A Two-Level Architecture for Semantic Protection of Persistent Distributed Objects

Mark Evered
University of New England

email: markev@turing.une.edu.au

Abstract

In this paper we formulate requirements for achieving flexibility and security for distributed persistent objects. We argue for the separation of a user-view level from an implementation level and for the integration of capabilities for semantic protection. We describe a system based on these criteria which allows simple, secure programming of multimedia applications using Java objects distributed across the internet.

1. Introduction

Multimedia applications always involve in some form the processing of persistent data, that is, the processing of data which exist longer than the program activations which create or process them. Conventional systems and conventional programming languages distinguish between transient data within programs and persistent data stored in files. Object-oriented languages generally maintain the same dichotomy: data stored as objects within programs must be converted to a linear form and stored in files if they are to persist. Similarly, the file data must be reconverted to object form when reloaded. This conversion requires a great deal of programmer effort and also implies the loss of type security.

The programming language Java [5] (since version JDK 1.1) offers a mechanism for alleviating both of these problems: if a class and all the classes it (transitively) uses implement the special interface 'java.io.Serializable', then an object of that class, together with all objects reachable from that object, can be serialised to a byte stream and stored as an ordinary file. Similarly, the whole graph of stored objects can be read and reconstructed from the file. The dichotomy still exists but the code for the conversions is generated automatically and type information is stored and retrieved together with the data. This approach is sometimes known as 'light-weight' persistence.

In combination with object-oriented programming, persistence can promote a new way of looking at files in a software system. Rather than being flat, typeless streams of bytes accessed via 'read' and 'write' operations, files can be viewed as persistent objects, accessed via operations meaningful for the application. A patient-records file in a hospital, for example, is then a persistent object with operations for registering new patients, retrieving the address of a patient etc. as shown in Fig.1.

Such a semantic *file object* can be simulated in Java by providing the class with 'load' and 'save' methods which internally use light-weight persistence to load or store the object contents from a file. The main limitation of this approach is that the file object loses its identity if it is used concurrently by more than one program. Each program loads a different copy of the object and only one of the programs will succeed in writing its changes back as a file.

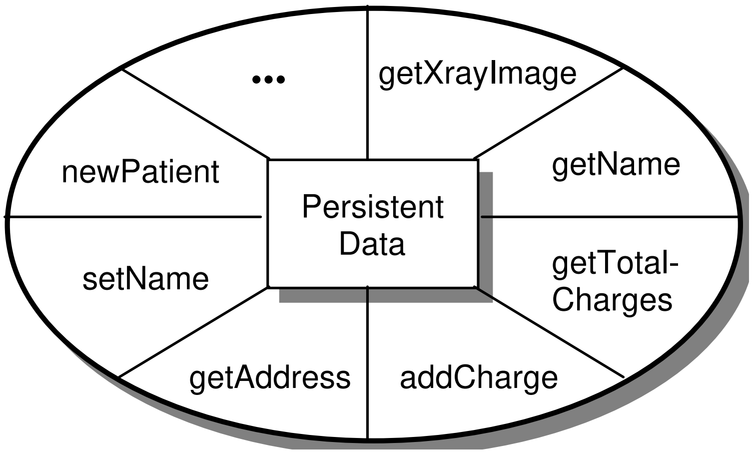

Fig. 1: A semantic file object for patient records

Almost as self-evident as the need for persistent data in multimedia systems is the need for distribution. To extend the above example, it is clear that many computer systems distributed within and beyond a hospital may need to access and/or modify the data in the patient-records file. With the growth of the internet, applications involving persistent, distributed Java objects can be expected to become increasingly important.

The aim of the work presented in this paper has been to define and implement a software architecture for such applications. While a number of proposals exist for both persistence and for distributed objects in Java, our specific aim has been to combine both aspects and to place emphasis on *flexibility* and *security*.

In the following section we formulate requirements for supporting the concept of distributed semantic files in a flexible way. In the third section we address the question of protecting such objects. Section four describes the user-view level of an architecture which aims at fulfilling the requirements in the context of distributed multimedia Java systems. In section five we describe a system which provides the implementation level via the use of Java's remote method invocation mechanism.

2. Requirements for Flexible Distributed File Objects

Our first requirement results from the problem of sharing and object identity as in Java's light-weight persistence.

Requirement 1: *The file object must be sharable by multiple independent programs without loss of object identity*

A further well-known principle formulated in research on persistent systems is the principle of 'persistence independence'. This means that the use of persistent objects (at least after some initialisation) should be exactly the same as the use of non-persistent objects. This can be supported in a Java system by ensuring that the persistent object implements a Java interface type which describes the semantic operations but has nothing to do with persistence.

Requirement 2: *The use of the file object should be exactly the same as for a non-persistent object.*

In contrast to the 'minimal approach' of light-weight persistence, languages have been developed which provide the concept known as orthogonal persistence. In particular, PJava [2], a persistent version of Java, uses an extended class library, an extended interpreter and an underlying persistent object store to provide orthogonal persistence. This system fulfils our first two requirements but, apart from lacking support for distribution, has two problems with respect to semantic file objects: protection and deletion.

In PJava, any existing object in a program can be promoted to a persistent status and stored as a 'persistent root'. This means that the object and all objects reachable from that object are transferred to a special persistent store. The objects are then sharable and concurrent access is managed by an automatic transaction mechanism. This approach removes the persistence dichotomy but gives rise to another kind of problem: we want the contents of a semantic file to be persistent but we also want to be sure that *no more than the contents* of the semantic file are persistent (and sharable). One reason for this is protection: we want to be able to define exactly which data are to be made persistent rather than having the system store everything reachable from some root object. Protection will be discussed more fully in the next section. Another reason is garbage collection: we want to be able to specify exactly when persistent data are deleted. The 'transitive persistence' approach simply retains objects in the persistent store until they are no longer reachable from a persistent root.

The intuitive concept of a file, whether object-oriented or not, is that it *contains* information and that this information is *within* the file. Only if this is in fact the case, can the protection of a file mean the protection of the information in the file. For this reason, it is necessary to distinguish between file objects which are persistent and protected and 'ordinary' objects which are persistent and protected only by virtue of being contained within a file object. This leads to our next requirement:

Requirement 3: *The file object should be self-contained, i.e. contain no references to objects which are not part of the file (except for references to other file objects)*

The key phrase here is 'part of the file'. An object should be reachable from the file object only if it is conceptually part of the file or is itself explicitly defined as a file object in its own right. This requirement has been satisfied by some special computer architectures such as the Monads PC [10] but it is difficult to enforce without architectural support. Our solution is to allow only a small set of fixed types as parameters and result types of the methods of a file object. In our system for Java these are primitive types, strings and 'handles' for other file objects (which will be described more fully in the next section) as well as a small number of types for multimedia information.

A requirement similar to Requirement 2 can be also formulated for distribution. Just as we want 'persistence independence', we also want 'distribution independence'. It should be possible that a persistent object is located at a remote site without influencing the mode of its use (except again perhaps during initialisation).

Requirement 4: *The use of a remote file object should be exactly the same as for an object located on the same node as the program using it.*

If an object is shared then its use must be synchronised. The 'heavy-weight' persistence of PJava not only automatically handles reachability and deletion but also automatically provides a transaction-based synchronisation of concurrent access. This is fine if a transaction mechanism is required but in many cases where persistent objects would be useful it is an overkill. Java already provides a mechanism for synchronisation which can be used to control concurrent access in a very flexible way.

Requirement 5: *The file object should be able to be synchronised in the same way and with the same flexibility as a non-persistent object.*

A user view which fulfils our requirements so far can be implemented by many different mechanisms. There are many ways of achieving persistence including normal files, light-weight persistence and databases and there are many forms of distributed objects such as Java's RMI [11] and Corba [9]. A programmer writing code which uses a file object should not need to know how the persistence and distribution are achieved. Indeed, it should be possible to mix different mechanisms for persistence and distribution within a single application. This allows a maximum of flexibility and allows optimisations to be made without extensive, error-prone modifications to programs. This leads to our final requirement:

Requirement 6: *The use of the file object should not reflect the particular mechanism used to achieve the persistence or the distribution.*

3. Semantic Protection

Persistent objects and especially distributed persistent objects are generally objects which are intended to be shared and the users of shared objects should not, as a rule, all have the same degree of access to the object. Files in a traditional file system, for example, may allow read or write access to be specified for the owner, for members of the owner's 'group' and for the rest of the world. The security aspect has been largely neglected in connection with persistent languages and is often given only secondary consideration in distributed systems.

The discussion of requirements for security in open systems in [4] provides a good basis for evaluating a security mechanism for file objects. The main criteria are that the underlying mechanism:

- should be independent of specific security algorithms
- should allow control of access by individual users or whole groups of users
- must not be bypassed in any way
- should be flexible, simple and efficient

A simple, powerful protection mechanism for fulfilling these criteria can be offered via capabilities [8,12]. In particular, a form of password capabilities [1] can be used, avoiding the need for special architectural support. In addition to a unique object name, password capabilities contain a long (unguessable), system-generated password which allows access to the object. The owner of a persistent object can distribute various passwords for the object and can define the mode of access associated with each password.

The mode of access for a semantic file can be much more finely specified than for a normal file. Instead of a simple read/write distinction, we can specify exactly which of the semantic operations on the file may be executed by holders of a particular password. So, for example as in Fig. 2, we can arrange that programs used by the administrators in a hospital have access to the patients-record file via a password capability which allows access to the 'addCharge' and 'getTotalCharges' operations but not to the 'getXrayImage' operation. This provides a very flexible basis for implementing security policies.

A further level of security can be gained by also storing the capabilities themselves in semantic files. Such a semantic file is in fact a directory with semantic operations such as insert and retrieve. While capability systems inherently offer discretionary protection, the use of directories with appropriate operations [6] can also enforce mandatory access control policies such as the Bell/LaPadula model [3].

An important question in any capability system is how the capabilities themselves are protected. In our system, the capabilities for our semantic files (or for the directories if this further level is added) are simply stored as normal files and protected by the file system security of the host system. Encryption can be used when capabilities are transferred across a network.

4. An Architecture for Secure Persistent Objects

In this section we present a capability-based system for persistent objects in Java which complies with the requirements formulated above. In accordance with Requirement 6, we describe here only the user's view of such a system. (In the following section we describe a particular system for creating class definitions for persistent objects which conform to this view. Here we simply assume the existence of a class 'PHashPatientRecords' for persistent (possibly remote) objects representing patient-records files.) By a user we mean the programmer of a program or another file object which uses or creates a patient-records object.

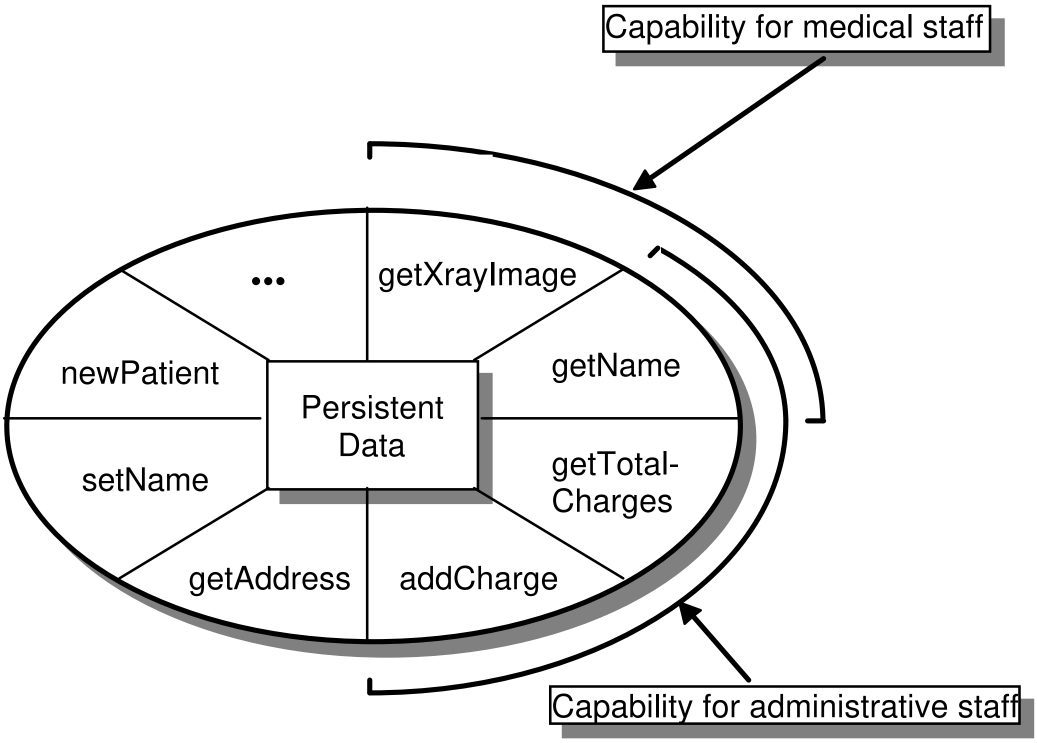

Fig. 2: Two different capabilities allowing invocation of a subset of the methods

The user view of a file object has two aspects: the semantic operations on the object (which are the same as if it were a non-persistent object) and the operations which treat it as a file object. The semantic operations are defined by a Java interface type, for example:

```
public interface PatientRecords {
   void newPatient(int patientId);
   void setName(int patientId, String name);
   /*enquiry*/ Image getXrayImage(int patientId);
   ...
}
```

The word 'enquiry' can be seen here simply as a comment indicating that the method 'balance' only returns information about the state of the object without changing that state. The file-specific operations are defined in a special predefined interface type which is implemented by all file objects:

```
public interface Persistent {
   // static Capability create(String uniqueObjName);
   void close();
   void delete();
   Capability refineCapability(String[] methods);
   Capability readOnlyCapability();
}
```

The commented static method indicates that all file object classes must offer a static method with exactly this signature. That is, the class 'PHashPatientRecords' has the form:

```
public class PHashPatientRecords implements Persistent,
                                             PatientRecords {
  static Capability create(String uniqueObjName);
  ...
}
```

The create method of the class creates a persistent object implemented by this class and returns an initial capability for the object (allowing access to all methods). As a parameter it is given a unique name for the persistent object. For example:

```
Capability prCap=
  PHashPatientRecords.create("/server.mainklinik.com/patients");
```

In a distributed system, the unique name is a URL as seen here. The type 'Capability' is a predefined class type defined as:

```
class Capability implements java.io.Serializable {
  ...
  public void saveAs(String fileName) { ... }
  public static Capability loadFrom(String fileName) { ... }
  public Object open() { ... }
}
```

The methods 'saveAs' and 'loadFrom' can be used to save or restore a capability from a file (via Java's light-weight persistence):

```
prCap.saveAs("mainklinikcap");
```

The 'open' method of the capability is used to open the object to which this capability allows access, that is to locate the object and return a pointer to it. Once opened, the file object can be used as any other object but with the difference that an entirely different Java program can be using the same object at the same time. For example:

```
Capability prCap=Capability.loadFrom("mainklinikcap");
PatientRecords p=(PatientRecords) prCap.open();

p.newPatient(123456);
p.setName(123456,"Smith, Samuel");
...
Image img=p.getXrayImage(123456);
Printer.output(img);
```

When a program is finished using a file object, it closes it by calling the 'close' method:

```
((Persistent) a).close();
```

Some implementations may not require this call but others (including the implementation presented in the next section) need to keep track of how many programs are currently using the object. The 'delete' method can be used to delete the file object entirely. Any other program which is currently using the object will receive an exception from the next method it attempts to call.

The remaining two methods defined in the interface 'Persistent' allow the creation of further capabilities with reduced access rights. In both cases the operations to which the new capability allows access are a subset of those accessible by the capability used to open the file object. The method 'readOnlyCapability' returns a capability which allows access only to operations which do not change the state of the object. This is determined from the methods marked as 'enquires' in the interface definition. The method 'refineCapability' returns a capability which allows access only to operations listed in the array of method names passed as a parameter (plus 'open' and 'close'). For example:

```
String[]
allowedMethods={"addCharge","getTotalCharges","readOnlyCapability"};
Capability refinedCap=((Persistent) p).refineCapability(methods);
refinedCap.saveAs("mainklinikcap2");
```

The following code:

```
Capability accCap=Capability.loadFrom("mainklinikcap2");
PatientRecords p=(PatientRecords) prCap.open();
p.getXrayImage(123456);
```

will then lead to the exception 'AccessDenied'. This exception and the exception 'PersistenceError' which indicates an error in the underlying persistence or distribution mechanisms can potentially be thrown by any method call to a persistent object. If one of these exceptions occurs, then the file object is automatically closed. Both exceptions are defined as subclasses of the standard exception class 'RuntimeException'. This is important for the principles of persistence independence and distribution independence (Requirements 2 and 4 above) since RuntimeExceptions need not necessarily be declared in the method or caught in the calling code. An ordinary exception would demand a 'catch'-clause which would imply that the same code could not be used for calls to persistent and non-persistent objects. In fact, according to the static semantic rules of the Java type system, a class which threw these exceptions as ordinary exceptions could not even be defined as implementing the interface 'PatientRecords'.

5. The Implementation Level: The Persistence Tool 'jshare'

There are many ways of providing persistence and fulfilling the user's view of persistence described in the previous section. Indeed one of the advantages of our requirement that file objects be partitioned, ie. have no common contents, is that each file object can potentially be made persistent in a different way. On the other hand, it is tedious for the programmer of a file object class, such as 'PHashPatientRecords' above, to program all the details of the persistence mechanism, the protection mechanism and the distribution mechanism for each class anew. Inheritance can help to avoid some of the repetition but there remains a significant amount of programming work which is specific for a particular file object class but which is tedious and mechanical to program.

In order to eliminate this work and to reduce the possibility of errors, we have developed a tool, 'jshare', which takes a Java class written to provide 'light-weight' persistence and creates a new class which provides the same functionality but with real sharable persistence, with remote access and with protection via capabilities as described above. This allows the programmer to concentrate on efficiently providing the functionality specified in an interface such as 'PatientRecords'. A programmer can, for example, simply write the class 'HashPatientRecords' as shown in Appendix A and the tool 'jshare' will automatically create a class 'PHashPatientRecords' as used in the examples of the previous section.

The tool also checks that the parameter and return types on the interface of the file object are only primitive Java types, the types 'String' and 'Capability' or one of the special multimedia datatypes such as 'Image'. Strings, capabilities and multimedia data are passed by copying their contents. This ensures the partitioning of file objects as required above.

In contrast to the persistence of the Monads-PC or the persistence of PJava, 'jshare' is intended to create file object classes in the context of an ordinary system architecture and the standard Java compiler and runtime system. Because of this, it can not be assumed that separate Java programs can directly share memory. The persistence mechanism used by 'jshare' therefore creates a new process for a file object when it is used and organises communication between user programs and the file object by using Java's remote method invocation facility. The use of RMI also allows file objects to be distributed around the internet without any special knowledge of the distribution in the code of either the user programs or the file object (except for a naming convention based on URLs).

Given a class such as 'HashPatientRecords', 'jshare' creates the invocation architecture in Fig. 3.

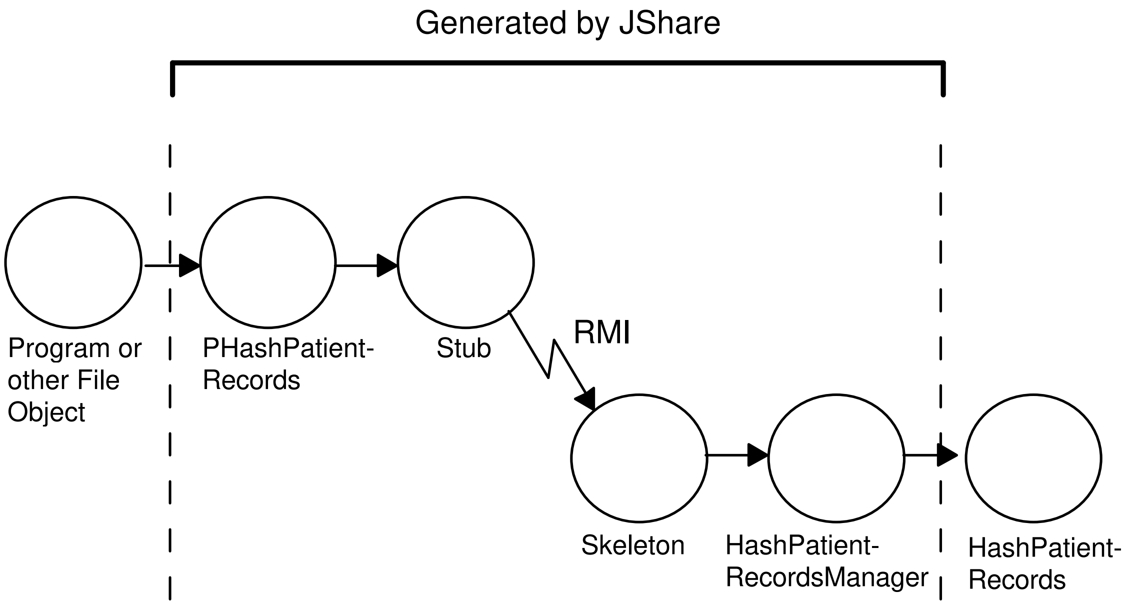

Fig. 3: Method Invocation architecture

Only the class 'PHashPatientRecords' is visible to a user. This class directs the method call via RMI to the remote object, and attaches the password of the capability with which the object was opened as an extra parameter. It also hides from the user the details of errors from the underlying RMI and light-weight persistence mechanisms by initiating retries and, in the worst case, converting errors to the generic exception 'PersistenceError'.

The 'HashPatientRecordsManager' object is responsible for checking whether the method call is allowed for the capability being used. This check is performed on the server side for security reasons. The client side of the invocation architecture is set up for each program when it opens the file object. The server side is common to all clients and is set up as long as some program has the file object open. When the file object is no longer in use, its contents are saved in a file via light-weight persistence.

As can be seen in Fig. 3, the implementation of 'jshare' can be seen as a security and persistence layer around an underlying RMI communication layer.

6. Related Work

Most work on persistent object systems and persistent programming languages has been based on the principle of transitive persistence or 'persistence by reachability' as it was originally called. This includes older persistent languages such as Napier [7] as well as PJava. As argued above, this approach differs from our aims in a number of ways. In particular, control over exactly which objects are made persistent is removed from the programmer and given to the system. As well as problems with protection and deletion, this approach relies on a modified virtual machine and does not support distribution.

The Java RMI mechanism, which we use in our implementation, does not fulfil the requirements on its own. As well as lacking semantic protection, the use of RMI is not transparent to the user who must, among other things, be prepared to catch and handle RMI-specific exceptions. Corba (which could be used as an alternative communication layer in our approach) also provides no semantic protection and involves a considerably greater overhead than our mechanism.

7. Conclusion

We have formulated a number of requirements for the use of distributed, persistent Java objects. In particular, we have stressed the need for flexibility and for a simple, integrated, semantic protection mechanism. We have presented an architecture in which a user level providing capability-based protection is separated from a hidden lower level which provides the implementation of persistence and distribution.

Finally we have described the 'jshare' system which allows simple, flexible programming of multimedia applications over the internet by automatically transforming Java classes written for light-weight persistence into a capability-protected object invocation structure implemented via remote method invocation.

Appendix: Example Source Code

```
import java.util.Hashtable, java.awt.Image;

class PatientRecord implements java.io.Serializable {
  int patientId;
  String name;
  int totalCharges;
  Image xray;

  public PatientRecord(int patientId) {
    this.patientId=patientId;
    totalCharges=0;
  }
}

public class HashPatientRecords implements PatientRecords,
java.io.Serializable {
  private Hashtable ht;
  public HashPatientRecords() { ht=new Hashtable(); }

  public void newPatient(int patientId) {
    Integer key=new Integer(patientId);
    PatientRecord rec=new patientRecord(patientId);
    ht.put(key,rec);
  }

  public void setName(int patientId, String name) {
    Integer key=new Integer(patientId);
    PatientRecord rec=(PatientRecord) ht.get(key);
    rec.name=name;
  }


  public Image getXrayImage(int patientId) {
    Integer key=new Integer(patientId);
    PatientRecord rec=(PatientRecord) ht.get(key);
    return rec.xray;
  }
}
```

References

1. Anderson, M., Pose, R.D., Wallace, C.S. A Password-Capability System, The Computer Journal, 29,1, pp.1-8, 1986.
2. Atkinson, M.P., Jordan, M.J., Daynes, L. Spence, S. Design Issues for Persistent Java: a Type-Safe Object-Oriented, Orthogonally Persistent System, Proc. 7th Intl. Workshop Persistent Object Systems, Cape May, 1996.
3. Bell, D.E., LaPadula, L.J. Secure Computer Systems: Mathematical Foundations, Mitre Corp., Bedford, Ma., Technical Report, ESD-TR-73-278, 1973.
4. Fairthorne, B. et al., eds. Security White Paper, OMG TC Document, 1994.
5. J. Gosling, B. Joy and G. Steele, The Java Language Specification, Reading, MA: Addison-Wesley, 1996.
6. J. Leslie Keedy, K. Vosseberg Persistent Protected Modules and Persistent Processes as a Base for a More Secure Operating System, Proc. 25th Hawaii International Conference on System Sciences, IEEE Computer Society Press, S. 747-756, 1992
7. Morrison, R., Brown, A.L., Carrick, C. et al. The Napier Type System, Proc. 3rd Intl. Workshop on Persistent Object Systems, Newcastle, 1989.
8. Mullender, S.J., Tanenbaum, A.S. The Design of a Capability-Based Distributed Operating System, Computer Journal, 29,4, pp.289-299, 1986.
9. Mowbray, T.J. & Zahavi, R. The Essential Corba - Systems Integration Using Distributed Objects, Wiley, New York, 1995.
10. Rosenberg, J., Abramson, D. A. The MONADS Architecture: Motivation and Implementation, Proc. First Pan Pacific Computer Conference, p. 4/10-4/23, 1985.
11. Sun Microsystems Inc. Java Remote Method Invocation Tutorial, http://java.sun.com/docs/books/tutorial/rmi/index.html
12. Wilkes, M.V., Needham, R.M. The Cambridge CAP Computer and its Operating System, North Holland, 1979.