ESPRIT Project No. 25 338

Mobile Object Workbench Internal Deliverable 1.0

Work package B

Mobile Object Workbench

MOW Design and Interface Specification DB3, DB4

ID:	DB3, DB4	Date:	30.12.97
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

Introduction

Background

This document contains the first release of the API for the mobile object workbench. An overview of the MOW purpose and function, together with a pre-release API may be found in the FollowMe Architecture document (DA1.1).

Current 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.

The methods detailed in this document, form the MOW API.

Structure

This document consists of an overview of the design of the MOW, followed by JavaDoc documentation for each of the primary classes.

Design

The Mobile Object Workbench supports a notion of object encapsulation. The unit of encapsulation is a cluster. Within a cluster, objects communicate by standard Java mechanisms, i.e. method invocation and field access. Between clusters, objects communicated via method invocation on exported interfaces. This notion of tightly coupled communications within a cluster, and looser communications between clusters allows us to manage clusters individually, and in particular, by de-coupling intra- and inter- cluster communication we can perform additional processing on inter-cluster calls, for example access control and redirection due to cluster migration.
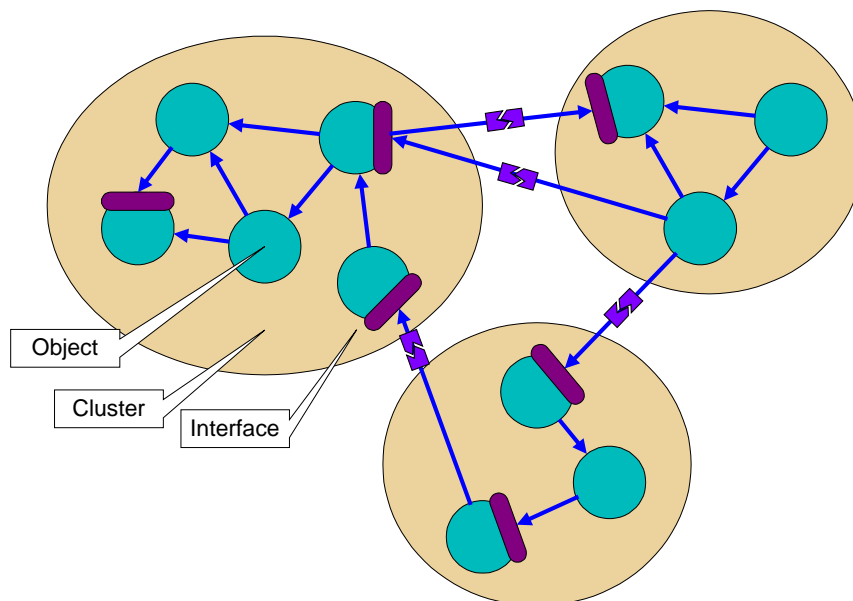

Figure 1 Intra- and Inter- cluster communication

Three key classes are provided to support this abstraction

Cluster

This is the top level class for encapsulated objects. All mobile or other managed clusters subclass this.

Mobile Object

A specialisation of a cluster that has the ability to move between places.

Place

An interface to a location at which a cluster may reside. The initial place implementation (PlaceImp) allows unrestricted movement of mobile objects between places, and unrestricted creation of new clusters. It is envisaged that the MOW security work will result in a place implementation which enforces restrictions according to some policy. In addition it is envisaged that the Autonomous Agents work package will sub-class Place or PlaceImp to provide a place with agent management functions.

The following sections give JavaDoc specification for these, and other, classes.

Approach to Encapsulation

Each cluster may be thought of as a 'virtual JVM'. It has its own top level thread group, and communication between clusters takes place by remote method invocation. The encapsulation provided by these two mechanisms prevents one cluster from adversely affecting another. The encapsulation is not complete, however, as one cluster may starve another of resources, and cluster may communicate using covert channels in the form of static methods on shared classes. Such communication is to be avoided, and in future implementations, clusters may be managed by different class loaders, or even different JVMs in order to prevent this crosstalk. Remote method invocation is performed using FlexiNet RMI. Indeed, the whole of the mobile object workbench is part of the larger FlexiNet architecture. There were many reasons for this choice, the primary one being the necessary flexibility in order to transparently located objects that have moved, and to enforce encapsulation for local (same JVM) calls.

An advantage in the approach to encapsulation chosen is to de-couple the notion of 'Place' from 'JVM'. This allows a place to be implemented by a set of co-operating JVMs (for example to enforce security), or several places to be implemented in the same JVM (although this is not possible in the initial implementation).
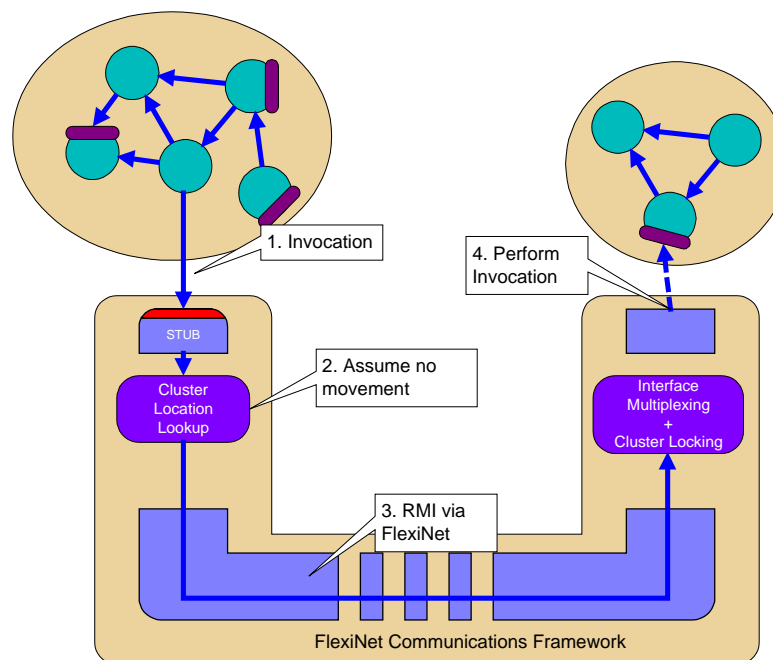

Figure 2 Implementation of inter-cluster calls

Approach to Mobility

In order to provide cluster mobility, there are three main MOW issues.

- It must be possible to serialize the state of the cluster and to recreate the cluster from a previously serialized state. This is exactly equivalent to the requirements for passing objects by value in an RMI system. Mobile clusters must

therefore be serializable by FlexiNet. Currently, FlexiNet can serialize objects providing that they have a constructor that takes no arguments, and that all of their fields are transient or public. The latter requirement will be removed in JDK 1.2. FlexiNet does not make use of `java.io.Serializable`.

- It must be possible to redirect access to interfaces exported from one instance of a cluster to a new instance of that same cluster. This redirection should be transparent. This equates to the movement of a callee being transparent to the caller.

- It must be possible to ensure that movement only occurs from a consistent manner. I.e. only one instance of the cluster is current at any time. This is managed by a combination of locks on the cluster, to prevent calls, and measures to ensure that a cluster may only move when there are no threads active within it. This is described in detail in the following section.
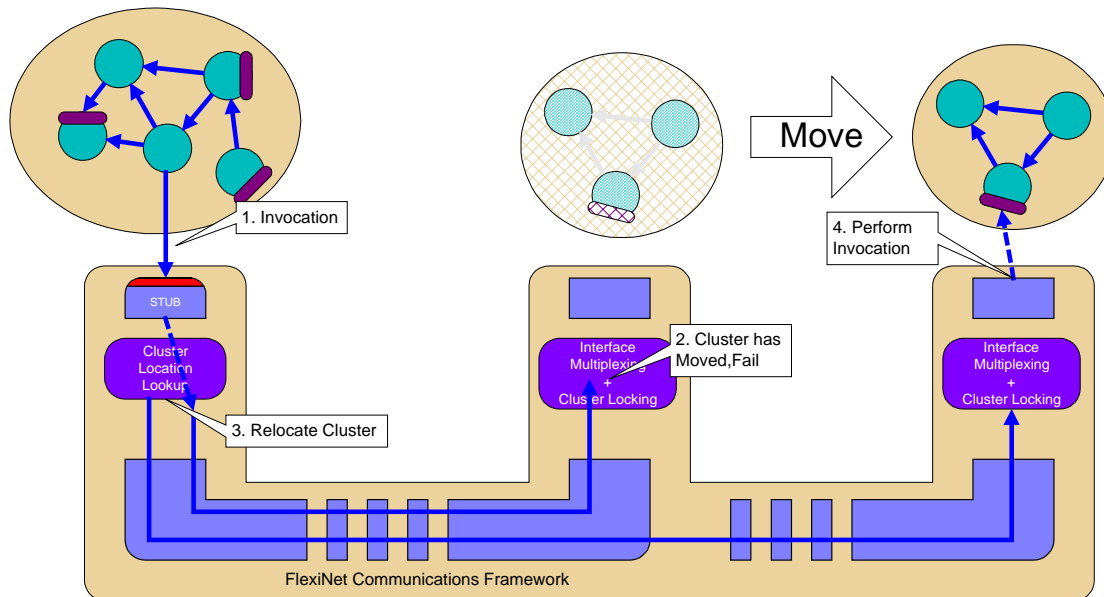

Figure 3 Back off and retry due to cluster movement

Ensuring Consistent Movement

Figure 4 illustrates the states that an instance of a mobile object may be in. The object is initially created in state A1. This state represents an *active* object that has *one* thread in it (the thread that calls `init(…)`). 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.

When a mobile object invokes the `pendMove` or `syncMove` operation, it will enter 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 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 error status is given as a parameter to the `restart` method.

The newly moved object is an exact replica of the original, and in addition it assumes the original's name (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 should obtain an additional lock prior to calling `pendMove` or `syncMove`.

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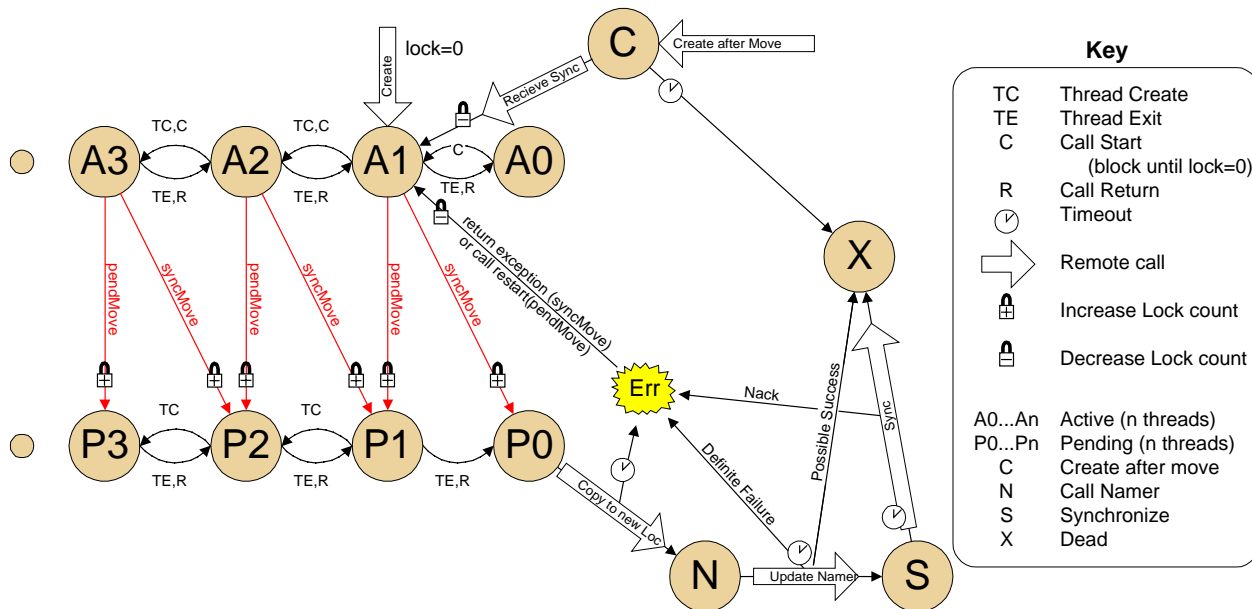

Figure 4 State transitions of a mobile object instance

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 MOW infrastructure. It should be noted that the callee is able to interrupt a thread making a call, but that this will not affect the caller. This is important to prevent the caller from blocking the callee's progress.
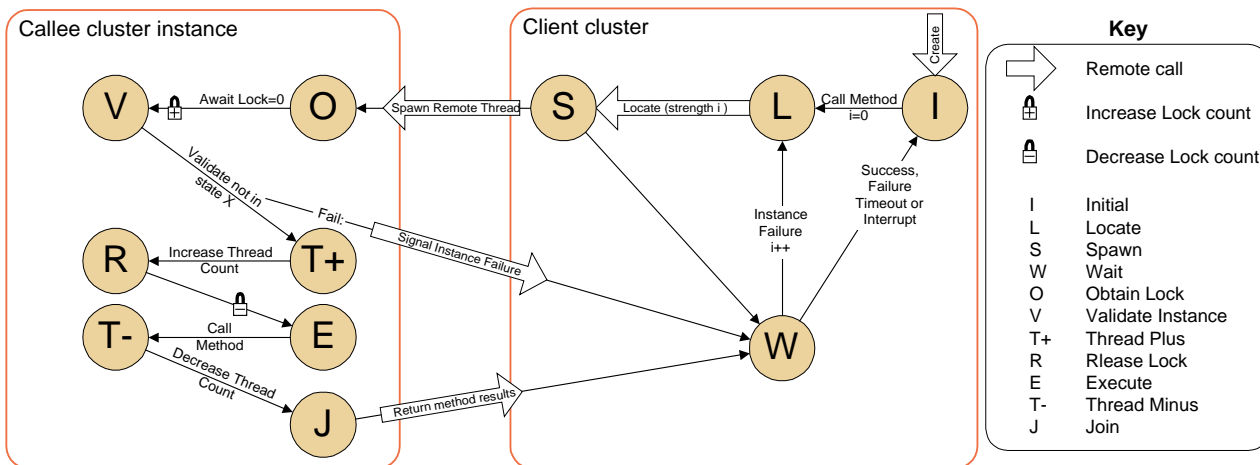

Figure 5 State transitions for an invocation on a remote cluster

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 clusters latest location, to speed future lookups. This simple scheme is used by the majority of existing mobile object 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 connectable 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 may *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 periodically refresh all references in order to keep the hop count low, and in order to reduce the reliance on hosts that a cluster has moved from.

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

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

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

The MOW has been designed to be 'relocation mechanism independent'. There is nothing in the API, or the majority of the code, to favour one implementation over another. The interface UK.co.ansa.flexinet.mobility.Mobile-Namer gives an interface to which relocation or naming services should correspond, and a simple originator based implementation is given in package UK.co.ansa.flexinet.mobility.namer.

In the FollowMe scenario, it is envisaged that there may be two kinds of host; relatively small numbers of reliable server hosts (previously called docks) that may manage many thousands of clusters over a period of time, and many thousands of untrusted, unreliable users owned hosts (eg browsers).

As part of ongoing MOW work, we are refining the design of a relocation service optimised for this environment. In particualr we are reluctant to use a Tombstone based approach (as it places an unrealistic reliability requirement on browsers), and even more reluctant to use one of the variants of Tombstoning that requires large scale co-operation between hosts in order to prune old tombstones (it requires unrealistic integrity and security). Instead we are looking at directory based approaches where only the directory need retain information about an object's location.

Unresolved API Issues

- The mechanisms by which a callee can ensure that a caller is local/has not moved have yet to be finalised.

- Services provided as part of the MOW work package (e.g. Event notification) have yet to be designed. They are not included in the API.

- Access to the AWT must be wrapped to enforce encapsulation. It is possible to use the AWT at present, but as the MOW is unaware of the AWT, it will not properly co-ordinate AWT event notification and mobility. This may lead to unexpected behaviour.

- The mechanisms for the retrieval of contextual information may change in light of work on the Information Space / Personal Profiles.

- The interface for the creator of a place to establish its security (or other) policy have not been defined.

- The API may need to be restricted (or have restricted implementation) in a browser compatible implementation.

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

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

public class **Cluster**

extends Object

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.

Constructors

🟡 **Cluster**

```
public Cluster()
```

> Standard no-args constructor. Allocates a name for debugging purposes. The cluster cannot be used until _init has been called.

Methods

🔴 **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.

🔴 **getPlace**

```
public Place getPlace()
```

> Return the place at which this cluster is currently residing.

● 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

● 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().

● 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.

● 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

● 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.

● 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 distinguish between failure related restarts and non-failure related restarts, non-failure related restart exceptions are subclasses of NonFailureRestart.

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.

init

```
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. 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
```

public abstract class **MobileObject**

extends Cluster

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

Constructors

🟡 **MobileObject**

```
public MobileObject()
```

Methods

🔴 **copyOrMovePending**

```
public synchronized 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.

Returns:
> true if a move or copy is pending.

🔴 **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 guaranteed 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.

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.

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.

Methods

🔴 **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

🔴 **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

● getProperty

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

Return contextual information. This is similar to the Java properties system, except that arbitrary 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.

Exceptions

UK.co.ansa.flexinet.mobility.MOWException

```
java.lang.Exception
    |
    +----UK.co.ansa.flexinet.FlexiNetException
            |
            +----UK.co.ansa.flexinet.mobility.MOWException
```

A tag to allow matching of all MOW exceptions in one go.

UK.co.ansa.flexinet.mobility.MoveFailedException

```
java.lang.Exception
    |
    +----UK.co.ansa.flexinet.FlexiNetException
            |
            +----UK.co.ansa.flexinet.mobility.MOWException
                    |
                    +----UK.co.ansa.flexinet.mobility.MoveFailedException
```

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

UK.co.ansa.flexinet.mobility.MoveOrCopyInProgress

```
java.lang.Exception
   |
  +----UK.co.ansa.flexinet.FlexiNetException
             |
            +----UK.co.ansa.flexinet.mobility.MOWException
                    |
                   +----UK.co.ansa.flexinet.mobility.MoveFailedException
                          |
                         +----UK.co.ansa.flexinet.mobility.MoveOrCopyInProgress
```

Simultaneous moves and copies are not allowed.

UK.co.ansa.flexinet.mobility.UnMatchedLockException

```
java.lang.Exception
    |
   +----java.lang.RuntimeException
             |
            +----UK.co.ansa.flexinet.FlexiNetRuntimeException
                    |
                   +----UK.co.ansa.flexinet.mobility.UnMatchedLockException
```

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

UK.co.ansa.flexinet.mobility.NonFailureRestart

```
java.lang.Exception
   |
   +----UK.co.ansa.flexinet.FlexiNetException
        |
       +----UK.co.ansa.flexinet.mobility.MOWException
              |
             +----UK.co.ansa.flexinet.mobility.NonFailureRestart
```

The superclass of all exceptions passed to restart that do not related to failure conditions There are currently two non-failure restart exceptions, Moved and Copied.

UK.co.ansa.flexinet.mobility.Moved

```
java.lang.Exception
    |
    +----UK.co.ansa.flexinet.FlexiNetException
              |
              +----UK.co.ansa.flexinet.mobility.MOWException
                        |
                        +----UK.co.ansa.flexinet.mobility.NonFailureRestart
                                  |
                                  +----UK.co.ansa.flexinet.mobility.Moved
```

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

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

```
java.lang.Exception
    |
    +----UK.co.ansa.flexinet.FlexiNetException
              |
              +----UK.co.ansa.flexinet.mobility.MOWException
                        |
                        +----UK.co.ansa.flexinet.mobility.NonFailureRestart
                                  |
                                  +----UK.co.ansa.flexinet.mobility.Copied
```

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.