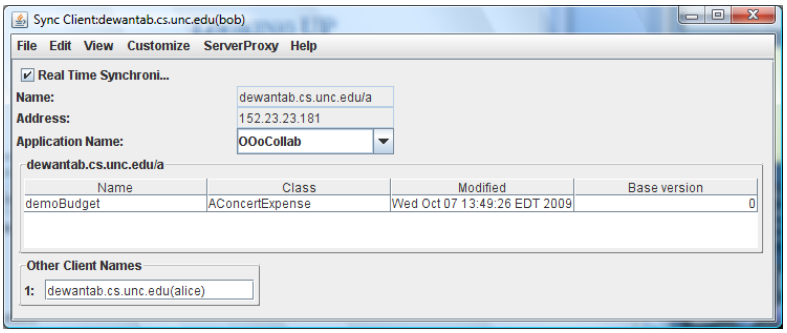


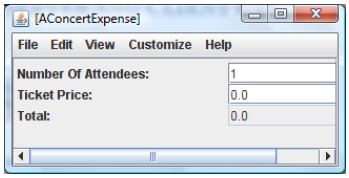
The following program illustrates the use of the lookup operation.

lookup (name)

```
package budget;
import bus.uigen.ObjectEditor;
import edu.unc.sync.Sync;
public class SyncBudgetUser {
static String CLIENT_ID = "bob";
    public static void main(String[] args) {
        String[] syncArgs = {"--oe"};
        ConcertExpense model = (ConcertExpense)
           Sync.lookup(SyncBudgetCreator.SERVER_NAME,
              SyncBudgetCreator.MODEL_NAME,
              CLIENT_ID,
              syncArgs);
        ObjectEditor.edit(model);
    }
}
```

The program is much like the one used by the associate creator except that it returns rather than registers a model. As before two ObjectEditor user interfaces are created by it.





Sync uses Java Serialization to create replica copies. Hence the registered objects must implement this interface and declare appropriate transient variables. If the class of the object defines the method, initSerializedObject(), then Sync calls it after receiving a serialized version of it from a remote location.

Sync does not currently handle cyclic logical structures – hence you must break cycles using transient variables and re-create them after receiving a serialized copy.

In the following outline of the concert expense example, the lines in blue indicate code written to support explicit associate binding in Sync.

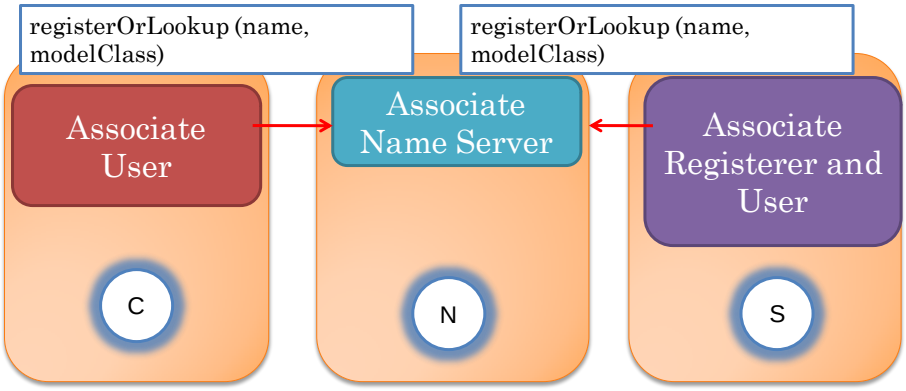
```
public class AConcertExpense implements ConcertExpense,
    Serializable {
      float unitCost = 0;
      int numberOfAttendees = 0;
      transient PropertyChangeSupport propertyChange =
            new PropertyChangeSupport(this) ;
      public float getTicketPrice() { return unitCost; }
      ...
      public void initSerializedObject() {
            propertyChange = new PropertyChangeSupport(this) ; }
}
```

Symmetric Associate Name Server

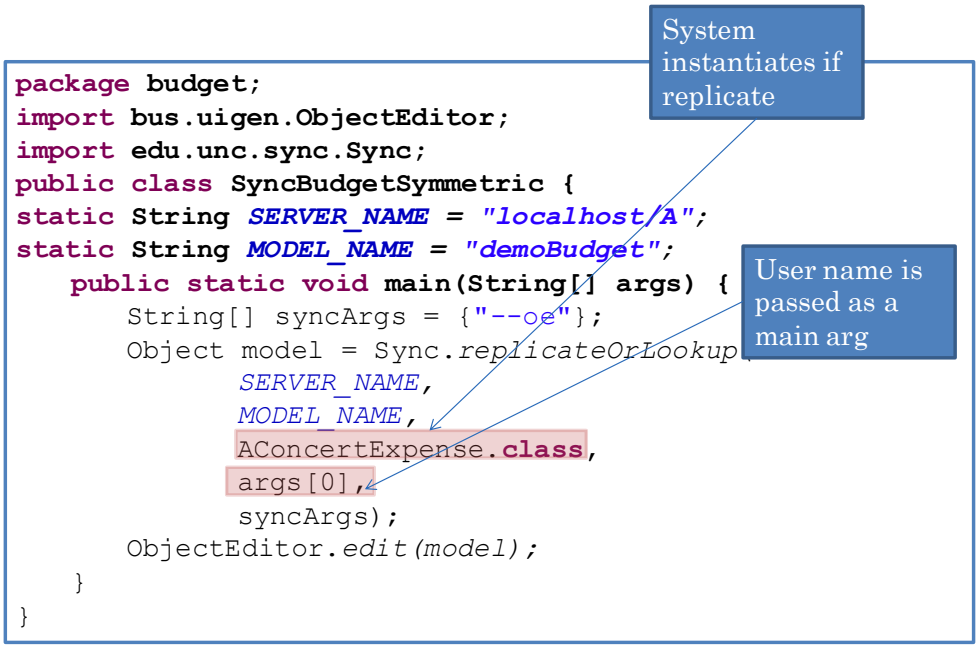
Explicit name binding, as defined above, requires the developers of each new application to write two different programs: one that registers an associate, and another that looks it up. The former program is required to start a collaborative session with the program, and the latter is required to join it. This approach is consistent with asynchronous manipulation of some shared artifact. One user creates the artifact and allows/invites others to access it, while other users simply access a created artifact. For instance, to share a directory in Groove, one user creates the directory on his/her computer, and others simply replicate it on their computers.

However, this approach does not work well in several synchronous collaborative sessions, especially those involving lab experiments. The users must coordinate with each other to decide who will start the session, and race conditions can occur if the creating and joining programs are executed about the same time, that is, a joining program may invoke the lookup call before the creating program executes the register call.

A solution to this problem is to combine the register and lookup operations into a single operation and let the name server decide whether the combined operation does a register or lookup based on when the combined operation is executed. As it may behave like a lookup, it must return an associate and not take an associate as an argument. However, when it behaves like a register, it must have an associate to register. Since this object is not passed to it as an argument, it must create the object. To do so, it must know how to create the instance. As shown below, it can take the class of the associate as an argument.



When it is called, it sees if an object with the specified name exists. If the associate exists, it behaves like the lookup operation. Otherwise, it creates a new instance of the specified class and registers it. We can now replace the two application programs above with a single one.

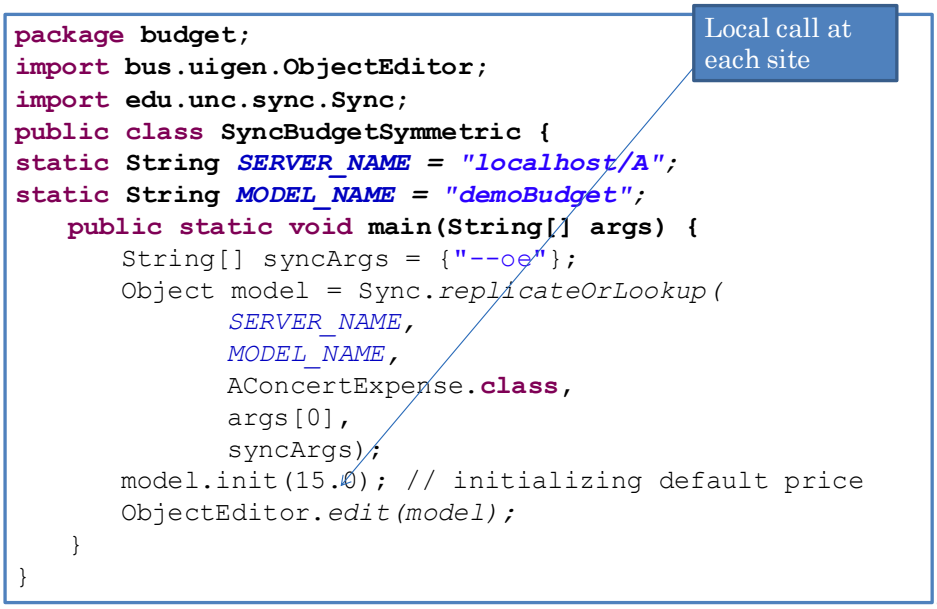


The client identifier is now passed as a main argument so that each user can run the same program. Of course, users on different devices will run different programs, but even in that case the combined operation does not have race conditions.

Factory-based Name Server

The symmetric operation requires the system to instantiate the class of the replicated object. This means it must choose a constructor of the class for the instantiation. As we have not passed constrictor arguments to the call, the system must try to find a parameter-less constructor and instantiate this constructor. This raises two problems. First, there may be no parameter-less constructor in the class. Second, the constructor may throw an exception application-specific handling.

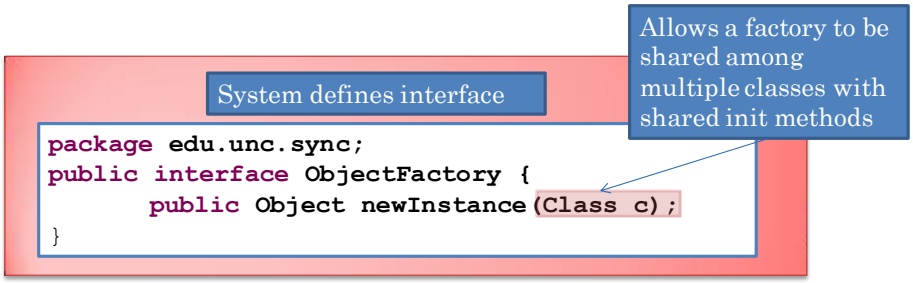
One way to address the first problem is to allow the programmer to pass constructor arguments to the symmetric operation. However, as constructors of different classes can take different arguments, there is no way to ensure, at compile time, that the actual and formal parameters of the constrictor match. Another approach to address this problem is to require all replicated classes to provide parameter-less constructors. Once the object has been instantiated, the application program can make a local call to a public init method it to initialize the object.



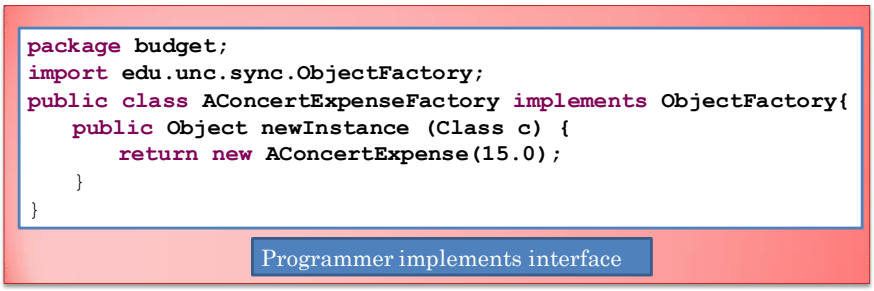
This approach, however, has several problems, which arise from the fact that this call is made at each site. If the method is computationally expensive, computing resources at all but one site are unnecessarily used. If the method is not idempotent, then the initialized state of the associates may diverge. Most important, if an application program joins the session after the initial state has been changed, it will reinitialize this state! Finally, this approach does not address the problem of constructors throwing exceptions.

The key to solving these two problems is to realize that the system knows *when* a class should be instantiated and the application knows *how* to instantiate it. So instead of passing a class to the replicateOrLookup() operation, the application should pass a method that does the instantiation. Several object-oriented languages such as Java do not support method parameters. However, they do support object parameters, and an object is just a collection of methods and some optional state. Thus, we can pass to the operation an object whose sole purpose is to instantiate a class. Such an object is called a factory, as it churns out new instances. The interface of the factory must be predefined by the infrastructure, which must call some well-known method in the object to perform the instantiation.

To illustrate, consider the factory interface assumed by Sync.

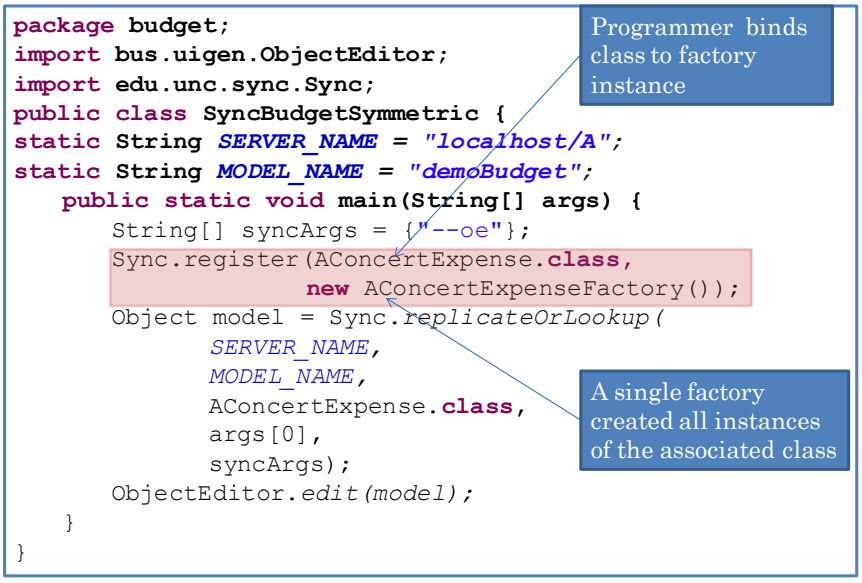


The interface defines a single operation, newInstance(), which is responsible for creating the object to be registered. The following is an example implementation of this interface.



Sync passes to this method the class of the object to be instantiated to allow the factory to be used to create instances of multiple classes. The factory above simply ignores this argument. The default factory in Sync tries to call a parameter-less method in the class.

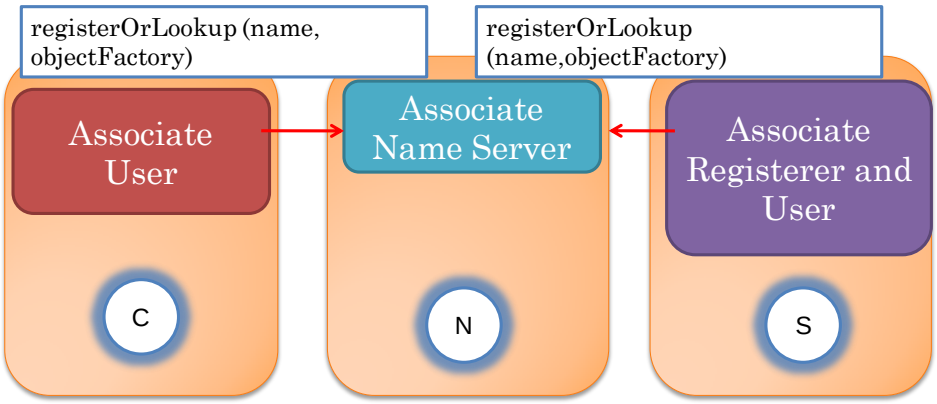
The following code shows how the default factory may be overridden.



The application program associates a model class with a factory using the register (Class, ObjectFactory) method. It still uses the replicateOrLookup() operation that takes a class as an argument. To instantiate the class, the system searches for an associated factory, and if it does not find it, it uses the default

factory. In either case, it creates the new object by calling the newInstance method of the factor, passing it the class.

The register(Class, ObjectFactory) operation allows programmers to essentially pass a factory rather than a class to the replicateOrLookup() operation. The following figure abstracts out this point.

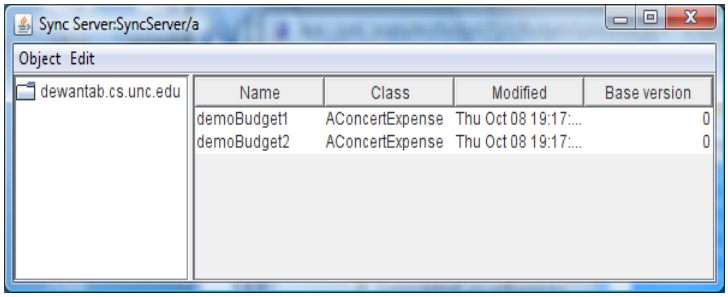


Aggregate register and lookup

It is possible for a single factory object to create multiple instances of a class. This is illustrated in the following variation of the Sync replicateOrLookup() call, which is used to atomically register or lookup multiple associates. This operation takes as arguments arrays that give the names and classes of these objects, and returns an array of instances, one for each element of the class array. In this example, the two classes in the array are the same. The factory object associated with the class is used to create both instances of the class.

```
package budget;
import bus.uigen.ObjectEditor;
import edu.unc.sync.Sync;
public class SyncBudgetsSymmetric {
    static final String SERVER_NAME = "localhost/A";
    static String[] modelNames = {"demoBudget1",
    "demoBudget2"};
    static Class[] classNames = {AConcertExpense.class,
                                 AConcertExpense.class};
    public static void main(String[] args) {
        String[] syncArgs = {"--oe"};
        Sync.register(AConcertExpense.class,
                new ConcertExpenseFactory());
        Object[] models = Sync.replicateOrLookup(
            SERVER_NAME,
            modelNames,
            classNames,
            args[0],
            syncArgs);
        for (int i = 0; i < models.length; i++)
            ObjectEditor.edit(models[i]);
    }
}
```

The server a UI given below shows the effect of the aggregate operation.



Replicated Logical Structures

We have address two issues in the design of replicated objects:

1. Associate binding: How are associates connected to each other in an association?
2. Synchronization: How are bound associates synchronized with each other.

These two issues are independent of each other. Thus, it is possible to use implicit and explicit binding with both multicast calls and broadcast method declarations. While the implementation of the current Sync explicit binder uses neither of these two synchronization schemes, in principle it could support them.

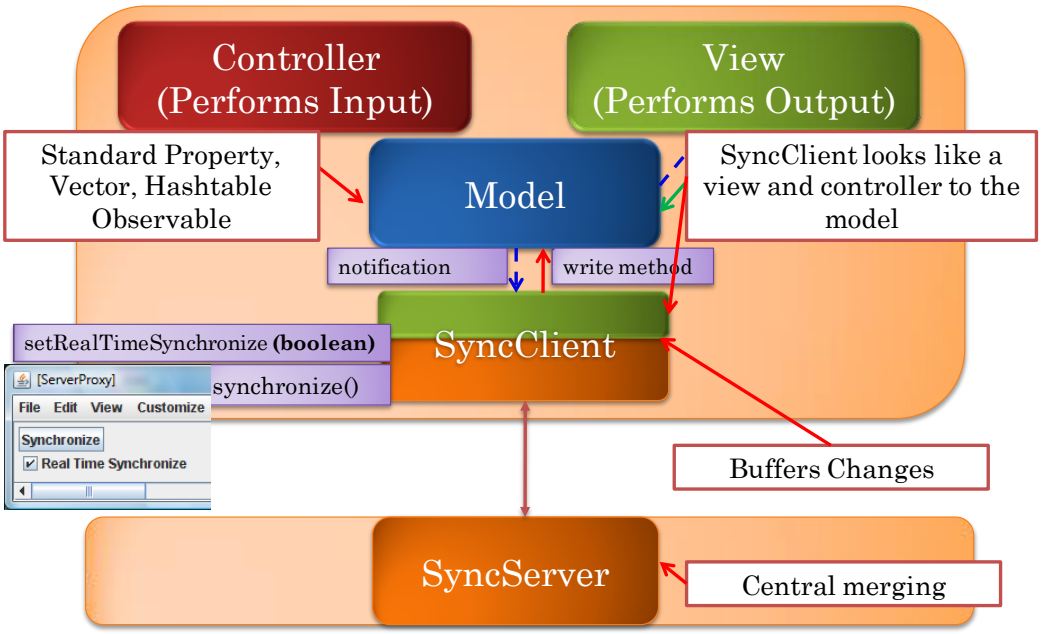
To motivate the actual synchronization scheme supported by Sync, consider the following limitations of multicast calls and broadcast methods.

- Display consistency: They do not ensure that concurrent input does not lead to divergent state at the two sites. These are problems we saw in replicated window systems, and in fact, arise in any replicated system in which the local input is processed immediately at the site at which it is generated.
- Asynchronous interaction/merging: They do not allow asynchronous interaction/disconnection and merging. In principle, it is possible to buffer non-local calls when a user is disconnected or working asynchronously and later, on connection or start of synchronous interaction, replay them in other connected associates. However, this approach requires a way to handle conflicting changes. Multicast calls and broadcast methods associate no semantics with methods, and thus do not know which methods conflict with each other.
- Fine-grained Locking: One way to prevent conflicts is to lock different parts of an structure of an object. However, systems supporting multicast calls and broadcast method declarations consider an object as a collection of methods rather than a manipulator of a data structure.

It is to overcome these problems, we introduced, in Chapter 3, the notion of logical structures of objects, and programming patterns to automatically identify them in programmer-defined encapsulated objects. ObjectEditor used these concepts to display objects. Sync uses them for synchronization. It uses the Bean, list and hashtable programming patterns, and associated notifications, defined in Chapter 3.

This means that that the model is required to follow these patterns and fire notifications – requirements already imposed by ObjectEditor for creating the user interface.

The following architecture of Sync shows how the patterns and notifications can be used to provide synchronization without the limitations above.

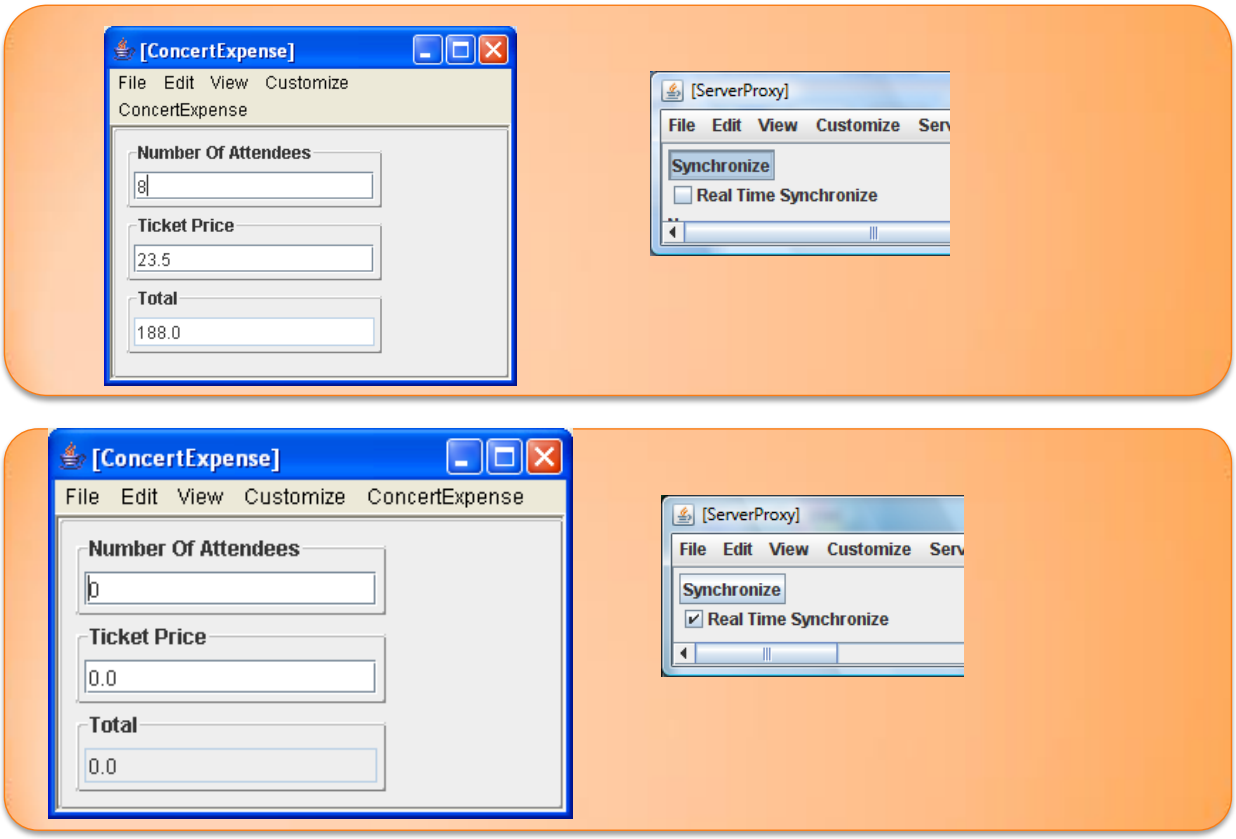


A notification sent by an associate is received by an infrastructure client library, which runs as part of each replicated process in the collaborative session. Information about the notifications is conveyed to the Sync client libraries of all other replicated processes in the session, which invoke corresponding write methods in peers of the associate that generated the notification. As the notifications specify the logical component of the associate that changed, the infrastructure can map them to write methods, and provide default policies for locking, and merging concurrent/asynchronous changes to unlocked components. Since merging is supported, each client library offers (a) the setRealTime() operation to switch between synchronous (real-time) and asynchronous interaction, (b) and the synchronize operation to flush buffered changes. As shown in the figure, its ObjectEditor user-interface provides a way to interactively invoke these commands.

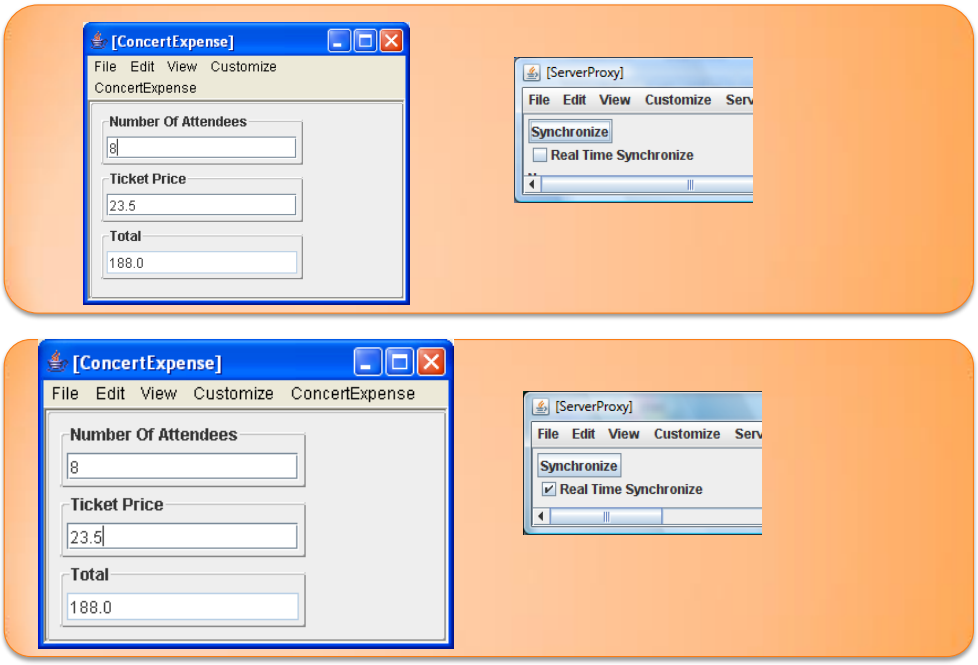
In principle, client libraries can directly communicate with each other after they have retrieved associates from Sync name server. However, correct merging in a fully replicated system is still a research issue. Therefore, the Sync name server also performs centralized merging. All communication between clients is performed through the Sync Server. The exact nature of a merging algorithm is the subject of a future chapter. In the interaction below, we will get a flavor of some of the policies that can be supported by such an algorithm.

Let us begin by understanding integrated asynchronous and synchronous collaboration. The following figure shows two users, Alice (top user), and Bob (bottom user), interacting with the concert expense

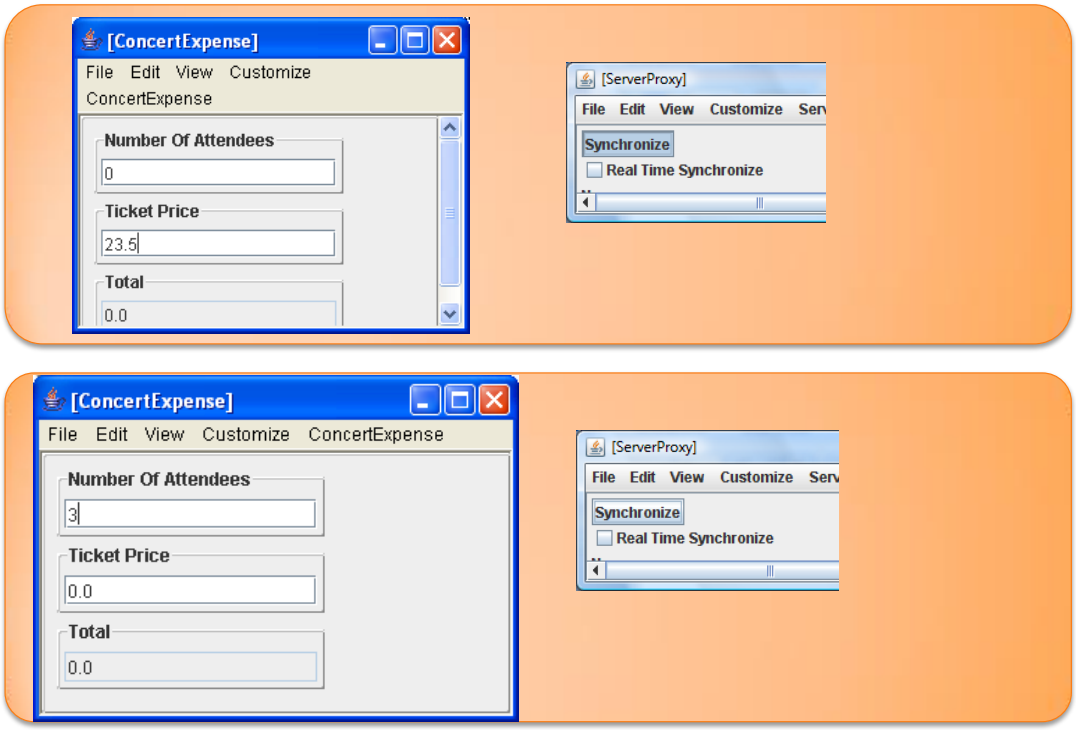
application. Alice is working asynchronously and Bob one is working synchronously, as indicated by the user interface provided by Sync to control the synchronization.



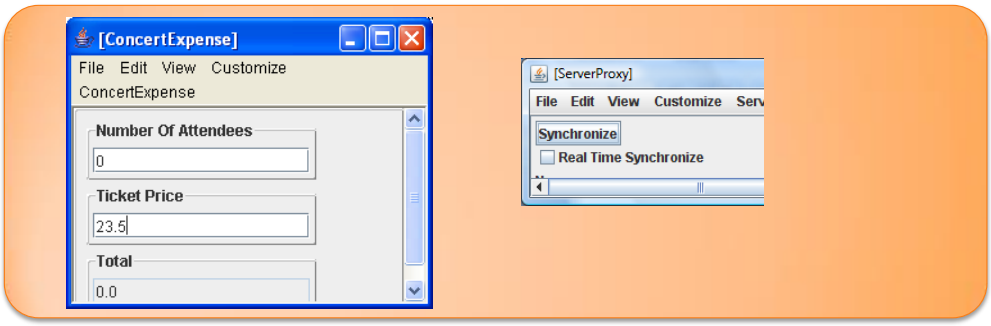
As indicated by the Sync user interface, Alice is about to invoke the synchronize operation. This operation sends the buffered changes to the Sync server, which in turn, sends them to all users working in the synchronous mode. Thus when the operation completes, both users see the same display.

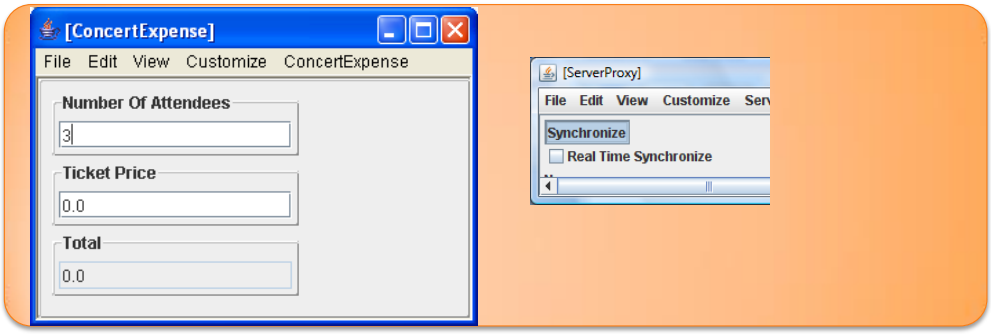


To better understand the event flow and merging in Sync, assume that both users are working in the asynchronous mode, and have made concurrent changes to their associates. Suppose Alice synchronizes first.

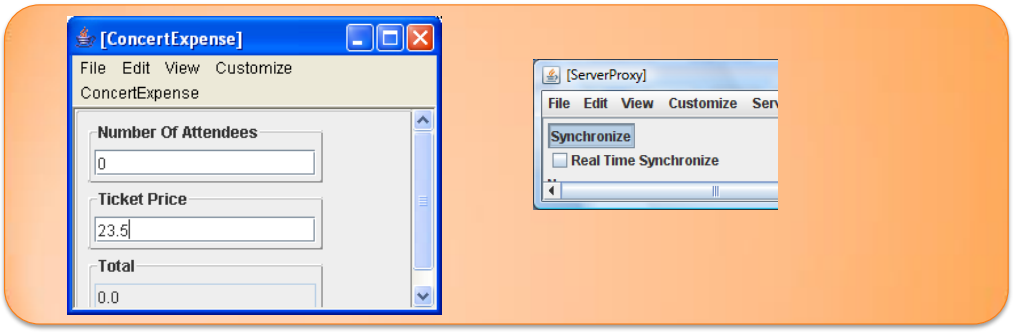


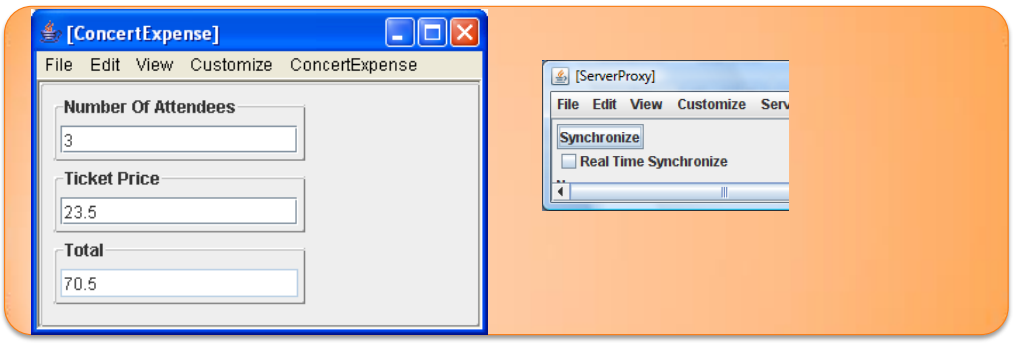
At this point, neither associate changes. The reason is that when the Sync server receives changes from Alice, it does not send it to Bob, as he is working in the asynchronous mode.



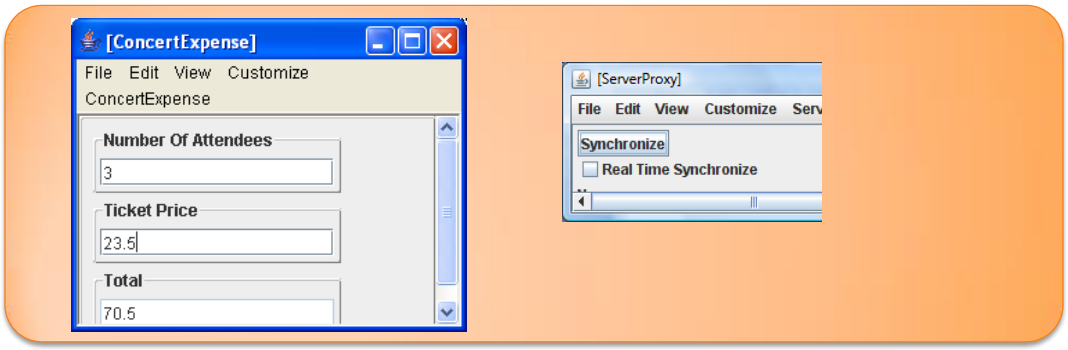


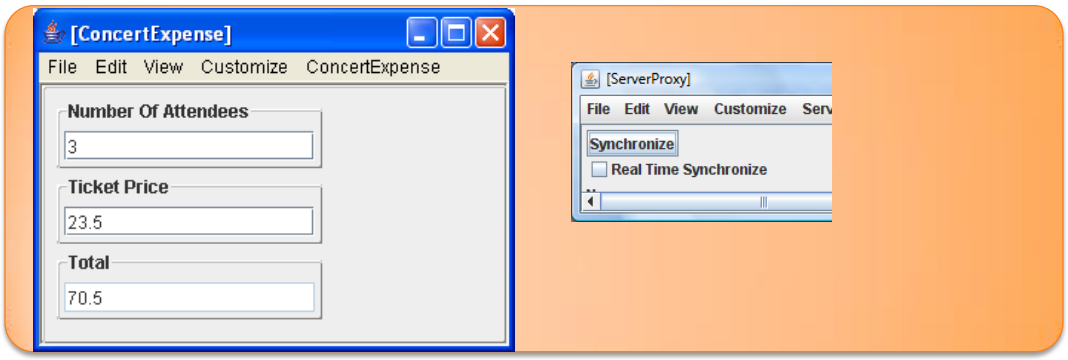
When Bob synchronizes, he sends the Server his changes, and receives Alice's changes. However, Alice does receive Bob's changes. She must execute the synchronize command again to do so.





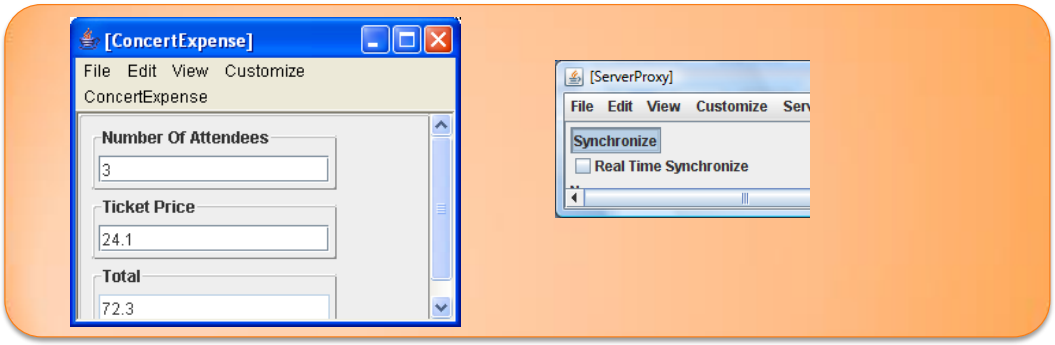
After the three synchronize commands have completed, both associates have the same state.

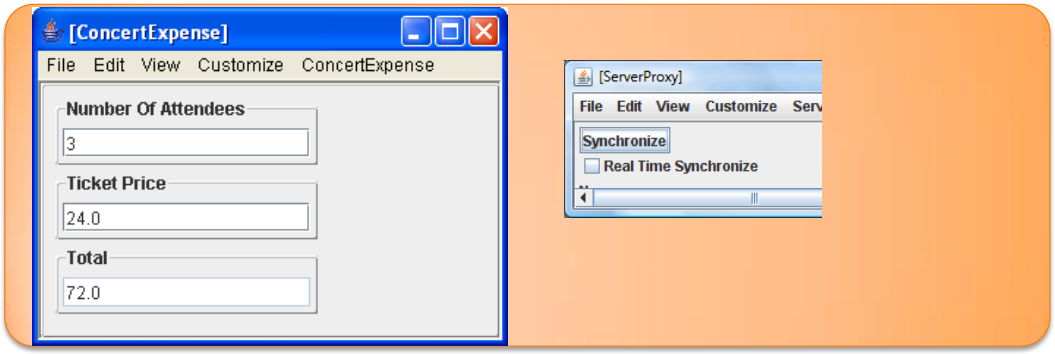




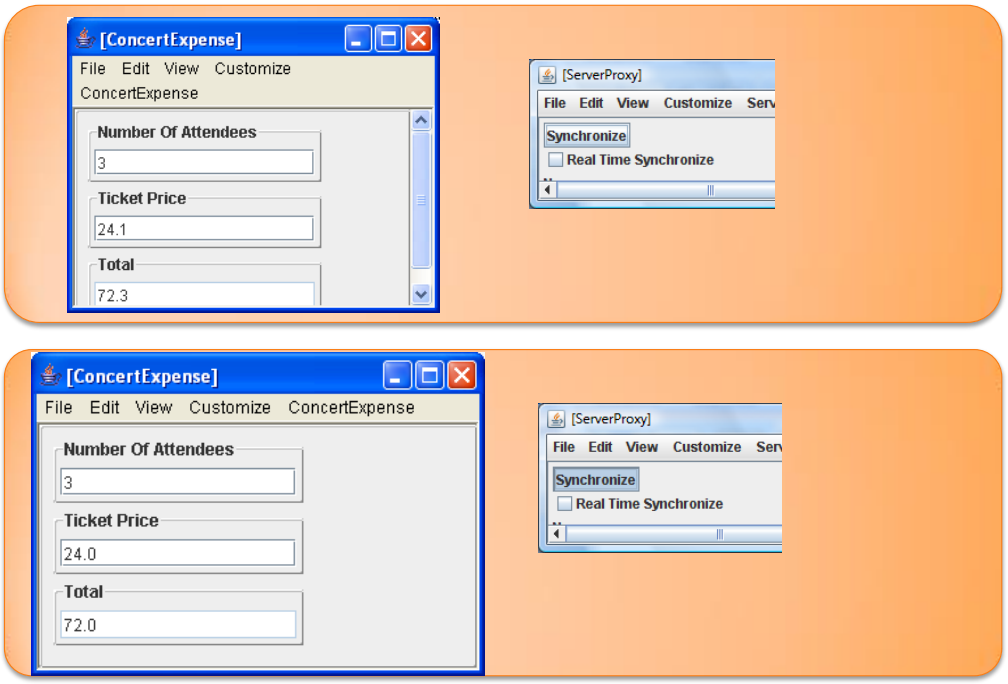
We see here the use of logical structures to synchronize associates. Because different components of the associates were edited by the two users, it was possible to compose them.

Consider now concurrent changes to the same component, shown below.

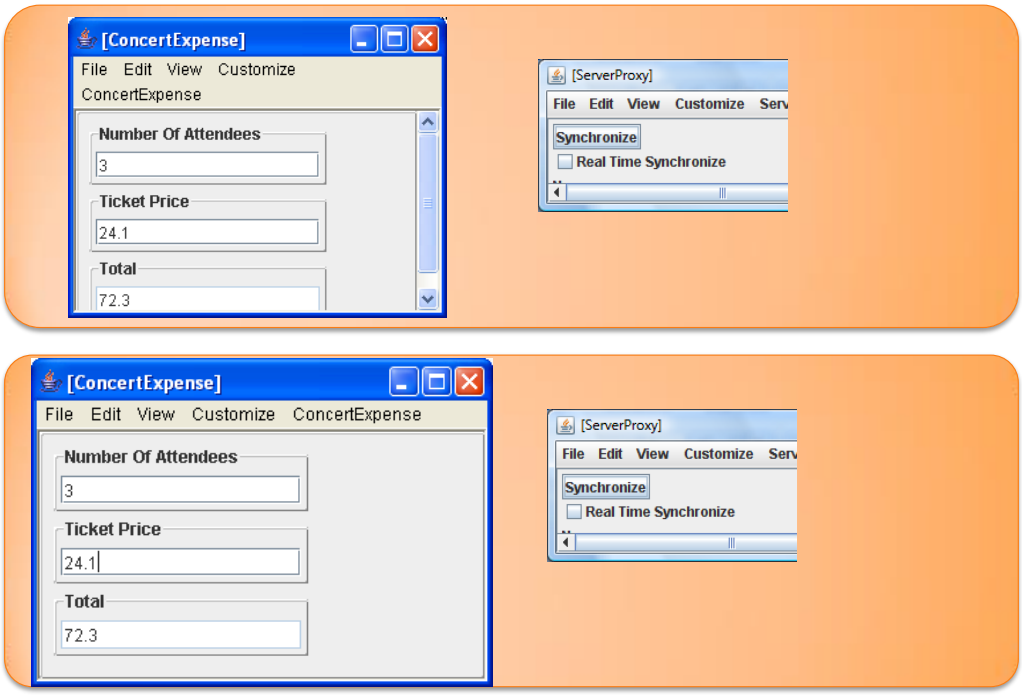




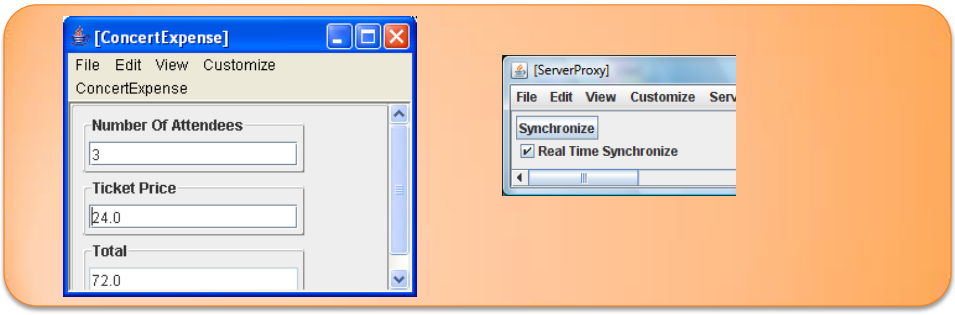
Here, Alice and Bob have the changed the ticket price to 24.1 and 24.0, respectively. When Alice synchronizes, her value is recorded by the server, but Bob does not receive it as he is working asynchronously.

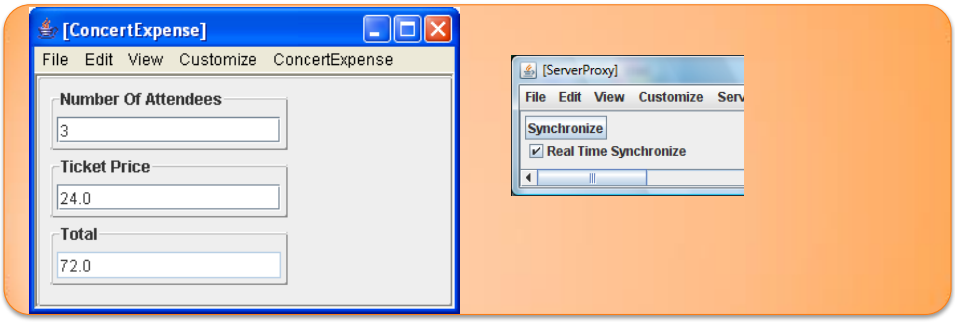


When he next commits, the server knows his change was made without seeing Alice's value. As both changes manipulate the same atomic component, it knows this is a conflict. It resolves the conflict by choosing the earliest conflicting change received by the Server. Thus, it overwrites Bob's change with Alice's value.



Why not choose the later value? When there are conflicts, there is no correct way to merge the changes. As you have seen in Google Docs, conflicts in the document editor and spreedsheet are handled differently – the former uses the Sync approach of choosing the earliest value, while the latter takes the opposite approach of choosing the latest one. In a system supporting disconnection/asynchronous interaction, it can be argued that the latest value should not win. To explain why, consider our example. When Bob synchronizes, if he sees no change to his value, and receives no other notification, then he will simply assume there was no conflict. Alice, on the other hand, may not synchronize for a while, and when she does synchronize, may not check if all of her previous changes had any conflicts. Thus, the conflict may go undetected. When Bob sees Alice's concurrent change, he knows there is a conflict. At this point, he can either accept her value, or override it with his value. If he overrides, the server will accept his change, as it was not made concurrently with Alice's change – Bob saw Alice's value before making his change.


[ConcertExpense]
File Edit View Customize
ConcertExpense
Number Of Attendees
3
Ticket Price
24.0
Total
72.0
[ServerProxy]
File Edit View Customize Serv
Synchronize
Real Time Synchronize


[ConcertExpense]
File Edit View Customize ConcertExpense
Number Of Attendees
3
Ticket Price
24.0
Total
72.0
[ServerProxy]
File Edit View Customize Serv
Synchronize
Real Time Synchronize

Collaboration Unawareness and Non WYSIWIS Coupling

Let us look at the full code of the concert model used in the interaction above.

```
public class ConcertExpense 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", null,  newVal);
            propertyChange.firePropertyChange("total", null,   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", null, newVal);
            propertyChange.firePropertyChange("total", null, getTotal());
    }
    public float getTotal() {return unitCost*numberOfAttendees;   }
    public void addPropertyChangeListener(PropertyChangeListener l) {
       propertyChange.addPropertyChangeListener(l);
    }
   public void initSerializedObject() {propertyChange = new PropertyChangeSupport(this) ; }
}
```

It is identical to the model of chapter 3 except for the blue lines, which are required to make it serializable. One can argue that it has no distribution awareness, as it needs to be serializable to be saved in a file and loaded from it. Moreover, it has no awareness of the fact it is interacting with multiple users. Thus, it is collaboration-unaware. Similarly, the model and controller are also distribution and collaboration unaware, as they communicate with local models. In other words, replicated logical structures can be used to support a collaboration and distribution unaware MVC structure. This is not the case with multicast calls and broadcast method declarations, which require the model, controller, and/or view to be use method declarations or calls that are ware that multiple users are interacting with the code. If we use implicit associate binding, replicated logical structures allows the whole application to be collaboration unaware.

This is an interesting result. The conventional wisdom has been that if we want to share existing collaboration-unaware applications, we must use a shared window system, which supports WYSIWIS coupling. We see here that is possible for different users sharing a collaboration-unaware application to create different views of it. Thus, it is not the case that collaboration-unawareness implies identical user views.

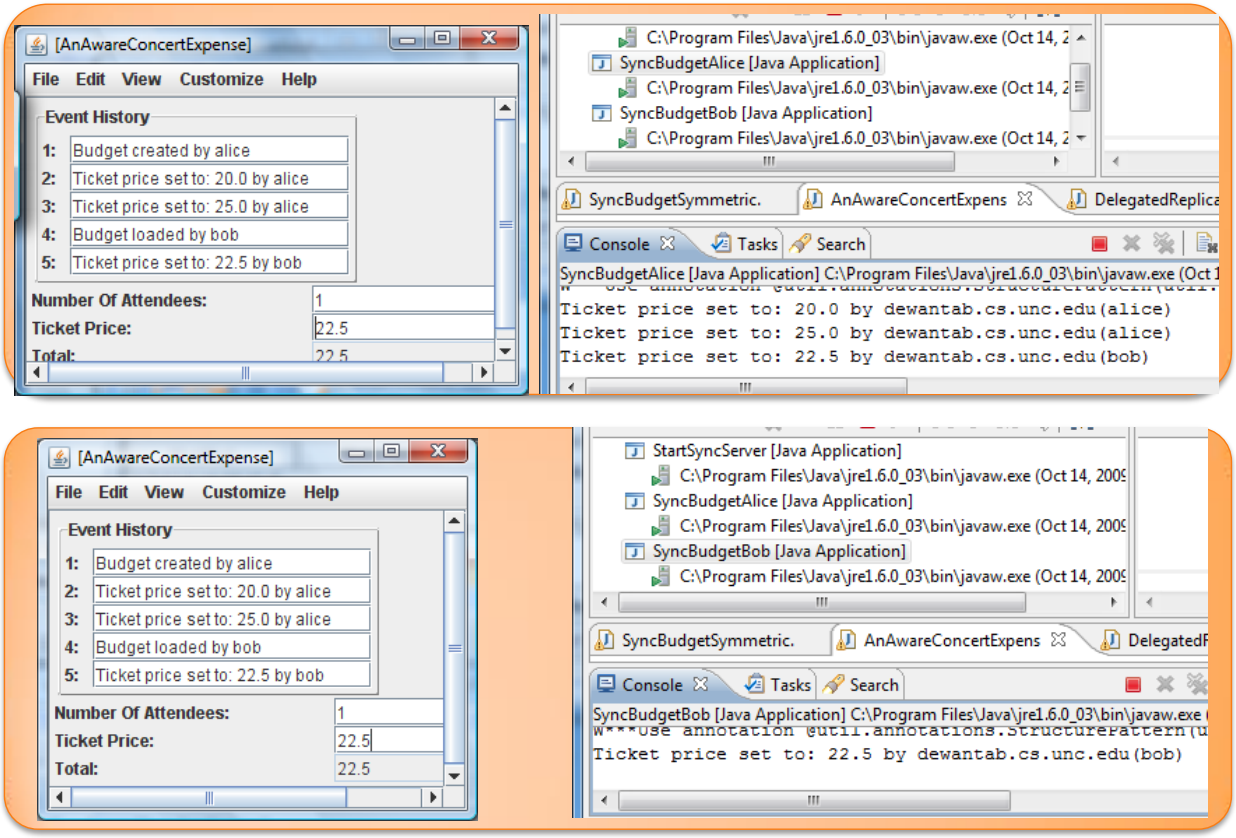
On the other hand, we cannot simply use existing applications written for some window system and create multiple views for different users. The applications must use programming patterns and send notifications based on them. It is currently uncommon to find such applications. Thus, unless this changes, *in practice*, collaboration awareness does imply WYSIWIS coupling.

Server and Client Awareness and Event Flow in Sync

Returning to the comparison of synchronization based on multicast calls, broadcast method declarations and replicated structures, while the first two approaches do not support collaboration-unaware applications, they do allow certain useful user-interfaces that cannot be supported by logical structures. For instance, we saw earlier how they can be used to create a MUDs like user interface in which a message shows up differently in the histories of the associates of the message creator and other users. Replicated logical structures, as described above, do not support such asymmetric synchronization.

It is possible to overcome this problem by providing collaboration-aware primitives that allow application programmers to override the replication scheme provided by replicated logical structures. Let us see a few of such primitives provided by Sync that allow associate methods to be aware of whether they are executing in the server or client and, if they are executing in the client, some details of the client. The use of these primitives requires understanding of explicit binding and replicated logical structures supported by Sync.

The following figure illustrates the use of these primitives. Again, the top user is Alice and the bottom one is Bob. The replicated processes interacting with them execute on the same machine in this example.



The histories of both users indicate that Alice created the first associate in the association and made two changes to its ticket price. Subsequently, Bob looked up the associate to create his own serialized copy, and changed the ticket price. The consoles of the two users also show changes to the ticket price. Bob does not see all three changes in his console because he joined the collaboration late.

How did Alice's associate know (a) when it was created, (a) it was the first object in the association. Recall that Sync's explicit binding scheme creates the first associate by instantiating its class, and subsequent associates by creating serialized copies of the associate stored in the server. Thus, when the first associate is created, the instantiating constructor is called. The message that indicates that a new budget has been created must then be in this constructor.

How did Bob's associate know when it was loaded by its Bob's process? Recall that Sync client library calls the initSerializedObject() in every serialized associate it receives from the server. The message that indicates that the budget was loaded must then be in this method.

Both the constructor and initSerializedObject() need to know the name of the client of the associate in which they are called. The collaboration-aware primitive, Sync.getClientId(),can be called by an associate to determine the client id of the user who registered or looked it up.

A user change to the ticket price results in a call to the corresponding setter method in each associate. it is important to ensure that only one of these adds information about the change to the history – in particular the one called by the local controller. This can be done by using the Sync.isRemote()

primitive, which can be invoked by a method to determine it was called by Sync or the local controller. Thus, we are able to avoid the problem of spurious broadcasts mentioned earlier.

Recall, that for each association, the server keeps a serialized associate, which is synchronized with other associates by calling its setter methods. The user-interface above does not generate a message when the server receives the serialized copy or an update to this copy. Sync allows programmers to distinguish calls to a method in the server and client associates. The routine Sync.isServer() can be called by the method to determine if it was been called in the server or a client.

Finally, the messages on the console show both the client id and host name of the user who made the change. The Sync.getSoutceName() call provides this information.

The following code illustrated the use of these primitives to generate the user-interface above.

```
public class AnAwareConcertExpense extends AConcertExpense{
ListenableVector<String> history = new AListenableVector();
    public AnAwareConcertExpense() {
        super();
        history.add("Budget created by " + Sync.getClientId());
    }
    public void setTicketPrice(float newVal) {
        super.setTicketPrice(newVal);
        String eventMsg = "Ticket price set to: " + newVal;
        System.out.println(eventMsg +" by " + Sync.getSourceName());
        if (!Sync.isServer() &&!Sync.isRemote())
            history.add(eventMsg + " by " + Sync.getClientId());
    }
    public void initSerializedObject() {
        super.initSerializedObject();
        if (Sync.isServer()) return;
        history.add("Budget loaded by " + Sync.getClientId());
    }
    public ListenableVector<String> getEventHistory() {
        return history;
    }
    public void setEventHistory(ListenableVector<String> newVal) {
         history = newVal;
    }
}
```

The Sync collaboration-aware primitives used in this example are highlighted. The constructor prints out the object creation message when it is called in the first associate. The initSerializedObject() method prints the object loading message in all serialized client copies of the first associate. Whenever a user changes the ticket price, the associated setter method call in (a) each associate prints a message to the console, and (b) the local associate stores the message in the history.

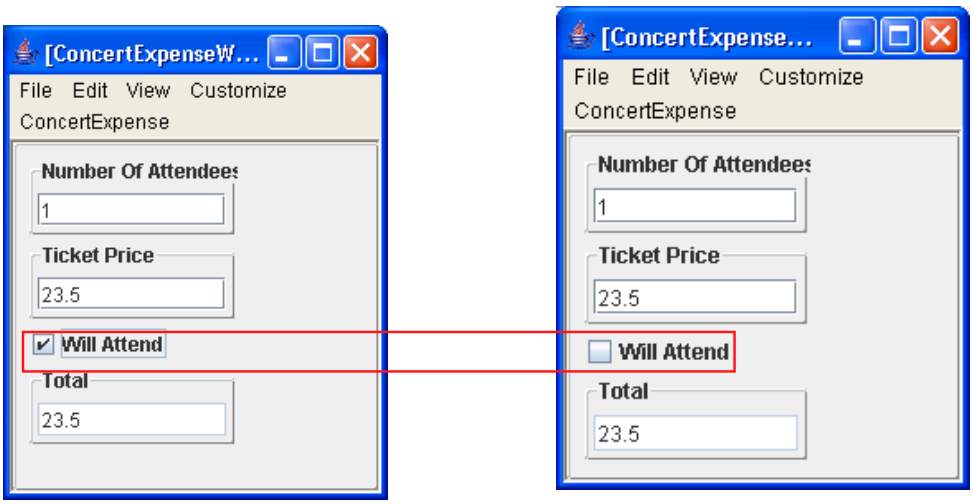
Ignoring Sent and Received Notifications

The example above helps us better understand how events flow between the model, Sync client and Sync server. To further understand this flow, consider model notifications generated when the value of a readonly component such as the total property changes.

```
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", null,  newVal);
                propertyChange.firePropertyChange("total", null,   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", null, newVal);
                propertyChange.firePropertyChange("total", null, getTotal());
      }
      public float getTotal() {return unitCost*numberOfAttendees;   }
      public void addPropertyChangeListener(PropertyChangeListener l) {
         propertyChange.addPropertyChangeListener(l);
      }
     public void initSerializedObject() {propertyChange = new PropertyChangeSupport(this) ; }
}
```

The value of this property is important to the view, as it must update its value. However, it is irrelevant to the replicating infrastructure as there is no corresponding write method to invoke on remote associates. The remote users do see this property update in their user interfaces. This happens because of invocation of the write methods, setNumberOfAttendees() and setTicketPrice(), that update properties on which the readonly property depends.

Thus, the replicating infrastructure processes only notifications regarding changes to editable components. Sometimes it is useful to ignore even some of these notifications, as illustrated by the following example.



In this example, each user sets the WillAttend property to indicate if he or she plans to attend the concert. The side effect of this update is that the value of the NumberOfAttendees property is changed. (Should the NumberofAttendees property still be editable?)

Here, the WillAttend property is private to each associate. Thus, changes to it made by the associate of one user should not be reflected in the associates of other users. We need a way to ensure that the replicating infrastructure does not broadcast these updates.

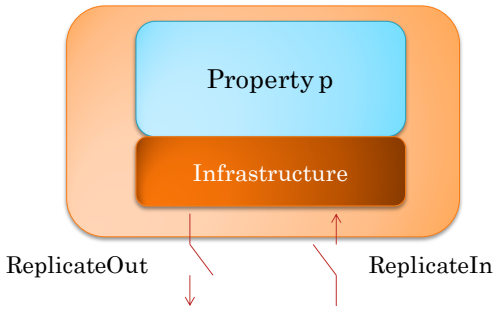
One way to ensure this is to not generate notifications when the property is updated.

```
public class AConcertExpenseWithAttendance extends AConcertEexpense
    implements ConcertExpenseWithAttendance {
     boolean attend;
     public boolean getWillAttend() {
          return attend;
     }
     public void setWillAttend(boolean newVal) {
          if (attend == newVal) return;
          attend = newVal;
          if (newVal)
                    setNumberOfAttendees (getNumberOfAttendees() + 1);
           else
                     setNumberOfAttendees (getNumberOfAttendees() - 1);
          }
   }
```

However, a disadvantage of this approach is that the view and other local observers who are interested in changes to this property do not receive notifications when it changes. Another approach is to use a different protocols to inform local observers and the replicating infrastructure, but this requires programmers to write more notification code and more important, goes against our goal of using standard programming patterns and associate notifications to promote program understanding and pattern-based local and remote tools.

The answer, then, if for the replicating infrastructure to allow application programs to control communication of each kind notification. Below, we see control of property updates. Similar control can be provides for each kind of vector and hashtable change.

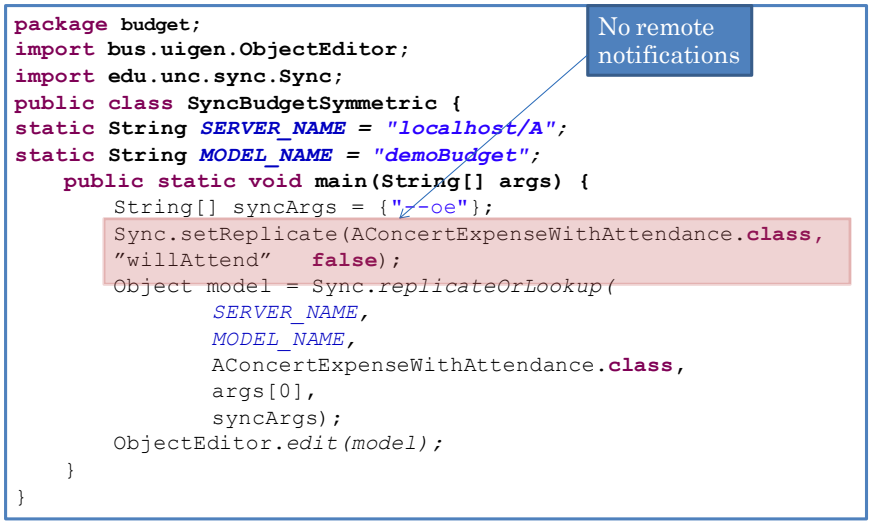
The figure shows a way to control both transmission and receipts of property updates. Each property of each class is associated with a ReplicatedOut and ReplicatedIn boolean attribute. The former determines if local updates to the property are sent to remote associates, while the latter indicates whether updates to it sent by remote associates cause changes in the local associate.



Sync provides collaboration-aware primitives, setReplicateIn() and setReplicateOut(), to set each of these attributes individually. It also provides a call, setReplicate(), to set both of them together. In addition, it provides the call, setReplicateAllProperties(), to set both of them for all properties of a class. Their signatures are given below.

```
Sync.setReplicate(Class c, String property, boolean newVal)
    Sync.setReplicateIn(Class c, String property, boolean newVal);
    Sync.setReplicateOut(Class c, String property, boolean newVal)
Sync.setReplicateAllProperties(Class c, boolean newVal)
```

In the example above, a Sync client should neither send or receive notifications to the WillAttend property. Therefore, the main program of each replicated process uses setReplicate() to set both ReplciateIn and RepliateOut attributes of this property to false.



As a result, the model is able to announce the notification to the private property to local observers.

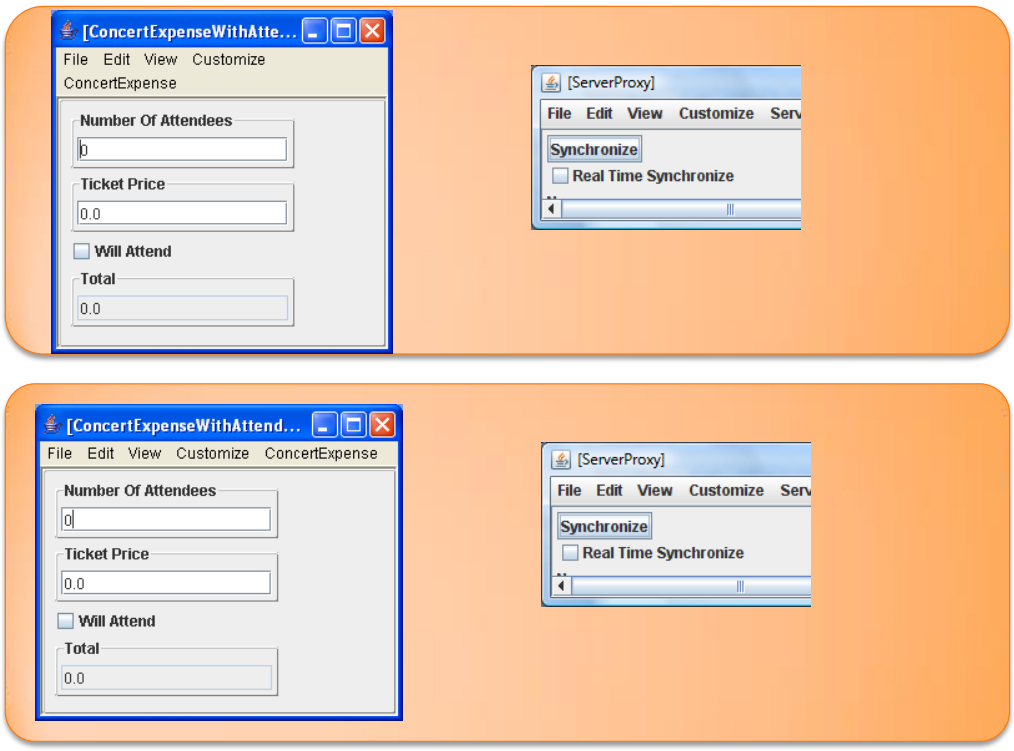
```
public class AConcertExpenseWithAttendance extends AConcertEexpense
   implements ConcertExpenseWithAttendance {
    boolean attend;
    public boolean getWillAttend() {
        return attend;
    }
    public void setWillAttend(boolean newVal) {
        if (attend == newVal) return;
        propertyChange.firePropertyChange("willAttend", attend, newVal);
        attend = newVal;
        if (newVal)
                setNumberOfAttendees (getNumberOfAttendees() + 1);
         else
                 setNumberOfAttendees (getNumberOfAttendees() - 1);
        }
  }
```

Wait Free Synchronization using a Memory Slot per User

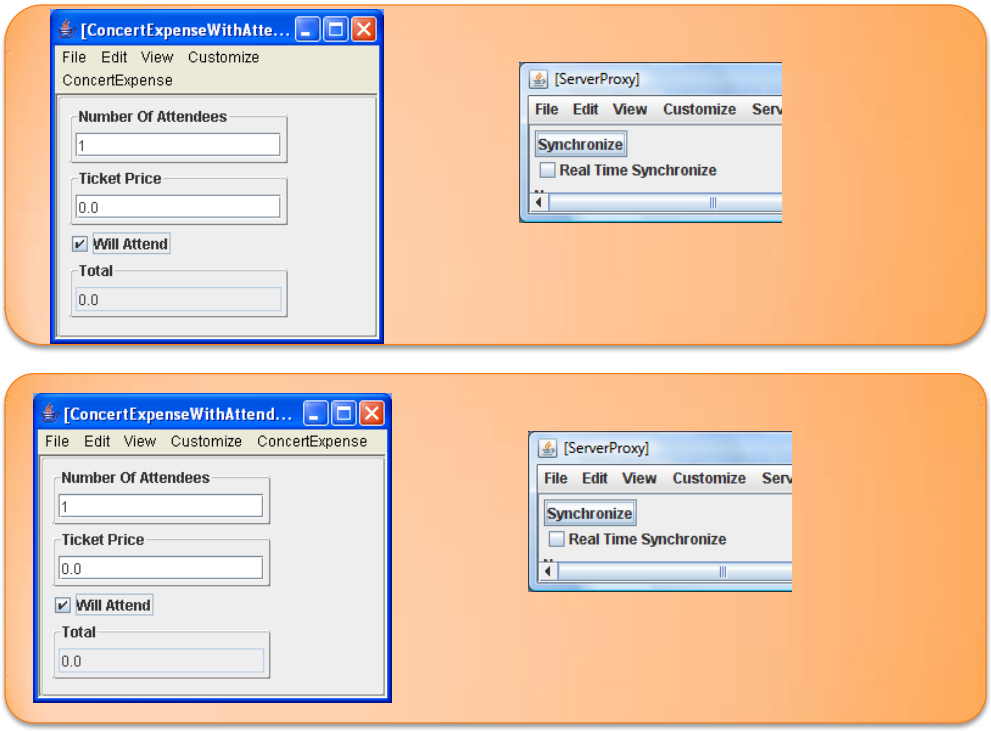
The setReplicate() allows us to make the WillAttend property private. However, let us consider a more fundamental question. Does the model correctly update the NumberOfAttendees property when a user changes the WillAttend property?

Let us look more carefully at setWillAttend(). It increments or decrements the NumberOfAttendees property based on whether the user decides to attend or not. If only one user interacts with the model at a time, this algorithm works correctly. However, concurrent interaction can lead to race conditions that make it fail. The problem is that the increment of NumberOfAttendees, which involves both reading and writing of the property, is not performed atomically. The following interaction illustrates why we need atomicity.

To ensure that race conditions occur, Alice and Bob have turned off real time synchronization.



Both of them indicate they will attend. The setter for this method read the local value of NumberOfAttendees, which is zero in both cases, and increments it to 1.



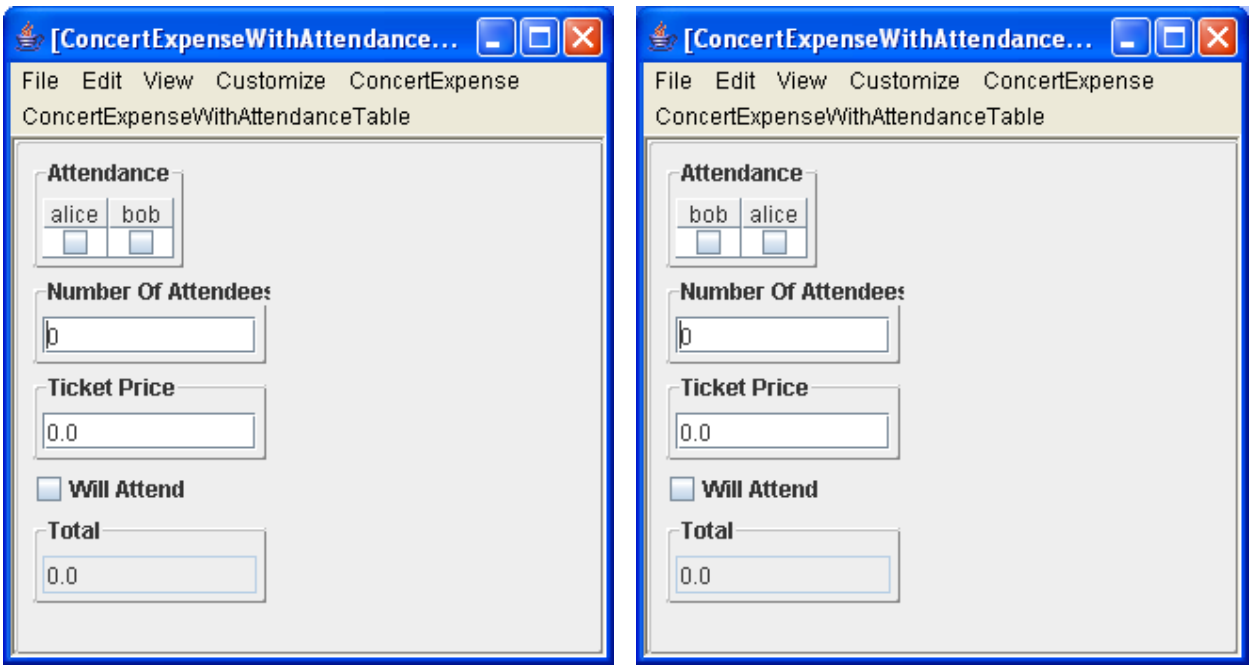
As both users have changed NumberOfAttendees to the same value, after they synchronize their changes, the value of this property remains 1, and thus does not reflect that both users will attend.

We can solve this problem by having the model keep track of who is attending in a table, as shown below.

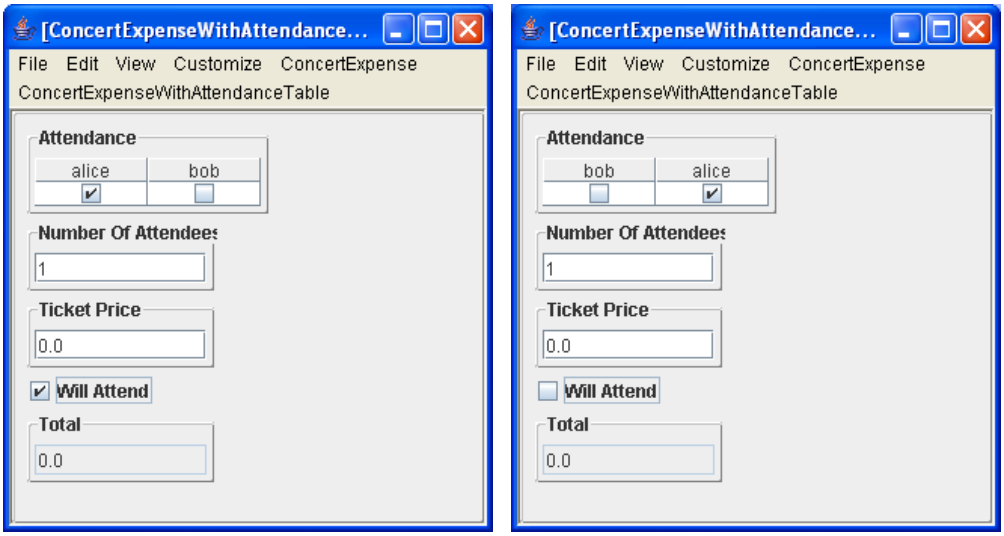
```
public class AConcertExpenseWithAttendanceTable extends AConcertExpense {
    HashtableInterface<String, Boolean> attendance = new AListenableHashtable();
    public HashtableInterface getAttendance() { return attendance;}
    public void setAttendance(HashtableInterface newVal) {attendance = newVal; }

    public int getNumberOfAttendees() {
      Enumeration<Boolean> elements = attendance.elements();
      int retVal = 0;
      while (elements.hasMoreElements()) {

            if ((elements.nextElement()).equals(true))  retVal++;
            return retVal;
      }
   }
    public boolean getWillAttend() {return attend;  }
    public void setWillAttend(boolean newVal) {
            if (attend == newVal) return;
            attend = newVal;
            attendance.put(Sync.getClientId());
    }
```

The table has a slot (key) per user. The method setWillAttend() now fills the slot of the user who calls it, using the collaboration-aware primitive, Sync.getClientId().The method, getNumberOfAttendees(), returns the number of filled slots.As each user fills a different slot, there is no worry about conflicts.

The following examples show that he model now correctly computes NumberOfAttendees. As before, let us assume that Alice and Bob are working in a disconnected fashion.

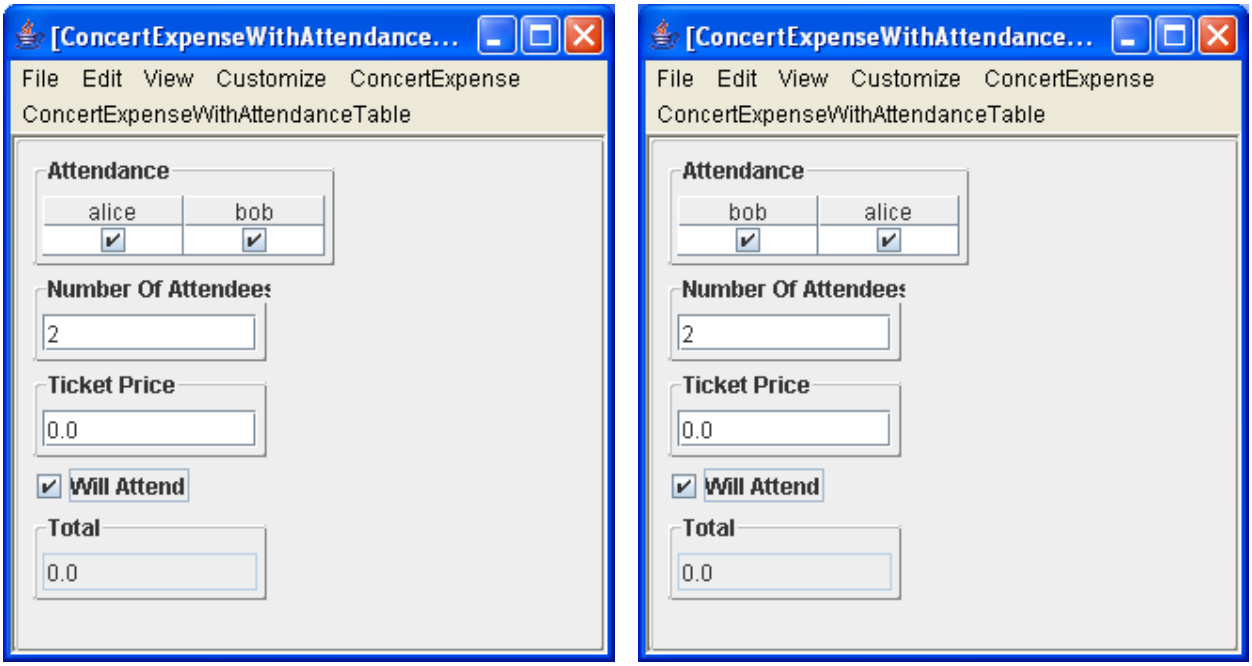


Alice and Bob , both concurrently indicate that they will attend. As their models are disconnected, the NumberOfAttendees property has the value 1, as in the previous interaction without the table.

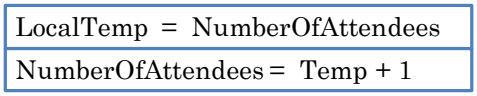


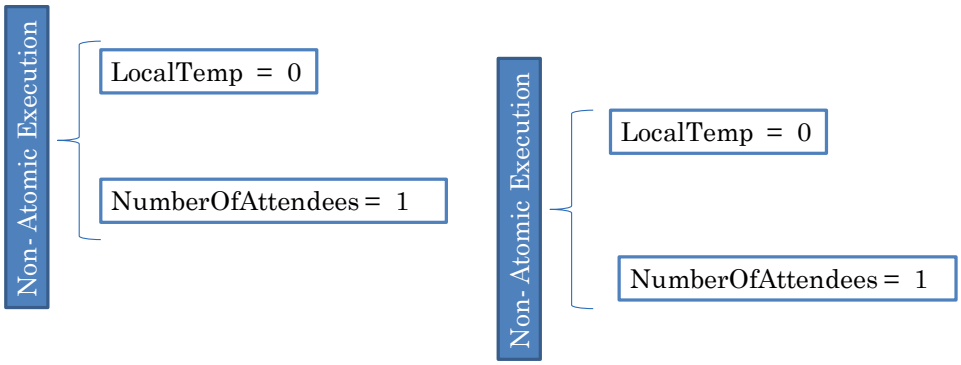
However, after they synchronize their changes, this value changes to 2. The reason is that each user's table gets the (key, value) slot inserted by the other user, and getNumberOfAttendees returns the

number of filled slots, which is now 2.



The example above illustrates an important problem and solution in concurrent computing. The problem with the first solution was that the increment operation was not executed atomically, that is, two executions of this operation could interleave their execution of the read and write steps in this operation. As a result, each of them read the same value, and adds 1 to it, thereby cancelling the other execution of it.





There are several solutions to this problem that require an execution of the operation to wait for a concurrent execution of it. Our solution, on the other hand, does not involve a wait and even allows disconnected concurrent interaction. It is a general example of wait free synchronization achieved by

replacing a shared slot with multiple slots, one per concurrent execution thread. This idea, as far as I know, was invented by Jim Anderson.

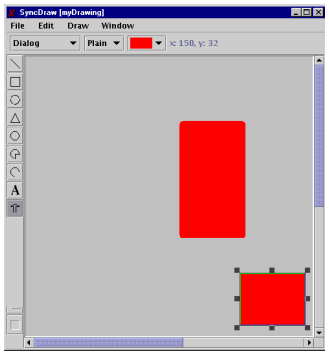
Flexibility of Replicated Logical Structures

Let us wrap up our discussion of replicated logical structures by evaluating their flexibility. They were introduced to remove limitations of broadcast method declarations and multicast calls. Do they have limitations of their own?

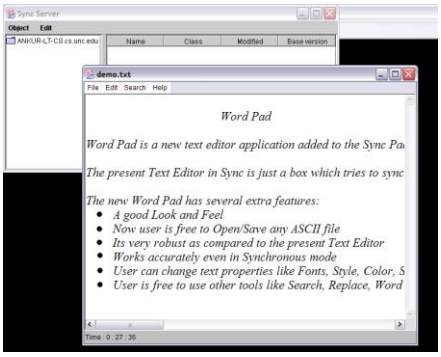
In theory, the answer is yes. They do not allow associates to be arbitrary objects – they must be editors of a predefined set of logical structures, which in our example are Beans, lists and tables.

To understand the impact of this limitation, in practice, let us see try and understand the practical range of objects that can and cannot be supported by replicated logical structures.

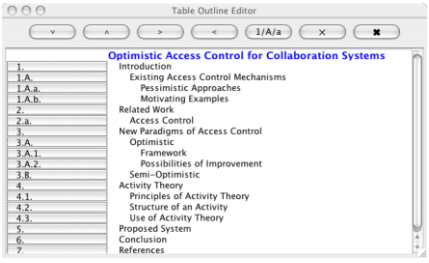
The following figures illustrate the range that can be supported using some examples of actual Sync student projects that build collaborative versions of popular applications.



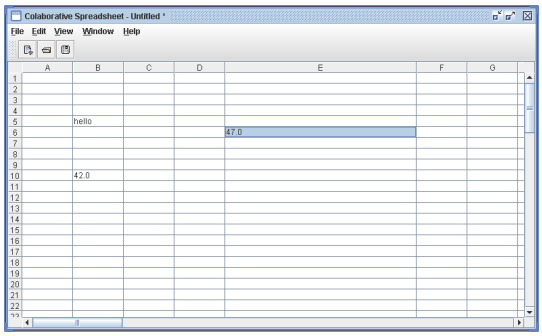
(a) Drawing Tool (Munson)



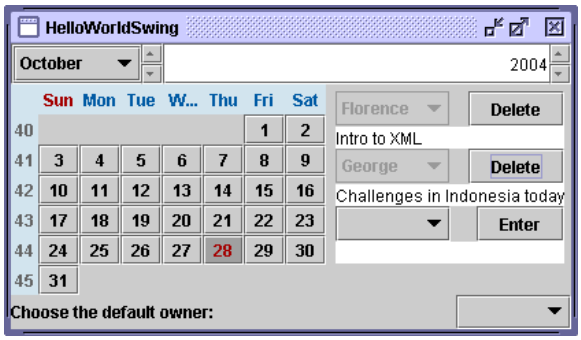
(b) Word Processor (Agiwal)



(c) Outline Editor (Porwal)



(d) Spreadsheet (Luebke)



(e) Calendar (McCuen)

None of these application uses ObjecEditor to create the view and controller, thereby showing that replication and user-interface generation are independent, even though in the examples used so far, we have combined their use.

To better understand the flexibility of replicated logical structures, let us see the logical structures of the models of these applications. The drawing editor is a table whose values are various shapes. Each shape defines geometric properties used by the view to display it. The word processor is a hierarchical sequence whose leaf elements are text objects. Each text object has properties such as font and size used by the view to display it. The outline editor is similarly a tree of text, shown using a different view to display the tree. The spreadsheet and calendar are similarly tables with different behavior and views.

What are some common shared objects that cannot be supported by Sync in particular and the general idea of replicated logical structures in general? Consider first, a stack.

Method Summary

boolean **empty**()
Tests if this stack is empty.

Object **peek**()
Looks at the object at the top of this stack without removing it from the stack.

Object **pop**()
Removes the object at the top of this stack and returns that object as the value of this function.

Object **push**(Object item)
Pushes an item onto the top of this stack.

int **search**(Object o)
Returns the 1-based position where an object is on this stack.

This object cannot be directly supported by Sync because it does not follow any of the programming patterns we have seen so far. However, this is not a limitation of the general idea of replicated logical structures, as it is possible to define a variation of the Vector pattern to support this structure.

A n incrementable counter is a much better example to illustrate the problems of replicated logical structures. We see below its use in overcoming the race condition problem involving concurrent executions of the setWillAttend() method. With multicast calls and broadcast methods, this operation could simply be executed in each associate to give the correct result without requiring a table per user.

```
public interface Counter   {
    public int getCounter();
    public void increment(int amount);
}
```

```
Counter numAttendees = new ACounter();
 public void setWillAttend(boolean newVal) {
      if (attend == newVal) return;
      propertyChange.firePropertyChange("willAttend",  attend,  newVal);
      attend = newVal;
      if (newVal)  counter.incmenet(1)
      else  counter.increment(-1);
```

Replicated logical structures cannot handle the Counter because its increment operation does not indicate how the state of the counter is changed. The write methods supported by logical operations had arguments that indicated how the (external) state was changed. As a result, the infrastructure could synchronize (diff and merge) the state changes of different associates. The increment operation is a blackbox to the infrastructure as it changes the state in an unknown manner. Thus, the example shows the limitation of replicated logical structures – they communicates state changes rather than events, and thus cannot support replication of write operations that compute state changes in an application-dependent fashion.

This discussion, in turn, shows that each of the three synchronization schemes has its advantages and limitations.

Systems Supporting Replicated Types

Multicast calls, broadcast methods, and replicated logical structures have been invented in research systems, and to the best of our knowledge, do not exist directly in industrial strength systems. However, predefined replicated types, exist in several current industrial strength and research systems – in particular Groove, LiveMeeting, and GroupKit. LiveMeeting and GroupKit, like Colab, supports only synchronous updates of associates, while Groove, like, Sync, supports both synchronous and asynchronous updates.

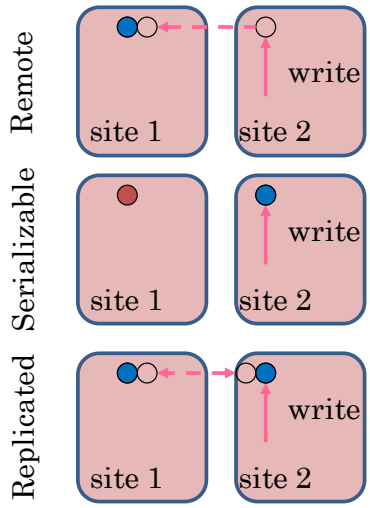
Groove and LiveMeeting support replicated mutable atomic types such as integers and Booleans, and a replicated sequence type roughly equivalent to AListenableVector. Groove also supports replicated tuples, which have the logical structure of Beans. Finally, Groove and GroupKit support a replicated hashtable type roughly equivalent to AListenableVector. As we saw earlier, GroupKit also offers multicast calls.

GroupKit has been a very popular toolkit, and a primary reason for this is the multicast calls and replicated hashtables supported by it. The emergence of LiveMeeting may also be owed partly to replicated types. It is an evolution of a product called PlaceWare, developed by a small start-up of the same name, started by a couple of Xerox researchers, and acquired by Microsoft in 2003. The replicated types supported by it probably were a factor in it being able to compete with the much bigger Webex, as they allowed relatively easy construction of the Whiteboard and user-interface of the PowerPoint Presenter.

Multicast calls, broadcast methods, and replicated logical structures are more powerful than the notion of predefined replicated types, as they allow replication of programmer-defined types such as the concert example above or a new table that, unlike current hashtables, supports keys that are mutable objects. Any of these three concepts can be used to implement any of the predefined types mentioned above.

Techniques for Communicating Parameters

Replicated types are related to serialized and remote types. All three techniques allow processes on different computers to exchange complete objects. The following figure shows the differences between them.



Remote types make it possible for a site to send object references to other sites. Proxies created at the sending and receiving sites ensure that methods invoked on the reference at the receiving sites are executed at the sending site. Thus, all sites share a single copy of the object, but method invocation is a costly operation for the receiving sites. These types essentially support “by reference” passing of distributed parameters.

Serialized types make it possible to send copies of objects to other sites. These types essentially support “by value” passing of distributed parameters. They go far beyond “by value” parameter passing supported in non distributed languages as they dereference pointers, creating isomorphic structures at the sending and receiving parameters. Like value parameters, a received serialized copy at a receiving site can be changed independently of the original object at the sending site.

Replicated types combine elements of remote and serialized types. As in the case of replicated types, the sending and receiving sites share state, but as in the case of serialized types, they access separate local objects storing this state. When the local copy of the shared state is shared, corresponding changes are made to the peer objects in the other sites.

Role of Server in Replicated Objects

The notion of a server seems at odds with the notion of replicated objects. Therefore, it is an optional component of such a system. It is useful to provide session management, merging, explicit binding, and an always connected repository from for downloading and uploading of client state. However, all of these facilities can be provided by a fully replicated system, through a completely distributed synchronous merge algorithm is still a matter of research. To illustrate, GroupKit provides replicated

session management, implemented using replicated tables, and Grove supports asynchronous merging without requiring a central server for storing and merging this state.

Replication Summary

- A replicated system creates associations of objects created by different processes running on the computers of different users.
- Associates can be implicitly bound to each other based on the order in which they were created by their process.
- Explicit binding supports associations consisting of objects create by replicated processes run different programs.
- Factories – objects that create other objects – allow programs registering and looking up associates to execute the same program.
- Associates can be synchronized using broadcast method declarations, multicast calls, and replicated logical structures.
- When a local object invokes a broadcast method in an object, the infrastructure invokes the method also in all remote associates of the object.
- Broadcast methods should not call other broadcast methods to avoid spurious calls.
- Broadcast methods of bind the “what and when” of coupling of an associate class when the class is written.
- Multicast calls explicitly indicate the subset of associates on which a method should be called.
- They bind the “what and when” of coupling of an associate class either at execution time or when the code accessing the class is written.
- Replicated logical structures assume that an object is a manager of a predefined logical structure – in particular a bean, sequence, or hashtable.
- They support merging of fine-grained changes to objects made in disconnected or asynchronous collaboration.
- Broadcast methods and multicast calls require collaboration-aware views, controllers, and/or models. Replicated logical structures support collaboration-transparent models, view, and controllers, to which collaboration-awareness can be optionally added.
- Replicated logical structures do not allow replication of write operations that change object state in an application-defined manner.
- Wait free synchronization can be achieved by replacing a shared memory slot with a slot per concurrent activity.
- Replicated types combine elements of serialized and remote types.

Acknowledgments

Thanks to Kelli for her corrections