Poseidon House
Castle Park
Cambridge CB3 0RD
United Kingdom

TELEPHONE: Cambridge (01223) 515010
INTERNATIONAL: +44 1223 515010
FAX: +44 1223 359779
E-MAIL: apm@ansa.co.uk

ANSA Phase III

Design and Implementation of a Persistence Service for Java

Scarlet Schwiderski

Abstract

A persistence service has been implemented as part of the Reflective Java project in the ANSA programme. This document describes the design and the implementation of this persistence service. The purpose of this work is to provide a persistence service for the Java programming language, and to demonstrate and evaluate the benefits of Reflective Java.

APM.1940.02	**Approved**	27th January 1997
	Technical Report	

Distribution:
Supersedes:
Superseded by:

Copyright © 1997 Architecture Projects Management Limited
The copyright is held on behalf of the sponsors for the time being of the ANSA Workprogramme.

Design and Implementation of a Persistence Service for Java

Design and Implementation of a Persistence Service for Java

Scarlet Schwiderski

APM.1940.02

27th January 1997

The material in this Report has been developed as part of the ANSA Architecture for Open Distributed Systems. ANSA is a collaborative initiative, managed by APM Limited on behalf of the companies sponsoring the ANSA Workprogramme.

The ANSA initiative is open to all companies and organisations. Further information on the ANSA Workprogramme, the material in this report, and on other reports can be obtained from the address below.

The authors acknowledge the help and assistance of their colleagues, in sponsoring companies and the ANSA team in Cambridge in the preparation of this report.

APM Limited

Poseidon House
Castle Park
CAMBRIDGE
CB3 0RD
United Kingdom

TELEPHONE UK	(01223) 515010
INTERNATIONAL	+44 1223 515010
FAX	+44 1223 359779
E-MAIL	apm@ansa.co.uk

Copyright „ 1997 Architecture Projects Management Limited
The copyright is held on behalf of the sponsors for the time being of the ANSA Workprogramme.

APM Limited takes no responsibility for the consequences of errors or omissions in this Report, nor for any damages resulting from the application of the ideas expressed herein.

Contents

1	**1**	**Introduction**
2	**2**	**Overview of the Persistence Model**
2	2.1	The Key Issue
2	2.2	The Objectives
4	**3**	**The Implementation**
4	3.1	Existing Java Technology
4	3.2	The Object Serialization API
4	3.2.1	**Usage**
5	3.2.2	Limitations
5	3.3	The Architecture
6	3.4	Implementation Options
7	3.4.1	Option One
7	3.4.2	**Option Two**
8	3.4.3	Option Three
8	3.5	The Activation and Deactivation Processes
9	3.5.1	Activation
9	3.5.2	Deactivation
9	3.5.3	**Cancellation**
9	3.6	The Persistence Metaobject
9	3.7	The Persistence Daemon
11	**4**	**Examples**
11	4.1	The Person Demonstration
11	4.2	The Crossword Demonstration
12	**5**	**Summary**

1 Introduction

A persistence service has been implemented as part of the Reflective Java project [WS96] in the ANSA programme. This document describes the design and the implementation of this persistence service. The purpose of this work is

1. to provide a persistence service for the Java programming language.
2. to demonstrate and evaluate the benefits of Reflective Java.

The motivation for providing a persistence service for Java is to make Java objects survive “normal” program execution, and to make Java applications failure resilient. We would like to achieve this by supporting one persistent store for Java objects at each computing site, usable by any number of applications running concurrently at this site. Other issues such as security, consistency, and redundancy are to be tackled. The main goal is to provide a powerful persistence service and make it easy to apply for end users.

Persistence can be regarded as a non-functional requirement of an application. Therefore, using Reflective Java seems suitable for implementing the persistence service. The advantage is that persistence can be provided to the end user with causing only little overhead, and without changing the Java language. Further, the end user can use his application program with any persistence service implemented according to the MOP (MetaObjectProtocol) approach, and change the binding to the persistence service at any time, even dynamically at runtime. The current binding depends on application requirements, for example, the required security model or method of serialisation.

This document is organised as follows. Section 2 describes our model of a persistence service for Java. After discussing the design, Section 3 illustrates the implementation according to the Reflective Java paradigm. Specific implementation problems are pointed out and provide necessary feedback for the work on Reflective Java as such. Two example applications have been implemented, which are outlined in Section 4. The document concludes with a summary.

2 Overview of the Persistence Model

2.1 The Key Issue

The main concern of our persistence model is to maintain a persistent store (*POS: Persistent Object Store*) for Java objects in order to

- make Java objects reusable by the same or by different applications
- provide failure resilience to applications

For this purpose, the state of a Java object and sufficient information on its type need to be kept in the POS. The following section identifies the objectives and presents the overall model of the persistence service.

2.2 The Objectives

The following objectives for a persistence service for Java objects can be identified:

1. writing Java objects to the persistent store
2. reading Java objects from the persistent store
3. supporting concurrent access to the persistent store
4. providing security measures
5. ensuring consistency of stored Java objects

In order to make an application object persistent, the application object and all objects that are reachable from that object (the whole hierarchy of objects) need to be stored in the POS. Reusability of an application object can be achieved by writing it to the POS at the end of program execution, whereas providing failure resilience implies that the application object has to be written when it is updated and in a consistent state.

A persistent Java object is to be read from the POS when the state of the corresponding application object needs to be restored, that is, at the beginning of program execution (for root-level objects in the object hierarchy) or whenever the object is needed throughout program execution (for lower-level objects in the object hierarchy). In order to read an object from the POS, the correct object has to be identified; its type has to correspond to the type of the Java object to be restored and a specific instance has to be chosen.

We want to support one POS for each computing site. Applications running concurrently at this site should be able to access the POS concurrently, if they are not interfering with each other (that is, if they are not accessing the same objects in the POS). Having applications accessing the POS concurrently means there needs to be one central persistence manager. Hereafter, we will call this persistence manager *persistence daemon*.

In order to avoid forgery, the POS needs to be protected from unauthorised access. One way to provide security is to encrypt Java objects as they are

written to the POS and decrypt them as they are read from the POS. Also according to the encapsulation rule, applications should only be able to access the POS through the persistence daemon. Further, there should always be at least one valid copy of a Java object. Hence, a shadow copy is written to the POS before the main copy is written. This ensures that there is always at least one valid copy of an object, even if the system failed while writing the main copy.

One way of ensuring consistency is to keep one and only one main version of a Java object. Having applications working concurrently with the same object would raise the risk of having two different, inconsistent versions. Hence, concurrent access can only be allowed in a controlled manner.

We attempt to achieve all these objectives without changing the Java language. In this way, there will only be little overhead for the end user. Also, if there is existing Java technology which achieves some of those objectives, we attempt to reuse this technology, rather than implementing everything from scratch.

Figure 2.1: Model of the Persistence Service

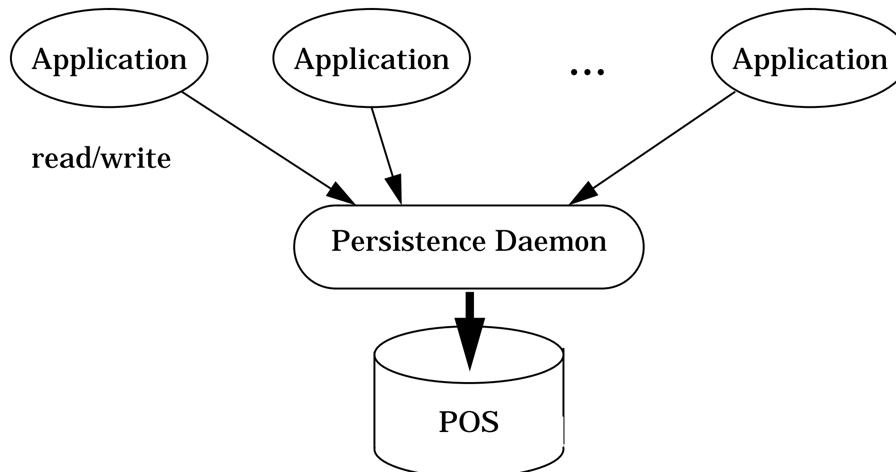


Figure 2.1 presents our model for the Java persistence service. Multiple applications contact the site's persistence daemon in order to read/write Java objects. The persistence daemon coordinates the access to the POS; it is responsible for naming and locating objects and provides concurrency control.

3 The Implementation

Persistence can be regarded as a non-functional requirement of an application. Hence, *Reflective Java* [WS96] is suitable for the implementation of a persistence service for Java. Using Reflective Java, persistence can be provided to applications without causing a large overhead for end users.

3.1 Existing Java Technology

There are two Java APIs which can be used for the implementation of the persistence service:

- Object Serialization API (Sun) [OS96]
- Cryptix API (Systemics) [CRYP96]

The purpose of the Object Serialization API is to support remote method invocation, that is, provide the means for marshalling and unmarshalling object arguments; objects and all objects that are reachable from that object are stored in a flat file, type checking is provided.

The purpose of the Cryptix API is to support encryption and decryption of objects. Different protocols are supported, for example SHA, and IDEA.

3.2 The Object Serialization API

3.2.1 Usage

- Standard usage - read:
  - create an object input stream `oin`
  - make object class serialisable (by implementing `java.io.Serializable`)
  - `object = oin.readObject();`
- Standard usage - write:
  - create an object output stream `oout`
  - make object class serialisable
  - `oout.writeObject(object)`
- Advanced usage - read:
  - provide own `readObject` method for the object class (signature: `private readObject(ObjectInputStream stream) throws ...`), using `defaultReadObject` for achieving default deserialisation behaviour
  - apply `readObject` as above
- Advanced usage - write:

- provide own `writeObject` method for the object class (signature: `private writeObject(ObjectOutputStream stream) throws ...`), using `defaultWriteObject` for achieving default serialisation behaviour
- apply `writeObject` as above

3.2.2 Limitations

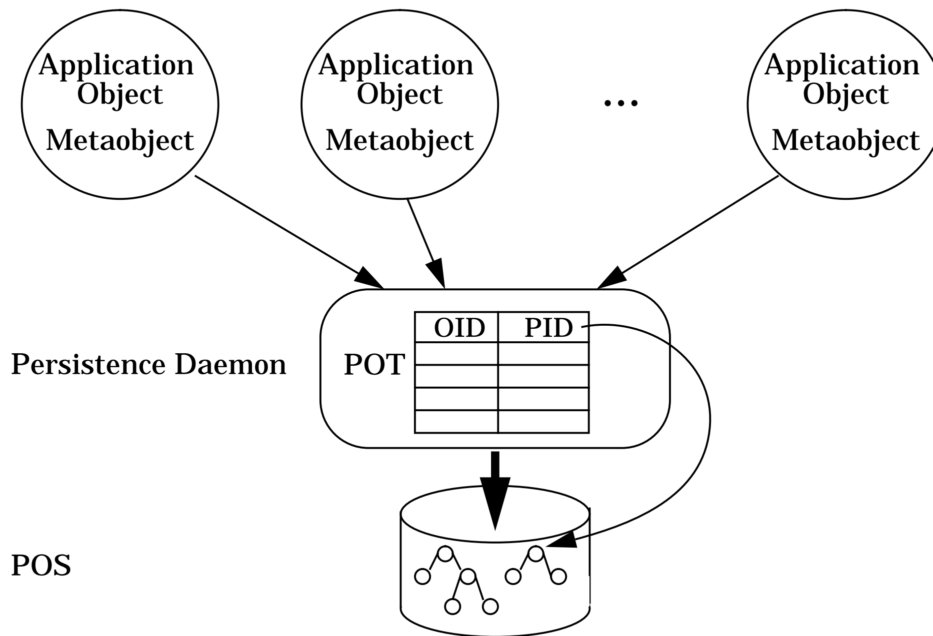
The purpose of Sun's Object Serialization API is to support remote method invocation. Therefore, there are a number of limitations with respect to supporting persistence:

1. When an object is serialised (i.e. `writeObject` method), the object and all objects that are reachable from that object (the whole object hierarchy) are written to one flat file. That is, there is no way in splitting object serialisation up, keeping each object of the object hierarchy in a separate file and therefore allowing it to be reused by other applications.
2. When an object is deserialised, the `readObject` method always returns a new object. That is, there is no straightforward way of returning only the state of a serialised object and copying it to another object. Hence, `readObject` cannot be applied within the constructor of an object class to restore its old state.
3. There is the possibility to provide own `readObject` and `writeObject` methods. Within these methods `defaultReadObject` and `defaultWriteObject` can be used to perform standard object deserialisation and serialisation. However, `defaultReadObject` and `defaultWriteObject` only work if they correspond to the first statements in `readObject` and `writeObject` respectively. The motivation for this is to allow users to append serialisation information at the end of a serialised object only. Moreover, the user-defined `readObject` and `writeObject` methods are `private`, that is, they are not visible outside the object.

3.3 The Architecture

The architecture of the persistence service is shown in Figure 3.1. Each application object is attached to a *persistence metaobject* which is responsible for reading Java objects from the POS and writing Java objects to the POS. Hereafter, reading a Java object from the POS via the persistence daemon is called *activating*, and writing a Java object to the POS via the persistence daemon is called *deactivating* the Java object.

Figure 3.1: Architecture of the Persistence Service



The persistence daemon is a well-known server at the local site. It maintains the POS and coordinates applications in accessing Java objects in the POS. The responsibilities of the persistence daemon include managing multiple applications concurrently, allocating disc space for new Java objects, locating Java objects, and providing security measures. The latter consist in allowing access to the POS via the persistence daemon only, encrypting Java objects as they are written to the POS, and writing shadow copies of Java objects. In order to fulfil these tasks, the persistence daemon maintains a hashtable, called *persistent object table (POT)*, containing the unique name [LINDEN93] of an object (i.e. the OID), its location in the POS (i.e. the PID), and its current state, that is, whether the object is currently activated and if, by which application.

3.4 Implementation Options

Figure 3.2: Basic Reflective Java Paradigm

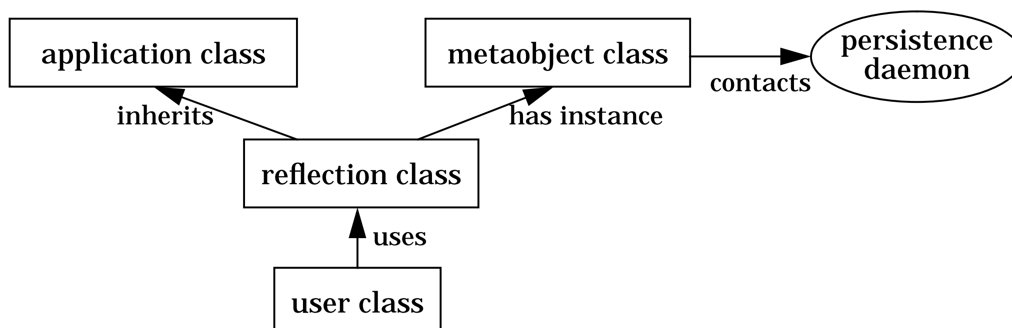


Figure 3.2 presents the basic Reflective Java paradigm [WS96]; the application class is to be made persistent. As stated earlier, the metaobject class is responsible for activating and deactivating Java objects. It does so by consulting the local persistence daemon.

In Reflective Java, method calls are made reflective, that is, the method arguments and the method's return value can be manipulated in the `metaBefore` and `metaAfter` operations of the metaobject class.

3.4.1 Option One

Complying with the Reflective Java paradigm, the first option we investigated attemps to make specific methods of the application class reflective. For activating an object from the POS the constructor, and for deactivating an object to the POS the `finalize` method seemed suitable. However, reflection works by manipulating the parameters of method calls.

1. The constructor - activate
   In the metaobject class, we would need the unique name of the object to be restored, create that object, and return it to the reflection class. However, the unique name is no parameter of a constructor and therefore not available in the metaobject class. Also, there is no way of returning the new object to the reflection class, since `this` (the object itself) is no parameter of a constructor and `metaBefore` and `metaAfter` do not have an appropriate return type either.
2. The `finalize` method - deactivate
   In the metaobject class, we would need the unique name of the object to be deactivated plus an object reference. Although the object reference is available (as part of the standard data structure of metaobjects), the unique name is not.

On the whole, this option is not suitable for implementing the persistence service for Java.

3.4.2 Option Two

As stated above, the user can provide own `readObject` and `writeObject` methods for serialisable objects. The second option is providing those methods in the reflection class and the end user using those methods through the standard deserialisation and serialisation mechanism in the user class. The idea is that the end user would use a default/null stream to call `readObject` and `writeObject` respectively and this stream parameter would then be manipulated in the metaobject class/persistence daemon before the default de/serialisation is performed. We have encountered the following problems with this approach:

1. In this way, the reflection class is made persistent and not the application class, because it is not possible to isolate the superclass from the class itself.
2. In the reflection class' `writeObject` method, the stream variable has to be manipulated before `defaultWriteObject` is called. However, since `defaultWriteObject` has to correspond to the very first statement in the method body, this is not possible.
3. This approach is not completely transparent to the end user, since s/he has to create a default/null stream and call the de/serialisation methods.

On the whole, this option is not suitable for implementing the persistence service for Java.

3.4.3 Option Three

Figure 3.3: Adapted Reflective Java Paradigm

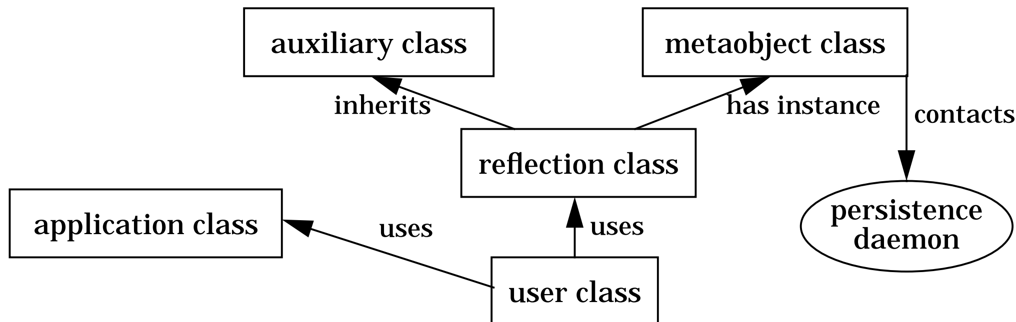


Figure 3.3 visualises the third implementation option, in which a simple auxiliary class `pers_service` is made reflective instead of the application class itself. `pers_service` provides four basic methods for using the persistence service:

1. `public Object activate(String name, boolean read_only)`
2. `public void write(Object oid, String name)`
3. `public void deactivate(Object oid, String name)`
4. `public void cancel_activation(String name)`
   This method is used when an exception occurs in an end user program after activating an object. Hence, the activation for the object is cancelled and its old state remains.

The auxiliary class `pers_service` defines these methods with empty method bodies. The real behaviour is achieved by binding the auxiliary class to a metaobject class implementing persistence. The “empty” class `pers_service` is available to any end user working with Reflective Java. However, for using the “real” persistence service an appropriate persistence metaobject class has to be obtained.

1. This approach does not correspond to the basic Reflective Java paradigm, since the auxiliary class and not the application class is made reflective.
2. This approach is not completely transparent to the end user, since s/he has to use the auxiliary class `pers_service` in order to use the persistence service.

Since options one and two are not suitable for implementing the persistence service, this approach has been taken. Despite of the problems mentioned above, the service is easy to use and causes only little overhead to the end user.

3.5 The Activation and Deactivation Processes

When the end user wants to utilise the persistence service (implemented regarding option three), s/he can use the following commands:

3.5.1 Activation

- `<object> = pers_service.activate(<name>, <read_only>)`
  - `<read_only> = true?` reads the object named `<name>` from the POS and returns it to the end user for read access only
  - `<read_only> = false?` reads the object named `<name>` from the POS if it is not currently activated, activates it, and returns it to the end user for write access

3.5.2 Deactivation

- `pers_service.write(<object>, <name>)`
  writes `<object>` to the POS and stores it under `<name>`
- `pers_service.deactivate(<object>, <name>)`
  writes `<object>` to the POS, stores it under `<name>`, and deactivates it

3.5.3 Cancellation

- `pers_service.cancel_activation(<name>)`
  cancels the current activation for `<name>` (necessary, if an exception is raised in the end user program)

3.6 The Persistence Metaobject

When a persistence metaobject is created, it links to the local persistence daemon. Hence, the services of the persistence daemon are made available to the end user. Calls to the persistence service are then passed from the end user program via the persistence metaobject to the persistence daemon. In each case, the persistence metaobject adds the identifier of the current application to the other parameters, in order to identify where the request originates.

3.7 The Persistence Daemon

When a persistence daemon is created, it registers with the rmi registry [RMI96] and creates an empty POT. Applications can then use the services of the persistence daemon.

- Activation:
  The persistence daemon receives a unique name (i.e. the name of an object), a boolean (indicating read or write access), and an application identifier from some persistence metaobject. Its reaction depends on whether the object is to be retrieved for read or write access. In the first case, the persistence daemon locates the corresponding persistent object, decrypts it, and returns it to the metaobject. Therefore, no update is made to the POT. In the latter case, the persistence daemon evaluates the information in the POT before taking any further action. If the persistent object is currently activated by some other application, the evaluation blocks until it becomes available. Otherwise, the corresponding persistent object is located, decrypted, and the entry in the POT is updated, that is, the persistent object is activated for this specific application. Then it is returned to the metaobject.

- Deactivation:
  The persistence daemon receives a unique name (i.e. the name of an object), an object reference, and an application identifier from some persistence metaobject. First, it checks whether the application is authorised to deactivate the object, comparing the application identifier with the one stored in the POT. If the application is authorised, the object is encrypted and a shadow copy is written to the POS before the main copy is written. Finally, the POT is updated, that is, the object is deactivated.

Writing an object is similar to deactivating it, except that the object's entry in the POT is kept active. Cancelling an object's activation means deactivating the object without writing a new version to the POS.

4 Examples

Two examples have been investigated for demonstrating the persistence service for Java, one regarding the storage of simple person objects, and one regarding the storage of more complex data structures, namely crosswords. These examples are sketched in the following two sections.

4.1 The Person Demonstration

The purpose of this example is to demonstrate the basic use of the persistence service, keeping in mind that the service can be used concurrently by different applications. Simple `person` objects, consisting of a `name` and an `age` attribute, are to be stored in the POS. These objects can be retrieved for read or write access. The persistence service has been tested extensively with several applications running concurrently, manipulating the same or different Java objects in the POS.

4.2 The Crossword Demonstration

The crossword demonstration constitutes a more complex example of using the persistence service. A crossword server runs at the local site and displays a crossword (Times Two Crossword No 955) together with the clues. The client consists of two player windows for manipulating the crossword. RMI [RMI96] is used to propagate updates from one of the player windows to the crossword server.

The persistence service for Java is employed in order to make the crossword persistent. A partially solved crossword is written to the POS whenever a new update is made, and deactivated when the crossword server is closed down. It will then be reloaded when the crossword server restarts.

The purpose of this demonstration is to illustrate how the persistence service deals with more complex data structures, depending on numerous other objects.

5 Summary

This document described the design and the implementation of a persistence service for the Java programming language. The purpose of this persistence service is twofold; first, to provide a powerful mechanism for making applications persistent, tackling problems such as the concurrent access to persistent objects, security, and consistency, and second and most importantly, to show and assess the use of Reflective Java.

By using Reflective Java, we have achieved a number of positive features. First, we are able to provide a persistence service to Java applications with causing only little overhead to end users. The end user can exploit the persistence service with adding as few as three commands (two for read only applications) to his application program (one for creating the persistence service, one for activating an object, and one for deactivating it), keeping in mind that all persistent objects must implement the `java.io.Serializable` interface. Therefore, no specific knowledge on persistence is required for the end user. Second, we have achieved great flexibility. The end user can bind to any persistence service implemented according to the MOP approach and even change that binding dynamically at runtime. The current binding will depend on the execution environment and other, application-specific requirements. Third, a persistence service implemented according to the MOP approach is generic, that is, can be used by any application. Finally, it is to mention that the work on the persistence service has given valuable feedback to the work on the Reflective Java preprocessor.

On the other hand, we have encountered some problems with implementing the persistence service according to the MOP approach. It was not possible to implement the persistence service exactly as in the original model (see Figure 3.2); the model had to be slightly adapted (see Figure 3.3). As a result, the persistence service is not completely transparent to the end user. Although, as mentioned above, the overhead could be kept to a minimum. Some problems are due to our basic reflection model, providing only behavioural reflection and not structural reflection. However, persistence affects the structural aspects of an object (that is, the state) and its behavioural aspects. Also, the implementation is based on Sun’s Object Serialization API. Some problems occurred because of the limitedness of the provided methods. For example, it was not possible to split the serialisation of an object hierarchy up, so that different objects would be written to different files. Hence, lower-level objects cannot be reused by different applications, which may result in redundancies/ inconsistencies.

References

CRYP96]

Systemics, *The Systemics Software Library*, http://www.systemics.com/.

[LINDEN93]

van der Linden R. J., *The ANSA Naming Model*; **AR.003.01**, APM Ltd., Cambridge U.K., February 1993.

[OS96]

JavaSoft, J*avaTM Object Serialization Specification*; JavaSoft, November 1996.

[RMI96]

JavaSoft, *JavaTM Remote Method Invocation Specification*; JavaSoft, November 1996.

[WS96]

Wu Z. and Schwiderski, S., *Design of Reflective Java*; **APM.1911**, APM Ltd., Cambridge U.K., December 1996.