ESPRIT Project No. 25 338

Mobile Object Workbench 1.1

Work package B

Mobile Object Workbench

Deliverables DB2, DB3, DB4, DB7.2 DB8.2

ID:	DB2,3,4,7.2,8.2	Date:	16.04.98
Author(s):	Richard Hayton	Status:	Internal Release
Reviewer(s):		Distribution:	Project confidential

Change History

Document Code	Change Description	Author	Date
MOW Design and Interface Specification DB3 , DB4	Draft version included in DA1.1.	Richard Hayton (APM)	27.Nov.97
	New version. Minor updates to semantics. Major rewrite of explanation.	Richard Hayton (APM)	19.Dec.97
DB 2,3,4,7.2,8.2	New version. Semantics remain unchanged. Additional topics covered, including events and naming.	Richard Hayton (APM)	16.Apr. 98

Introduction

This document contains the rational, design and API for the Mobile Object Workbench (MOW).

Current Status

This document reflects version 1.1 of the Mobile Object Workbench, as delivered in April 1998. It also covers the design of several aspects of the Mobile Object Workbench that have yet to be implemented, and some that a currently in production. In particular the details of the security API are liable to change.

API Status

There is an initial implementation to this API, and the API is expected to grow, rather than to change. It should be noted, however, that some methods on MOW classes do not form part of the API, and are liable to change. In particular, the internal methods for object migration are likely to be extended to include security information.

The Autonomous Agents work package is expected to subclass some of the MOW classes, in order to add agent management functions. This will require access to some non-API aspects of the MOW. For this reason, the JavaDoc version of the API includes details of many additional methods, some of which may need to change to reflect security, or other issues. We will attempt to minimise changes, and liaise with interested parties.

Structure

In the following three sections, we discus the background to the design of the Mobile Object Workbench, and discuss the requirements, both for mobile objects themselves, and for supporting autonomous agents built out of mobile objects. We then present the design of the MOW, and summarise the programmer API. The next four sections consider the design and implementation of specific areas of the MOW. The penultimate section provides notes for application programmers wishing to use the MOW. The document concludes with a summary of the features of the MOW. The full API is included as an appendix.

Background

The mobile object workbench was designed with the view that mobility should be a natural extension to distributed computing. Within APM we had already developed a Java middleware platform called FlexiNet[1], which provided a "sea of objects" abstraction. The approach taken to mobility was to extend FlexiNet to enable 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.

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.

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[2]. 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.

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.

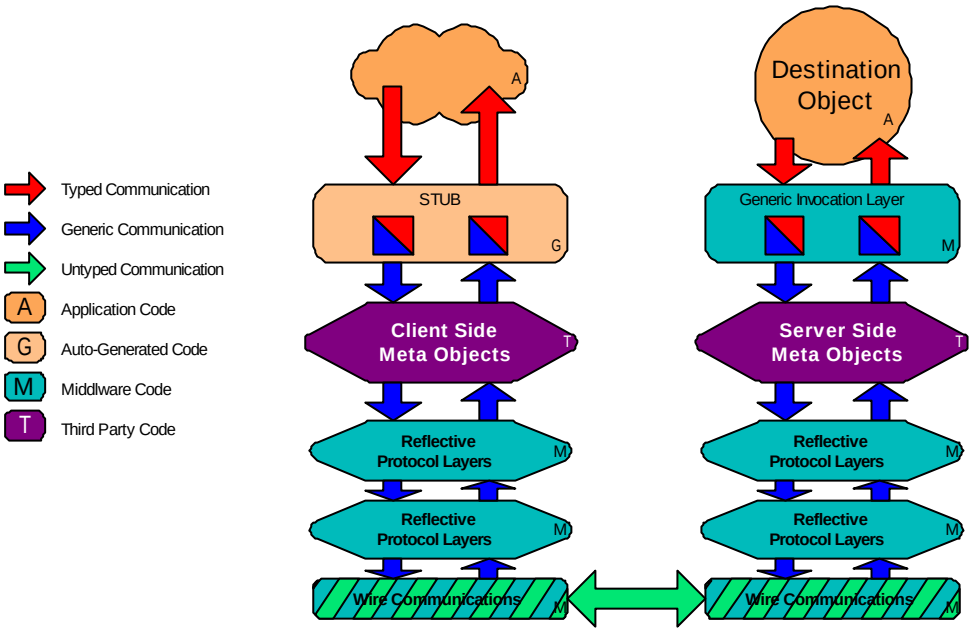

Figure 1 A Reflective FlexiNet Communication Stack

Naming Interface References

FlexiNet allows great flexibility, both in the design of a communications stack and the wire format of messages. In order to take advantage of this, the format use for interface references must be equally flexible. Traditionally, interface references are represented by ‘ bundles of bits’ , and extracting fields from these bundles must be done with care, and without the aid of language level typing. This makes it difficult, if not impossible, to extend a system by adding new types of names, as this leads to changes in the name format and has repercussions throughout the code.

In FlexiNet we took the approach that interface references are themselves objects. As objects they may be converted to and from a serial byte representation in the same way as other objects are converted. This removes the need for a specific wire format for names, and allows us to use the full generality of object typing to subclass and otherwise manipulate interface references. This is only possible because Java provides the run time type information required for serialization.

As an example the decomposition of the name representing an interface on a mobile object is shown in Figure 2. This is necessarily complex, but it considerably easier to manipulate that its ‘ bundle of bits’ representation. As the components for storing and manipulating interface references all use class `Name`, and `MobileName` is a subclass of this, mobile names may be treated like any other in the majority of code.

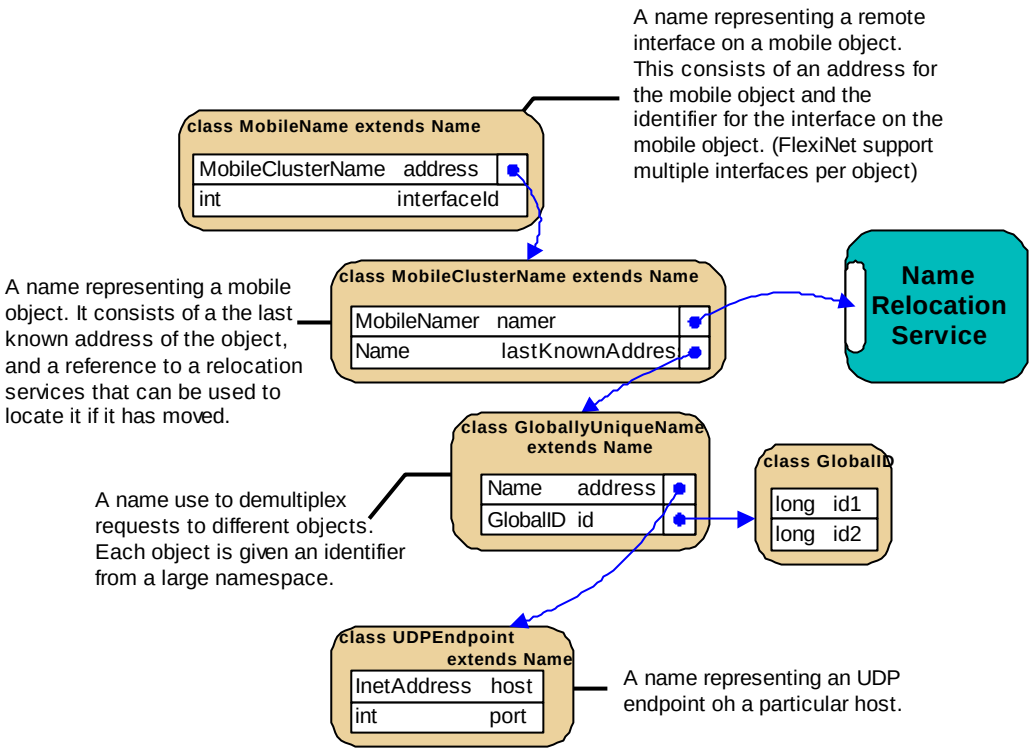

Figure 2 The format of a mobile name

Requirements for Mobile Objects

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

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. This was not an feasible option for the MOW.

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 used. 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.

Consistency and Threads

At any point in time, an object may be *active* or *passive*. An active object is one that has one or more threads 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.

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.

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.

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 later in this document.

Requirements for Mobile Agents

The mobile object workbench is not a mobile agent system, however it is intended as a platform on which Mobile Agents will be developed. When designing the Mobile Object Workbench, it was important to consider the relevant requirements for mobile agents.

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.

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. A later section discusses security issues in more detail.

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.

MOW Concepts

Figure 3 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.

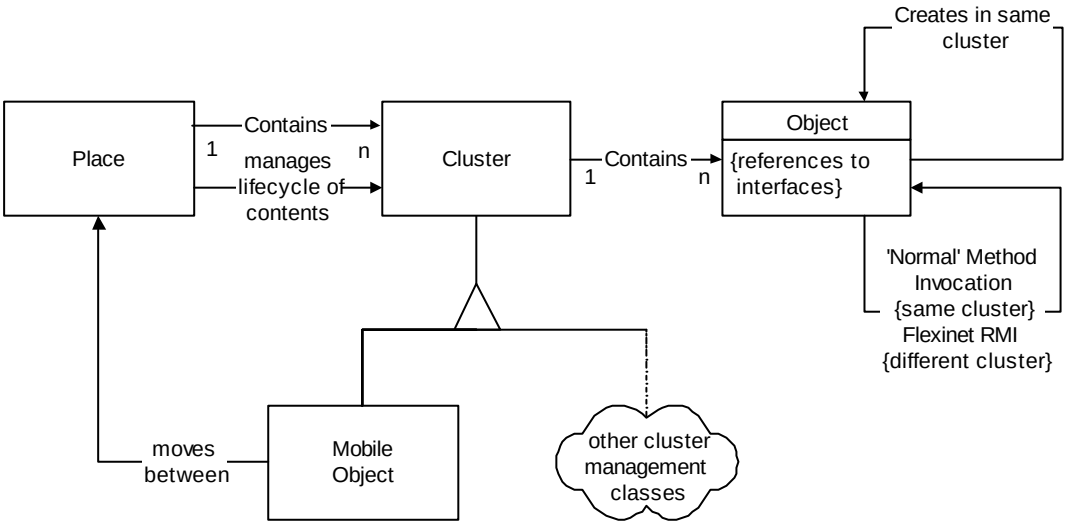

Figure 3 Objects, Clusters and Mobile Objects

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.

API

The Mobile Object workbench API is entirely concerned with the lifecycle of clusters - communication takes place using application-level exported interfaces, and transparent remote invocation.

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

`public Place getPlace()`

Return a reference to the place at which this cluster is currently located.

`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. This method is used internally by the MOW infrastructure, but is also available for specialist use by a subclass of Cluster. A particular subclass may arrange that a cluster is always restarted in a locked state by invoking lock() prior to any method that may lead to `restart()` being called (i.e. move or copy operations).

`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 (optionally) return an interface on any object in this cluster. This interface is returned to the creator of the 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. A destroyed cluster is conceptually removed from the system and should not attempt any further processing. Different version of the MOW may enforce this destruction to different degrees. (JDK 1.1 does not provide sufficient mechanisms to enforce destruction).

`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. A restarted cluster has the same lock status as it did prior to the method call which lead to restart.

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. If the move fails then *either* an exception is thrown prior to the call returning, *or* `restart()` is called with the reason for failure as an argument.

`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. This call should not normally be made within the body of a method servicing a request from outside the cluster, as the thread will not never return if the call is successful. In practice, after a successful call, the caller will receive an exception indicating that the thread was terminated.

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.

`public Object getProperty(String name)`

return a property associated with this place. A place may provide arbitrary properties. It is still undecided as to which properties should be provided by all places, and which are optional.

Place Implementations

There is currently one place implementation within the MOW (`UK.co.ansa.flexinet.mobility.place.PlaceImp`). This provides basic cluster creation and object mobility functions. It is likely that other place implementations will be created within the MOW (e.g. "SecurePlace"). In addition other work packages (particularly WP C - Autonomous Agents) may wish to subclass `PlaceImp`, or create alternative implementations in order to provide additional, agent related, functionality. As a side effect of this, it is likely that the Place interface itself will be sub-classed in order to add additional methods.

PlaceImp

`PlaceImp` is a simple default implementation. In addition to supporting the Place interface, it provides the following properties.

- `(FNetTrader) MOW.trader`
  An interface reference to the Trivial Trader. This is a simple name service used for development purposes. It may be used to lookup remote services, or advertise interface for remote access.
- `(String) Place.name`
  A name given to this place by its creator.
- `(Boolean) MOW.AWTenabled`
  A boolean indicating if this place instance supports the Java Abstract Windowing Toolkit.

Sub-classing Place and PlaceImp

A subclass `PlaceImp` will probably require that `Cluster.getPlace()` returns a subclass of `Place.` In order to do this, the subclass of `PlaceImp` must overwrite the field `Class publicIface = Place.class;` with the subclass of Place. This is because FlexiNet (quite correctly) prevents a client from widening an interface reference to a remote service, and Places are effectively remote from clusters. It is therefore essential that the place indicates the precise class of the interface to be implicitly exported on cluster creation.

Implementation

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 4.

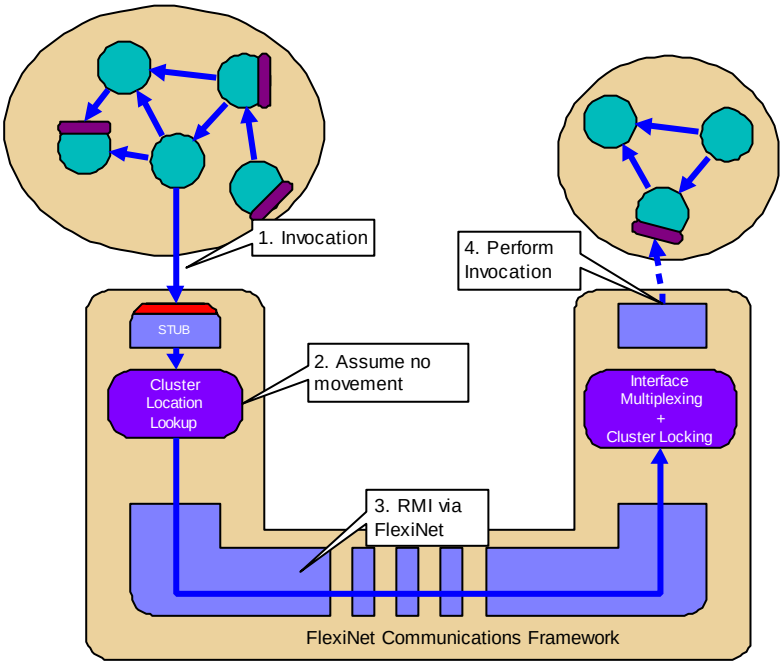

Figure 4 Implementation of inter-cluster calls

If a cluster has moved since it was last accessed by a particular client, then the server side encapsulation layer will throw an exception indicating this. The client side cluster location layer catches this exception, and then contacts a relocation service to determine the new address for the cluster. The procedure is then repeated. Figure 5 Illustrates.

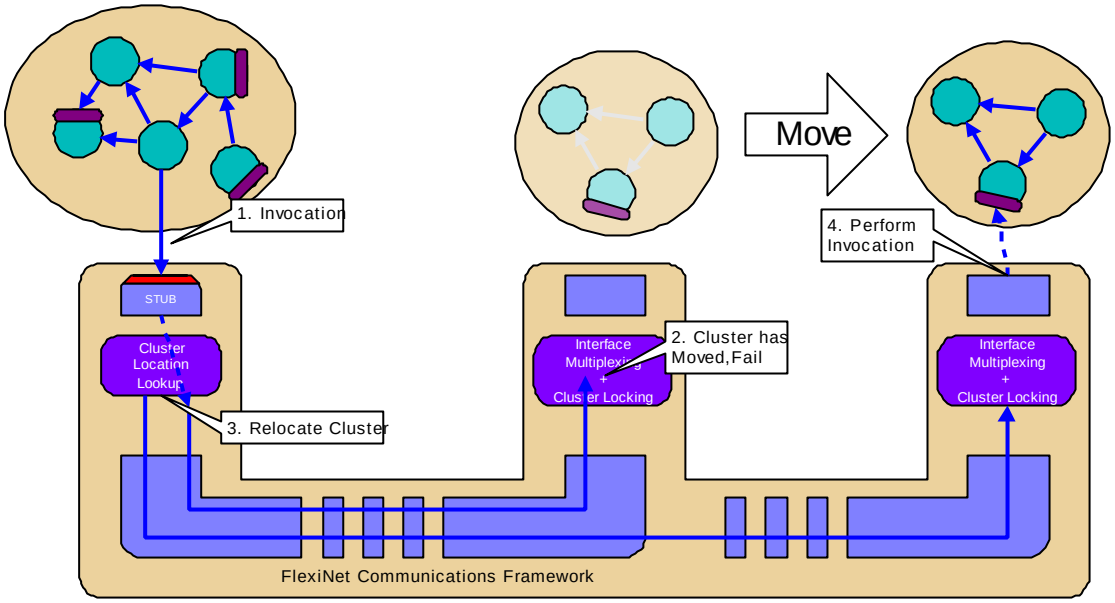

Figure 5 Back off and retry due to cluster movement

Orchestrating Mobility

Figure 6 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.

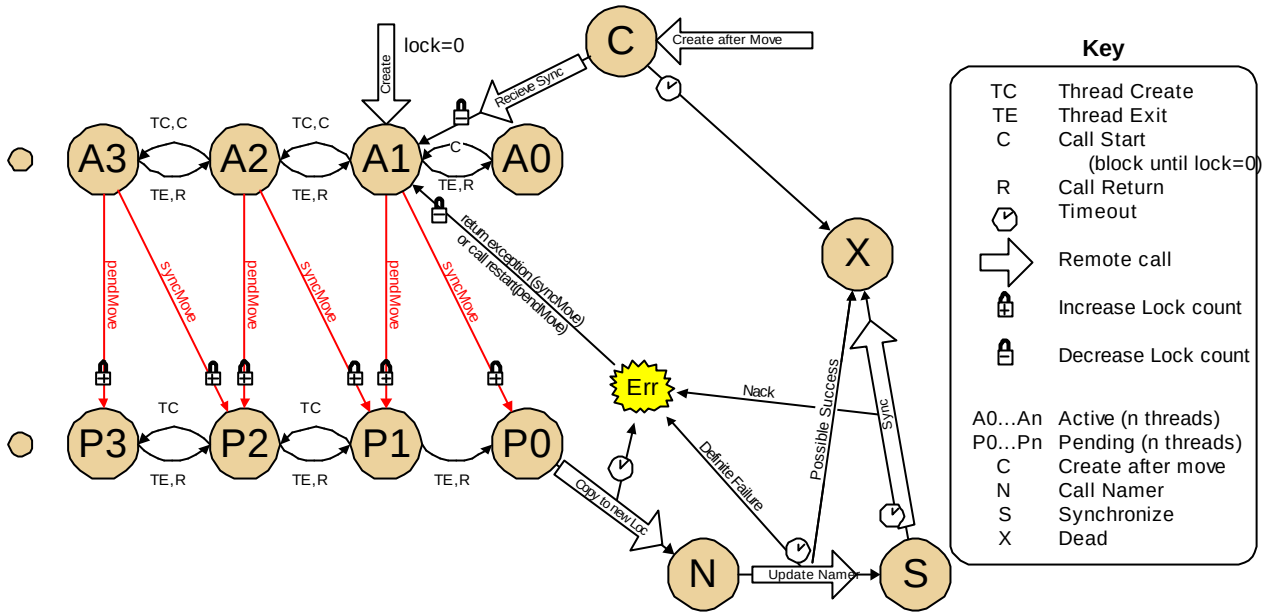

Figure 6 State transitions of a mobile object instance

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.

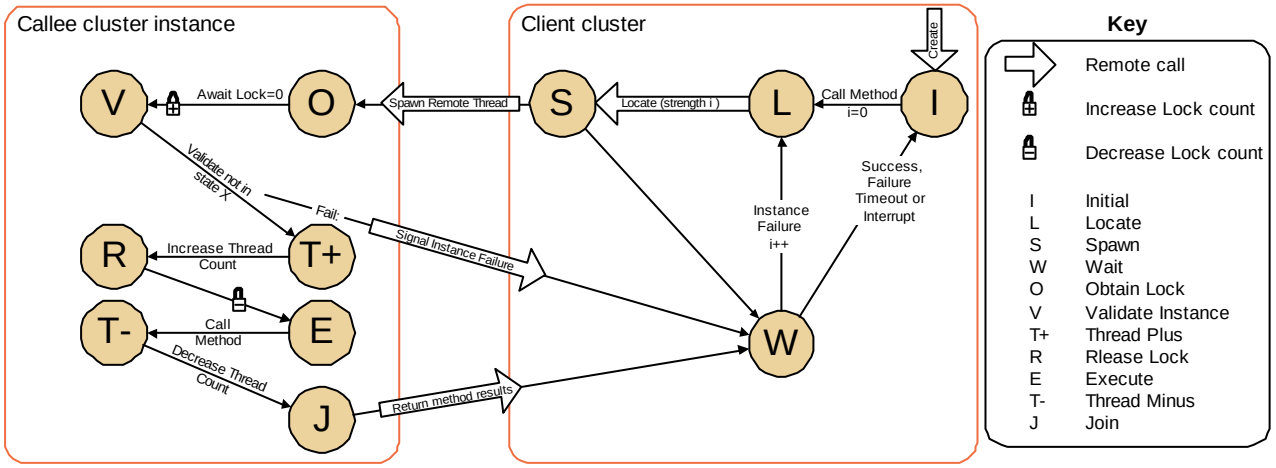

Figure 7 State transitions for an invocation on a remote cluster

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[3] 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 8 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.

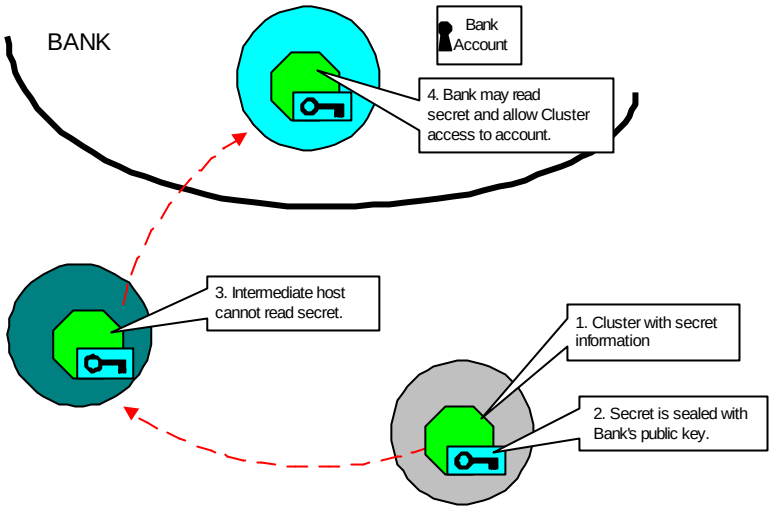

Figure 8 Clusters with Secrets

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[4]. 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.

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 in-definitely. 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 af-fect 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.

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:

1. 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.

2. 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.

3. 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).

4. 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 9*, *Figure 10* and *Figure 11*. In *Figure 9*, 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.

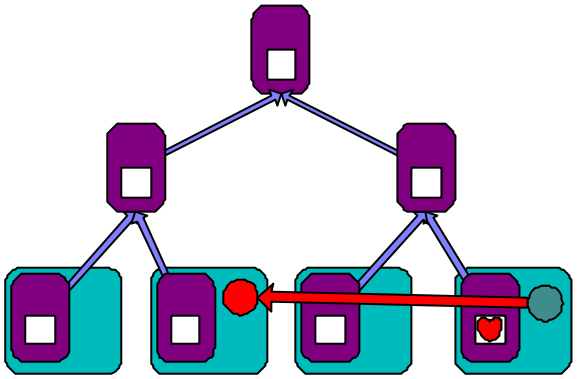

Figure 9 - Relocation service: Mobile Object moves

In *Figure 10*, 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.

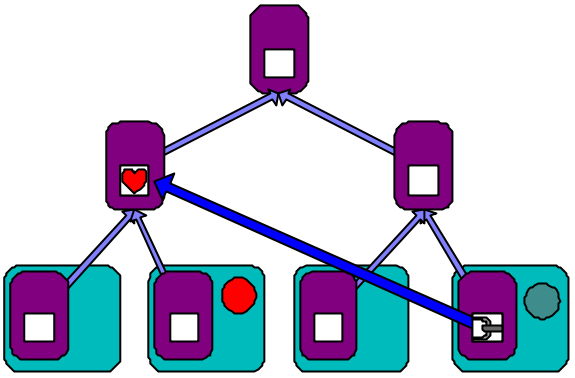

Figure 10 - Relocation service updated

In *Figure 11*, 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.

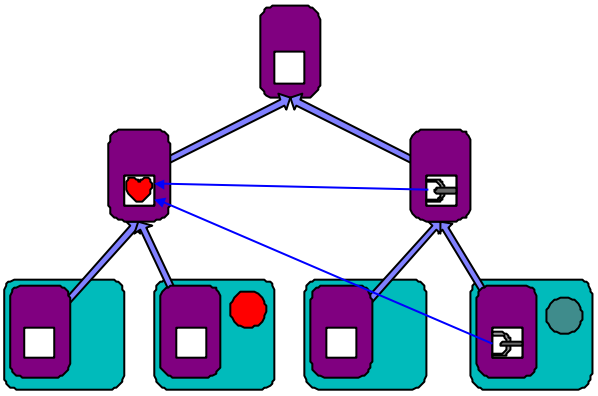

Figure 11 - Relocation service - caching and copying

Events

An event service has been constructed using primitives from the Mobile Object Workbench. This may be used by mobile (and non-mobile) object to signal events to each other.

JDK 1.1 Event Model

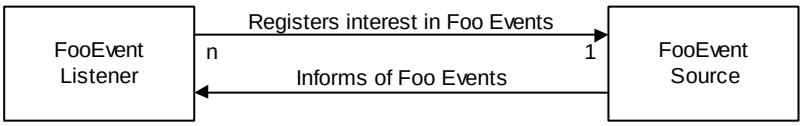

Figure 12 JDK 1.1 Event Model

JDK 1.1 contains a definition of an event model whereby *listeners* can register interest in certain events by invoking an appropriate `addListener(…)` call on a service object. The service object may then notify the listeners by calling each back in turn using an interface supplied at registration time. In the JDK model, event notification is synchronous, and the object signalling the event can therefore determine when all parties have been notified. (Figure 12)

The JDK model may be viewed as a template for a particular event implementation. It prescribes method signatures, but does not provide implementation, nor does it discuss distribution related issues, such as the action to be taken if a listener is unavailable, or how low-level mechanisms such as message multi-cast may be used.

Mobile Event Issues

When considering an event service for the Mobile Object Workbench it was decided that the service should provide an *engine* for event transmission, to be hidden behind an interface appropriate for the particular application (for example the JDK 1.1 event model). The issues to be considered in the design of such an engine are the mechanisms for maintaining a set of listeners, and the mechanisms for performing callback on event occurrence.

In a distributed environment, it may take a considerable time to inform all listeners of an event occurrence. Indeed, if some listeners are no longer accessible, or have been destroyed, it may not be possible to inform them at all. It is application dependant what action should be taken in this circumstance. A secondary point is that a mobile object is unable to move whilst a thread is active in it. Together these features could lead to a mobile object blocking indefinitely when signalling an event, and effectively been prevented from moving. Many potential application of events require simultaneous event notification and object movement, this requires *asynchronous* notification, so that a mobile object may signal an event, and then move whilst some background task completes the actual notification.

For these reasons, the event service was designed as two complementary services. The first, (BroadcastGroup) maintains a set of listeners. An implementation of an addListener method may utilise a local instance of this service to store a set of listeners for a particular event class. The BroadcastGroup implementation is basically an atomic set implementation.

The second service is a broadcast service that will invoke a method on a set of clients. This is used by the BroadcastGroup implementation to perform actual event notification. Typically the broadcast service is implemented as part of a non-mobile service (e.g. a Place). In order to inform its listeners of an event, an event source generates a suitable method invocation (signature + arguments) and then passes this to the local BroadcastGroup managing the listeners. This in turn synchronously invokes the broadcast service passing it the invocation description, and the current population of the listener set. The broadcast service returns an identifier for this group invocation, and then asynchronously invokes the method on each of the listeners. During this time, the event source is at liberty to move (if it is mobile). When the group invocation is complete, the broadcast service calls back to the event source to inform it of the completion status. The event source will also be called back if any individual invocation failed. This is illustrated in Figure 13.

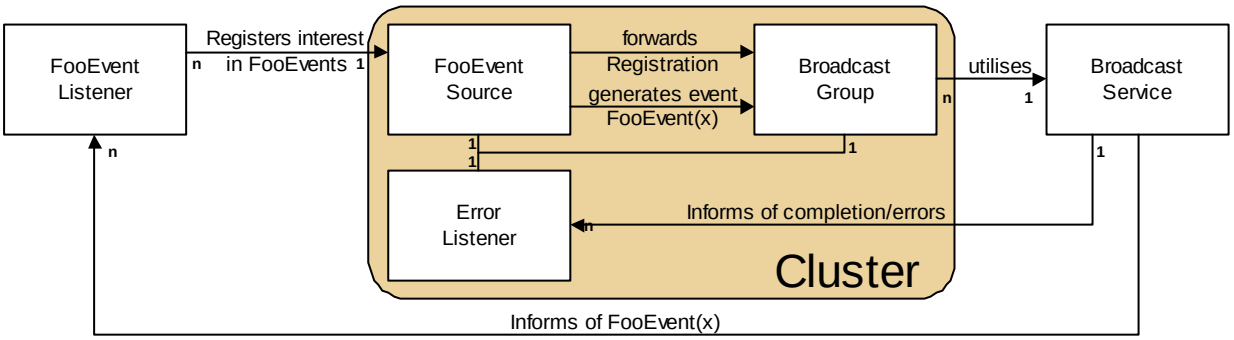

Figure 13 Event Notification in Practice

Using the Mobile Object Workbench

This section contains additional information for users of the Mobile Object Workbench. In particular it considers details of how to create, destroy and communicate with mobile objects. To use the mobile object workbench, programmers should be aware of the concepts described in this paper, in particular, the concepts of *object*, *interface* and *cluster* shown in Figure 14.

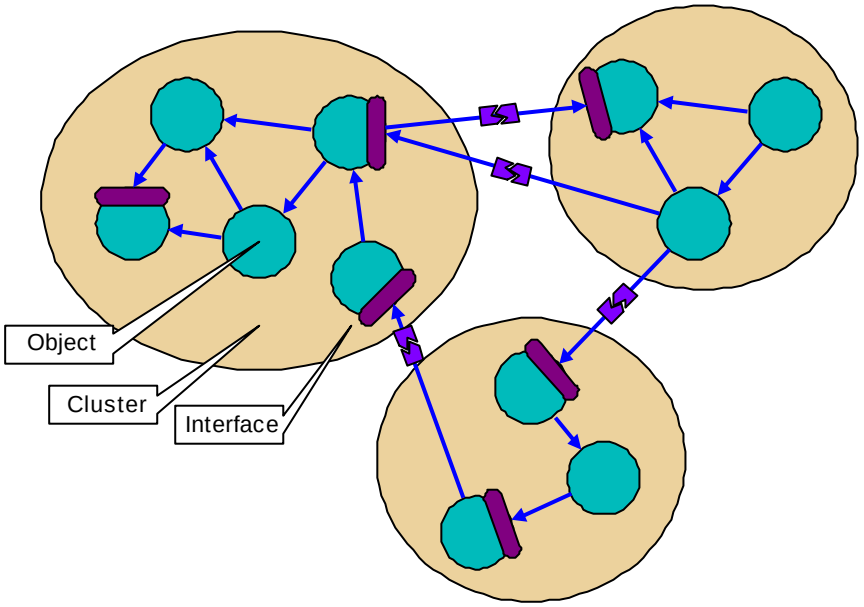

Figure 14 Objects, Interfaces and Clusters

Generic (untyped) Interfaces

There are times when an object (client) calling another object (server) wishes to pass a parameter in a generic (untyped) way. In normal Java programming, this is easy - the server is passed an object as if it were of type `java.lang.Object` , and this can later be cast back to the appropriate type. An example of a simple use of this is `java.lang.Hashtable` . In a distributed system, it is inappropriate to allow client of a service to widen (cast) an interface reference to that service. This might allow the client access additional methods which were intentionally not specified in the original interface reference. For this reason, FlexiNet *does not allow* reference to interfaces on remote objects to be widened (cast from superclass to subclass). An unfortunate side effect of this, is that you cannot pass an interface as it if were of type `object`, and then cast it back to its actual type, even when this is the ‘ right and proper’ thing to do.

The Tagged Class

To get round this limitation FlexiNet provides a Tagged() class. This is a simple tuple of a reference to an arbitrary interface, and the class of this interface. This may be used to pass interfaces in a generic way, and is treated as a special case, so that the interface can be widened to the indicated class. In circumstances where it is necessary to pass an interface in a generic way, the Tagged class should be used. An example of this is when adding or removing interface references from a trader. (However, the Trivial Trader implementation provided with the Mobile Object workbench provides a wrapper class to perform the tagging for you).

There are two ways to create a new `Tagged` object. Many objects implement more than one interface, and the developer can choose which of these interfaces to make available via the Tagged object, or allow the Tagged object to choose - in which case it will choose the first interface. The first interface is described as "the first interface name to appear in the 'implements' clause of the object's class definition, or if it implements no interfaces directly, then this is the first interface implemented by its direct superclass, or nearest ancestor that implements an interface (in other words, when in doubt, specify which one you want!).

Creating Tagged objects

Assuming that class `blob` implements the interfaces `Simon` and `Bolivar`, and that object `bb` is of class `blob`:

1. either interface may be used

   `Tagged j = new Tagged(bb, Simon.class)` or `Tagged j = new Tagged(bb, Bolivar.class)`

2. or the shorthand constructor may be used

   `Tagged j = new Tagged(bb)` in this case, a reference to the first interface implemented by `bb`.

Multiple Interfaces on one Object

In FlexiNet and the Mobile Object Workbench, objects are accessed through *opaque* interfaces. That is to say that the callee cannot determine which object implements an interfaces, nor whether two interfaces are implemented by the same object. This allows the implementor of a service with several interfaces the freedom to change the internal design. Although done for good reason, this approach can seem confusing to a programmer. In particular it is not possible to cast from one interface to another (for the same reasons as the restrictions on widening described above). The programmer may find it easier to program as if each interface were on a different object. In particular, if an cluster/service wishes to export several interfaces, them a strategy must be adopted for a client to obtain references to these interfaces. There are many possibilities

1. One interface may be returned to the creator of the cluster, as a return value from `Place.newCluster()`
2. One or more interfaces may be exported to a Trader, or passed to a third party
3. One or more interfaces may contain methods used to obtain references to the other interfaces

Trivial Trader

A simple trading service is provided as part of FlexiNet and the Mobile Object Workbench. This can be used by MOW clients services and mobile objects as a simple directory to locate interfaces on other MOW objects.

`TrivTrader`, the basic Trader provided with MOW v1.1, is a simple name server which allows the look-up of interfaces by name. These names are those specified when the interface is exported to the trader, and clashes are not resolved - it has a simple hashtable implementation. Only interfaces may be stored in the trader. Figure 15 gives an example of the Trader in use.

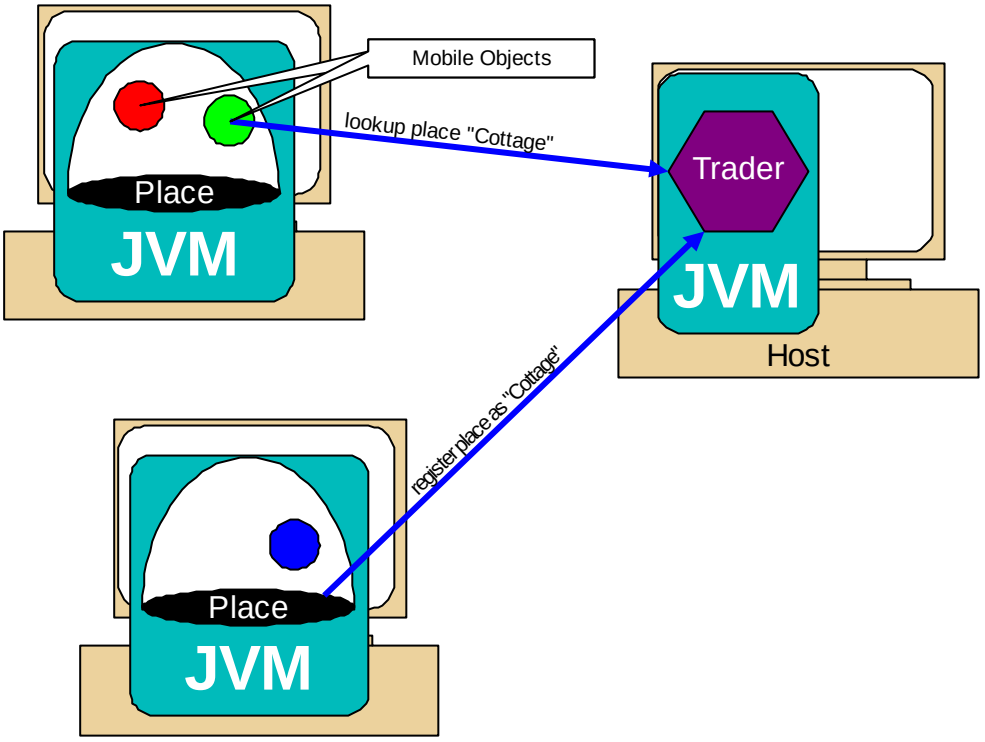

Figure 15 The Trivial Trader

Exporting an interface to the trader

`void FNetTrader.put(String name, Object obj, Class ifaceclass)` or
`void FNetTrader.put(String name, Object obj)` , where:

- `name` is the name for the interface in the trader (for instance "MyService")
- `obj` is the object that implements the interface being exported
- `ifaceclass` is the class of the interface being exported.

Note. `FNetTrader` is actually a (local) wrapper class for the interface `UK.co.ansa.flexinet.trivtrader.Trader` . This provides wrapping of interfaces using the `Tagged` class.

Importing and interface from the trader

`Object FNetTrader.get(String name)`

Setting up a trader

When running MOW objects, a trader is generally required, and clients, services and other objects wishing to use the trader must have the address of a trader set in the system properties. For Sun's Java interpreter this may be done using the -D flag. It is generally worthwhile setting this address up as a system environment variable, and this is described below.

Steps to running a trader (NT):

1. run the trader for the first time - typically with '`java UK.co.ansa.flexinet.trader.TrivTrader` '
2. note the output, which will be of the form '`flexinet.trader=(aaa.bbb.ccc.ddd:pppp)(n)` ', where "`aaa.bbb.ccc.ddd` " is the IP number of the machine running the trader, "`pppp`" is the port number being used (initially

random), and "n" is the exported interface number. (Note, in later releases of the MOW, the trader address may be of a different format (e.g. a CORBA IOR).

3. set an environment variable, e.g. "FT", with the value "-Dflexinet.trader=(aaa.bbb.ccc.ddd:pppp)(n) " (where the values of are as above)
4. you may stop the trader - next time it is restarted, give it the address to run at:
   'java %FT% UK.co.ansa.flexinet.trader.TrivTrader '
5. next time you run any MOW components, run them as "java %FT% blob ", where blob is the class you are running.

Steps to running a trader (UNIX):

run the trader for the first time - typically with 'java UK.co.ansa.flexinet.trader.TrivTrader'

1. note the output, which will be of the form 'flexinet.trader=(aaa.bbb.ccc.ddd:pppp)(n) ', where "aaa.bbb.ccc.ddd " is the IP number of the machine running the trader, "pppp" is the port number being used (initially random), and "n" is the exported interface number. (Note, in later releases of the MOW, the trader address may be of a different format (e.g. a CORBA IOR).
2. set an environment variable, e.g. "FT", with the value "-Dflexinet.trader=(aaa.bbb.ccc.ddd:pppp)(n)" (where the values of are as above - you'll need the double quotes to escape out the parentheses!)
3. you may stop the trader - next time it is restarted, give it the address to run at:
   'java $FT UK.co.ansa.flexinet.trader.TrivTrader '
4. next time you run any MOW components, run them as "java $FT blob ", where blob is the class you are running.

Alternatively, you can just use "java -Dflexinet.trader=(aaa.bbb.ccc.ddd:pppp)(n) blob " each time you run an MOW component, where the trader address is that of the current instance of the trader, as it may change between instantiations.

Creating Places

There is currently only one implementation of a place provided in the MOW. This is UK.co.ansa.flexinet.mobility.place.PlaceImp .

A new place may be created by creating an instance of this class. Note the following restrictions

- There may be only one place per JVM. This is to make the engineering easier, and may be lifted in a later implementation.
- Mobile Objects cannot create places. This is non-sensical, as well as being a special case of the above restriction.
- Places cannot be created in browsers. This is due to security restrictions and bugs in current browser implementations.

Note that mobile objects may be created in remote places, as well as local places (using Place.newCluster) and that it is not necessary for all services that are to be accessed remotely to live in the same JVM as a place (or indeed be mobile). The utility class UK.co.ansa.flexinet.mobility.MOWClient may be used to obtain a reference to the TrivTrader, and hence mobile objects and remote services without the need to create a local place. It is envisaged that many services (for example the Trader itself) will exist in non-Place JVMs. There is however a restriction, a single JVM may contain at most one Place or make use of MOWClient. Under no circumstances should use of the two mechanisms be combined (this will be enforced when the Security manager is completed). A corollary of this is that a mobile object should not use MOWClient to obtain a reference to the trader. Instead

```
MobileObject.getPlace().getPropery("MOW.Trader")
```

should be used.

Writing a mobile object

There are a number of specific issues to be remembered and addressed when writing a mobile object: some of these are listed below. These issues include:

- providing a no-args constructor
- what should the init(...) signature include?
- what should the init(...) return type be?
- locking and unlocking
- movement
- restarting
- copying
- stop and destroy

Initialisation and constructors

When a Cluster, or MobileObject, is created, *one of* the methods called `init` is invoked.

Generally a subclass of `MobileObject` will provide its own `init` method, with appropriate arguments, and an `init` method will be dynamically chosen at run time that matches the caller's arguments. If more than one init method matches a set of arguments the behaviour is undefined.

If desired, the init method may return any *interface* to the caller. This interface will typically be on one of the newly created objects within the cluster. As mobile objects are constructed by calling `Place.newCluster()` this interface is returned to the caller after wrapping using the Tagged class.

init and restart

The `init(…)` method is called exactly once - when an object is created. The `restart()` method may be called many times, typically after an object has been moved to a new location, although it may also be called after some recoverable system error (such as a failed move). Note by default, `restart()` is not called on object creation, although the `init()` method may invoke it explicitly.

no-args constructors

Note that due to restrictions imposed by the Java language on serialisation, all objects passed by value (including Clusters and MobileObjects) are required to have a no-arguments constructor. This need not perform any action, as it will only be used to create an empty object prior to the deserializing the object's fields. In particular if the no-arg constructor sets non-transient fields, then these will be overwritten.

Locking and unlocking

The `lock()` and `unlock()` methods on a Cluster or a MobileObject allow calls made on interfaces exported by the cluster to be blocked. Typically this is to allow the cluster to move. Calls which have already started processing continue to be executed. An implementor of a MobileObject may wish to utilise locking for other reasons. As locks may be held recursively, a particular mobile object may acquire a lock prior to calling `syncMove` or `pendMove`, and in this case the newly moved object will be restarted while it is still locked. This may be useful to allow it to perform re-initialisation undisturbed. When `syncMove`, `pendMove` or `copy` is called, one lock is automatically acquired. This may not be released by the programmer, and is automatically released one the move is complete, or has failed.

Moving & restarting

Two methods are available to request a move by a MobileObject (these are not available to Clusters!), `syncMove` and `pendMove`. Neither of these is exported from a MobileObject by default, and so a MobileObject is therefore autonomous in the sense that other objects cannot force it to move. These two methods can be briefly characterised as follows:

- `syncMove` - this will attempt to move using the thread from the method call, which will therefore not return. However, the move will not take place until there are no other threads in the cluster. If the move fails an exception is thrown. This should not normally be called in situations where the thread should return, for example when servicing a remote request from another cluster. If `syncMove` *is* used in this circumstance, the client is likely to receive an exception indicating that the thread has been destroyed.
- `pendMove` - in this case, a new thread will be spawned to perform the move. Again, the move will not take place until there are no other threads in the cluster. An exception may be thrown or may later be passed to the restart method as the Cluster is restarted in the same Place. This method is typically used in situations where `syncMove` is inappropriate, for example when servicing a client request.

In both cases, an exception will be thrown if:

- a move is already in progress
- a copy is already in progress
- the move fails due to a remote error.

On arrival at the new place, the restart method is called - see the API for details of possible exceptions.

NOTE - on a move, only public fields are serialised, due to restrictions imposed by the Java language (JDK 1.1). However, this is a minor restriction, as fields which might otherwise be declared private or protected may be declared public in a MobileObject due to the strong encapsulation: they will not be visible outside of the cluster. In JDK 1.2, it will be possible to serialize private fields, and this restriction will be lifted.

Copying

When a Cluster is copied, the new version is created, and then the `copied()` method called. The current default behaviour is for this to call the restart method. It is possible for a sub-class of MobileObject to override this, and in particular the copy may return an interface on itself to the original cluster.

Stop and destroy

stop() is a method called by the local Place if it wishes a Cluster to cease processing. The Cluster is expected to clean up and then return. When the call returns, the Place will invoke destroy(). This then destroys the Cluster as effectively as possible. In the current implementation, this prevents new calls from outside the Cluster, and attempts to prevent the Cluster from continuing processing.

It should be noted that neither of these methods addresses issues of thread clean-up. The implementation of stop() should include management and shut-down of threads.

The Java Abstract Windowing Toolkit

Generally, the strong encapsulation provided by the Mobile Object Workbench means that calls made between Clusters (and therefore MobileObjects) are safe, in as much as they maintain strong encapsulation and thread separation. However when other non-MOW APIs are used, there is a potential for problems.

The Java Abstract Windowing Toolkit is a pathological example of this. Firstly, it calls the cluster (rather than visa versa) when it informs it of windowing events. As this call is not under the control of the mobile object workbench, there is no possibility of enforcing the thread encapsulation needed. This would allow one cluster to block when servicing an AWT event, and prevent other clusters from using the AWT. Another side effect of this is that the mechanisms used for thread counting, which is necessary for determining when it is possible to move a cluster, no longer work. Finally the AWT is pathological in that the ThreadGroup (ownership) of the thread used for event callback is effectively set at random.

To overcome these problems, we have created an abstraction of the AWT to be used for event callback. This effectively acts as a buffer between the ‘ real’ AWT and a cluster that is a client of it, and enforces the appropriate encapsulation.

AWTEventSource

This is an abstraction of the AWT used for event processing. Each Cluster needing AWT events should have one `AWTEventSource` , which can register Listeners of any of the types currently within the `java.awt.events` package. Please note that `AWTEventSource` is not designed to work with Java 1.02 type events, but Java 1.1+ events only.

Once an AWT component has been instantiated, rather than registering an object within the cluster as a Listener on the component using

```
component.addFooListener(myListener)
```

The `AWTEventSource` object should be used. I.e.

```
myAWTEventSource.addFooListner(myListener,component)
```

Unregistering follows the same pattern. When an event occurs, the use of AWTEventSource is completely transparent *except* that the correct encapsulation takes place.

Note: currently, a Cluster has to create an `AWTEventSource` for itself. This is likely to change, and in later releases, it is expected that a Cluster will get a reference to an `AWTEventSource` from its current Place.

In detail

1. create an AWTEventSource -

   `AWTEventSource myAWT = new AWTEventSource(this);` (where 'this' is a Cluster)

2. create a one ore more AWT components -

   `Button blob = new Button("blob");` (as usual)

3. register with the component using `AWTEventSource` -

   `myAWT.addActionListener(myListener, blob);` (where, `myListener` is an `ActionListener`)

   Note - Buttons require `ActionListeners` - other components produce different `AWTEvents`. Please check `java.awt` specifications for details.

4. proceed as usual!

When moving

1. unregister -

   `myAWT.removeActionListener(myListener, blob);`

   otherwise we may collect 'dangling' references which cannot be garbage collected.

2. dispose of the component -

   `blob.dispose();`

   This is generally good practice anyway - but particularly so within the context of mobility.

Note: When a Place is first create, a ‘ splash screen’ is created. This is actually to overcome the ‘ randomness’ of AWT thread creation. It may be safely disposed of by hand.

Summary

The Mobile Object Workbench was designed primarily to add *mobility transparency* to the distribution transparency provided by FlexiNet. To do this we have added 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.

Implementation Status

The Mobile Object Workbench version 1.1 has been deployed to project members and is currently in use. The current implementation supports clusters, mobility and transparent communications as described in this document. 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 this document. 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 added support for creating secure bindings (using SSL), and interworking with CORBA clients/services using IIOP. Transactional support for FlexiNet is underway[5] and we hope to integrate this, and other FlexiNet enhancements into later releases of the MOW.

Appendix 1: JavaDoc API

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

```
java.lang.Object
   |
   +----UK.co.ansa.flexinet.mobility.Cluster
```

Clusters are encapsulated groups of objects. Each cluster resides at a place, and communicates with other clusters via method invocation on exported interfaces. Subclasses of Cluster may be able to move between places.

Author: Richard.Hayton@ansa.co.uk
See Also: MobileObject, Place

Methods

o **toString**
`public String toString()`
Return a unique name for debugging
Overrides:
toString in class Object

o **getPlace**
`public Place getPlace()`
The place at which this cluster is currently residing.

o **lock**
`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.

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

Throws: UnMatchedLockException
The lock was unlocked too many times

o **stop**
`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().

o **destroy**
`public final void destroy()`
This call destroys the cluster as effectively as possible. In the current implementation this prevents new calls from outside of the cluster, and attempts to prevent the cluster from continuing processing.

o **startCall**
`public synchronized boolean startCall()`
Indicate the start of a call from outside the cluster. This will block if the cluster has been locked. This call is normally only used by the communications infrastructure.
Returns:
false if the cluster has been destroyed

o **endCall**
`public synchronized void endCall()`
Indicate the end of a call from outside the cluster. This call is normally only used by the communications infrastructure.

o **restart**
`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. A cluster may be restarted for many reasons, for example after movement or after failure recovery. To distingish between failure related restarts and non-failure related restarts, non-failure related restart exceptions are subclasses of `NonFailureRestart` .

o **copied**
`public Tagged copied()`
Called on a newly created copy of an object. By default this will call `restart` with `Copied` as the restart reason. A sub-class that wishes to return an interface to the copier of the mobile object, should override this method. The interface returned should be tagged to distinguish its class.
Returns:
A tagged interface to return to the original.

o **init**
`public void init( ExampleArgumentClass arg1)`
Called on 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. If more than one `init()` method matches a set of arguments the behaviour is undefined. The `init` method may return an interface on any object in this cluster.

Class UK.co.ansa.flexinet.mobility.MobileObject

```
java.lang.Object
   |
   +---- UK.co.ansa.flexinet.mobility.Cluster
           |
           +----UK.co.ansa.flexinet.mobility.MobileObject
```

MobileObject are clusters that have the ability to move between places

Author: Richard.Hayton@ansa.co.uk
See Also: Cluster, Place

Methods

o **unlock**
`public synchronized void unlock() throws UnMatchedLockException`
Decrease the number of locks held. If the number of locks held is zero, wake any blocking calls. `syncMove` , `pendMove` and `copy` all automatically acquire a lock. This cannot be released by client code.
Throws: UnMatchedLockException
The lock was unlocked too many times
Overrides:
unlock in class Cluster

o **copyOrMovePending**
`public boolean copyOrMovePending()`
Determine if an operation is pending. Return true if this mobile object is currently in a pending state. When in a pending state, the cluster will move (or be copied) as soon as all other threads have terminated.

o **pendMove**
`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. If a move or copy is already in progress, or if the move is guarenteed to fail, then an exception is thrown. If during the move, an exception is raised, restart() is called By default, this method is not exported from a cluster, and a mobile object is therefore autonomous, in the sense that other clusters cannot force it to move.
Parameters:
dest - The place to move to.
Throws: MoveFailedException
The move failed.

o **syncMove**
`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. If a move or copy is already in progress, or if the move fails, then an exception is thrown. By default, this

method is not exported from a cluster, and a mobile object is therefore autonomous, in the sense that other clusters cannot force it to move.

Parameters:

dest - The place to move to.

Throws: MoveFailedException

The move failed for some reason.

o **copy**

```
public  Tagged copy( Place dest) throws  MoveFailedException
```

Copy the object. The object is copied to the given place. This will block until there are no other threads active within the cluster. If the mobile object overrides the `copied` method, then the new object may return an interface to itself to the old (parent) object. **Note** it is not possible to have a pending copy and a pending move at the same time.

Parameters:

dest - The place at which to create the copy.

Returns:

A tagged interface to the new copy, or null.

Throws: MoveFailedException

The copy did not succeed.

Interface UK.co.ansa.flexinet.mobility.Place

public interface **Place**

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

Author: Richard.Hayton@ansa.co.uk
See Also: Cluster, MobileObject

Methods

o **newCluster**

```
public abstract  Tagged newCluster(Class cls) throws InstantiationException
```

Create a new cluster at this place Once created, init() will be called on the new object.

Parameters:

cls - The class of the cluster to be created

Throws: InstantiationException

The cluster could not be created, or init() raised an exception

o **newCluster**

```
public abstract  Tagged newCluster(Class cls,
                                   Object args[]) throws InstantiationException
```

Create a new cluster at this place Once created, init(arg0,arg1,...) will be called on the new object. The place will attempt to find a matching method. If more than one method matches, the resulting behaviour is undefined. **Note.** As all arguments are being passed as objects (not interfaces) the default mechanism will be to copy their value. If the intended semantics was to pass interfaces, then these

should be wrapped using the `Tagged` class. Similarly, the returned interface (if any) is wrapped.

Parameters:
cls - The class of cluster to create
args - The arguments to pass to `init(...)`
Returns:
The interface returned by `init(...)`
Throws: InstantiationException
The cluster could not be created, or init() raised an exception

o **getProperty**

```
public abstract Object getProperty(String name)
```

Return contextual information. This is similar to the java properties system, except that arbitary data, not just strings, may be stored. If the property is not known, or the place does not wish to release it to the callee, then null is returned.

o **receiveMove**

```
public abstract  MobileClusterName receiveMove( MobileClusterName oldname,
                                                Cluster cluster,
                                                ExportEntry exports[])
```

receive a cluster an recreate it at this place. `name` is a name the cluster previously had at a different location. It will not be restarted until a matching `sync` is received. This method is only normally called by the internals of class MobileObject. **It is liable to change in later releases of the MOW.**

Parameters:
name - The name of the cluster prior to it's move.
obj - The cluster itself (it will be copied)
exports - A list of exported interfaces from this cluster.
Returns:
The name of the newly recreated cluster.

o **sync**

```
public abstract boolean sync( MobileClusterName name)
```

Restart a previously received object. This method is only normally called by the internals of class MobileObject. **It is liable to change in later releases of the MOW.**

Parameters:
name - The name of the cluster that was returned by receiveMove
Returns:
true on possible success, false on definite failure

o **receiveCopy**

```
public abstract  Tagged receiveCopy( Cluster cluster,
                                     ExportEntry exports[])
```

Create a new cluster based on this image. When recreated, `copied` will be called. If this returns an interface, this will be returned from this call. This method is only normally called by the internals of class MobileObject. **It is liable to change in later releases of the MOW.**

Parameters:
obj - The cluster itself (it will be copied)
exports - A list of exported interfaces from this cluster.

Returns:
The interface returned by `copied`

Class UK.co.ansa.flexinet.mobility.place.PlaceImp

```
java.lang.Object
   |
   +----UK.co.ansa.flexinet.mobility.place.PlaceImp
```

An implementation of a Place This class is primarily concerned with the creation and destruction of clusters, and with the instantiation of clusters from serialized forms. In order to enforce access control, or resource usage policies, this class should be subclassed, and the appropriate methods overridden.

Constructors

o **PlaceImp**
`public PlaceImp(String name)`
Create a place. A System property is used to find the trader location.
Parameters:
name - A name to help application objects orient themselves.

o **PlaceImp**
`public PlaceImp(String name, String traderIdent)`
Create a place.
Parameters:
name - A name to help application objects orient themselves.
traderIdent - Location of the trader.

Methods

o **parseIfaceName**
`public static  Place parseIfaceName(String sname)`
Create a reference to the remote place represented by the stringified name sname. This is provided for debugging purposes.

o **getIfaceName**
`public String getIfaceName()`
Return a string represting the name of this place, suitable for use by parseIfaceName.

o **setProperty**
`public void setProperty(String name, Object value)`
Add contextual information. This is similar to the java properties system, except that arbitary data, not just strings, may be stored. If the property is not known, or the place does not wish to release it to the callee, then null is returned.

o **callbackArrived**

```
public void callbackArrived( MobileClusterName cluster, Exception reason)
```
receive notification of of an event. This should be overriden by subclasses interested in this occurence.

Parameters:

cluster - The cluster that has arrived

reason - The reason for the arrival (eg Moved, Copied..)

o **callbackLeft**

```
public void callbackLeft( MobileClusterName cluster)
```
receive notification of of an event. This should be overriden by subclasses interested in this occurence.

Parameters:

cluster - The cluster that has left.

o **callbackCreated**

```
public void callbackCreated( MobileClusterName cluster)
```
receive notification of of an event. This should be overriden by subclasses interested in this occurence.

Parameters:

cluster - The cluster that has been created.

o **callbackDestroyed**

```
public void callbackDestroyed( MobileClusterName cluster)
```
receive notification of of an event. This should be overriden by subclasses interested in this occurence.

Parameters:

cluster - The cluster that has been destroyed.

Class UK.co.ansa.flexinet.mobility.MOWClient

```
java.lang.Object
   |
   +----UK.co.ansa.flexinet.mobility.MOWClient
```

This is a static utility class for client applications of the Mobile Object Workbench that do not themselves contain places. It creates the required communications infrastructure so that such clients can communicate with mobile objects, and offer services for use by them. A single JVM may use *either* MOWClient *or* PlaceImp not both.

Methods

o **getTrader**

```
public static  FNetTrader getTrader()
```
Return a reference to the trader.

Class UK.co.ansa.flexinet.mobility.awt.AWTEventSource

```
java.lang.Object
```

```
|
+----UK.co.ansa.flexinet.mobility.awt.AWTEventSource
```

This class acts as a gateway between a cluster and the AWT. It must be used by clusters wishing to receive AWT events.

Constructor

- **AWTEventSource**
  `public AWTEventSource( Cluster myCluster)`
  Parameters:
  myCluster - the cluster that created this AWTEventSource

Methods

For each method, the arguments are the listener for the event, and the AWT component object which is the source of the events.

- **addActionListener**(ActionListener, Object)
  register with as a Listener on an object
- **addAdjustmentListener**(AdjustmentListener, Object)
  register with as a Listener on an object
- **addComponentListener**(ComponentListener, Object)
  register with as a Listener on an object
- **addContainerListener**(ContainerListener, Object)
  register with as a Listener on an object
- **addFocusListener**(FocusListener, Object)
  register with as a Listener on an object
- **addItemListener**(ItemListener, Object)
  register with as a Listener on an object
- **addKeyListener**(KeyListener, Object)
  register with as a Listener on an object
- **addMouseListener**(MouseListener, Object)
  register with as a Listener on an object
- **addMouseMotionListener**(MouseMotionListener, Object)
  register with as a Listener on an object
- **addTextListener**(TextListener, Object)
  register with as a Listener on an object
- **addWindowListener**(WindowListener, Object)
  register with as a Listener on an object
- **removeActionListener**(ActionListener, Object)
  deregister with as a Listener on an object
- **removeAdjustmentListener**(AdjustmentListener, Object)
  deregister with as a Listener on an object
- **removeComponentListener**(ComponentListener, Object)
  deregister with as a Listener on an object
- **removeContainerListener**(ContainerListener, Object)

deregister with as a Listener on an object

- **removeFocusListener**(FocusListener, Object)
  deregister with as a Listener on an object
- **removeItemListener**(ItemListener, Object)
  deregister with as a Listener on an object
- **removeKeyListener**(KeyListener, Object)
  deregister with as a Listener on an object
- **removeMouseListener**(MouseListener, Object)
  deregister with as a Listener on an object
- **removeMouseMotionListener**(MouseMotionListener, Object)
  deregister with as a Listener on an object
- **removeTextListener**(TextListener, Object)
  deregister with as a Listener on an object
- **removeWindowListener**(WindowListener, Object)
  deregister with as a Listener on an object

Class UK.co.ansa.flexinet.mobility.events.BroadcastGroup

```
java.lang.Object
   |
   +----UK.co.ansa.flexinet.mobility.events.BroadcastGroup
```

Maintain a set of clients for an event source.

Constructors

- **BroadcastGroup**
  `public BroadcastGroup( BroadcastService bcast, ErrorListener errListener)`
  Parameters:
  bcast - The service to be used to broadcast events. (optional)
  errListener - an interface on an object to be informed of the status of event notification (optional)

Methods

- **setBroadcastService**
  `public synchronized void setBroadcastService( BroadcastService bcast)`

- **setListener**
  `public synchronized void setListener( ErrorListener errListener)`

- **invoke**
  `public synchronized int invoke(Method method, Object args[])`
  Invoke this method on all members in the broadcast group. Ignore any results or errors. The method will not return until all invocations are complete or abandoned due to error.
  Parameters:
  method - The method to invoke. This must be provided by each member of the group.

args - The arguments to pass to the method.

Returns:

an identifier for this invocation. This is used in callbacks to errListener.

- **addMember**

```
public synchronized void addMember(Object member)
```

Add a member to the broadcast group. If an invocation is in progress the member will not be added until the invocation is completed. This method may be called by an event handler.

Parameters:

member - The member itself.

- **addMember**

```
public synchronized void addMember(Object member, Class iface)
```

- **addMember**

```
public synchronized void addMember( Tagged member)
```

- **removeMember**

```
public void removeMember(Object member)
```

- **removeMember**

```
public void removeMember(Object member, Class iface)
```

- **removeMember**

```
public synchronized void removeMember( Tagged member)
```

Interface UK.co.ansa.flexinet.mobility.events.BroadcastService

An interface to a service used to broadcast method invocations (or events) to a number of clients. An error Listener will be informed of the status of the invocation. A particular implementation may inform the clients simultaneously or in an arbitrary order.

Methods

- **invoke**

```
public abstract int invoke( Tagged dest[],Method method, Object args[],
                            ErrorListener errListener)
```

Invoke a method on a number of interfaces, and inform errListener of the completion and any errors.

Parameters:

dest - An array of interfaces on which the method should be invoked.

method - the method to invoke. The return value of this method (if any) is discarded.

args - an array of arguments to pass as parameters to the method invocation.

errListener - a client to inform of the progress of the call (or null).

Interface UK.co.ansa.flexinet.mobility.events.ErrorListener

An interface to an object to be informed of the status of asynchronous invocations performed by Broadcast-Service.

Methods

- **invokeFailed**
  `public abstract void invokeFailed(int invokeID, Tagged client)`
  Callback to indicate that an invocation failed for an unspecified reason.
  Parameters:
  invokeID - The identifier for the invocation, returned from invoke()
  client - The client on whom the invocation failed.
- **invokeComplete**
  `public abstract void invokeComplete(int invokeID)`
  Callback to indicate that an invocation is complete. After invokeComplete() has been called, no more failures will be notified usung invokeFailed()
  Parameters:
  invokeID - The identifier for the invocation, returned from invoke()

Class UK.co.ansa.flexinet.mobility.events. SimpleBroadcastService

```
java.lang.Object
   |
   +----UK.co.ansa.flexinet.mobility.events.SimpleBroadcastService
```

A simple broadcast service. This may be part of a cluster, or an independent service.

Constructors

- **SimpleBroadcastService**
  `public SimpleBroadcastService()`

Exceptions

Class UK.co.ansa.flexinet.mobility.MOWException

extends FlexiNetException

A tag to allow matching of all MOW exceptions in one go. Note FlexiNet exceptions may also need to be caught.

Class UK.co.ansa.flexinet.mobility.NonFailureRestart

extends MOWException

The superclass of all exceptions passed to restart that do not related to failure conditions. When a cluster is restarted after a non-failure condition, the MOW does not automatically take an additional lock. There are currently two non-failure restart exceptions, `Moved` and `Copied` .

Class UK.co.ansa.flexinet.mobility.Copied

extends NonFailureRestart

Not actually an exception, but a status passed to **restart** to indicate that restart has been called after an object was created as a copy of some other object.

Class UK.co.ansa.flexinet.mobility.Moved

extends NonFailureRestart

Not actually an exception, but a status passed to **restart** to indicate that an object is restarting after a sucessful move.

Class UK.co.ansa.flexinet.mobility.MoveFailedException

extends MOWException

An exception generated to indicate that an attempted move has not succeeded. A failed move is guarenteed not to result in an active object at the destination place, althoug the destination host may be aware of the move attempt (and indeed may have actually blocked it). This exception may be subclassed to indicate more precise reasons for failure.

Class UK.co.ansa.flexinet.mobility.MoveOrCopyInProgress

extends MoveFailedException

Simultaneous moves and copies are not allowed.

Class UK.co.ansa.flexinet.mobility.UnMatchedLockException

extends FlexiNetRuntimeException

A lock was unlocked more times than it was locked. This is a runtime exception, and callees do not need to explicitly test for it.

References

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

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

3 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

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

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