Working With Alien Objects: An Object Oriented Viewpoint

Ashok Malhotra

IBM Client/Server Strategy
M/D 2339, Route 100, Somers, NY 10589
MALHOTRA @ VNET.IBM.COM 914-766-2555

Abstract

This paper describes a framework that unifies several different styles of working with distributed objects. Most systems support just one style, typically data shipping or function shipping. We show that multiple styles are necessary and reasonably straightforward to provide and integrate. The style of working with alien objects can be changed without changing the class definition and with very little effect on the application code.

A different style of access makes copies of alien objects and then disconnects. Updating the copies and attempting to reflect the updates to the original objects can lead to consistency problems. This paper presents a simple, practical algorithm for reconciling conflicting updates.

1 Introduction

Alien objects are objects that live in another address space. In this paper we describe a uniform formalism for working with alien objects. Our goals are to allow the programmer to work with alien objects as easily as she works with objects in her own address space and to allow broad flexibility in choosing options on protocols with regard to performance, reliability, availability and data integrity. A key objective is to provide several styles of working with alien objects without having to change class definitions. Similarly, the application code should be as independent of the style of access as possible.

Basically, all classes that can be accessed remotely must inherit from a special class. A distinguished variable that is automatically included in every method specifies the style of access. This variable can be set and changed independently of the application logic to change the style of access.

In allowing the user to intermix styles that move the object to the method or the method to the object this paper occupies the same intellectual territory as Mariposa[13] although the approaches are different. Mariposa is concerned with queries i.e. operations over collections of objects while our concern is with single object operations and navigation. Further, unlike Mariposa, our emphasis is more on user facilities and less on system structure.

A different set of problems arise when an application makes copies of alien objects and then disconnects. If the copies are updated and the application subsequently reconnects and attempts to reflect the updates to the original objects, then problems arise if some of the objects have been updated since the copy was made. We discuss several strategies to cope with these problem situations.

Section 2 introduces the styles of access we discuss. Section 3 outlines our approach to supporting the various access styles simultaneously. Section 4 discusses replication in more detail. Section 5 contains implementation details and may be skipped without losing the thread of the discussion. Section 6 discusses serializability issues in our design. Problems of disconnected operation are discussed in section 7 and some solutions are developed.

2 Connected operation

Persistent object systems can be implemented either by allowing multiple users access to a single copy of an object or by creating multiple copies, one for each user.

2.1 Single copy

A simple and elegant approach is to maintain a single copy of the object in its original address space (we call this the original object) while users in other address spaces contain proxies for the object. The remote sites operate on the proxies as if they were
operating on the original object. In reality, a method call on the proxy causes a remote call to be made on the original object. This is the approach advocated by OMG’s CORBA (Common Object Request Broker Architecture) [7] and implemented by various vendors such as IBM’s DSOM (Distributed System Object Model) [6]. The problem with this function shipping approach is that remote calls are expensive, typically several thousand instructions each, and can result in a great deal of network traffic.

The AS/400 single address space model [10] also keeps a single copy of each object but the architecture allows applications in different address spaces to attach to the object and operate directly upon it.

To maintain data integrity in the face of multi-user access requires some form of concurrency control. DSOM does not provide any concurrency control. The AS/400 persistent object model provides a variation of two-phase readOnly/readWrite locking [4,10]. In fact, a little reflection shows that if serializability is desired, then two-phase locking is the only concurrency control option available with a single copy model. Optimistic concurrency control and time-stamp ordering are not sensible.

2.2 Multiple copies
In this approach, the application gets a copy of the original object in its address space. This requires transmitting the object to the application and if the object is modified, transmitting it back to the source. Most databases use the data shipping or copy model and provide only this model to users. In this paper we argue that users should have multiple models to choose from and, indeed, these are not difficult to provide.

A copy model is more efficient than the proxy model for small objects with many calls per object. To ensure serializability, several concurrency control options are available. Alternately, the user may want to give up serializability for what is called replication, where a single object in an object is instantaneously visible to all users of that object either automatically or (a new option we provide) on demand. This is also sometimes called groupware. Typically, applications that use replication would run without transactions.

2.3 Replication
Arjuna [11] provides two styles of replication: active and passive. In the first, every method that updates the object is run simultaneously on each replica or copy. In the second, updates are only made to a master replica which then broadcasts the new values to all the other replicas. RSOM [6], the SOM Replication Framework, calls these operation and value logging respectively.

Like Arjuna and RSOM we use a master copy scheme for replication. Other schemes, such as voting, have been developed and are discussed in the literature. The consensus seems to be that master copy is the simplest and most reliable scheme [3].

3 Implementation
Objects that can be accessed from other address spaces must inherit from a special class. Let us call this class RemoteEnabled. This class provides several methods as well as data structures to implement remote access and provide control over update behavior. We expect distributed access to be pervasive. Thus, we recommend that all classes be enabled for remote access i.e. the functionality provided by the RemoteEnabled class be included in the base object class.

3.1 Context object
CORBA [7] provides a context object to allow the passing of information that is awkward or inconvenient to pass as arguments. For operations whose definition contains a context clause, the context object is added as a distinguished parameter to skeleton and stub interfaces.

The context object consists of a list of name-value property pairs. Context objects may be created or destroyed and individual context properties may be set or queried. Our proposal is to use the context object to pass information about style of access requested to a remote object. If a context clause is specified for all operations (methods) then calls to remote objects will look exactly like calls to local objects. The context variable, which will be ignored for local calls, will provide the additional information required to determine access style for remote calls. For this purpose we define the following three properties of the context object:

ACCESS with possible values PROXY or COPY
REFRESH with possible values NO, ACTIVE or PASSIVE
WHENREFRESH with possible values AUTOMATIC or ONDEMAND

The REFRESH and WHENREFRESH properties are only meaningful if ACCESS = COPY and are
ignored otherwise. If replication is requested i.e. REFRESH = ACTIVE or REFRESH = PASSIVE, the default is WHENREFRESH = AUTOMATIC.

Context objects should be created in a special section separate from the application logic. This way the style of access can be changed easily without affecting the main body of the application.

Locking information must be included in each method i.e. whether it requires a readWrite, readOnly or a readDirty lock on the object. This does not change with the style of access: proxy, copy or replica. ReadWrite and readOnly are traditional database locks. ReadDirty is, essentially, read without a lock.

Transaction directives, if desired, are included in the application. These may change with the style of access desired.

Notice that access to a complex remote object is as simple as access to a single remote object. Assuming that the component objects of the complex object are linked together by object references, all the application has to do is to access each object with the same context object to get the same style of access for all components. Or, if it so desires, it can change the context object and access components of the complex object in different styles i.e. some by proxy and some by copy.

ACCESS = PROXY: If ACCESS = PROXY, or if no context object is specified or if the context object has no ACCESS property specified, a proxy for the object is implicitly installed in the application when it attempts to access a remote object and we get the default CORBA behavior if no locking is specified in the methods. If locking is specified, then we get additional behavior.

If transactional behavior is desired, the object class must participate in the Object Transaction Service. Details of the Object Transaction service are discussed elsewhere [8]. We shall assume a simple model that represents a transaction by a transient object that can contain a set of object references and responds to methods such as Create, Destroy, AddObject, RemoveObject, Start, Checkpoint, Commit and Abort.

ACCESS = COPY: If ACCESS = COPY, then a copy of the object is implicitly installed in the application when it attempts to access a remote object. Method calls on the object update this local copy. If the application is running within a transaction then at commit time the updated copy of the object is transmitted to the original and replaces it. If the application is not running in a transaction then it has to make explicit UpdateObject calls to update the original object.

If locking is not used, then this policy leads to a sort of optimistic concurrency control with the difference that updates to the object may come from applications that have read earlier or later copies of the object. Locking within a transaction discipline or with carefully coordinated read and update calls would guarantee serializability.

Can copies and proxies be intermixed?: If locking is not used the situation is potentially chaotic. The applications that have proxies can run methods and update the object at any time (as in DSOM today) while the applications that have copies will update at the end of the transaction or with explicit calls.

If locking is used, the design works fine. A readWrite lock may be held by a proxy or by a copy. In either case, no one else can read while the object is being updated. After the lock is released the object can be made available to multiple copies or proxies to read or to one copy or proxy to write.

4 Replication

While the access styles discussed above are server-to-consumer, replication is a peer-to-peer concept and oriented towards groups of users working together with immediate access to one another’s changes. Looking below the surface, though, the master copy replication protocol uses one designated replica as the master, or server, copy. This copy controls and coordinates the behavior of the other copies (replicas).

Our approach is to use the master copy as the server in the access protocols in an attempt to unify the styles of working with distributed objects. For multiple applications accessing a persistent object, the persistent object serves as the master copy. If the network becomes partitioned and one of the partitions has to elect another master [5] then, in effect, we have disconnected operation and independent updates to two master replicas have to be reconciled. This is discussed in section 7.

In master copy replication, all replicas read from the master copy. If a replica wants to update, it asks for a readWrite lock on the master copy. In passive replication, the replica updates itself and sends its state to the master copy and the master copy then broadcasts the updated state to all the replicas that have requested automatic refresh and releases the readWrite lock. Alternately, the replica asks the
master copy to run an update method and then broadcast the updated state to all the replicas including itself. This is the essential algorithm. Many variations are discussed in the literature. See, for example, [1].

In active replication, the replica updates itself and then sends the update method(s) and parameters to the master who broadcasts it to all the replicas that have requested automatic refresh. Replicas that want "on demand" refresh must send a RefreshCopy request to the master whenever they want to be updated. The master responds by sending the latest copy of the object for passive replication or the list of update methods and their arguments since the last refresh for active replication. "On demand" replication can also be used for replicas that disconnect and then reconnect. On reconnection, the application sends a RefreshCopy request for every replica (either immediately or lazily) and updates their state.

From a concurrency control viewpoint, replicas read dirty data: their data is refreshed whenever any of the proxies or copies change their data or when a replica requests a refresh by running the RefreshCopy method. To write data, the replica requests a readWrite lock, writes the data and releases the lock. If it wishes, it can run under transaction control and release all its readWrite locks at the end of the transaction.

5 Class Structures

5.1 At the remote application
Alien objects are accessed either via proxy or via copy. The copy or proxy objects are instances of a class that inherits from the Copy or Proxy class and the object class in question. They are installed automatically as discussed later. Proxies and copies present the same interface as the original object as well as some additional interface to support their function. Both classes store a reference to the original object. The Copy class also remembers when the copy was made as well as a set of update objects consisting of a timestamp, the updated image, a list of methods that caused the update and their arguments and the identity of the application responsible for the update. These are for purposes of reconciling updates to disconnected copies and for refreshing replicas on demand. A disconnected copy may be updated several times before reconciliation. A replica, which is a replicated copy, must also remember that it is a replicated copy, whether replication is active or passive and whether it should be refreshed automatically or on demand.

In addition to proxies and copies the application must also maintain a structure (perhaps implemented as a hash table) that remembers the remote objects for whom it has copies or proxies. Before creating a new proxy or copy, the application checks whether one already exists for the object. If so, the structure returns a reference to the proxy or copy. For simplicity we assume that the application uses the object reference returned even if it is not of the access style that was requested. Without such a mechanism the application can have multiple copies or proxies of a single remote object. With locking, this can create a deadlock if, for example, one copy holds a readWrite lock and another requests a readOnly lock. If locking is not used then multiple copies can lead to lost updates.

5.2 At the original object
The following object structures are required for an object to be accessed remotely. As discussed earlier, the object class must inherit from the RemoteEnabled class. Copies and proxies are represented by RemoteReference objects that contain the object reference of the copy or proxy object. Replicas are represented by RemoteRefreshedReference objects which contain additional information about the RefreshStyle.

We show lock information stored with the original object for simplicity. In fact, it may be better to store this information in a LockManager object that would handle all lock requests and maintain a global picture of the locks requested and in force for deadlock detection.

Similarly, the replication information may be stored in a ReplicaManager object rather than in the original object.

5.3 First reference
The application attempts to run a method using an object reference that may, possibly, be remote. It may have come upon this reference as an instance variable of another object or it may have created the object or it may have received the reference as the return parameter of a method call or as a the result of a query. Alternatively, it may have received a stringified reference through any number of ways such as reading from a file or a screen.
In any case, the application asks the ORB to run the method. Our objective is that the syntax of the request should not change, nor should the class have to change, whether the object is local or remote or on the style of access desired for a remote object.

Assume that the reference is, in fact, remote, and a proxy is requested. In this case, unless a proxy already exists, a proxy for the remote (or original) object is installed in the application by creating an instance of a class that inherits from a proxy class and the object class in question. The proxy presents the same method interface as the original remote object except that the methods are redirected to run remotely on the remote object instead of on the
proxy. The proxy also presents some additional interface as we shall see later.

The elegant solution is to have the proxy class generated and installed dynamically, without the application having to anticipate the classes it will access remotely. DSOM[6] provides a method to create a proxy class in the application. If dynamic installation of classes is not possible, as in standard C++, then the application will have to define proxy and copy classes and include them in the application.

At the remote location, a RemoteReference object is created to represent the proxy and used to update the state of the original object. If the method requests a readWrite lock it sends a LockRequest method to the object. If the lock is available, the lockType of the original object is changed to readWrite and the value of the writer attribute is set to the RemoteReference and the lockRequest method returns TRUE. A copy of the object is stored in the beforeImage attribute. If the lock is not available, the RemoteReference is added to the set of potential writers and the method waits.

Similarly, if the method requests a readOnly lock and the object already has a readOnly lock on it, then the lock is granted and the RemoteReference object created and added to the set of readers. If the object has a readWrite lock the RemoteReference is added to the set of (potential) readers and the method waits. A request for a readDirty lock is granted immediately and does not change the state of the object.

Consider, now, the case where a copy is requested. If the method requests a readWrite lock and the lock is available then a copy object is created in the application and its whenCopied attribute set to the current time. If the lock is not available the method waits. A copy object is an instance of a class that inherits from a copy class and the object class in question. The copy presents the same method interface as the original object as well as some additional interface that we discuss later. The issues for creating copy classes dynamically are the same as discussed above for proxy objects.

Similarly, if a readOnly lock is requested and is available, the state of the object is updated as above and a copy is installed in the application. If the lock is not available the state of the original object is updated and the application waits. ReadDirty requests are granted immediately and result in a copy being installed in the application but no change to the state of the object. If the REFRESH property of the context object is NO, then the actions at the original object are exactly the same for a copy as they are for a proxy. The difference between the proxy and the copy case is that while the proxy is installed when the first lock is requested, the copy is not installed till the lock is granted.

If a replica is requested i.e. a copy is requested and the REFRESH property in the context variable is not NO then, instead of a RemoteReference a RemoteRefreshedReference object is created and, in addition to the above actions, it is filed in the refresh set. The refresh set contains information about all replicas and their style of replication.

5.4 Locking
Consider first the situation where there are no applications requesting replication i.e. the refresh set is empty. In this case, the applications use standard 2-phase readOnly and readWrite locking.

When a readWrite lock is released we check if the set waiting to read is empty. (It is usually better to satisfy the readers first. Other policies may be better in specific situations.) If not, the lockType is set to readOnly and the RemoteReference objects in the waiting to read set are removed from the set, granted readOnly locks, and filed in the set of readers. If the set waiting to read is empty and the set waiting to write is not empty then the first RemoteReference object on the waiting to write set is removed from the waiting to read set, granted a readWrite lock, and installed as the value of the writer attribute and the current state of the object is stored as the beforeImage. The lockType is set to readWrite.

When a readOnly lock is released, we remove the RemoteReference from the readers set. If the set is not empty we need to do nothing. If the set is empty and the set waiting to write is also empty then, again, we need to do nothing. If the readers set is empty and the waiting to write set is not empty, then we remove the first remote object from the waiting to write set, grant it a readWrite lock and change the locking information in the original object as discussed above.

5.5 Update processing
As the application runs, objects are updated. Depending on whether the ACCESS is by copy or by proxy, the original object is updated (via remote method calls) or a copy of the object is updated. In either case, an Update object is created that contains the list of update methods and their arguments, an updated copy of the object if it is a copy, the identification of the application and a timestamp. The application conveys that the original object should be updated by sending an UpdateObject request with the
Update object as a parameter. This causes a copy of the Update object to be created and filed in the updates set of the original object and the lastUpdate attribute set to the current timestamp. The updated copy of the object is not official, however. That is, a log record has not been written out. This is done when the Commit method is executed. If the application is running under a transaction the Commit method is called by the transaction object. Otherwise it has to be called by the copy or proxy. When the Commit method is received, the beforeImage is discarded and the value of the lastCommit attribute is set to the current timestamp.

The transaction or the copy/proxy may also choose to abort. In this case the updated version of the object is replaced by the beforeImage, the entries in the updates set since the last Commit are removed and the value of lastUpdate is set to lastCommit.

The method list in the Update object is required to support active replication. One way of creating the method list is to use a mechanism like the SOMMBeforeAfter metaclass[6]. This provides before and after methods that can be customized and run before and after the execution of each method of a class. To create the method list the before method is written to update the method list with the name of the method and its flattened parameters every time a method is run.

After a successful update, refresh information is sent to each of the copies represented by RemoteRefreshedReference objects on the refresh list with whenRefresh = AUTOMATIC. If refreshStyle = PASSIVE, then the updated state of the object is sent to each copy represented on the refresh list. If refreshStyle = ACTIVE, then the list of methods including their arguments is, similarly, sent to each copy represented on the refresh list.

Replicated objects with whenRefresh = ONDEMAND can send a RefreshCopy request to the original object at any time. If the replica has refreshStyle = PASSIVE, then the current image of the object is sent to the copy. If the replica has refreshStyle = ACTIVE then it is sent the list of all methods run on the object since that replica was last updated. This, in fact, is sent as one or more Update objects i.e. those Update objects that have a timestamp later than the lastUpdate attribute of the replica. Update objects are discarded when no replicas remain with lastUpdate attributes earlier than the timestamp of the Update object.

Consider, now, the case where a replica wants to update the object. It would ask for a readWrite lock on the master copy exactly as if it was running in a non-refresh mode. It may be granted the lock immediately or it may have to wait in the waiting to write set. When it gets the lock it locks out all readers and writers for the duration of the lock i.e. for write operations, replicas behave exactly like copies.

Read operations using ReadOnly locks for replicas are also exactly the same as for copies. In the spirit of replication however they may want to use readDirty locks instead of ReadOnly locks. This creates a dilemma in our design. Either we create two versions of every method: one with readDirty locks for applications with REFRESH = NO and the other with readOnly locks for applications with REFRESH = NO. The latter solution is to interpret requests for readOnly locks as readDirty locks of REFRESH = NO. The other solution has, however, the unfortunate consequence that an application can never use a readOnly lock if it runs with REFRESH = NO. Given this choice, the better option seems to be not automatically reinterpret the lock statements and have the user create almost duplicate copies of methods where required.

6 Serializability
Consider a concurrent set of operations \( A \) with ACCESS = COPY or PROXY but with REFRESH = NO, running under transaction control and using readOnly and readWrite locks that are requested at the start of each method and released at the end of the transaction. It is clear that the set of operations in \( A \) will produce a serializable schedule because they adhere to the conditions required for serializability [9].

Violating any of the conditions, for example, using readDirty locks can make the schedule nonserializable although it is still possible to carefully construct correct transactions.

Now consider a set \( O \) consisting of the union of a set of operations \( R \) where REFRESH = ACTIVE or PASSIVE running concurrently with \( A \). The set \( R \) does not have serializable schedules because it may read dirty data, but the subset \( A \) in \( O \) still has serializable schedules because none of the operations on \( R \) can change data locked by an operation in \( A \). There is a theoretical caveat to this. The updates created by \( R \) may be invalid as they are based on dirty data. Since \( A \) reads objects updated by \( R \) its updates may also be invalid as they are based upon data created from dirty data. In practice, however, \( R \) can be written to generate correct transactions that maintain data integrity.
7 Disconnected Operation
There is increasing interest in systems that maintain copies of persistent objects for availability and reliability. For example, an insurance company maintains its enterprise data on a large, central mainframe. At the end of each day, data relevant to each branch office is extracted and loaded onto branch office machines. Salesmen download data pertaining to specific customers onto their workstations and use it to sell and change policies. As businesses struggle to improve efficiency by making data more available, this is a typical scenario. Shedletsky and Rofanno [12] call it data staging. Data staging is one example of disconnected operation.

7.1 Data staging
For simplicity, assume that the salesman connects his workstation to the enterprise database or main object store. He then runs an extract application that copies several objects onto his workstation. He may want to store these copies out on persistent memory and later run one or more applications against these objects.

Sometime later he reconnects to the main object store and attempts to install the updates onto the original objects. Problems arise if some of the objects have been updated since the copies were made. The remainder of this section develops solutions for coping with these situations.

7.2 Background
There has been a great deal of interest in this problem but no satisfactory general solution has emerged. One interesting line of investigation maintains a log of the transactions run on each extracted set of objects and then uses dependence relations between the objects read and written by each transaction to determine if the updates made on one copy can be made on the other copy without conflict [2]. We discuss this approach later in more detail.

7.3 Rerunning applications
In this paper we investigate another approach: rerunning applications. In data staging, consistent collections of objects are copied from one address space or persistent store to one or more others. Let us call such a collection of objects an object set or OSet. Now, if OSet1 and OSet2 are copies of one another and applications (A1, A2) are run on OSet1 and applications (A3, A4) are run on OSet2, then, somewhat naively, we could create a consistent OSet with all the updates by running applications (A3, A4) on O1 or by running applications (A1, A2) on O2. This makes two assumptions:

- That all the inputs to the applications are available.
- That all the inputs to the applications will be the same even if the application is run on a OSet in a different state.

Object oriented systems provide a great advantage in that all changes of state take place through methods. Thus, if we can capture the methods and their arguments then we can capture all that happens in an application. This is essentially correct except that arguments passed by reference must be flattened and converted into arguments by value as discussed earlier and the input should not change with the state of OSet.

Some of the input to an application is always entered from a terminal. This may be the command that starts the transaction or it may be subsequent input that is entered on visual interfaces. If we capture the methods and their parameters then we capture such input as well. A problem is that the input may have been entered into a dialog box by cutting and pasting from another application or more simply just copied from another window. In so far as these inputs represent states of the application that may be different for different states of OSet this represents a limitation of our methodology.

The next question that arises is "Which methods should we rerun?" Since methods call other methods, rerunning every method would cause massive duplication. Thus, either we structure the application so that objects are updated only by low-level methods that do not call other methods and rerun these methods, or we rerun only the top-level method.

Rerunning the top level method: Every application has a top-level object. This is used to start the application and may be used to start a transaction and set up the initial screen. Applications that have the ability to be rerun would inherit from a special class that would provide two attributes and three methods.

```java
interface RepeatableApplication {
    attribute sequence<objectReference> accessLog;
    // The objects referenced
    attribute char * keystrokes;
    // The input keystrokes
    bool Run ();
    bool RunKeystrokeCapture ();
    bool ReRun ();
};
```
All three methods run the application. The RunKeystrokeCapture method intercepts the keyboard input and makes a copy of it in the keystrokes attribute. The ReRun method runs the method but takes its input from the keystrokes attribute rather than from the keyboard. The Run method also writes an application log. This consists of entries containing a timestamp, the identity of the application and the arguments passed to the run method.

The application log may be implemented as a file. Or it may be implemented as an attribute of a single, global WorkDone object implemented as a set of ApplicationExecution objects:

```java
interface ApplicationExecution
{
    attribute RepeatableApplication application;
    attribute timestamp whenRun;
}

interface WorkDone
{
    attribute sequence<ApplicationExecution> applicationLog;
    bool ClearLog(); //Empties application log
}
```

### 7.4 Dependency

This is the basic approach discussed by Davidson[2] from an object oriented point of view. Essentially, we keep a log of the remote objects accessed by the application. To reconcile the updates the timestamps of the entries in the access log are compared with the timestamps of the original remote objects. If the original objects have not been updated or deleted since the copies were made then there is no conflict and the updates can be applied. If the original objects have changed since the copies were made then there are conflicts and the application must be rerun.

To implement this we need to create an entry in the access log for every remote object that is accessed. This entry can be written when the first method is run on the object. One way of doing this is to use a mechanism like the SOMMBeforeAfter metaclass mentioned earlier. To write an access log, a metaclass can be created as a subclass of the SOMMBeforeAfter metaclass with the before method overridden to write the log. If the Copy class uses this metaclass then the before method would be run before each method of the Copy class was run. To ensure that the access log has only one entry for each remote copy accessed, the after method would change the class of the copy object so it no longer used the metaclass descended from SOMMBeforeAfter and did not run any before or after methods. This can be done by:

```java
newCopy -> somRenewNoInitNoZero(this);
```

### 7.5 Strong updates

Several types of applications have semantics where the updates are valid regardless of the state of OSet. Additions to a patient's medical history can be made regardless of its current state. This is an instance of a class of application that keeps a history for each transaction by appending records to the end. For example, if you order some furniture and the company is not able to deliver because no one is at home, a record is added to the history of that order. At the same time, a different machine may add a record that shows that an item in your order has been backordered or replaced by a comparable model because of availability. These two updates may have been created by reading several other objects but they can be applied regardless of the state of other objects.

Other applications take decisions where somewhat stale input data does not matter. For example, you may download your retirement information onto your workstation and spend a couple of days investigating alternative scenarios. You may then generate orders to change your investment direction. When you convey these orders back to the enterprise database your investment information may have been updated but the orders you generated are valid nevertheless.

### 7.6 Reconciliation

We use strong update identification, reinstalling updates after dependency analysis and rerunning applications to reconcile independent updates. A system that contains copies of objects that it expects to update and later reconcile writes a log of every application that is run. For each application, it writes an access log containing the reference of every copy accessed.

The reconciliation program first checks the application log. The applications that create strong updates can be installed in any order. It removes them from the log and installs them last.

For each application remaining in the application log the reconciliation program checks every reference on the access log and compares the whenCopied attribute on the copy with the lastUpdate attribute on the original object. If no original object was updated since its copy was made, then the application is free of dependency conflicts and the updates it created can be installed on the original objects. This is done by
going through the access list again to select the objects that have been updated by looking at their update set. For objects that have been updated the updated image is copied onto the original object.

If one or more of the original objects have been modified in the main store since the copy was made, then the updates cannot be installed and application must be rerun either from the keystrokes or by rerunning the update methods. The following pseudocode shows the algorithm:

```
forEach ApplicationExecution
    in WorkDone.applicationLog {
        foreach Copy in
            ApplicationExecution.application.accessLog {
                if (Copy.whenCopied < Copy.objRef.lastUpdate)
                    goto rerun;
            }
            foreach Copy in
                ApplicationExecution.application.accessLog {
                    for each Update in Copy.updates {
                        //Find update created by this application
                        if Update.application = ApplicationExecution.application {
                            Copy.updateObject(Update); break;}
                    } //Update the original object
                rerun Application.ReRun();
        }
    WorkDone.ClearLog(); //Reset
```

If two salesmen come back with updated objects on their workstations then applying the updates from salesman 1 will advance the timestamps on some objects. This may invalidate some of salesman 2's updates, causing him to rerun some applications. A different sequence of events may result if salesman 2's updates are applied before salesman 1's. There is room for optimization here. Other types of optimizations are also possible. For example, we assumed that all data written was also read. This may not be the case and adding additional information to avoid this assumption may reduce the conflicts significantly.

8 Conclusion
This paper has presented a unified model for working with distributed objects. An application can access a remote object either by proxy or by copy or as a replica. We have shown that these styles are reasonably straightforward to implement and simple to use.

An application that wants to change the style of access can merely change the attributes of the context object. This can be done outside the main logic of the application. No changes to the class definitions or method code are required.

This paper has also developed an algorithm for reconciling updates made independently on disconnected copies of object sets. The algorithm is simple and practical, based on strong updates, applying updates that are free of dependency conflicts and rerunning applications.

Bibliography

6. IBM Corporation, S0M objects Developer Toolkit Publications, Order No. S96F-8649-00, Jan. 1994