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 straight-forward programming paradigm which embraces programming language semantics such as strong typing, method invocation and encapsulation. We have built a Mobile Object Workbench on top of a flexible Java middleware platform, which can be used as a the basis for a Mobile Agent System. 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.

1 Introduction

Software agents, and in particular mobile agents, are currently an active area within both the research community and the commercial sector [1], [2]. However, despite obvious interest, the commercial take up of mobile agent technology has been slow, and the search for a 'killer app' continues. We believe that one of the reasons for this caution is that the mobile agents approach has been billed as an alternative to the traditional distributed systems paradigm. Typically, the transition to agent based programming requires a programmer to learn a new programming language, and discard existing programming tools and network services.

This sets up a chicken and egg situation for agent systems: until they can provide all the services and integration capabilities of current middleware such as CORBA [3] and DCOM [4], developers will be reluctant to build large systems using them.

We believe that the correct approach is to extend existing programming paradigms by adding autonomy and mobility facilities. This will allow developers to use these facilities where appropriate, and standard distributed systems techniques elsewhere. We therefore present 'agent facilities' as a natural extension to the programming environment used for existing distributed system development.

This approach has many additional advantages when coupled to a network-oriented object programming language such as Java: method invocation extends naturally to remote method invocation, and strong typing support offers greater safety over ad-hoc messaging found in some current agent systems. Java has specific features that benefit our approach. *Interfaces* may be considered as the published access points for services, and *objects* as the means by which these services are implemented. This separation is important, as it allows some other object to act as a *proxy* for a real service object - by implementing the same interface, and providing the same semantics by communicating with the remote service. The construction and use of proxy objects can be made transparent to the application programmer, and it is easy to imagine how proxies may be used to represent services that are themselves mobile.

The notions of object and interface are fundamental to the design of distributed systems, and is formalized in the ODP reference model [5]. RM-ODP defines a number of *distribution transparencies*. Existing platforms such as CORBA provide *access* and *location* transparencies - the ability to communicate with an object regardless of its location or network address. The Mobile Object Workbench that we have constructed adds two additional transparencies: *relocation* transparency - a client need not be aware that a service has moved - and *migration* transparency - a service need not be aware that it itself has moved. Together these form a basis for the construction of mobile agents. The other ODP transparencies are replication, persistence, transactions and security. These are being addressed within the wider context of our project.

The mobile object workbench described in this paper is an implemented system that provides objects with access, location, relocation and migration transparency within the Java object model. It is not in itself an agent system; however it provides many of the basis facilities required by a mobile agent framework, and provides an easy transition path for programmers wishing to incorporate agent behaviour within the context of a larger distributed system. The work on the Mobile Object Workbench has been undertaken as part of the FollowMe ESPRIT project (No. 25,338) on support for mobile users. Other partners within the project are designing a fuller agent system on top of our Mobile Object Workbench [6].

1.1 Related Work

Many existing agent systems have been built from scratch, and have had to contend with both agency and distribution issues. This division of effort has led to general weaknesses in distribution abstractions: for example untyped message passing is common (e.g. [7]).

Many agent system designers have tackled the problem of implementing mobile code by basing their systems on scripted languages, simplifying the mobility of code across heterogeneous systems, and allowing control of its execution (e.g. suspension and resumption) (e.g. [8]). The disadvantage of this approach is that introduces new programming languages, which currently have no tool support, and no integration with other services.

Java solves many of the problems of agent system designers. Code mobility is relatively straightforward, although there remain problems of controlling execution. This has prompted some designers to port and upgrade their systems to make use of the new facilities (e.g. [9]), but few have taken the approach of using Java as the agent programming language, and adding to it facilities for object movement and remote method invocation.

An exception to this is Voyager [10] which takes an approach similar to ours. However this makes use of the Java RMI service [11] which add its own restrictions. RMI divides objects into two categories, service objects and data objects. Data objects may be copied in a remote method invocation, but service objects may not. This precludes the possibility of a service object moving - effectively ruling out the most natural approach for mobile agents. Voyager overcomes this with some complex wrapping of service objects, we avoid the problem entirely by using our own remote invocation and binding mechanisms [12].

The OMG's Mobile Agent Facility [13] is an important piece of standardization work being done in the area of mobile agents. Whilst we see the Mobile Object Workbench 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 option for future work.

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 build 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 Clusters

Language level objects are typically too small to be a useful unit for mobility. For example, 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 base 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.

Clustering is not a new concept. The notion of clusters was present in ANSAware [14] and RM-ODP [5]. In Java, each applet is effectively a cluster, and programming using clusters is no more complex than applet programming. As clusters provide units of encapsulation for mobility, security and other functions, each mobile agent would normally be constructed within its own cluster.

2.2 Encapsulation

The cluster boundary provides a strong encapsulation boundary between clusters. Access to objects within one cluster from threads in another is restricted to interfaces that have been passed (directly or indirectly) between the clusters. Clusters cannot examine the internals of each other, nor may arbitrary methods on objects in one cluster be called from another. We can (and do) take this approach even further. When a thread in one cluster invokes a method on an exported interface from another cluster, we de-couple the threads so that the callee and caller cannot adversely affect one another by blocking or thread termination. In addition each cluster may effectively be given a separate Java security manager, and class loader. Effectively, each cluster is a ‘virtual process’ that is de-coupled from other clusters in terms of privileges, code base and management. Although one cluster may crash the entire process or starve it of resources, clusters are isolated from each other as effectively as possible, making them suitable vessels for storing wandering agents.

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 local *stubs* to represent interfaces to object in remote clusters. 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 Mobile Object Workbench API is entirely related to the life cycle and movement of clusters.

Unlike RMI or CORBA we do not require an off-line stub generator. Instead we allow the generation of stubs on demand, whilst allowing caching of previously generated code for performance. The generation of stubs on demand has the important

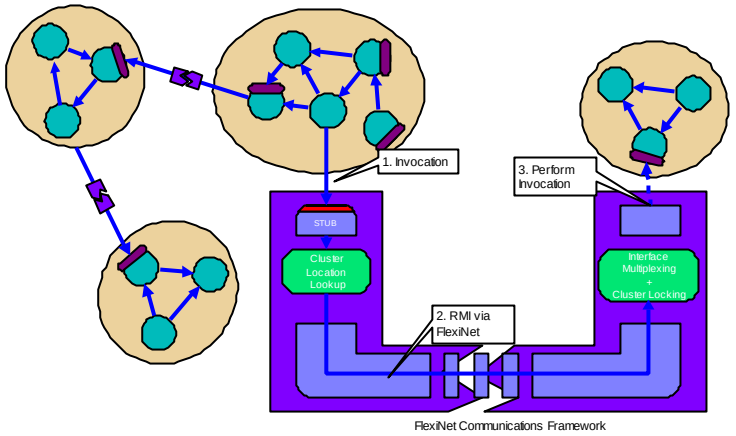

Fig. 1. Clusters, showing internal and external communications.

benefit that since stubs are locally generated they may be treated as trusted, rather than potentially hostile, code.

It has long been a point of contention in distributed systems research as to whether distribution transparency is achievable, or indeed desirable. In FlexiNet, 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 when resolving inter-cluster interface references; 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.

3 Implementation

Figure 1 shows an example set of objects and clusters, and how the engineering framework provides the transparency of remote references. 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.

In order to encapsulate clusters, we must distinguish references between objects within the same cluster, and 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.

3.1 Location transparent communications

All references to interfaces within the Mobile Object Workbench are treated identically by the programmer, that is, the application/agent performs the same action, whether a reference is to an interface on a local or remote cluster, or whether it is to a cluster that is currently in transit. A call to a remote cluster may result in an exception if the cluster 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 cluster 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.

When a cluster is to move, we must carefully orchestrate the behavior of its threads, and any calls made to it, to ensure that a consistent version of the cluster is moved, and that the move is atomic with respect to calls made from other clusters. As we have built our system on top of the Java virtual machine, we have little control over thread execution, and in particular it is not possible to serialize a thread during execution in order to move it. This is the price we have had to pay for using a standard language implementation, rather than providing our own interpreter. However, in practice the price is not high: a cluster must close down any internal threads prior to movement, and restart them after movement. The infrastructure monitors thread activity and ensures that the move is atomic. This is described in detail in [15].

3.2 Mobile Object Workbench API

The Mobile Object Workbench API (figure 2) is wholly concerned with the lifecycle and management of clusters. Communication between object in clusters takes place using application level remote method invocation and is therefore not specified in the

```
public abstract class MobileObject
{
 void pendMove(Place dest) throws MoveFailedException;
 void syncMove(Place dest) throws MoveFailedException;
 Object copy(Place dest)   throws MoveFailedException;
 abstract 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);
}
```

Fig. 2. Mobile Object Workbench API.

API. An abstract class MobileObject gives a ‘ hook’ on which the implementation of a particular mobile object may hang (in much the same way as the class Applet). Each subclass may provide instance initialization by providing an init method, which allows arbitrary arguments to be passed to the mobile object upon creation. The restart method is called on completion of a move, in order to allow a mobile object to restart threads. The Tagged type represents a most general interface reference type (whereas the Java Object type represents the most general object reference type). The interface Place is exported by system provided objects that represent execution environments for mobile objects. Mobile objects are created in places, and may move between them. Places also provide properties which mobile objects can use to interrogate their environment.

3.3 Location-transparent naming service

When a call is made on a 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 by the infrastructure in the callee’ s cluster. 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 five key properties:

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

their current network host. This prevents fraudulent changing of naming records by "spoof" hosts or clusters.
2. We provide a hierarchy of directories, for scale and robustness. This means that an instance of the relocation service may decide to copy the naming record for a cluster up the hierarchy to increase its stability, or to reduce the load placed upon it.
3. We allow naming records to be moved between directories so that an optimal directory location can be chosen for the record (e.g. following the movement of a cluster around the network).
4. We allow caching for performance. A naming record can be kept at a previous directory, as well as being passed up the hierarchy, to reduce look-up time.
5. We arrange that a client can locate the appropriate directory for a cluster rapidly, without having to search other directories.

4 Support for Agents

As described so far, the Workbench is does not provide a suitable platform for mobile agents; two additional capabilities are required. First, agents are generally considered as autonomous. Support is therefore required to ensure that a cluster's movement is under its own control. Second, unlike a single application with mobile parts, a mobile agent system typically consists of agents with mutual distrust between each other, and with complex trust relationships between agents and hosts. Security covers a whole range of issues from data protection to code integrity, and a mobile agent system must deal with these if it is to be viable in the wider network.

4.1 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.

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 one.

4.2 Security

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 [16]. 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 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** - 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, 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. Recent Java releases provide a number of cryptographic capabilities which have aided this work.

4.2.1 History and provenance

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, object capabilities are controlled within a Security Manager [17], and policies are typically granted based on the provenance of these classes. Classes may be signed to provide extra confidence of provenance, and a JVM may assign policies to instantiations of these classes based on this signing.

However, 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, as they can solely give information about the initial "birthplace" of a set of classes, and not about the history of a particular instantiation.

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

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

4.2.2 Integrity and confidentiality concerns

Cluster integrity and confidentiality 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 which it visited earlier. In addition to this, we must ensure that clusters are not dissected, as otherwise a malicious host could 'steal' parts of the cluster that represented encrypted passwords and use them to build its own clusters. To do this, we require a mechanism for specifying and validating integrity statements. For example we may annotate a cluster's definition to indicate that a particular field may only be modified by certain hosts. We may then use digital signature techniques to ensure that whenever the field is modified it obtains a signature from the current host, and when other hosts attempt to read this field we can throw an exception if the signature is incorrect.

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

Cluster authority can be implemented by leveraging cluster integrity and confidentiality. Together these allow a cluster to carry with it a password or other secret information, without the concern that this secret can be read at any host which is visited. Clearly, once the secret *is* revealed to a host, there is nothing that can be done to prevent the host from misusing it. For this reason we have a model that the mobile object moves into a secure environment before revealing a secret. Figure 3 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.

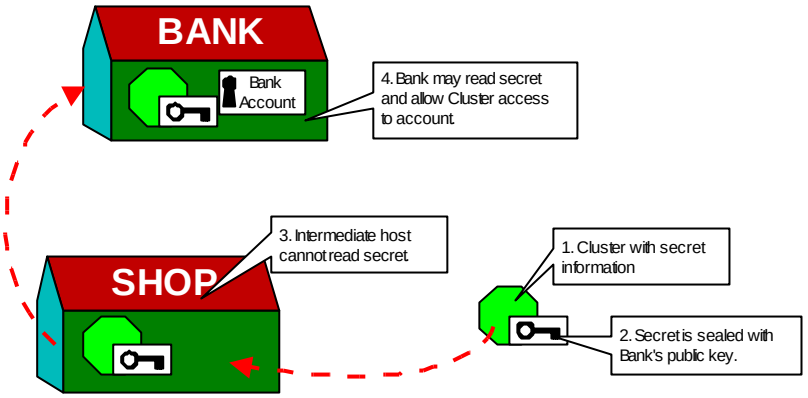

Fig. 3. Clusters with secrets.

4.2.3 Method access and communications

Method 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 its 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 [18]. Secure communication between mobile clusters may also take place using SSL, but requires that clusters reveal the information used to prove their identity to the host from which they are communicating. This is reasonable in some circumstances, but should be used with caution.

5 Status

The FollowMe project, and the design of the Mobile Object Workbench, started in October 1997. We now have a fully functioning mobile object system written using 100% pure Java. The limitations of this system are primarily an incomplete implementation of the security infrastructure described, and limited scalability within the relocation service. We are concentrating our efforts in these directions. Other FollowMe project members are working on an agent system, pilot applications and other services underpinned by the mobile object system.

Although relatively young, the Mobile Object Workbench has undergone three releases, and has been in used by FollowMe project members and the ANSA consortium for the past six months.

6 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.

References

1. MA 97: First International Workshop on Mobile Agents 97, Berlin, April 7-8, 1997.
   http://www.informatik.uni-stuttgart.de/ipvr/vs/ws/ma97/ma97.html
2. The Agent Society: Agent Product and Research Activities
   http://www.agent.org/pub/activity.html
3. Object Management Group: CORBA/IIOP 2.1 Specification. Aug. 1997.
   http://www.omg.org/corba/corbiiop.htm
4. Brown, N. and Kindel, C.: 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
5. International Standards Organisation : Open Distributed Processing - Reference Model. Sep. 1995. http://www.iso.ch:8000/RM-ODP/
6. FAST e.V.: FollowMe project overview. http://hyperwav.fast.de/generalprojectinformation
7. Lange, D.B. and Chang, D.T. : IBM Aglets Workbench - Programming Mobile Agents in Java, A White Paper (Draft). IBM Corp.. Sept. 1996. http://www.trl.ibm.co.jp/aglets/
8. General Magic Inc:Telescript Language Reference. 1995
9. General Magic Inc: Agent Technology http://www.genmagic.com/agents/
10. ObjectSpace: ObjectSpace Voyager Core Technology.
    http://www.objectspace.com/Voyager
11. Sun Microsystems: Java Remote Method Invocation (RMI) Specification.
    http://www.sun.com/products/jdk/1.1/docs/guide/rmi/ 1996
12. Hayton, R.J. and Herbert, A.J.: A flexible component oriented middleware system. SIGOPS European Workshop 1998
13. Crystaliz, GMD FOKUS, General Magic, IBM, The Open Group: Mobile Agent Facility.
    http://www.genmagic.com/agents/MAF/
14. O' Connell, J. Edwards, N. and Cole, R. A review of four distribution infrastructures. Distributed Systems Engineering 1 (1994) 202-211
15. Hayton, R.J. Bursell, M.H., Donaldson, D.I. and Herbert, A.J. : Mobile Java Objects Middleware 1998.
16. Wallach, D.S., Balfanz, D., Dean,D. and Felten, E.W.: Extensible Security Architectures for Java. Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles 31,5, Dec. 1997, pp. 116-128.

17. Erdos, M., Hartman, B. and Mueller, M.: Security Reference Model for the Java Developer's Kit 1.0.2. Sun Microsystems. Nov. 1996. http://java.sun.com/security/SRM.html
18. Netscape Inc.: The SSL Protocol. http://home.netscape.com/newsr ef/std/SSL.html