Full Technical Paper submission to Middleware '98

Mobile Java Objects

Richard Hayton, Mike Bursell, Douglas Donaldson, Andrew Herbert.
APM Ltd.

Abstract

In this paper we discuss the engineering requirements for adding object mobility to the Java programming language, and give an overview of the design and implementation of our mobile object system. We note that it is helpful to *cluster* objects for mobility, and that if these clusters represent untrusted pieces of code (for example Agents) then they must be *encapsulated* both to control their access and to control access to them. We note that managing large numbers of mobile objects in an open environment is a difficult problem, but has its roots in the management of large distributed name spaces. We propose an architecture for re-locating moved objects that is both scaleable and *tuneable*.

The mobile object system we describe has been implemented, and is currently in use as part of an ESPRIT agent project. We are currently evolving the design and implementation to provide additional security and distribution facilities.

Contact Information

Richard Hayton
APM Ltd
Poseidon House
Castle Park
Cambridge
CB3 0RD
United Kingdom

Tel: +44 1223 568921
Fax: +44 1223 359779
Email: Richard.Hayton@ansa.co.uk

Mobile Java Objects

Richard Hayton, Mike Bursell, Douglas Donaldson, Andrew Herbert.
APM Ltd.

Abstract

In this paper we discuss the engineering requirements for adding object mobility to the Java programming language, and give an overview of the design and implementation of our mobile object system. We note that it is helpful to *cluster* objects for mobility, and that if these clusters represent untrusted pieces of code (for example Agents) then they must be *encapsulated* both to control their access and to control access to them. We note that managing large numbers of mobile objects in an open environment is a difficult problem, but has its roots in the management of large distributed name spaces. We propose an architecture for re-locating moved objects that is both scaleable and *tuneable*.

The mobile object system we describe has been implemented, and is currently in use as part of an ESPRIT agent project. We are currently evolving the design and implementation to provide additional security and distribution facilities.

1. Introduction

Java is an ideal language for developing distributed applications. It provides both object and interface abstractions, which gives a useful distinction between an object's interface and implementation. It also provides language level introspection, and allows for dynamic code creation. These features make it particularly suited to the design of middleware systems. Current Java middleware offerings have largely ignored these language level features. Sun's RMI[1] supports remote invocation, but allows only one interface per object, and insists that objects are constructed from base-classes to indicate if they are passed by reference or value. This restricts distributed programs to being written using a specialised sub-set of Java, which is both unnecessary, and an added complication when distributing existing code. RMI does not make use of Java's powerful type introspection, but instead relies heavily on native methods. This increases the burden on a JVM implementer and gives a system which is hard to evolve.

The other Java middleware offerings are CORBA compatible ORBs[2,3,4]. These use CORBA notions of object and interface, rather than the native Java ones. From the point of view of Java programmers, CORBA IDL offers a very restricted subset of the facilities offered by native Java interfaces. Again, these products rely on specialist stub-compilers or other tools, rather than leveraging type-introspection. This increases the complexity of distributed programming, and the amount of additional knowledge that a programmer must have.

In the area of code mobility, most offerings have concentrated on either *agents*, which tend to discard Java's strong interface typing in favour of simple messages, or *applets*, which provide class, but not object, mobility. One exception to this is Voyager[5], which provides simple object mobility by post-processing Java classes.

With the design of our middleware platform, FlexiNet, we took the approach of extending Java language concepts by adding *selectively transparent* remote invocation to calls on any interface. We used Java's strong typing support and runtime introspection to allow us to build a strongly typed reflexive binding framework. We further used Java's support for runtime code loading to allow us to create stubs transparently *on the fly* during program execution. This gives a middleware system which is both extremely flexible and a natural extension of Java's features[6].

When we turned our attention to mobility, we took the view that this too should be a natural extension of the Java language. We required a system in which we maintained the FlexiNet view of a "sea of objects" but which allowed objects to move between hosts. This movement should be transparent to clients of the objects, who simply continue to use Java references to exported interfaces.

This approach has several advantages. As well as providing a straightforward, and familiar, programming paradigm, we remain within the well-understood domain of distributed systems. This allows us to leverage existing research when tackling issues of scalability, robustness and security.

In the following section we give a brief overview of the FlexiNet middleware platform that was used to create our Mobile Object Workbench. In Section 3 we outline the requirements of mobile objects. Section 4 briefly considers mobile agent requirements, as they are an obvious application of mobile objects. Section 5 outlines the Mobile Object Workbench itself and expands on the implementation choices. Section 6 discusses the issues related to rebinding to a moved object. Section 7 gives the current implementation status, and Section 8 concludes.

2. FlexiNet

The FlexiNet Platform is a Java middleware system built as part of a larger project to address some of the issues of configurable middleware and application deployment. Its key feature is a component based 'white-box' approach with strong emphasis placed on reflection and introspection. This allows programmers to tailor the platform for a particular application domain or deployment scenario.

The FlexiNet engineering model has three central concepts. Interfaces are represented by **proxies**. Proxies enforce the typing of the remote interface, and perform remote access by utilising **binders**. Each binder is an object capable of creating a *generic* binding to a local or remote object, and different binders embody different application requirements or engineering strategies. Binders may make use of other binders in a recursive way. This keeps individual binders small, and allows application domain-specific binders to be easily created. To manage the flexibility, FlexiNet supports the notion of **multiple name spaces** for interfaces, and names are both strongly typed and structured. Names may be constructed out of other names, or arbitrary data, making the management of aggregate and indirected names straightforward.

2.1. Multiple Binders

The FlexiNet model of multiple *recursive* binders, allows different reflective abstractions to be embodied. For example, we may have binders that create bindings that enforce transparent persistence, or that provide transactional access to objects,

or that bind to remote objects using standard protocols such as IIOP. Binders performing all of these functions are currently being developed as part of continuing FlexiNet work. As binders may call other binders recursively, we may also create binders to perform additional functions that are orthogonal to the actual communication, by recursively calling other binders. For example we have a recursive binder that chooses between a number of other binders, and one that performs access checks prior to binding.

2.2. Generic Communications

This is a reflexive technique central to the design of FlexiNet. Rather than using stubs to convert an invocation directly into a byte array representation, instead we leverage Java's runtime typing support to represent the invocation in a generic (but fully typed) form. The layers of the FlexiNet communication stack may then be viewed as reflexive meta-objects that manipulate the invocation before it is ultimately invoked on the destination object using Java core reflection.

This approach allows middleware (or application) components to examine and modify the parameters to the invocation using the full Java language typing support. Figure 1 illustrates how a communication stack can be considered as a number of meta-objects that perform reflective transformations on an invocation. Third part meta-objects can be fully general and are fully type-safe. For example a replication meta-object might extract replica names from an interface and then perform invocations on each replica in turn. As this processing is performed in terms of generic invocations, the is no need for each of these calls to pass through stubs and so the code can be both straightforward and efficient.

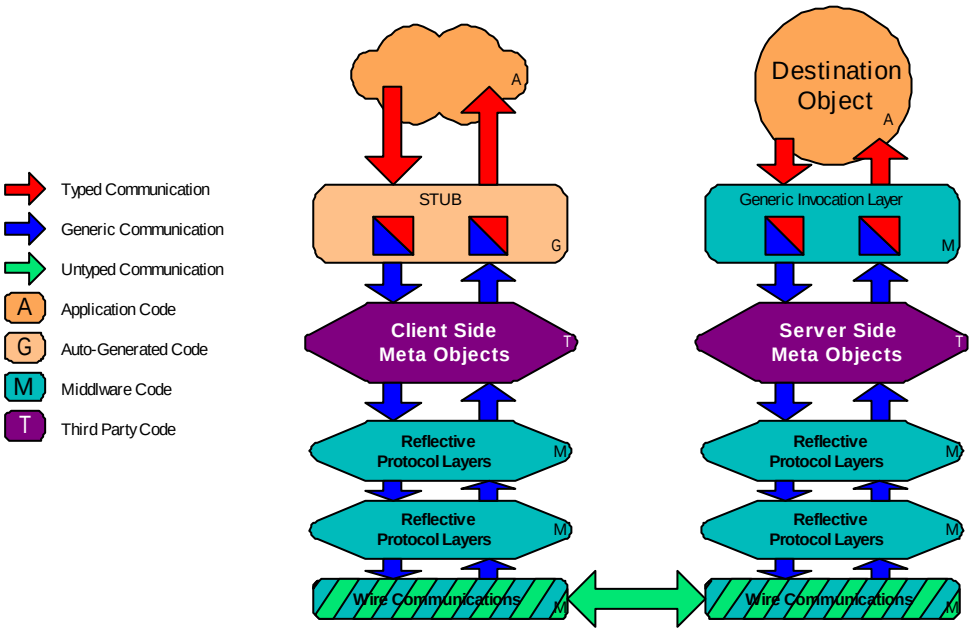

Figure 1 A Reflective FlexiNet Communication Stack

When designing the Mobile Object Workbench, we decided to implement it as a set of FlexiNet binding protocols and services. In particular the mechanism for communication with a mobile object is essentially communication with a static object together with mechanisms for rediscovering the object's location after it has moved.

3. Requirements for Mobile Objects

In this section we outline the requirements for mobile objects over and above the requirements of distributed objects.

3.1. Unbinding

The key feature of a mobile object is that it must be able to move. When we move an object, we effectively copy it to a new location, and then arrange that all references to the old object are replaced with references to the new object. In distributed system terms, this requires a mechanisms for *unbinding* a previous binding to an object. If objects are referenced directly using language level pointers, then this would not be possible without changing the implementation of the Java virtual machine, which would reduce the advantages of Java as a portable language. Instead we arrange that mobile objects (or more correctly interfaces on mobile objects) are referred to indirectly, using FlexiNet stubs. This level of indirectly allows us to re-plumb references dynamically when objects move. In addition, to avoid the need to track distributed references, we only perform this re-plumbing when a reference to a moved object is first de-referenced. The use of stubs and the re-plumbing is transparent to the application programmer, although they may reflect this process, for example to deal with errors, or if there is a requirement only to communicate with an object when it is in a certain location.

3.2. Consistency and Threads

At any point in time, an object may be *active* or *passive.* An active object is one that has a thread of control currently executing in it, or passing through it. A passive object is one that is not currently being executed, and hence has no threads active in it. When we move an object we must ensure that the move is *atomic.* To do this all processing of the object is halted until the move is complete. One approach to this would be to pause any threads running in an object, move the object, and then restart the object and the thread at the new location. Whilst this would seem an ideal solution, it is impractical, as Java does not allow us to determining the complete state of an active thread at an arbitrary point of execution. A more practical approach (and the one chosen) is to *encapsulate* the object and enforce a locking strategy which ensures that no threads are executing the object at the point of movement. This does not prevent the existence of active mobile objects, or those that contain completely internal threads, but it does require a degree of co-operation with such objects, so that they can be 'shut down' prior to movement, and then 'restarted' at the new location.

The encapsulation mechanism is integrated with the mechanism for transparent re-binding, so that external threads that have blocked pending an object's movement restart and relocate the newly moved object.

3.3. Grouping

In the discussion so far, we have been describing the migration of single Java objects. However there is little utility in moving a single Java object. A more useful unit for mobility is a set of related objects, and we need a mechanism for deciding which parts of a program should move together.

We introduce the notion of a *cluster* as both a grouping and encapsulating construct to address this issue. A cluster is an encapsulated set of objects in the sense that references that pass across a cluster boundary are treated differently from those entirely internal or external to it. In particular, when resolving an external reference, the system may have to locate a cluster on a remote machine (possibly after it has moved). References entirely within a cluster can be ordinary Java references, as no special action needs to be taken when they are de-referenced.

To a programmer, clusters are a surprisingly straightforward concept. A special mechanism is used to created the initial object populating a cluster, and after this any new object is created in the same cluster as its creator. For the most part clusters are completely transparent to the programmer.

3.4. Failure Modes

When designing distributed systems, there is always the possibility of host or network failure. In particular network partition can result in hosts incorrectly assuming that other hosts have failed. When designing a mobile object system, a key decision is the semantics in the worst case scenario of a network partition during object migration. There are three possibilities. We could allow the possibility of the object existing on both sides of the partition - this was rejected as it introduces an unwanted degree of complexity. The second possibility is to ensure that an object is destroyed if it cannot be uniquely determined which side of a partition it exists in. This is the default semantics chosen in the Mobile Object Workbench. The third possibility is to suspend use of the object until the network is restored. This is being considered in the context of the integration of transactional mechanisms into FlexiNet. Section 5.4 gives a fuller description of the state transitions required to ensure consistent movement.

3.5. Scalability

There are two issues relating to the scalablity of a mobile object system. Firstly, some design choices would require the registration either of all objects, or worse, of all references to all objects. We rejected these approaches as we wish to use the mobile object workbench in an Internet environment, which is both open and has no central administration. The second issue relates to the rebinding to interfaces on an object once it has moved. We would like this to be possible, even if the original host has since failed. This issue is discussed in detail in section 6.

4. Requirements for Mobile Agents

The mobile object workbench is not a mobile agent system, however it was developed as part of an ESPRIT agent project called FollowMe[7], and one of the other partners is developing an agent system on top of it. As mobile agents are an obvious application of mobile objects, it is worthwhile to consider their specific requirements.

4.1. Autonomy

Mobile agents are generally considered to be 'autonomous'. That is to say that it is the agent itself that determines the actions it takes, and in particular controls its movement. In terms of mobile objects and clusters, this gives a requirement for cluster mobility to be initiated only from the cluster itself. The encapsulation mechanism gives provision for this; only threads within a cluster have access to the

objects within it, and by giving one of these objects a handle to the infrastructure which controls mobility, this is effectively hidden from the outside world. There are two exceptions to this. Firstly, a malicious implementation of the infrastructure can overcome the FlexiNet encapsulation mechanisms; this is a necessary evil of distributed computing - you have to trust the host. The second exception is that a host may 'legally' destroy an object and reclaim the resources it is using. This is necessary to allow hosts to be manage their own resources. The 'normal' procedure is for a host to inform a cluster that destruction is imminent, in order to allow it to move or shut down cleanly, but a host must always be able to perform a 'dirty shutdown' in order to protect it from malicious or erroneous agents.

4.2. Security

"Security" is a catch-all term used to describe a variety of issues, not all always well differentiated. Some autonomous mobile agent systems can be seen as lacking in their approach to some of these issues, and we have found that the approach of designing and engineering from the point of view of a mobile object system, allowed us build on established security principles.

We identify six basic areas of security concern within the Mobile Object Workbench:

1. host integrity - protecting the integrity of a hosting machine and data it contains from possible malicious acts by visiting objects.

2. cluster integrity - it should be possible to determine if a cluster has been tampered with, either in transit or by a host at which it was previously located. We may wish to allow hosts to modify parts of a cluster (e.g. data) but not others (e.g. code).

3. cluster confidentiality - a cluster may wish to carry with it information that should not be readable by other clusters, or by (some) of the hosts which it visits.

4. cluster authority - a cluster should be able to carry authority with it, for example a user's privileges, or credit card details. To provide this we need both cluster integrity and cluster confidentiality.

5. access control - hosts should be able to impose different access privileges on different clusters that move to it. Clusters and hosts should also be able to enforce access control on exported methods.

6. secure communications - clusters and hosts should be able to communicate using confidential and/or authenticated communication. Some applications may also require other security features, such as non-repudiation.

We believe that unless all of these aspects of security are addressed, any mobile object system will not prove secure enough for real world applications, and we have therefore adopted the principle of including security issues from the outset, rather than as an "add-on", bolted on at a later date. Section 5.6 discusses our approach to these issues.

4.3. Thread Encapsulation

As cluster representing agents represent potentially distrusting pieces of code, it is important that one cluster cannot adversely affect another. In particular one cluster must not be able to invoke a method on a second cluster, and then destroy the thread performing the call, so as to leave the second cluster in an inconsistent state. Equally, if a cluster crashes or intentionally blocks whilst servicing a request, the client must be able to recover, and must not also fail or block indefinitely. In order to achieve this degree of strong encapsulation, we de-couple all threads that enter or leave a cluster, so that the failure of the caller and callee are independent. Again, this thread de-coupling is integrated with the binding system and is transparent to the application programmer.

5. The Mobile Object Workbench

The Mobile Object Workbench is being built within the context of the FollowMe European ESPRIT project (no. 25,338), which commenced in October 1997[7].The Mobile Object Workbench is being constructed as an extension of a Java middleware platform called FlexiNet, which was developed at APM during the last eleven months[8].

5.1. Concepts

Figure 2 shows the relationship between (Java) objects, Clusters and Mobile Objects. A Cluster is a Java object containing a grouping of objects which are managed together. A Mobile Object is a specialisation of this which is able to move between Places. Places are themselves objects which abstract execution environments, typically with one Place per JVM. Protection, movement, destruction, charging and other management functions are considered in terms of the lifecycle of Clusters and the interaction between them. It is sometimes useful to consider a Cluster and its contents as a virtual process, and the encapsulation and security concerns around Clusters encourage this abstraction.

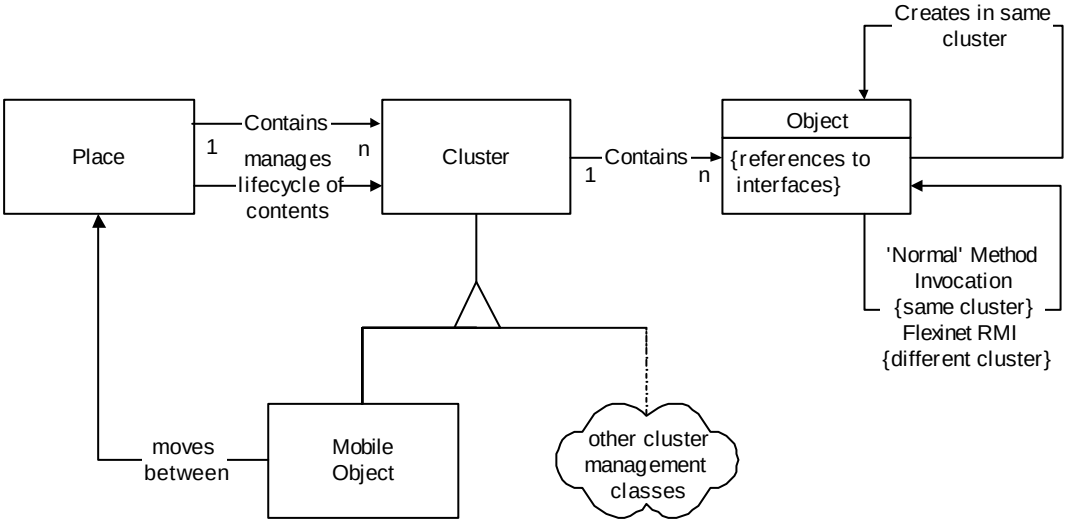

Figure 2 Objects, Clusters and Mobile Objects

An object is the basic building block out of which applications may be built. Objects may contain references to interfaces on other objects anywhere in the system. Objects

may directly create other objects, but only within the same Cluster. They may be able to arrange the creation of objects in other Clusters via communication with a place. Within a Cluster, access to methods/data on objects is determined by standard Java language protection means and takes place using standard Java method invocation. Between Clusters, encapsulation is enforced so that object in one Cluster may only access methods on objects in other Clusters if these methods form part of the interface passed between the clusters.

5.2. API

For completeness, the Mobile Object Workbench API is included. The key point of note is that the API is entirely concerned with the lifecycle of clusters - communication takes place using application-level exported interfaces, and transparent remote invocation.

5.2.1. Class UK.co.ansa.flexinet.mobility.Cluster

`public synchronized void lock()`
Increase the number of locks held on the object. Whilst a lock is held, new calls made on objects in this cluster from other clusters will block. Calls which have already passed a certain point will continue to be executed.

`public synchronized void unlock() throws UnMatchedLockException`
Decrease the number of locks held. If the number of locks held is zero, wake any blocking calls.

`public void init()`
Called upon object instantiation. A subclass that requires initialisation arguments, or wishes to return an interface to its creator, should provide an alternative `init(...)` method. The `init` method may take any arguments, and the appropriate `init` method will be chosen by matching the creator's arguments. The `init` method may return an interface on any object in this cluster

`public void stop()`
A call made by the place if it wishes the cluster to cease processing. The cluster is expected to clean up and then return. When the call returns, the place will invoke destroy().

`public final void destroy()`
This call destroys the cluster as effectively as possible.

`public void restart(Exception e)`
Called after the cluster is restarted. A subclass which wishes to take action after a restart should override this method.

5.2.2. Public Class UK.co.ansa.flexinet.mobility.MobileObject extends Cluster

Mobile objects are clusters that have the ability to move between places

`public synchronized void pendMove(Place dest) throws MoveFailedException`
Request a move to the identified place. A new thread will be spawned to perform the move. The move will not take place until there are no other threads within the cluster.

`public void syncMove(Place dest) throws MoveFailedException`
Request a move to the identified place. The current thread will attempt to perform the move. If successful it will exit. The move will not take place until there are no other threads within the cluster.

5.2.3. public interface UK.co.ansa.flexinet.mobility.Place

The interface representing a place at which a cluster resides, and between which mobile objects move.

`public Tagged newCluster(Class cls) throws InstantiationException`
Create a new cluster at this place. Once created, `init(arg0,arg1,..)` will be called on the new object. The `init` method may return an interface, which is passed to the creator.

5.3. Binding architecture

Communication between clusters takes place by remote method invocation using a special binding protocol. This is a 'standard' FlexiNet binder, together with two special reflexive layers. On the client side of an invocation is a "cluster location" layer. This examines the internal name used to represent the interface being accessed, and determines the host on which it resides. The procedure adopted is to try the last known location, and only contact the relocation service upon failure.

On the server side of the communication, there is a reflexive "encapsulation layer". This processes incoming calls, checks that they refer to clusters that are (still) located on the host, and performs the synchronisation required to ensure that the cluster is not in the process of moving. Part of the encapsulation process is to de-couple the calling thread, so that client and sever clusters cannot affect each other by killing or otherwise manipulating the thread. This is illustrated in Figure 3.

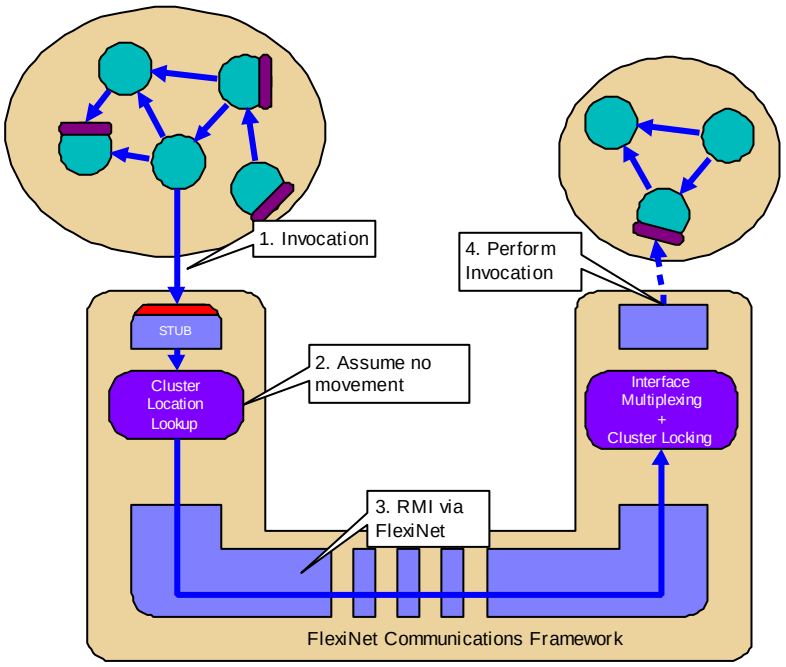

Figure 3 Implementation of inter-cluster calls

5.4. Orchestrating Mobility

Figure 4 illustrates the states that an instance of a mobile object may be in. The object is initially created in state A1. This state represents an *active* object that has *one* thread in it (the thread that calls the constructor). When active, the object may create other threads, and methods on its interface may be invoked by objects in other clusters. It will therefore move between active states.

In order to move, a mobile object invokes a `pendMove` or `syncMove` call. Both of these request a move 'as soon as possible', the different being whether the calling thread returns immediately (`pendMove`) or blocks and never returns (`syncMove`). When a move call is invoked, the object enters a pending state. These are identical to active states except that the object will be moved as soon as all executing threads exit (i.e. when it enters state P0). As a side effect of executing a `pendMove` or `syncMove`, the cluster becomes locked. When locked, calls from other clusters block until the cluster is unlocked. A cluster may lock itself any number of times, and an equal number of unlocks are required before it may be accessed by other clusters. Locking a cluster does not prevent it from calling other clusters. When in a pending state, a cluster is not able to remove the final lock.

When a mobile object enters the state P0 it will undergo a series of transitions that may result in the creation of a new mobile object at a different place. The original mobile object will then be discarded (it enters state X). If an error occurs during this process and it can be safely inferred that the new object has not been created, then this object is returned to state A1. If the move was initialised by a call to syncMove, then the error status is returned as an exception. If the move was initiated by a call to `pendMove`, the object is restarted by calling the `restart` method, and the exception is passed as a parameter.

The newly moved object is an exact replica of the original, and in addition all references to interfaces exported by the original cluster are re-mapped to the new (effectively the original object has moved). It is started in state A1 by a call to `restart`. The new object (or original after failure) will have the same lock status as the original - apart from the lock automatically taken when `pendMove` or `syncMove` was called, which is released. If an object wishes to restart in a locked state, then it may obtain an additional lock prior to calling `pendMove` or `syncMove`. This allows newly moved objects to perform start-up cleanly, before allowing external access.

Copying, rather than moving, an object follows exactly the same procedure as `syncMove`. The `copy` operation blocks until there are no other threads and the new object has been created, or a failure is detected. After successful synchronisation, or failure, the original object enters state A1 and the copy operation terminates. The newly created copy of the object commences operation with a call to `restart` in state A1.

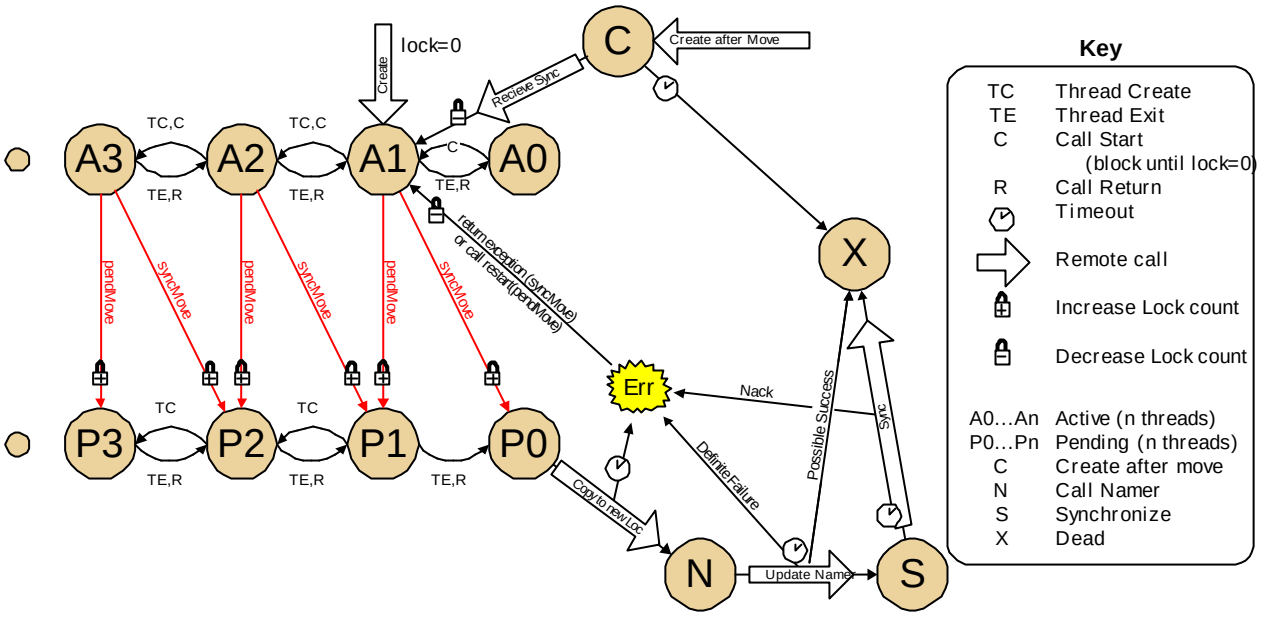

Figure 4 State transitions of a mobile object instance

5.5. *Method Invocation*

When an object in one cluster attempts to invoke a method on an object in another cluster, this must block if the callee cluster is in the process of moving. Equally it must not be possible for a caller to prevent a callee from moving, by bombarding it with requests. The following state diagram indicates the process through which a callee must go in order to meet the requirements, and in order to locate the current instance of a mobile object. This process is undergone automatically in the Mobile Object Workbench infrastructure. It should be noted that the callee is able to interrupt a thread making a call, but that this will not affect the caller. This is important to prevent the caller from blocking the callee's progress.

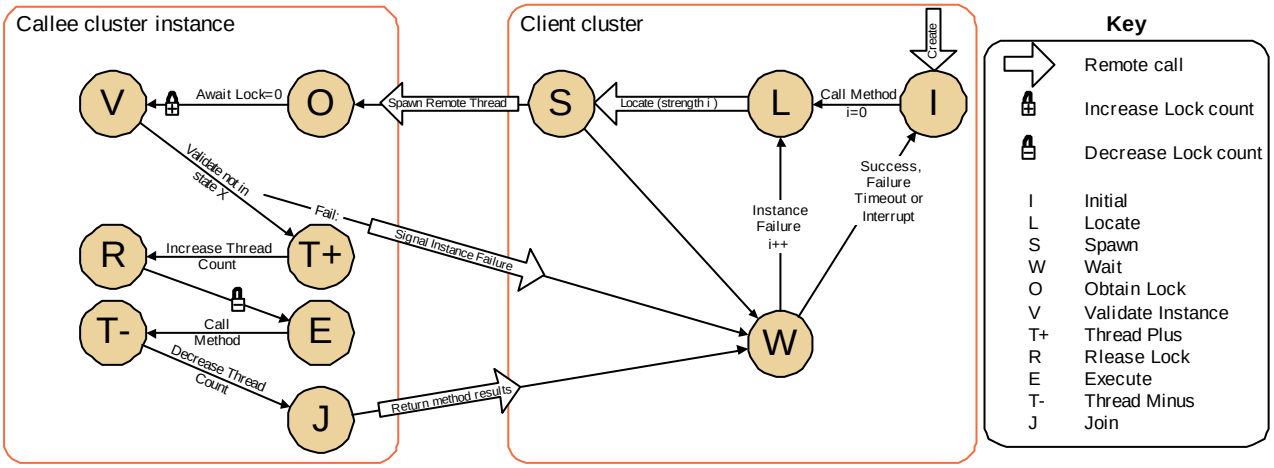

Figure 5 State transitions for an invocation on a remote cluster

5.6. Security

Within a distributed, mobile object context, issues of trust take on a different slant to non-mobile systems. In both mobile and non-mobile systems, questions of how much trust is placed in an object must be based on the provenance of that object - where it has come from, and its history. In a non-mobile object oriented system, such as the base Java implementation, objects are typically instantiated from class files, having no other state. In Java, these classes may be signed, and a JVM may assign policies to their instantiations based on this signing. In a mobile object context, this mechanism is not flexible enough, as the history of the object has not started at this JVM, and the initial signing of the class files does not reflect the full provenance of the object.

For this reason, we have designed and implemented a Security Manager[9] which extends Java's model by allowing policies to be assigned to instances of objects, rather than just their class. This has been possible because of the strong thread encapsulation we have employed within the Mobile Object Workbench, which gives each cluster its own thread group. As Java allows checking of the thread performing a particular operation, we may determine the cluster from which an invocation originated, and hence enforce the appropriate policies.

This security policy allows hosts to restrict operations allowed by particular clusters, thereby protecting their own integrity. It also provides a good base from which to extend cluster-to-cluster access restrictions.

Cluster integrity and confidentially are enforced by encrypting and/or signing certain objects within a cluster. This prevents a host without sufficient access privileges from examining a cluster's state, and allows one host to detect if a cluster has been modified by a host that it visited earlier. In addition to this, we must ensure that cluster are not dissected - or a malicious host could 'steal' parts of the cluster that represented encrypted passwords and use them to build its own clusters. To do this, we require a mechanism for specifying, and validating, integrity statements. For example we may annotate a cluster's definition to indicate that a particular field may only be modified by certain hosts. We may then use digital signature techniques to ensure that whenever the field is modified it obtains a signature from the current host, and when other hosts attempt to read this field we can throw an exception if the signature is incorrect.

We are currently developing a system to allow integrity policy statements to be specified. Once specified, the use of secure fields or objects can be made 'almost' transparent to the programmer. All that is required is that they use accessor functions to access the protected fields.

Cluster authority can be implemented using cluster integrity and confidentiality. Together these allow a cluster to carry with it a password or other secret information, without the concern that this secret can be read at any host which is visited. Clearly, once the secret *is* revealed to a host, there is nothing that can be done to prevent the host from misusing it. For this reason we have a model that the mobile object moves into a secure environment before revealing a secret. Figure 6 gives an example; a cluster may move between several hosts before eventually arriving at a 'Bank' host. At this host, it may reveal a password to allow it access to a bank account. However, as the Bank host already knew the password, revealing it has not given the bank any additional privileges, and the security of the password has not been weakened.

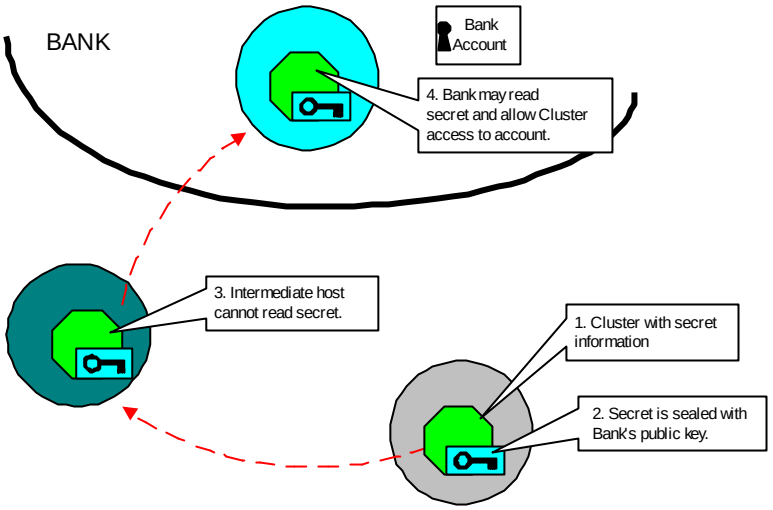

Figure 6 Clusters with Secrets

Access control and secure communications may be implemented using standard techniques. We use FlexiNet's reflective binding system to allow a cluster to receive notification of an invocation immediately prior to it's execution, so that it may implement its own security policy, and throw an access control exception if appropriate. Secure communication between places may takes place using a FlexiNet binder that support SSL[10]. Secure communication between mobile clusters may also take place using SSL, but requires that clusters reveal the information use to prove their identity to the host from which they are communicating. This is reasonable in some circumstances, but should be used with caution.

6. Relocating Moved Objects

There are several approaches to managing migration, the most common of which is called the 'Tombstone' approach. In this approach, when a cluster moves, it leaves behind a forwarding address, so that future calls can be redirected. When a cluster is located by a particular client, that client (optionally) remembers the cluster's latest location, to speed future lookups. This simple scheme is used by the majority of existing mobile object (and agent) systems. Although it has some deficiencies, it can be used as a standard to measure other approaches against. We note a number of important parameters when assessing a scheme for managing mobile names.

- **Move cost**. The additional overhead that must be performed whenever a cluster moves. In the tombstone approach, this is low as no additional hosts need to be contacted.

- **Call cost**. The additional overhead per call. In the tombstone approach, this may be small (if the cluster has not moved), but is unbounded. If the cluster has moved many times, there will be a cost associated with each forwarding call.

- **Dependence on hosts**. The number and type of hosts that must remain active and reachable in order for a call to succeed. The tombstone approach scores badly here. Each of the hosts at which the cluster has previously resided have to remain active and connectable. These hosts must *never fail* if it is to be guaranteed that the cluster can be contacted by all clients who have references.

- **Background load**. The amount of processing that must be done by a host in order to keep references 'live'. The tombstone approach has no background load, but a frequent extension of it is to refresh all references periodically in order to keep the hop count low, and in order to reduce the reliance on hosts from which a cluster has moved.

- **Garbage accumulation**. The amount of information that a host must keep about clusters that do not reside on the host, and are not referenced by objects at the host. In the tombstone approach, each host must remember forwarding information indefinitely. In extensions of this scheme, this may be traded off against increased background processing.

- **Security implications**. The effect the scheme has on ordinary access control or authentication. Security requirements tend to limit the use of schemes such as Tombstoning to messages used to locate a cluster. Once the cluster is located, a normal call is made directly from client to server. *I.e. normal messages are not forwarded, only 'locate' requests.*

- **Integrity Requirements**. The effect that a malicious or erroneous host can have on the smooth running of the system. This effect may be to prevent execution or to increase any of the costs or dependencies listed above. In addition any adverse affect may be limited to objects created at, or once located at, a malicious host, or it may not. In the Tombstone approach, a malicious host cannot affect the location of objects other than those which were once located at it. However, in variants on Tombstoning, that rely on honest accounting of remote references to perform background Tombstone pruning, a malicious host can play havoc.

The Mobile Object Workbench uses a relocation service to locate moved objects. This service is addressed via a FlexiNet interface reference, and a particular implementation may have one global relocation service, or as many as one per JVM. The mechanism used for relocation is hidden behind this interface and each FlexiNet interface reference for a mobile object contains a reference to the corresponding relocation service. This gives considerable flexibility, and allows a different scheme to be used for different deployment scenarios. We have developed a scaleable federated relocation service for use in an Internet environment. This is described in the following section.

6.1. Name Relocation Service

When a call is made on a mobile cluster, the encapsulation layer at the called host will determine whether the cluster (still) exists at this host. If it does not, then the host will raise an exception which is passed back to the callee and caught in the callee's locate layer. This then contacts the name relocation service to determine the new location of the object. The relocation service is a federation of a number of directories. Each directory contains a mapping from old to new cluster addresses. Our naming service was developed with four key properties:

i) we control what entities are able to update the directories - only hosts from which a cluster is moving may update the record for the cluster. This is possible as Cluster names (transparently to the applications programmer) contain information

about their current network host. This prevents fraudulent changing of naming records by "spoof" hosts or clusters.

ii) we provide a hierarchy of directories, for scale and robustness. This means that an instance of the relocation service may decide to copy the naming record for a cluster up the hierarchy to increase its stability, or to reduce the load placed upon it.

iii) redirection: we allow naming records to be moved between directories so that an optimal directory location can be chosen for the record (e.g. following the movement of a cluster around the network).

iv) we allow caching for performance. A naming record can be kept at a previous host, as well as being passed up the hierarchy, to reduce look-up time.

One possible scenario for using the naming service is shown in *Figure 7*, *Figure 8* and *Figure 9*. In *Figure 7*, a naming hierarchy is shown, with hosts (large, light-coloured boxes), naming services (smaller, dark boxes), a 'live' naming record (a heart-shape) and a Cluster (a circular object). In this figure, the Cluster moves from its original host to another.

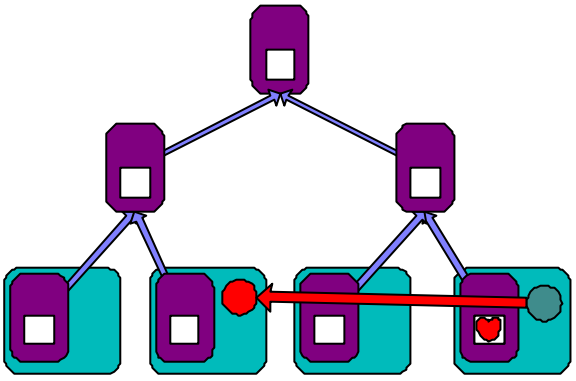

Figure 7 - Relocation service: Mobile Object moves

In *Figure 8*, the naming record for the Cluster is moved to another relocation service. In this case, the naming service is a 'close' to the host the Cluster has moved to. The move might have been initiated because the cluster is expected to move between a number of hosts close to the naming service, rather than staying at the new host. A link is provided at the previous naming service, to allow the cluster to continue to be found by other objects with references to its original directory. It should be noted that the link is not, however, to the cluster itself, but to the directory. Generally the naming record will move much less often than the cluster, so although we use tombstoning, it is only between naming records that move both infrequently, and usually to robust hosts, rather than between clusters, that may move often, and to unreliable hosts.

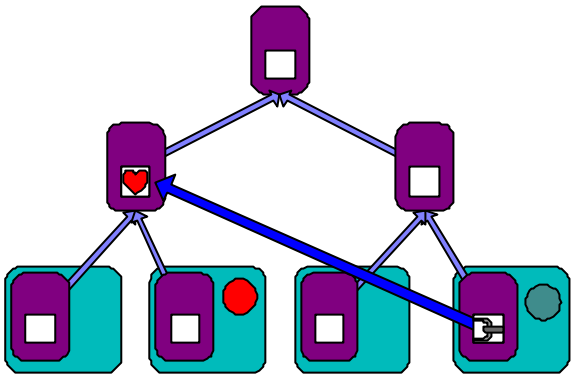

Figure 8 - Relocation service updated

In *Figure 9*, the link from the original host's naming service is copied from itself to its (well-known) parent. This means that in the event of failure or of garbage-collection by the original directory, the cluster can still be found by the infrastructure by searching back up the tree, but means that while the link still remains, it is cached and may provide a performance improvement on traversing the naming service hierarchy.

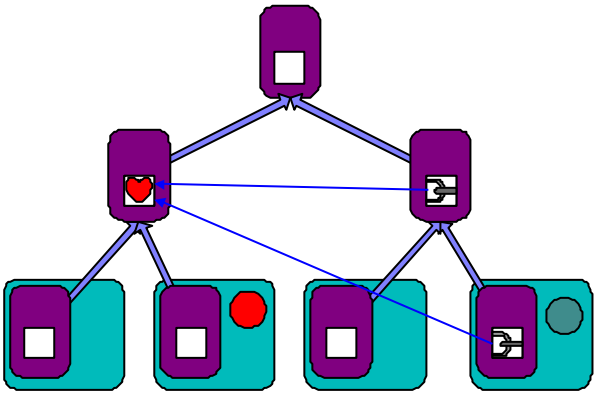

Figure 9 - Relocation service - caching and copying

7. Implementation Status

The Mobile Object Workbench version 1.0 is currently in use by the members of the ESPRIT FollowMe project. The current implementation supports clusters, mobility and transparent communications as described in this paper. Current work is enhancing the workbench in three ways.

1. We have started work on the design and construction of a federated network class loader, to allow support for mobile objects that have different views on the Java class name space. Currently a place cannot support two clusters that have different interpretations on the mapping between the class named 'A', and the code implementing this class. These enhancements are important in an Internet environment where there is no global consensus on class names - and to support evolution of classes.

2. We are currently implementing several of the security abstractions outlined in section 5.6. We have a Security Manager implementation which can enforce different security policies based on the identity of the cluster invoking a call.

3. The current implementation of the Relocation Service does not support the migration of directories. We intend to rectify this in the near future.

In addition to specific Mobile Object Workbench issues, work on FlexiNet is continuing. We have recently created secure bindings (using SSL) and both IIOP and transactional support are underway[11].

8. Summary

The Mobile Object Workbench was designed primarily to add *mobility transparency* to the distribution transparency provided by FlexiNet. In this paper we have shown that in order to do this we must add clustering and re-binding mechanisms. If mobile objects are to be used to support autonomous agent systems, then security requirements lead to the need for *encapsulation* mechanisms so that hosts and agents may communicate and co-operate without the need for complete trust. We have approach the design of the Mobile Object Workbench as a distributed system problem, as it has all the traditional issues related to distributed systems; scale, robustness, independent failure modes, distrust, decentralised administration, multiple name spaces and diverging code bases. This approach has lead us to design an architecture, and implementation, that can evolve to meet future needs, and we believe this gives it clear advantages over the ad-hoc approaches of existing mobile agent systems. In addition we have designed the system as a natural extension of the Java language. This makes it straightforward to use, and allows programmers of mobile objects to use the full language facilities.

---

1 "Java Remote Method Invocation (RMI)" Specification
http://www.javasoft.com/products/jdk/1.1/docs/guide/rmi/spec/rmiTOC.doc.html

2 “CORBA/IIOP 2.1 Specification” *Object Management Group*. Aug. 1997. http://www.omg.org/corba/corbiiop.htm

3 OrbixWeb - A Java ORB from IONA Technologies http://www.orbix.com/products/internet/orbixweb/

4 VisiBroker - A Java ORB from VisiGenics http://www.visigenic.com/

5 “ObjectSpace Voyager Core Technology”, *ObjectSpace.* http://www.objectspace.com/Voyager/

6 “FlexiNet - A flexible component oriented middleware system”, Richard Hayton, Andrew Herbert. SIGOPS ’98 (Submitted)

7 “FollowMe project overview”, *FAST e.V.* http://hyperwav.fast.de/generalprojectinformation

8 “FlexiNet - Automating application deployment and evolution”, *APM Ltd.* http://www.ansa.co.uk/Research/Flexinet.htm

9 Marlena Erdos, Bret Hartman, Marianne Mueller. “Security Reference Model for the Java Developer's Kit 1.0.2”, *Sun Microsystems*. Nov. 1996. http://java.sun.com/security/SRM.html

10 “The SSL Protocol”, *Netscape Inc.* http://home.netscape.com/newsr ef/std/SSL.html

11 “A Reflective Component-Based Transaction Architecture” Zhixue Wu, APM Ltd Middleware ’98 (Submitted)