A Mobile Object Workbench

Michael Bursell, Richard Hayton,
Douglas Donaldson, Andrew Herbert

APM Ltd
Poseidon House, Castle Park, Cambridge CB3 0RD United Kingdom
Email: apm@ansa.co.uk Fax: +44 1223 359779

Abstract

Existing mobile agent systems are often constructed with a focus on intelligence and autonomy issues. We have approached mobility from a different direction. The area of distributed systems research is quite mature, and has developed mechanisms for implementing a "sea of objects" abstraction. We have used this as our starting point, and added to this the ability for objects to move from host to host, whilst maintaining location-transparent references to each other. This provides a powerful and straightforward programming paradigm which embraces programming language semantics such as strong typing, method invocation and encapsulation, rather than rejecting them in favour of untyped messages. We have built a Mobile Object Workbench on top of a flexible Java middleware platform. The Mobile Object Workbench is being used to develop a mobile agent system, and to run application software. In this paper we examine the philosophy and design of the Mobile Object Workbench, and describe how this is being extended to provide a security framework oriented towards agents.

Corresponding author:
Michael Bursell
APM Ltd

Poseidon House
Castle Park
Cambridge CB3 0RD
UK
phone: +44 (0)1223 568919
fax: +44 (0)1223 359779
mike.bursell@ansa.co.uk

1. Introduction

Software agents, and in particular mobile agents, are currently an area of intense interest both within the research community and the commercial sector[1][2]. Current agent systems typically follow a message-based paradigm[3].

Distributed systems research takes an opposing view: building on a programming language's features such as method invocation and strong typing, it attempts to extend the language's capabilities transparently to support distributed applications. The most obvious example of this is in the area of remote procedure (or method) invocation. The abstraction this extension provides - the ability to invoke procedures on remote object - is often referred to as creating a "sea of objects". In such a "sea", objects may reference and invoke each other regardless of their relative locations. The most common examples of this are CORBA[4] and DCOM[5]. The overall architecture of this style is RM-ODP, which shows how to engineer in "abilities" such as *interfaces* on objects[6].

We believe that in order to develop mobile agents, it is first logical to extend the "sea of objects" abstraction to include the possibility that objects may move from host to host. This movement should be transparent, in that references and invocations made on the object which has moved should continue to function, without the application programmer having to take special steps to maintain their validity. Such a "Mobile Object Workbench" provides a generally useful distributed computing paradigm, one obvious application of which is to underpin mobile agents.

The Mobile Object Workbench we describe, and have implemented, is not an agent system, but, as in the ESPRIT[15] project within which it has been developed, will act as the basis for the implementation of an autonomous mobile agent system by other researchers.

2. Principles

In this section, we describe the basic principles which we have adopted in the design and implementation of the Mobile Object Workbench. Throughout, we have striven to extend, rather than replace our base language, Java, at both the language level and the distributed systems level. This principle, we believe, makes for ease of use (as we building on abstractions which are already familiar).

Where possible, we have also aimed for "selective transparency", particularly of engineering mechanism, so that application programmers need not concern themselves with the details of the implementation or design in order to use the system, although they retain the ability to set policy and respond to errors.

2.1. Encapsulation

Language level objects are typically too small to be a useful unit for mobility. It would not generally be useful to provide mobility for a simple string. A mobile agent is more likely to consist of several language level objects, with a single object as its 'root'. It is neither helpful or useful to move the constituent objects individually, and instead we need a grouping mechanism or policy 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) and may have to perform security checks, such as access control and auditing. Even when two clusters are located on the same host, they still communicate through the encapsulation boundary via system-provided mechanisms, although the bottom "transport" can be via local memory rather than the network. This allows clusters to protect themselves from each other, and gives them some degree of autonomy.

2.2. Interfaces

Given encapsulation, there must be some way for objects to communicate across encapsulation boundaries. We restrict references across encapsulation boundaries to be *interface* rather than object references, following the ODP model of the interface as the point of access to an object, and thus allowing the implementation of objects in different clusters to evolve independently. Binding at interfaces has the key property of preventing sharing of state between clusters. We arrange that when a reference to an interface is passed across an encapsulation boundary, then it is passed *by reference*. However, when a reference to an object is passed across the boundary, then the object is copied.

These semantics are different from both Java RMI and CORBA. Java RMI passes objects by reference unless they inherit from a particular class (java.rmi.Remote). CORBA currently lacks the ability to copy objects transparently, and passes interfaces by reference. Using our semantics we see identical treatment of both the local and remote case (apart from the possibility of communications exceptions). We anticipate that as ongoing work in the OMG on a Java to CORBA binding and support for "Objects by value" converges, a solution akin to ours will be selected.

No methods on an object are available for use by objects outside the encapsulation unless they are exported as an argument or result. For example, a cluster may export interfaces and these interfaces can be imported by other clusters. No other methods are available to any object outside the encapsulation boundary.

2.3. Location Transparent Communications

In order to maintain transparency of use of interfaces, some mechanism must be provided to allow communications between clusters which are remote from one another. Like other distributed object systems, the Mobile Object Workbench makes use of *stubs* to represent remote interfaces (i.e. interfaces from outside the cluster). This means that there is no special API for communications in the Mobile Object Workbench; communications is as close to the pure language semantics as possible, and transparency is to a large extent preserved. The MOW API is entirely related to the life cycle and movement of clusters.

Unlike RMI or CORBA we do not have an off-line stub generator. Instead we generate stubs on demand. This has the important benefit that since stubs are locally generated they are trusted, rather than potentially hostile, code.

It has long been a point of contention in distributed systems research as to whether true location transparency is achievable, or indeed desirable. In FlexiNet, the ORB upon which the MOW is built, the approach taken is for *selective transparency*. Ordinarily, remote communication is transparent. However, the application or middleware programmer can link in "binding objects" which describe special action to be taken; for example an application programmer may provide code for ruling on access control policy, or for explicitly choosing a communications protocol. These

binding objects are invoked when interface references are initially bound, rebound after Mobile Object migration, or when failures occur.

2.4. Autonomy of Movement

We have stated that a basic facility is for clusters to be able to be moved from one host to another. An important issue is who is at liberty to decide when a cluster should move. We note that a cluster cannot move at an arbitrary point in time - it may have to release resources cleanly, and it is therefore not reasonable to command a cluster's movement externally - rather, the process should be within the cluster. This does not, of course, preclude a *request* for movement being made from an external entity.
As agents may be malicious or erroneous, it is essential that a cluster can be *destroyed* by an external signal, and this is provided. Thus our Mobile Objects are autonomous, as might be expected to meet the needs of agent-based applications.

2.5. 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[7].
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.

3. Related Work

Several agent systems have provided script-based languages, rather than utilising a systems programming language (e.g. Telescript - Odyssey [8]). We believe this is harder for application programmers to use, and requires considerably more

infrastructure to allow the interworking of agents, with other non-agent technologies, such as JNDI [9], JDBC[10], CORBA[4], etc..
Other agent systems are built as extensions of an existing language, but with new abstractions for inter-agent communications. For example, Aglets[3] allows agents to be built using Java, but they must communicate using untyped messages.
The nearest to our approach is Voyager [11],which deals, however, with individual objects rather than clusters. We believe our approach gives more scope for the management and securing of mobile entities, as intra-cluster and inter-cluster communications are differentiated.
The OMG's Mobile Agent Facility[12] is an important piece of standardisation work being done in this area. Whilst we see the MOW as providing basic facilities to be extended by an agent implementation, in practice we provide the majority of the MAF facilities, and extension of the Mobile Object Workbench to provide a MAF-compliant platform is one possible implementation path in the longer term of the project.

4. Design

4.1. Encapsulation

In the Mobile Object Workbench, both threads and objects are encapsulated. The mechanisms are illustrated in *Figure 1 - Clusters, showing internal and external communications*, which shows four clusters. One of the communications bindings between clusters is expanded to show that invocation of an external interface reference is achieved using a proxy (stub) with a communications stack.

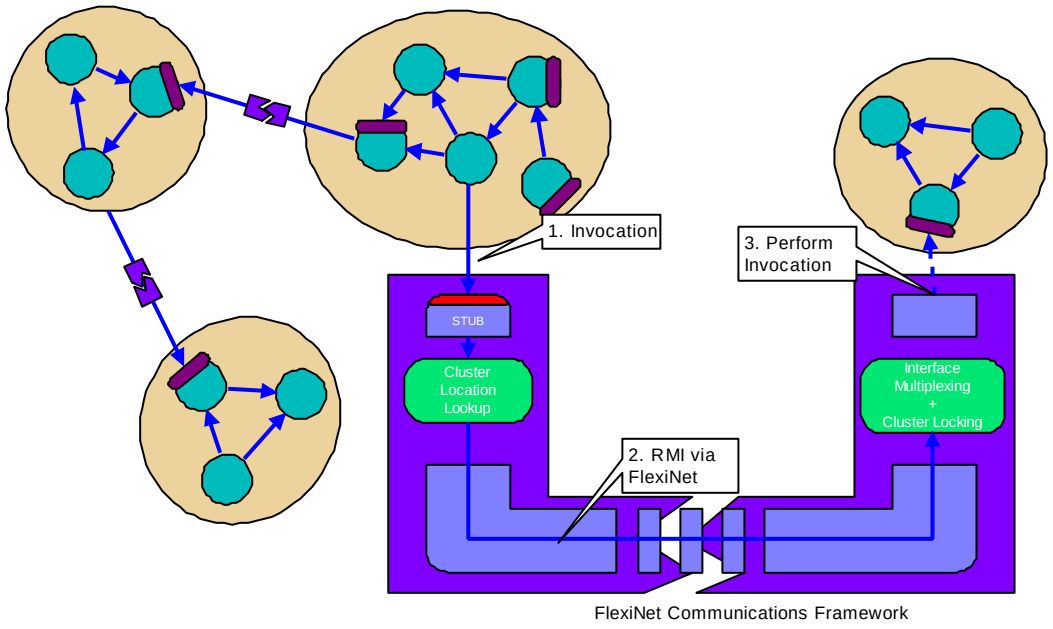

Figure 1 - Clusters, showing internal and external communications

In order to encapsulate clusters, we must distinguish between two types of references:
i) references between objects within the same cluster
ii) references between objects in different clusters.

As objects within a cluster share the same privileges, and are always collocated, these references can be ordinary language level references. References that cross the encapsulation boundary appear to the programmer to be the same, but in fact are implemented via interface proxies and (potentially) remote communication. Each cluster is created with a reflexive communications stack, and all references to external clusters or services are bound to the communications stack. We ensure that any references passed or returned in method invocations using proxied interfaces are also treated in a similar way. This ensures that a cluster remains encapsulated. Any objects created within a cluster become part of the same cluster as the object performing the creation, and a special mechanism is used to create new clusters. Objects and data may be passed in method invocations over the encapsulation boundary, and are passed by copying.

As clusters 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.

4.2. Location transparent communications

All references to interfaces within the Mobile Object Workbench are location transparent by default, that is, the application/agent performs the same action, whether a reference is to a local or remote object, or whether it is to an object that is currently in transit. A call to a remote object may result in an exception if the object is unreachable within a pre-set interval (or if the access is disallowed after access controls). These exceptions may be caught, or may be ignored and allowed to propagate through the client code.

When an interface reference is returned from a remote call, a local stub is transparently constructed on-the-fly from the interface definition using Java introspection. This contains references to a communication stack, and the name of the remote interface. When a call is made on the interface, we first attempt to locate the remote interface using the last known location. If the remote interface happens to be on a mobile object that has moved, the remote host will raise an exception. This is caught within the infrastructure on the client machine, and a secondary mechanism is then used to relocate the cluster. We are constructing a robust directory-based distributed location service to perform this task, though the architecture is flexible, and other approaches could be employed. This is an advantage over other systems which 'hard-wire' protocols such as forwarding tombstones, which are inappropriate for highly mobile long-lived objects.

4.3. Movement

The encapsulation provided by clusters allows a straightforward implementation of mobility, since both objects and threads are encapsulated. The basic process for moving a cluster is creating a snapshot of a cluster, copying it, restarting the new snapshot at the new place and discarding the old. The real issues are ensuring that a *consistent state* is moved, and providing robustness if a host or network fails during movement.

A Mobile Object is defined as a cluster with mobility capability, and a simplified view of migration process is as follows:

1) Mobile Object "decides" to move

In order to allow the move to proceed, we must guide the cluster towards a consistent state. To do this we block any new method calls being made on this cluster from outside its encapsulation boundary. Method calls that are already proceeding are allowed to complete, and it is the responsibility of the mobile cluster to ensure that any threads internal to the cluster are terminated, or allowed to run their course.

2) The infrastructure monitors the Mobile Object until there is no more thread activity within it

The encapsulation and thread management mechanisms provide introspection for this. When the activity has stopped, the object is in a consistent state.

3) The Mobile Object is copied to the destination host, and the relocation service is informed of the new location

This is performed in a number of stages to allow rollback or rollfoward if a failure is detected. A critical failure can result in an object being discarded by both the source and destination host. These semantics were preferred to the possibility that both hosts might restart the Mobile Object in a network partition, which would be likely to cause inconsistency in the system and applications based on it.

4) A synchronisation call completes the process

A restart method is called on the newly copied cluster, and the old cluster is discarded. The restart method may recreate transient resources, start internal threads, or perform any other processing. Any previously blocked method invocations are then allowed to proceed, and will throw *object moved* exceptions which will then lead to a rebind to the new host.

4.4. Autonomy

As a cluster has access to objects within it, in order to provide autonomy we must add sufficient introspection for a cluster to be able to determine when it has no active threads, and sufficient access rights to allow a cluster to create a remote copy of itself, and to update the relocation service.

As security was a primary concern of the MOW, the internal naming scheme for clusters is designed in such a way that a malicious program cannot spoof the relocation service, such that another entity might believe that a cluster had moved when it had not, or believe that it had moved to the wrong location. To prevent this form of denial of service attack, we arrange that cluster names include the current host name (to prevent malicious claims of ownership), and arrange that only the current host may inform the relocation service of a cluster migration (to prevent spoofing). We also use authentication of hosts, to prevent one host acting as another.

The standard encapsulation mechanism is all that is required to enforce the autonomy of movement. If a cluster never exports the interface containing the *move* method, then this is not accessible outside its encapsulation boundary. Of course a malicious host can always circumvent this (or any other) mechanism, which is why we employ

cryptographic security measures so that we can detect if a cluster has been tampered with, or if a malicious host is attempting to impersonate a trusted place.

4.5. Security

Host integrity and access control to host resources are managed by encapsulation, and extending Java security manager abstractions[13]. Secure communication is being addressed by using SSL[14] for authentication, confidentiality and integrity of communications.

Cluster integrity, confidentiality and authority are the unique issues to be addressed in the context of mobile agents, particularly in open systems like the Internet. We are designing a mechanism for cryptographic sealing of parts of a cluster, and declarative policy specification of which hosts may examine and modify which parts of a cluster, and the cross dependencies between the data items.

5. Implementation

5.1. Introduction

The Mobile Object Workbench is being built within the context of the FollowMe European ESPRIT project (no. 25,338), which commenced in October 1997[15].The Mobile Object Workbench is being constructed as an extension of a Java middleware platform called FlexiNet, which was developed at APM during the last eleven months[16].

5.2. Mobile Objects and Agents

As previously stated, agents are a particular specialisation of Mobile Objects.

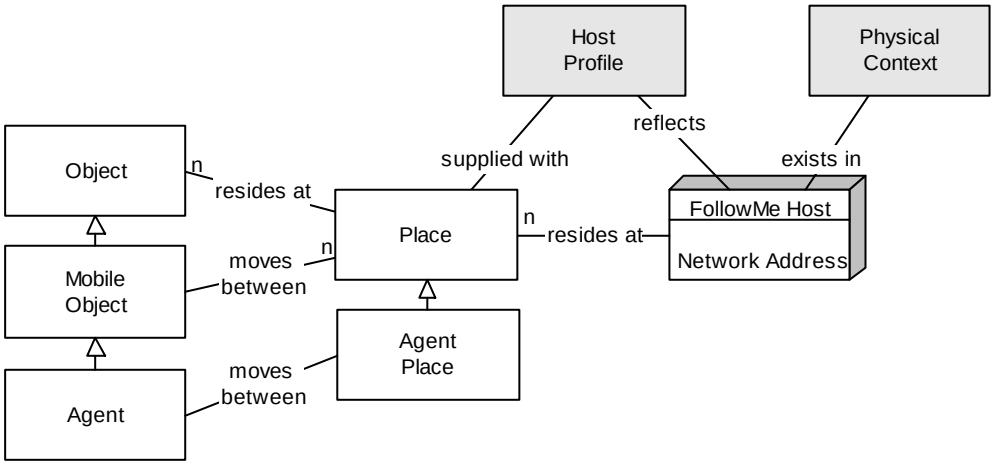

Figure 2 Places and Hosts

Figure 2 Places and Hosts shows the relationships between Mobile Objects (i.e. Clusters), Agents and Places. A Place is a logical execution environment for objects. Located objects reside at one Place. A Mobile Object can change its residence from one Place to another (Figure 2). A Place resides at a host for its lifetime. It is useful to consider a Place to be a distinguished object created within a host process (e.g. a virtual machine process which is part of a FollowMe system). The lifetime of the Place would then be the lifetime of the process. Places do not move between hosts; however hosts may be mobile, and a Place on a host which moves has support for dynamic change of the host's address.

5.3. Implementation of Clusters

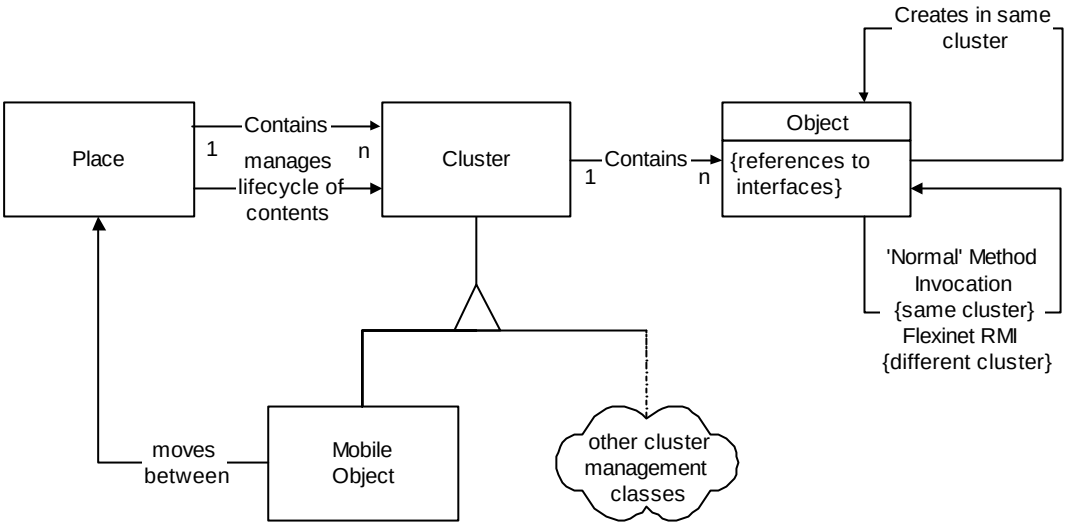

Figure 3 Objects, Clusters and Mobile Objects

Figure 3 Objects, Clusters and Mobile Objects 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. 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.

An object is the basic building block out of which applications or agents may be built. Objects may contain references to interfaces on other objects anywhere in a FollowMe 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 language protection means and takes place using standard 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. The method invocation semantics are those of FlexiNet RPC (see *Section 5.4 - Communications and Encapsulation).*

5.4. Communications and Encapsulation

The process by which we provide location-transparency for intra-Cluster communications revolves around three points: Cluster encapsulation; dynamic stub generation; and the infra-structure level name directory, which provides location-transparent naming. The encapsulation and stub generation are discussed in this section, and the naming service in *Section 5.5, Location-transparent naming service.*

Communications are all by FlexiNet RPC, and all communications between Clusters are handled by a FlexiNet communications stack. The Place on which a Cluster is created provides a reference to a default trading service, and the Place may also provide references to itself and/or interfaces on other Clusters. These references are bound to the FlexiNet communications stack for the Cluster. Via these interfaces or the trading service, the Clusters can gain references to interfaces on other Clusters. These interfaces will also be bound to the Cluster's communications stack.

When a Cluster wishes to make a method call on a remote interface (e.g. the trading service), the communications stack examines the Java class to which the interface belongs, using introspection provided by Java core reflection, and dynamically creates a stub out of Java bytecode. This stub, when invoked, marshals the method call according to the FlexiNet RPC protocol and passes it to the communications stack to fulfil the method invocation on the interface on the remote Cluster. If any such call passes an interface as an argument or result, the stub binds the interface reference to the relevant communications stack, so that if the reference is invoked, a stub can be created to manage the remote communications.

5.5. Location-transparent naming service

That the communications stack is able to find the Cluster on which the call should be made is due to the location-transparent directory-based naming service. There are four key points to our naming service:

i) we control what entities are able to update the directories - only Places from which a Mobile Object is moving may update the record for the Mobile Object. 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" Clusters.

ii) we provide a hierarchy of directories, for scale and robustness. This means that a naming service may decide to copy the naming record for a Mobile Object (or Cluster) up the hierarchy to increase its stability.

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

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

One possible scenario for using the naming service is shown in *Figure 4*, *Figure 5* and *Figure 6*. In *Figure 4*, a naming hierarchy is shown, with Places (large, light-coloured boxes), naming services (smaller, dark boxes), a naming record (a heart-shape) and a Mobile Object (a circular object). In this figure, the Mobile Object moves from its original Place to another.

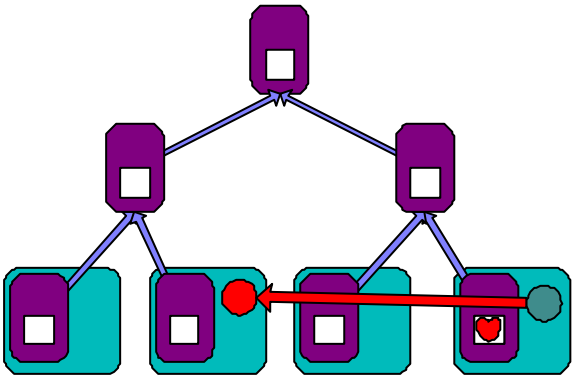

Figure 4 - Naming service: Mobile Object moves

In *Figure 5*, the naming record for the Mobile Object is moved to another naming service. In this case, the naming service is a parent to the naming service at the Place to which the Mobile Object has moved. This might be because the Mobile Object is expected to move between a number of branches in this hierarchy, rather than staying at the new Place. A link is provided at the naming service of the old Place, to allow

the Mobile Object 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 Object itself, but to the directory. This means that we do not rely on tomb-stoning from each Place, but point instead to naming services which themselves provide pointers.

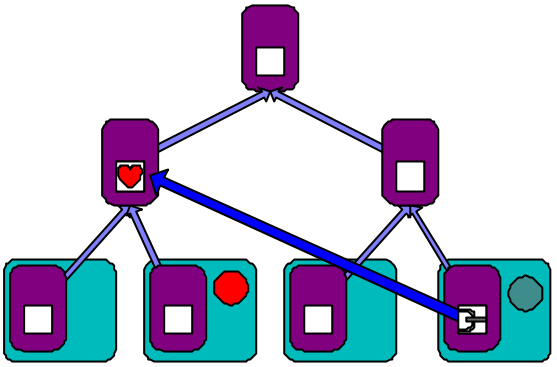

Figure 5 - Naming service updated

In *Figure 6*, the link from the original Place'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 Mobile Object 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.

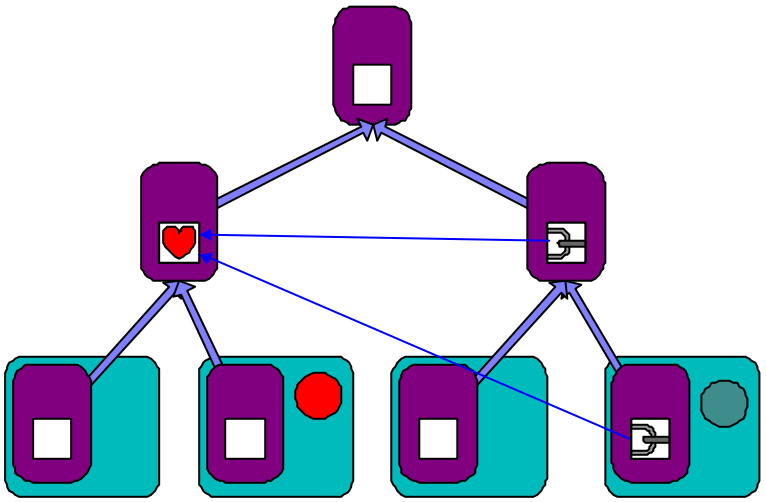

Figure 6 - Naming service - caching and copying

5.6. Mobile Object Workbench API

The constructor for a MobileObject is for use by the infrastructure, and any instance-specific initiation should be done within an "init" method. The "restart" method is called on completion of a move to a Place. The Tagged type represents a most general interface reference type (whereas the Java Object type represents the most general object or interface reference type).

```
public class MobileObject extends Cluster
{
 void pendMove(Place dest) throws MoveFailedException;
```

```
 void syncMove(Place dest) throws MoveFailedException;
 Object copy(Place dest)   throws MoveFailedException;
 Object init(...) throws InstantiationException;
 abstract void restart(Exception reason);
}

public interface Place
{
 public Tagged newCluster(Class cls,Object[] init_args)
    throws InstantiationException;
 public Object getProperty(String propertyname);
}
```

6. Status

The FollowMe project, and the design of the MOW, started in October 1997. We now have a fully functioning mobile object system written using 100% pure Java. Currently we are concentrating our efforts in this direction, and have prototype secure SSL communications[14] and a design to support integrity and confidentiality of mobile data and code.

Other ongoing work includes support for class evolution, again leveraging the cluster abstraction to allow different mobile objects to use different version of classes, despite being located in the same host.

The MOW has been released to project members and the ANSA consortium[17] and various demonstration applications have been written.

7. Summary

The Mobile Object Workbench has shown how to add mobility to an existing object language. The key principle has been strong encapsulation of both state and threads, which has meant that the addition of mobility has not been too difficult. With our simple computational model of passing all interfaces by reference and all objects by copy, we have provided transparency of remote communications, and by providing a robust directory-based location service, we have ensured location-transparency for communications as well.

The two other important principles of the system are the integration of security into the design framework from the outset, and local dynamic stub generation for interfaces. This has provided us with great flexibility for binding policies and allowed security to be explicitly or implicitly leveraged by the infrastructure and applications using it.

---

1 “MA 97” *First International Workshop on Mobile Agents 97*, Berlin, Germany, April 7 - 8, 1997. http://www.informatik.uni-stuttgart.de/ipvr/vs/ws/ma97/ma97.html

2 “Agent Product and Research Activities” *The Agent Society*. http://www.agent.org/pub/activity.html

3 Danny B. Lange and Daniel T. Chang. IBM Aglets Workbench - Programming Mobile Agents in Java, A White Paper (Draft)” *IBM Corp.*. Sept. 1996. http://www.trl.ibm.co.jp/aglets/

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

5 Nat Brown, Charlie Kindel. "Distributed Component Object Model Protocol -- DCOM/1.0 - Network Working Group INTERNET-DRAFT", *Microsoft Corporation*, Jan. 1998. http://www.microsoft.com/oledev/olecom/draft-brown-dcom-v1-spec-02.txt
6 "Open Distributed Processing - Reference Model", *International Standards Organisation*, Sep. 1995. http://www.iso.ch:8000/RM-ODP/
7 Dan S. Wallach, Dirk Balfanz, Drew Dean, Edward W. Felten. "Extensible Security Architectures for Java", in *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles 31,5*, Dec. 1997, pp. 116-128.
8 "Agent Technology" *Odyssey*, General Magic, Inc.. http://www.genmagic.com/agents/
9 "Java Naming and Directory Service (JNDI)", *Sun Microsystems*.
http://www.javasoft.com/products/jndi/index.html
10 "The JDBC database access API", *Sun Microsystems*.
http://www.javasoft.com/products/jdbc/
11 "ObjectSpace Voyager Core Technology", *ObjectSpace*.
http://www.objectspace.com/Voyager/
12 "Mobile Agent Facility (Draft)" *Crystaliz, GMD FOKUS, General Magic, IBM, The Open Group*. http://www.genmagic.com/agents/MAF/
13 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
14 "The SSL Protocol", *Netscape Inc.* http://home.netscape.com/newsr ef/std/SSL.html
15 "FollowMe project overview", *FAST e.V.*
http://hyperwav.fast.de/generalprojectinformation
16 "FlexiNet - Automating application deployment and evolution", *APM Ltd.*
http://www.ansa.co.uk/Research/Flexinet.htm
17 "The ANSA consortium", *APM Ltd.* http://www.ansa.co.uk/Research/