Mobile Java Objects

R.J.Hayton, M.H.Bursell, D.I.Donaldson, W.Harwood, A.J.Herbert
Citrix Systems (Cambridge)
Poseidon House, Castle Park,
Cambridge CB4 3JB
Tel: +44 1223 713111 Fax: +44 1223 359779
Email: apm@ansa.co.uk

Abstract

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

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

Keywords

Java, mobile objects, agents, security, middleware, binding

1 INTRODUCTION

Java is an ideal language for developing distributed applications. It provides both object and interface abstractions, which gives a useful distinction between an object's interface and implementation. It also provides language level introspection, and allows for dynamic code creation. These features make it particularly suited to the design of middleware systems.

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

When we turned our attention to mobility, we took the view that this too should be a natural extension of the Java language. We designed a system in which we maintained the FlexiNet view of a "sea of objects" but which allowed objects to move between hosts. Clients of mobile objects need not be aware if a service object is mobile, nor if it has actually moved; they 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.

2 RELATED WORK

There are an ever-increasing number of "agent" systems available, either commercially or as academic projects. Although some provide high level abstractions that might be considered as support for intelligent behaviour, the majority of these systems are basically mobile code platforms. They allow applet-like pieces of code to move through a distributed system, and allow varying degrees of communication between different agents and between agents and hosts or other resources. Examples of these systems include Telescript (GenMagic 1995) and Aglets (Lange 1996).

In general, communications in these systems takes place using messages sent between the various parties. We believe that the advantages of object oriented computing, and in particular distributed object computing, are well accepted, and it is equally advantageous to build mobile systems using objects. Until the advent of the Java programming language, it was difficult to create "deployable" objects so most mobile code systems relied on scripting technology. It is only recently that the possibility of combining mobility and object orientation has emerged.

Existing Java systems that claim some degree of object mobility, do so either by abandoning Java' s method invocation abstraction (and taking the message passing route) or by utilising Sun' s remote method invocation (Sun 1996). Unfortunately when designing RMI, Sun dictated that each object must be explicitly tagged as a server object or as a data object. Only data objects may be passed by value on a remote invocation. This precludes the possibility of moving server objects, and is a showstopper for most mobile object systems.

Voyager (ObjectSpace 1997) is an exception to this. This Java middleware platform supports mobile services by wrapping them with pre-processor generated classes. However, objects are moved individually and there is no notion of clustering. Although Voyager is closest system to ours, we believe we have a more flexible and scaleable approach. In FlexiNet, an introspection based serialization engine is used, which does not have the limitations of RMI' s serializer, and is more portable as it is written using 100% Java.

FlexiNet embraces ODP principles (ODP 1995), and by adding mobile objects to FlexiNet we are providing the ODP relocation and migration transparencies that most other middleware platforms lack.

3 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.

Interfaces on remote objects 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 between a local proxy object and the remote object it represents. Different binders embody different application requirements or engineering strategies such as choice of protocols and the imposition of security policies or atomicity. 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. 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.

3.1 Generic Communications

Rather than using stubs to convert an invocation directly into a byte array representation, we leverage Java' s runtime typing support to represent the invocation in a generic (but fully typed) form. This is a reflexive technique central to the design of FlexiNet. 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 and semantics of the invocation using the full Java language typing support. Third part meta-objects can be fully general and are fully type-safe. When designing the Mobile Object Workbench, we decided to implement it as a set of FlexiNet binders, binding protocols and services. In particular the mechanism for communication with a mobile object is essentially communication with a static object together with two reflective layers: a server side layer, which raises an exception if an object has moved, and a client side layer, which relocates and rebinds to the object.

4 REQUIREMENTS FOR MOBILE OBJECTS

4.1 Unbinding

The key feature of a mobile object is that it must be able to move. When we move an object, we effectively copy it to a new location, and then arrange that all references to the old object are replaced with references to the new object. This requires a mechanism for *unbinding* a previous binding to an object. If objects were referenced directly using language level pointers, unbinding would require changes to the implementation of the Java virtual machine, reducing the advantages of Java as a portable language. Instead, we arrange that mobile objects (or more correctly interfaces on mobile objects) are referred to indirectly, using FlexiNet stubs as proxies. This level of indirection allows us to rebind references dynamically whenever objects move.

In addition, to avoid the need to track distributed references, we only perform this rebinding when a reference to a moved object is first de-referenced. The use of stubs and the rebind is transparent to the application programmer, although they may reflect the process, for example to deal with errors.

4.2 Consistency and Threads

At any point in time, an object may be *active* or *passive*. An active object is one that has a thread of control currently executing in it, or passing through it. A passive object is one that is not currently being executed, and hence has no threads active in it. When we move an object, we must ensure that the move is *atomic*. To do this the object must be *passivated* - i.e. all processing of the object must be suspended. One approach to this would be to pause any threads running in an object, move the object, and then restart the object and the threads at the new location. This, unfortunately, is impractical, as Java does not allow us to determine the complete state of an active thread at an arbitrary point of execution.

We have chosen to *encapsulate* the object and enforce a locking strategy that 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.

4.3 Grouping

In the discussion so far, we have been describing the migration of single Java objects. However, there is little utility in moving a single Java object, as they are typically very small (for example, a linked list will consist of many objects). 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. 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 language references, as no special action needs to be taken when they are de-referenced.

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

4.4 Failure Modes

When designing distributed systems, there is always the possibility of host or network failure. In particular network partition can result in hosts incorrectly assuming that other hosts have failed. When designing a mobile object system, an important 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. Long term suspension requires transaction-like logging and recovery.

This will be considered once transactional mechanisms have been integrated into FlexiNet. Section 6.4 gives a fuller description of the state transitions required to ensure consistent movement.

4.5 Scalability

There are two issues relating to the scalability of a mobile object system. Firstly, some design choices would require the registration either of all objects, or worse, of all references to all objects. We rejected these approaches as we wish to use the mobile object workbench in an Internet environment, which is both open and has no central administration. The second issue relates to the rebinding to interfaces on an object once it has moved. We would like this to be possible, even if the original host has since failed. This issue is discussed in detail in section 8.

5 REQUIREMENTS FOR MOBILE AGENTS

The mobile object workbench is not a mobile agent system, however it was developed as part of an ESPRIT agent project called FollowMe (FAST 1997), and one of the other partners is developing an agent system on top of it. As mobile agents are an obvious application of mobile objects, it is worthwhile to consider their specific requirements.

5.1 Autonomy

Mobile agents are generally considered to be ‘autonomous’. That is to say, that it is the agent itself that determines the actions it takes, and in particular controls its movement. In terms of mobile objects and clusters, this gives a requirement for cluster mobility to be initiated only by 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 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 itself from malicious or erroneous agents.

5.2 Security

We have found that the approach of designing and engineering from the point of view of a mobile object system has allowed us build on established security principles. In particular 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 clusters.
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** - a host 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 communication 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. We have therefore adopted the principle of including security issues from the outset, rather than as an "add-on", bolted on at a later date. Section 7 discusses our approach to these issues.

5.3 Thread Encapsulation

As clusters representing agents are mutually distrustful 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, by spawning and rendezvous, so that the failure of the caller and callee are independent. This thread de-coupling is integrated with the binding system and is transparent to the application programmer.

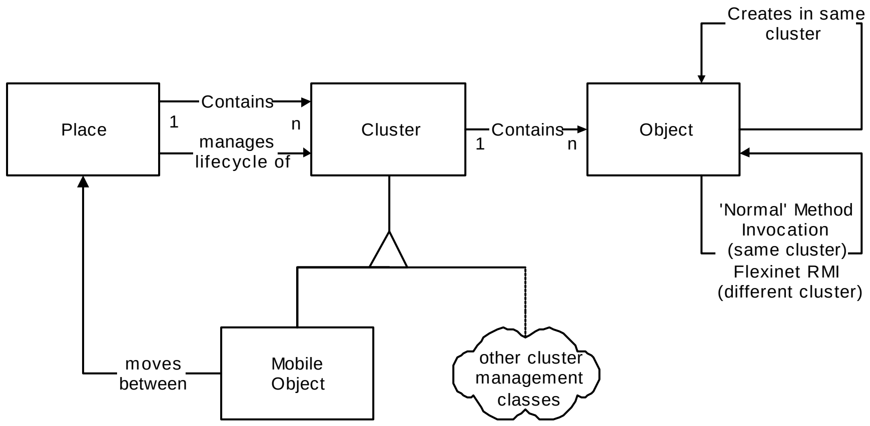


Figure 1 Objects, Clusters and Mobile Objects.

6 THE MOBILE OBJECT WORKBENCH

6.1 Clusters

Figure 1 shows the relationship between (Java) *Objects*, *Clusters* and *Mobile Objects*. A *Cluster* is a Java object containing a grouping of objects that are managed together. A *Mobile Object* is a specialisation of this that 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 view.

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 an interface passed between the clusters.

As communication between mobile (and non mobile) objects takes place using application-level exported interfaces, the Mobile Object Workbench API is entirely concerned with the lifecycle of mobile objects, and is analogous to the Applet or Bean API. The main classes and methods are illustrated in Figure 2.

public class Cluster

```
public synchronized void lock()
public synchronized void unlock()


public void init()

                                init(...)

public void stop()


public void restart(Exception e)
```

public class MobileObject extends Cluster

```
public void syncMove(Place dest)
```

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.

public interface Place

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

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

Figure 2 Mobile Object Workbench API.

6.2 Binding architecture

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

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

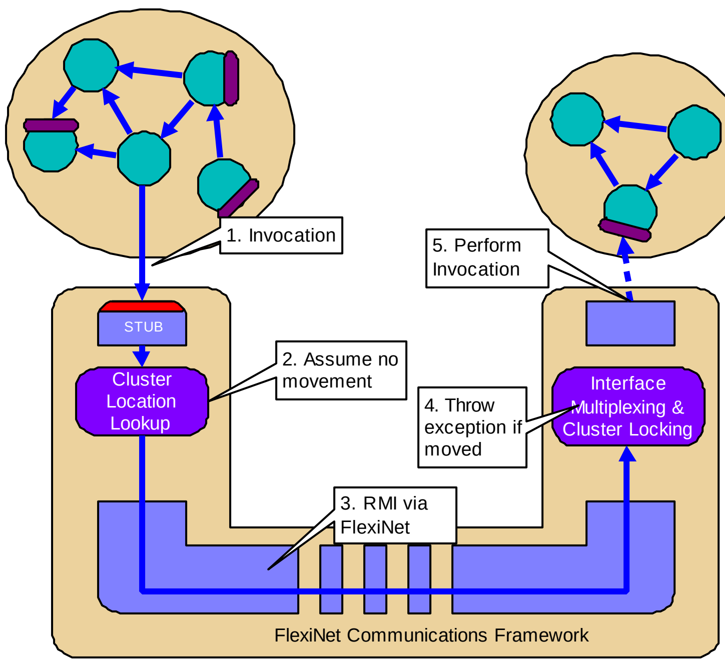


Figure 3 Implementation of inter-cluster calls.

6.3 Orchestrating Mobility

Figure 4 illustrates the states that an instance of a mobile object may be in. The object is initially created in state A_1. 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 difference being whether the calling thread returns immediately (`pendMove`) or 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 P_0). 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.

When a mobile object enters the state P_0 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 A_1. 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 cluster 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 one (effectively the original cluster has moved). It is started in state A_1 by a call to `restart`. The new cluster (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 a cluster wishes to restart in a locked state, then it may obtain an additional lock prior to calling `pendMove` or `syncMove`. This allows newly moved clusters to perform start-up cleanly, before allowing external access.

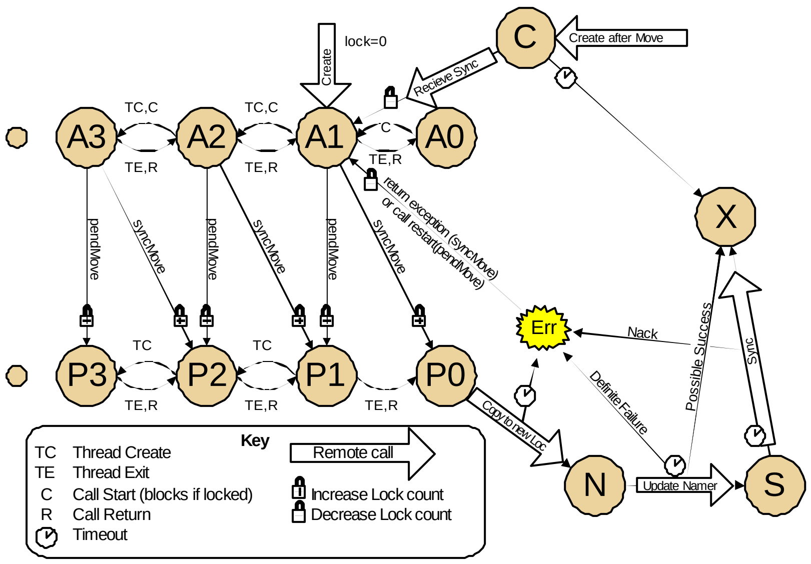


Figure 4 State transitions of a mobile object instance.

Copying, rather than moving, a cluster follows exactly the same procedure as `syncMove`. The `copy` operation blocks until there are no other threads and the new cluster has been created, or a failure is detected. After successful synchronisation, or failure, the original object enters state A_1 and the copy operation terminates. The newly created copy commences operation with a call to `copied` in state A_1.

6.4 Method Invocation

When an object in one cluster attempts to invoke a method on an object in another cluster, it must block if the callee cluster is in the process of moving. The state diagram in Figure 5 indicates the process followed by the 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 prevents the caller from blocking the callee’ s progress.

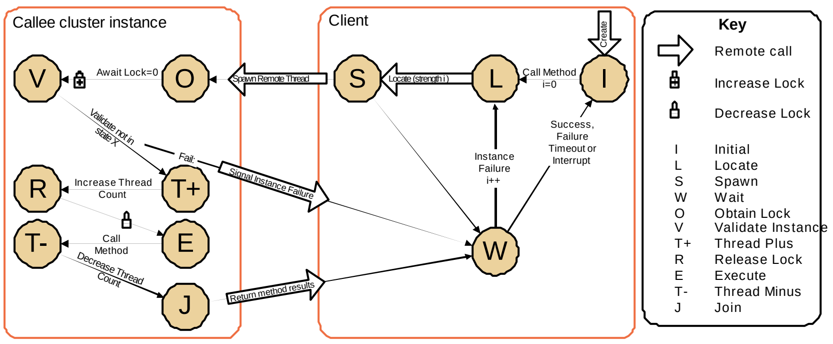


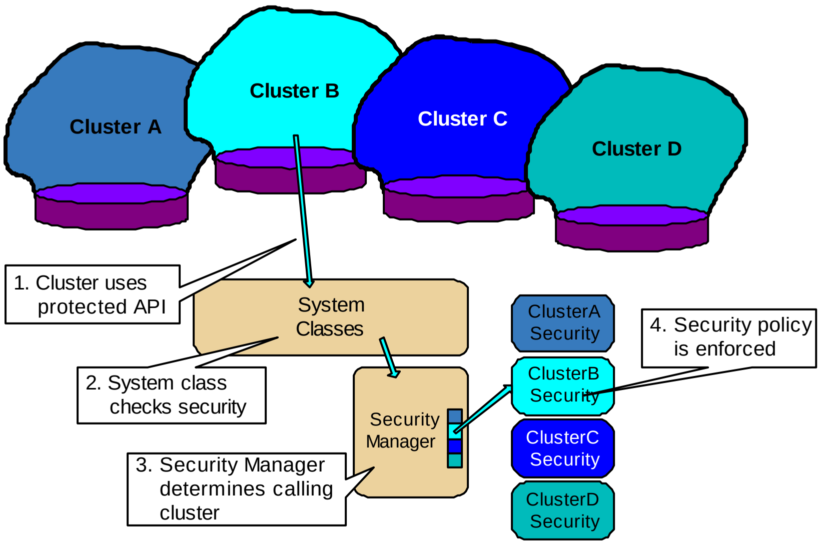
Figure 5 State transitions for an invocation on a remote cluster.

7 SECURITY

Within a 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 originated, 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, an object arriving from a remote host has a history that is not captured by the signature on the class file. Class signatures alone do not provide sufficient information on which to assign security policy.

For this reason, we have designed and implemented a Security Manager that extends the Java’ s security 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.

The cluster security manager is illustrated in Figure 6. It has two parts. The shared portion uses the identity of the calling thread to determine which cluster the call originated from. It then calls the per-cluster security manager to determine whether an operation should be allowed.



Cluster A
Cluster B
Cluster C
Cluster D
1. Cluster uses protected API
System Classes
2. System class checks security
Security Manager
3. Security Manager determines calling cluster
ClusterA Security
ClusterB Security
ClusterC Security
ClusterD Security
4. Security policy is enforced

Figure 6 Per-Cluster Security Managers

This security scheme 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 clusters 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 call a collection of objects within a cluster that are protected in this way a ' secure inner cluster' .

7.1 Secure Inner Clusters

A “secure inner cluster” is a cluster of objects within another cluster such as a mobile cluster. The secure cluster uses a security management interface on the host that it currently resides on and is accessed via an access interface by the containing cluster (Figure 7).

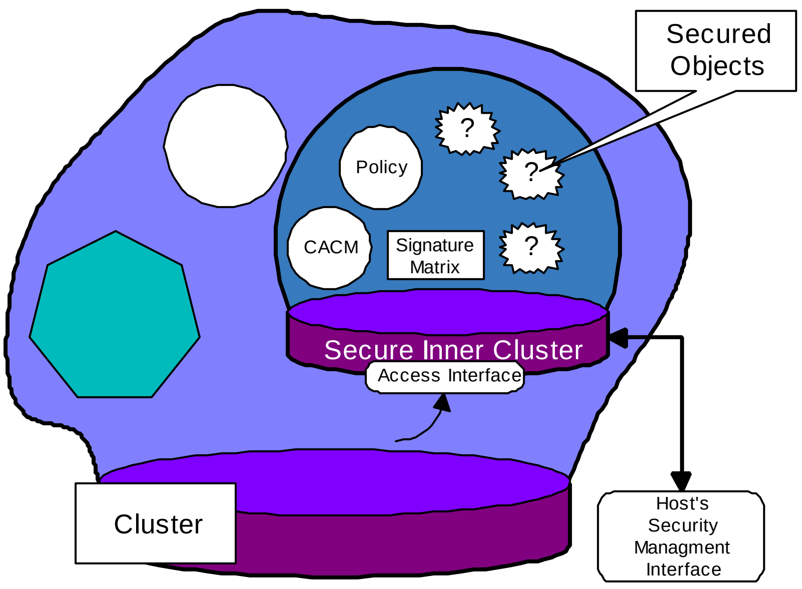


Figure 7 A secure inner cluster

The secure inner cluster is treated as an object by the outer cluster and therefore moves when the outer cluster moves.

The inner cluster does not share any objects with the outer cluster and the semantics of calls on its interfaces are call by value (i.e. copies are made of all arguments passed in and all results passed out).

Inner clusters are implemented as a specialisation of a standard management class that has specialised serialisation methods for serialising and deserialising the inner cluster’ s state. These perform cryptographic encoding and decoding of the state.

The inner clusters are used to create secure carrier objects. A secure carrier object contains a collection of named objects, referred to as the secured objects in the carrier, together a cryptographic access control matrix, a policy object and a signing matrix. The policy defines a state integrity check on the contained objects and the top-level cluster. The signing matrix is a table of principals against signatures. The carrier object supports the methods:

- `void initialize(Cluster context)`
  Initialise the secure carrier object and inform it of its current context (the cluster within which it resides). This method is called by the infrastructure when the secure carrier object is first created/deserialised. The carrier object may perform an integrity check on its context, for example to ensure that it has not been removed from its original cluster and associated with a new one.
- `void put(String name,Object object)`
  Put an object into the carrier and performs encryption and signing using the host's security infrastructure and the keys from the cryptographic access control matrix and, if required, the hosts signing key. (The actual encryption may be delayed until the carrier is serialised).
- `Object get(String name)`
  Return a secured object from the carrier using keys obtained from the cryptographic access control matrix.
- `void sign(String objectName)`
  Perform signing on the secured object in the carrier.

-
  ```
  void dependantSign(SignatureReferenceList l,
                           String objectName)
  ```
  Takes a list of signature references, l (see below), and a secured object reference, s, and produces a signature on s that is valid if and only if the signatures in l remain unaltered (see below).
- `boolean check(String objectName, Principle p)`
  Check whether a specified principal has signed the current value of the secured object in the carrier.

The signing matrix is a mapping from object names to principles to signature forms. It contains details of the object values that different principles are willing to commit to, together with digital signatures to prevent tampering. The signing matrix contains two different signature forms, simple signatures and compound signatures.

Simple signatures are constructed by signing a digest of the name and value of the object being signed. They are used to indicate that a particular principle has set (or agrees with) the value of a signed object. Compound signatures are used in dependant signing. This is where a principle makes a statement of the form "I will commit to this value if *x* remains committed to this value". A compound signature is a signing of the digest of an object name and value *together with* the list of dependant values – i.e. a list of (object names, principles) which represents which principles must remain committed to which values. This is illustrated in Figure 8.

Sig. Form = Signature | Compound Sig.
Compount Sig. = Sig. Reference List × Sig.
Sig. Reference List = Null | (Object Name × Principle) × Sig. Reference List

Figure 8 Definition of a Signature Form

Signature checking for compound signatures is recursive in that to check a compound signature is valid requires recursively checking the signatures in the signature reference list.

The basic operation of a secure carrier object is illustrated in Figure 9.

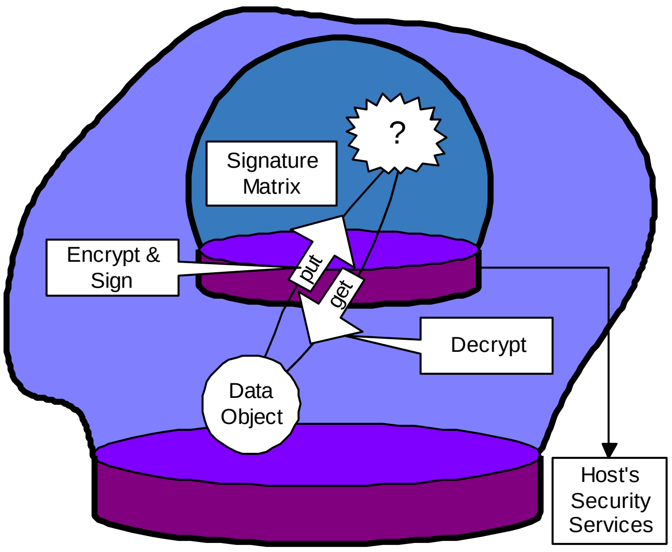


Figure 9 Operation of a secure carrier object

The security model requires that the code of a secure object, the Cryptographic Access Control Matrix (CACM) and the Policy (specification of movement itinerary and signing behaviour associated with a secure object) are signed by an accepted source. They must be mutually signed, so that the code, CACM and Policy cannot be detached from one another. This requires that the tuple

(secure carrier object class, CACM, policy)

is signed by an accepted authority. This is equivalent to applet signing. The class may be referenced by a Java `Class` object serialised by a FlexiNet serialiser that uses the FlexiNet Class Repository. The serialised object will actually be a secure reference to a class stored in the repository.

Figure 10 shows component interactions.

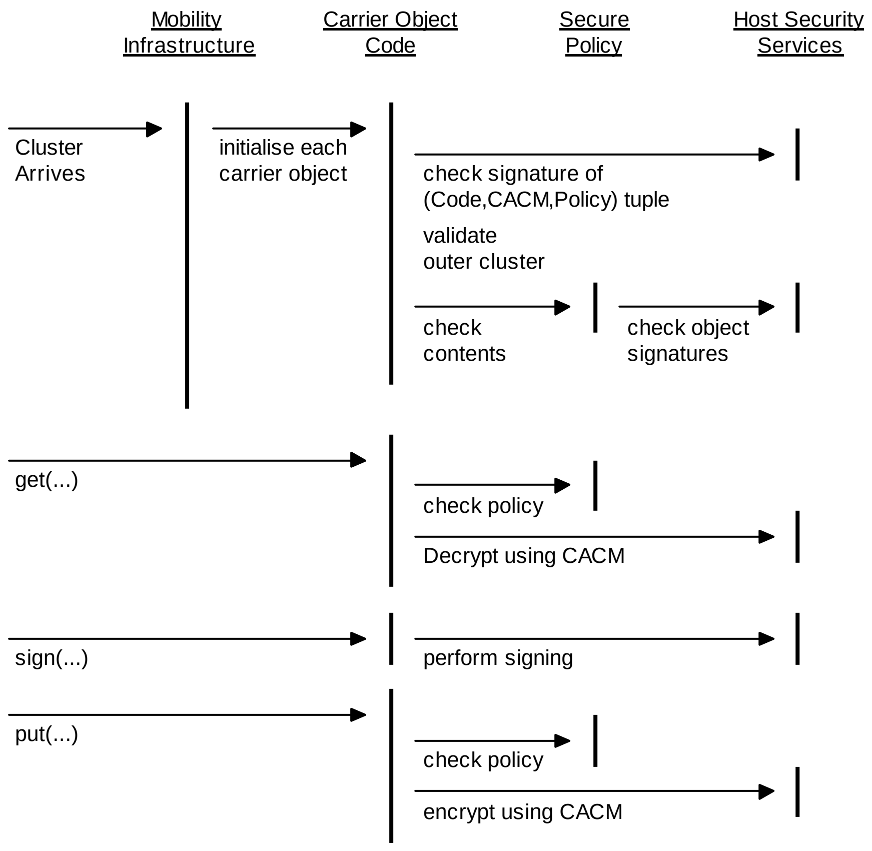


Figure 10 Interaction of Security Components

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.

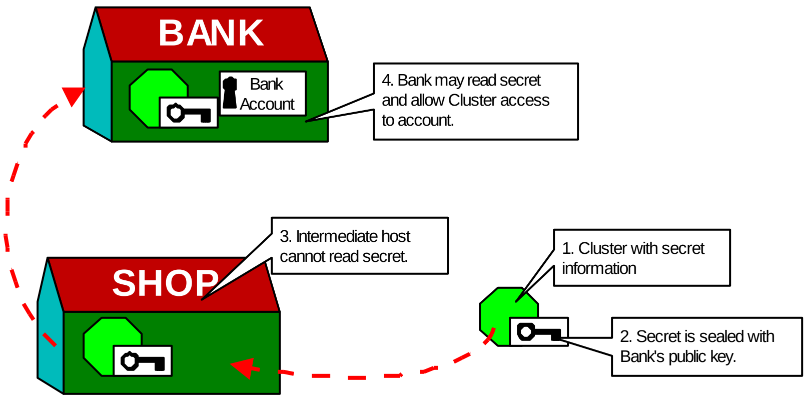

BANK
Bank Account
4. Bank may read secret and allow Cluster access to account.
SHOP
3. Intermediate host cannot read secret.
1. Cluster with secret information
2. Secret is sealed with Bank's public key.

Figure 11 Clusters with Secrets.

Secure carrier objects can be used to provide cluster authority. They 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 that 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 11 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.

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 take place using a FlexiNet binder that supports SSL (Netscape 1996). 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.

8 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 six key properties:

1. We arrange that a client can locate the appropriate directory for a cluster rapidly, without having to search.
2. We allow naming records to be moved between directories so that an optimal location can be chosen for the record (e.g. following the movement of a cluster around the network).
3. Different instances of the service may be implemented in different ways, or turned for different performance/robustness/scalability trade-offs.
4. We control what entities are able to update the records - only hosts from which a cluster is moving may update the record for the cluster. This prevents fraudulent changing of naming records by "spoof" hosts or clusters.
5. We detect and handle all potential ' identifier clashes' , whether accidental or malicious. This closes the security loophole, removes the reliance on a statistical probability (that randomly chosen numbers are unique), and therefore allows us to use smaller identifiers, as the possibility of a clash is no longer fatal.
6. We ensure that records relating to deleted clusters, or those that have been moved to a different directory can be archived and never need updating. We allow these records to be ultimately deleted.

The basic function of the relocation service is straightforward. It stores a simple table of mappings from old to new cluster addresses. There may be more than one instance of the service and each holds a portion of the mapping table. This spreads the load is spread between them. As each mobile name explicitly contains a reference to an instance of the relocation service (directory) this partitioning incurs no extra cost, and has dual advantages of scalability and decentralised management.

When resolving a mobile name in order to make a call; an optimistic client will first try the last known address, whereas a pessimistic client will first contact the relocation service. The relocation service cannot promise accurate information – the named interface may be moving faster than it can be resolved – and it is up to a particular client to decide if and when to give up.

To ease the burden of the relocation service, each cluster is given an identifier. A directory need only store a mapping from each identifier to the corresponding cluster' s current address, rather than a complete history of the cluster' s movements. Generally, the identity assigned to a cluster will not change over its lifetime. However, the possibility of renaming is introduced to prevent security attacks whereby two or more clusters with the same identity are created in logically remote servers, and then moved to the same location. There is also a (slight) possibility of two clusters accidentally been given the same identifier.

8.1 A Toy Relocation Service

The current implementation of the Mobile Object Workbench relies on a ‘toy’ relocation service. This is insecure, and cannot cope with accidental or malicious ‘clashing’ of cluster identifiers.

The implementation is based on a simple hashtable, which maps cluster identifiers to addresses. For clusters which have never been assigned a new identifier, and that still use the original directory, then this requires exactly one entry per cluster, regardless of the number of moves. It takes exactly two additional (remote) method lookup for a client to access a cluster that has moved. The first call is the failed invocation using the old address; the second call is the lookup at the relocation service.

For a cluster that has moved to a different directory, but has not changed identity, then one entry is stored in each directory. At worst, clients will have to contact each of these in turn before correctly locating a cluster. However, if moving between directories is rare (as is expected) then clients will normally have a reference to the most recent directory.

Clusters are rarely assigned a new identity. This is only done when two identifiers for different clusters are found to ‘clash’. In the toy implementation, this is only detected if a cluster is moved/recreated at a location where a cluster of the same identity already resides. If the relocation service is asked to store two different mappings from identifier to address, then it will discard the first one. This is clearly incorrect behaviour and is the primary security attack against this implementation.

A directory therefore only ever has knowledge of one cluster with a particular identifier. If this cluster changes identifier, the toy relocation service simply stores a record within the hashtable that maps from the original cluster identifier to the new identifier. On resolution, this leads to an extra (local) lookup to determine the cluster’s address.

The toy implementation has many advantages over some schemes, such as tombstones. However it also has a number of failings when used in a large distributed system.

- It is insecure. An attacker could arbitrarily add, remove or change information to disrupt normal activity. In addition, accidental or malicious clashing of identifiers will lead to errors.
- It is not robust. Even if the records were stored persistently, an instance of the relocation service is a single point of failure for clusters it manages. This is mitigated somewhat by optimistic strategies and the use of many instances of the relocation service.

The use of many instances of the relocation service partitions the object population, and each instance is only responsible for a proportion of the whole. By using an appropriate number of instances, the effect of failure can be reduced. In addition, as the relocation service is only contacted by clients to locate clusters that have moved, the effect of a failure will be limited to those cluster that have moved since last being contacted by their clients.

We have designed a more complete relocation service that is secure, robust and can recover from malicious or accidental clashing of identifiers. The implementation of this service is currently in progress.

9 IMPLEMENTATION STATUS

The Mobile Object Workbench version 2.0 is currently in use by the members of the ESPRIT FollowMe project. The current implementation supports clusters, mobility and transparent communications as described in this paper. The implementation of the ‘secure cluster’ abstraction, and the implementation of the security aspects of the relocation service, are incomplete.

In addition to specific Mobile Object Workbench issues, work on FlexiNet is continuing. We have recently created added support for secure bindings using SSL and support for CORBA interoperability using IIOP (OMG 1997). Transactional support is underway.

10 SUMMARY

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

REFERENCES

FAST (1997) FollowMe project overview

http://hyperwav.fast.de/generalprojectinformation
General Magic (1995) Telescript Language Reference
General Magic Inc.
Hayton, R.J. and Herbert, A.J. (1998) A flexible component oriented middleware system. *SIGOPS European Workshop'98*
Lange D.B and Change D.T (1996) Aglets Workbench
http://www.ibm.co.jp/trl/aglets
Netscape Inc. (1996) The SSL Protocol
http://home.netscape.com/newsref/std/SSL.html
ObjectSpace (1997) Voyager Core Technology
http://www.objectspace.com/Voyager/
Object Management Group (1997) CORBA/IIOP 2.1 Specification
http://www.omg.org/corba/corbiiop.htm
ODP (1995) ODP Referenec Model
http://www.dstc.edu.au/AU/research_news/odp/ref_model/
Sun Microsystems (1996) Java Remote Method Invocation (RMI)" Specification *http://www.sun.com/products/jdk/1.1/docs/guide/rmi/*

BIOGRAPHY

Richard Hayton graduated from Cambridge University in 1991 and joined the department as a research associate investigating LAN multimedia. He was awarded a PhD in 1996 for work on an Open Architecture for Secure Interworking Services (OASIS). He joined Citrix Systems (Cambridge) in October 1996 and has since been leading work on a flexible Java middleware platform (FlexiNet). His current research interests include support for complex trust relationships in distributed systems.

Douglas Donaldson graduated in Engineering and Computing from Oxford University in 1991. He worked in the Distributed Software Engineering section of the Department of Computing, Imperial College, London from 1992 to 1996, gaining a PhD in object modelling and design of distributed systems. Since 1996, he as worked for the ANSA research group in Citrix Systems (Cambridge), investigating flexible, object-oriented middleware to support distribution transparencies.

Michael Bursell graduated from Cambridge University in 1994 and spent two years with Cambridge University Press engaged in electronic publishing projects and Internet development. Since joining Citrix Systems (Cambridge) in 1996 he has worked on Java middleware, database integration and FollowMe. Much of his work has focused on issues of mobile code, from Java applets and RMI to push technologies and mobile agents, and he has a particular interest in issues of security around mobile object systems.

Andrew Herbert, is Director of Advanced Technology for Citrix Systems Inc. He led the ANSA industry sponsored programme of research and advanced development into the use of distributed systems technology from its start in 1985 through to its completion in 1998 with the work reported in this paper. His interests are in the application of object technology to the deployment and management of distributed systems.