FlexiNet Architecture

ARCHITECTURE REPORT

ANSA

FlexiNet Architecture

Richard Hayton
&
ANSA Team

February 1999

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 Citrix Systems (Cambridge) Limited on behalf of the companies sponsoring the ANSA Work Programme.

The ANSA initiative ran from 1985 to 1998. Further information an the ANSA Work Programme, the material in this report, and on other reports can be obtained from the addresses 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.

Citrix Systems (Cambridge) Limited

Poseidon House
Castle Park
CAMBRIDGE
CB3 0RD
United Kingdom

TELEPHONE UK	(01223) 515010
INTERNATIONAL	+44 1223 515010
FAX	+44 1223 359779
EMAIL	andrew.herbert@ansa.co.uk
WORLD WIDE WEB	http://www.ansa.co.uk

CONTENTS

FIGURES

PART ONE:
OVERVIEW

1 INTRODUCTION

1.1 Background

This document is the final ANSA Technical Report. It describes the FlexiNet Java middleware platform forming the final deliverable of the ANSA Phase III Work Programme to the ANSA Consortium.

ANSA is an industry sponsored programme of collaborative research, advanced development, market development and standards generation in the field of distributed systems. Started in 1985 and completed in 1998 ANSA has been influential in academic research and ISO, ITU, OMG and W3C standards. The ANSAware system was an important precursor of Object Request Broker (CORBA) technology.

Much of the work on FlexiNet was additionally supported by the FollowMe ESPRIT project [FAST 87].

The aim of the FlexiNet project was to show how the concepts of component-oriented system could be applied to produce efficient re-configurable, extensible middleware platforms. FlexiNet provides both a generic binding framework and a set of engineering components to populate the framework. By making appropriate choices of which components are assembled within the framework a variety of different middleware facilities can be achieved including mobile objects, persistent objects, secure objects and transactional objects. FlexiNet represents an evolutionary step from contemporary CORBA [OMG97a] and Java Remote Method Invocation (RMI) [SUN96] and Enterprise Java Bean (EJB) [SUNc] middleware. It is a full demonstration of the ANSA architectural principles at work.

FlexiNet is particularly suited to applications that are deployed in a variety of different contexts (e.g., on the Internet, in Intranets or Extranets) since its enables the infrastructure to be tailored for the specific needs of each deployment and for inter-operability between application components in different environments.

FlexiNet is built in Java because of its portability, object-orientation, facilities for dynamic linking, reflection and introspection and increasing role as the language of choice for distributed applications development. Java is the language that currently comes closest to supporting the ANSA computational and engineering models.

1.2 Capabilities

The heart of FlexiNet is a remote procedure call (RPC) system for remote invocation of services implemented as interfaces on Java objects. In this respect FlexiNet is similar to CORBA and RMI. FlexiNet allows both object-by-value and interface-by-reference parameter passing. It uses introspection and dynamic code generation and linking in place of off-line stub generation.

The implementation of the RPC infrastructure is based in terms of sets of components called *binders*, which implement RPC semantics over underlying transport systems. In addition to FlexiNet specific binders, there is an IIOP binder that implements the OMG IIOP standard protocol to enable inter-working with CORBA clients and servers.

A simple Trader is provided to enable FlexiNet clients to locate FlexiNet servers.

A basic object location service is provided, principally to support the mobile object workbench (see below). A design for a more advanced version is presented.

A class repository sub-system is provided which enables classes to be dynamically linked across networks and locally cached, with advanced facilities to manage name clashes that might arise in federated environments.

A mobile object workbench sub-system is provided. This allows “clusters” of Java objects to move from one “place” (computer) to another. This sub-system was developed to enable an investigation of mobile agents in the FollowMe project [BURSELL98].

A persistent information space sub-system is provided. This allows clusters of Java objects to be removed from the execution environment and placed in storage until next activated. The persistent information space provides an “object file system” for the mobile agents of the FollowMe project.

A visual application builder and associated infrastructure components are provided to enable transactional Enterprise Java Beans to operate in the FlexiNet environment. These facilities enable transactions to be used to enable objects to manage concurrent access and recover from failures transparently. The FlexiNet transaction infrastructure is more powerful than current EJB implementations because it includes the transaction facilities within the infrastructure rather than offloading transaction control to a database management system.

Security is provided primarily in the form of an implementation of the SSL protocol as an additional binder. This can be used both for secure access control and secure communication. Strong encapsulation is an intrinsic feature of the FlexiNet framework. In addition, there is a design for secure carriage of mobile objects across trust boundaries.

Basic support for high availability is provided as a binder for a reliable multicast protocol between members of an object group. This implementation is currently unfinished, but includes sufficient components to give a proof of concept.

Finally a declarative configuration tool for assembling components into binders using the FlexiNet framework called “Blueprints” is provided.

1.3 Structure of this Report

The report consists on nine sections. It is assumed the reader is familiar with Java, distributed processing concepts and the broad principles of middleware design. The aim of the document is to give a complete, high level introduction to FlexiNet, it’s principles and design rationale as a companion to the source code distribution.

The sections are as follows:

I. **Introduction** – this section, consisting of background, description of capabilities and high level overview

II. **Technical Overview** – an introduction to the FlexiNet framework, its major components and sub-systems (mobile objects, persistent objects, transactional objects / EJBs), from an application developers / system designer’s perspective

III. **Architecture** – the detailed description of the FlexiNet binding framework from a binder implementor’s perspective.

IV. **Engineering Components** – detailed descriptions of individual binders that fit into the FlexiNet framework including those for basic RPC, secure communication and object groups

V. **Clusters and Capsules** – the management architecture for the FlexiNet mobile object and persistent object sub-systems

VI. **Managed Objects** – detailed description of the three FlexiNet sub-systems for mobile objects, persistent objects and transactional objects

VII. **Services** – detailed description of the services supporting FlexiNet – Trader and Class Repository

VIII. **API and Examples** – a summary of the principle FlexiNet API’s and simple examples of FlexiNet components

IX. **Advanced Topics** – the designs for mobile object security and advanced object relocation.

2 MOTIVATION

The FlexiNet Platform is a Java middleware system built 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 at all levels. This allows programmers to tailor the platform for a particular application domain or deployment scenario by assembling strongly typed components. The FlexiNet component developer operates within the language type system and is saved from having to conduct the book-keeping that would otherwise been needed to remember relationships between components.

In the following chapters we give an overview of the FlexiNet architecture, highlighting how its approach differs from other middleware architectures and illustrating the benefits that result from our approach. The core of FlexiNet is the binding mechanism by which components are linked together and our approach to encapsulating distributed objects.

The FlexiNet platform grew out of dissatisfaction with industrial middleware platforms. We therefore begin the description of FlexiNet with a review of the limitations of existing middleware.

Generally, research middleware platforms provide application programmers with facilities for just one model for distributed programming, for example remote procedure call, or message passing or process groups. Consequently, compact, efficient and scalable implementations are often achieved. By contrast, industrial middleware platforms address the need for a ubiquitous infrastructure and provide an integrated set of capabilities including, for example, transactions, replication, authentication, privacy, auditing and others. The result is typically monolithic, inefficient and complex.

Since different applications require different combinations of middleware features, a compositional approach in which only the middleware services needed by an application need be made available is appropriate. In CORBA [OMG97a], for example, a set of nested choices is offered by the CORBA Object Services. Each Object Service extends the core Object Request Broker with additional capabilities such as persistence and transactions. The benefit of the CORBA framework of Object Services is that it is comprehensive. The disadvantage is that it is unnecessarily rigid because the order in which capabilities have to be assembled is fixed and this rules out some implementation choices. Moreover, the core Object Request Broker is required to contain support for the data structures and protocols required by each Object Service whether it is used or not. Thus, in addition to causing bloat and inefficiency in the implementation, developers are forced to manage more capabilities than they necessarily need in any particular situation.

A further aspect of inflexibility comes from the use of stubs in Object Request Brokers to provide access transparent invocation. A stub converts an invocation into an untyped byte array representation to be passed on to a communications layer in the case of a remote service. Discarding language level typing and introspection facilities in this way, makes it hard to provide developer-written protocols and mechanisms that can coexist with standard stubs. Specifically, it can be difficult to tie together application level events, middleware events and communications events. For example, The Iona ORBIX Object Request Broker provides filters and transformers [IONA97] as a means to modify how communication events are handled. However, no conventions or data structures are defined for relating filter events (i.e., pre-stub events) to transformer events (i.e., post-stub events). Behaviour at the filter level is modelled by CORBA type codes and dynamic type checking of these has to be managed by the developer rather than delegated to the programming language.

Inherent in the design of distributed systems is the need to make appropriate trade-offs between the competing goals of abstraction and application control. Abstraction in middleware is generally associated with distribution transparency. Abstraction/transparency makes life easier for developers by hiding the engineering details of interaction models behind a generic invocation interface (e.g., method invocation). In essence, the infrastructure manages distribution. Application control, by contrast, allows developers to optimise the infrastructure when it is beneficial to do so, for example by providing heuristics for error cases. Control requires that implementation aspects of a distribution transparency should be exposed. Unfortunately current systems either impose a ‘one size for all’ transparency or expose the low level ‘systems’ mechanisms in all their complexity.

3 RELATED WORK

There are many research systems offering flexible binding, particular with respect to performance and resource tradeoffs. Traditionally this work has been driven either from a quality of service perspective [BLAIR97] or from an aim to simplify protocol implementation by building complex protocols out of simpler micro-protocol engines. In these contexts binding can be at the generic buffer and communication channel level. FlexiNet differs in that it provides flexibility at a higher level. In addition to controlling protocol level choices; management of higher level distribution transparency mechanisms based on transactions, replication, security, persistence, mobility and so forth can be managed. In providing these capabilities as components we have to intercept and transform application level invocations and tie together different protocols, transparency mechanisms and system services in a consistent and structured manner.

The key advantage that FlexiNet has over other schemes is the Java Virtual Machine itself. This provides a great deal of the support needed; in particular for introspection (runtime examination and control of types) and reflection (generic invocation of methods). It allows FlexiNet to provide a middleware framework that extends the core language features, and is internally strongly typed – rather than having to separately manage the infrastructure type system as with CORBA (i.e., via typecodes). The disadvantage of course is that effectively FlexiNet is a Java language specific platform. (While there are mappings of other languages to the Java Virtual Machine, none of then can be regarded as main stream implementations). However, by way of mitigation, in distributed Internet applications, Java is a common choice because of its suitability for network programming and platform independence. Since this is the focus of our work, we are prepared to trade the benefits gained against the restriction to a single language.

3.1 Computational Model

FlexiNet uses a different computational model for Java than Javasoft's Remote Method Invocation (RMI) [SUN96]. It adopts the ODP [ISO95] notion of interfaces as the access points for objects, and provides transparent interface proxies. When parameters are passed to a method, references to interfaces are passed by reference, and object values are passed by copying. This enables FlexiNet to follow Java language model as closely as possible, without introducing 'special' tag classes to indicate remote interfaces, or value objects, as is done in RMI. The benefit of determining the parameter semantics by choosing between an object or an interface (for copy or reference semantics respectively) within the context of a particular call can be seen when mobile objects are considered. Using the tagged object approach, each object would be statically classified as either a server or a data object. It is therefore not possible to have the concept of a mobile server object – since it would be tagged as a ‘server’ and could never be transmitted as data when moving from one location to another.

4 SELECTIVE TRANSPARENCY

Since FlexiNet remote method invocation is has similar semantics to local method invocation, we have access transparency at the lowest level. This uniformity helps keep application code separated from 'systems' code, making it easier to move applications from one environment to another. To take control, the application programmer can inject particular mechanisms, both at runtime using an explicit binding facility and at design time by controlling the mixture of protocols and transparency components used. Resources can be managed by restricting the allocation policies for, and sizes of, resource pools assigned to selected components. This capability is described as 'selective transparency', since the developer can choose how strongly system components are tied to (and visible to) application components. Binding decisions can be taken directly by system components or handed off to third parties where this is appropriate. The former is appropriate for autonomous systems, the latter for managed infrastructures (e.g., a trusted computing base).

4.1 Bindings

An interface on a remote object is represented in FlexiNet by a local proxy object. Typically, this is a simple stub object that turns a typed invocation into a generic (but fully typed) form and then passes the request to the top layer of a protocol stack. FlexiNet stubs are very lightweight compared to other systems, and in particular the stub is not responsible for the 'on-the-wire' representation of the invocation, and embodies no implicit or explicit policy about how the call will be handled. Since stubs are so simple, stub classes are generated on demand within client processes, by introspection on the interface definition, and link them dynamically. A stub class can be shared by all protocols that treat service objects in a similar way.

FlexiNet protocol stacks are correspondingly more complex than those of a traditional 'heavyweight stub' system, since we make them responsible for all aspects of call processing. This includes high level features, such as management of replication, in addition to basic actions such as the serialisation of invocation parameters and execution of a remote procedure call protocol.

The layers of a FlexiNet communication stack can be viewed as a set of reflective meta-objects. Each meta-object in turn manipulates the invocation as a data structure before it is ultimately invoked on the destination object using Java Core Reflection [SUNa] (thereby removing the need for a server-side stub or skeleton). Using reflection has a number of advantages. Middleware (or application) components may examine or modify the parameters to the invocation using, and protected by, the Java language typing system. Debugging is made straightforward as information is kept 'in clear'. Splitting omnibus middleware protocols into components is simplified, as the language typing support provides the necessary machinery to ensure consistency of use and to reduce cross dependencies in the code.

Figure 1 illustrates how a communication stack can be assembled as a number of meta-objects that perform reflective transformations on an invocation. Meta-objects can be fully general Java objects and are fully type-safe. For example a replication meta-object might extract replica names from an interface and then perform invocations on each replica in turn. As this processing is performed in terms of generic invocations, there is no need for each of these calls to pass through stubs and so the code can be both straightforward and efficient.

At the top of the client side stack, an invocation consists of the abstract name of the destination interface, the method to be invoked, and the parameters to the method as an array of objects. Interface names are arbitrarily complex objects that can be resolved by a protocol stack to provide a route from the client to the destination interface (or interfaces for replicated objects).

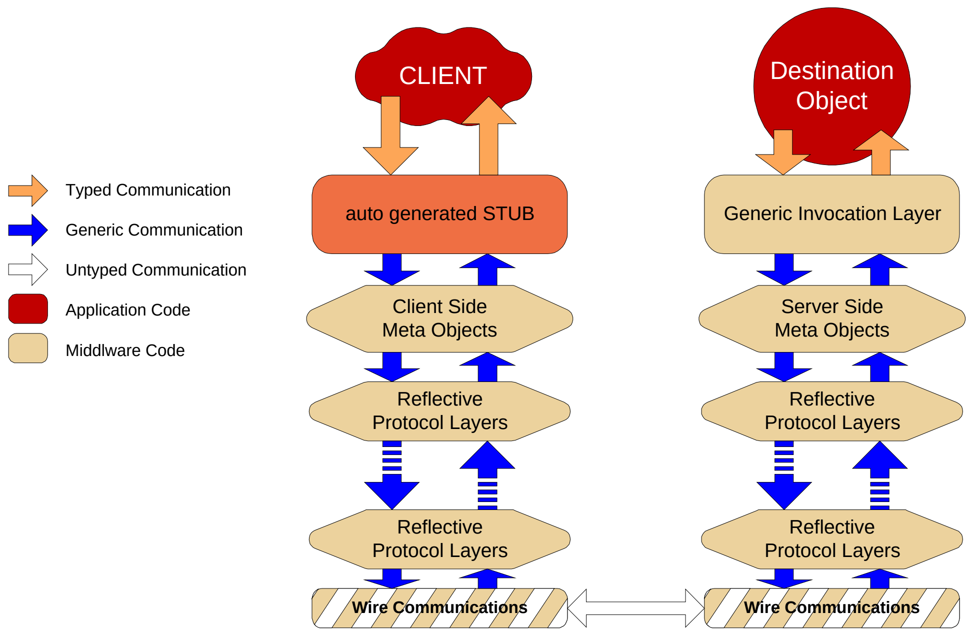


Figure 1 A Reflective Protocol Stack

As the call proceeds down the stack, each layer can manipulate the invocation. By the bottom of the stack, the original abstract name will have been resolved to an appropriate endpoint or connection identifier, and sufficient information will have been serialised into a buffer to allow reconstruction of the invocation on the server.

On the server, the reverse process takes place, so that ultimately the destination object, method and parameters are available.

Many RPC protocols maintain state across a number of calls, for example a UDP based protocol may keep a record of unacknowledged replies, and a TCP based protocol might maintain a connection. FlexiNet provides *sessions* as an abstraction for managing this information, so that a stack can retain client-related information over the duration of a number of calls. Sessions also provide an in-call mutex for use by the layers to ensure that per-client resources are cleanly allocated and freed across a number of essentially independent components. Using this mutex, conflicts between communication events on the way up the stack, and application events on the way down can be avoided. Other ANSA work has found that a suitable session structure greatly eases this kind of post-hoc protocol modification and its omission from CORBA and RMI is a considerable oversight.

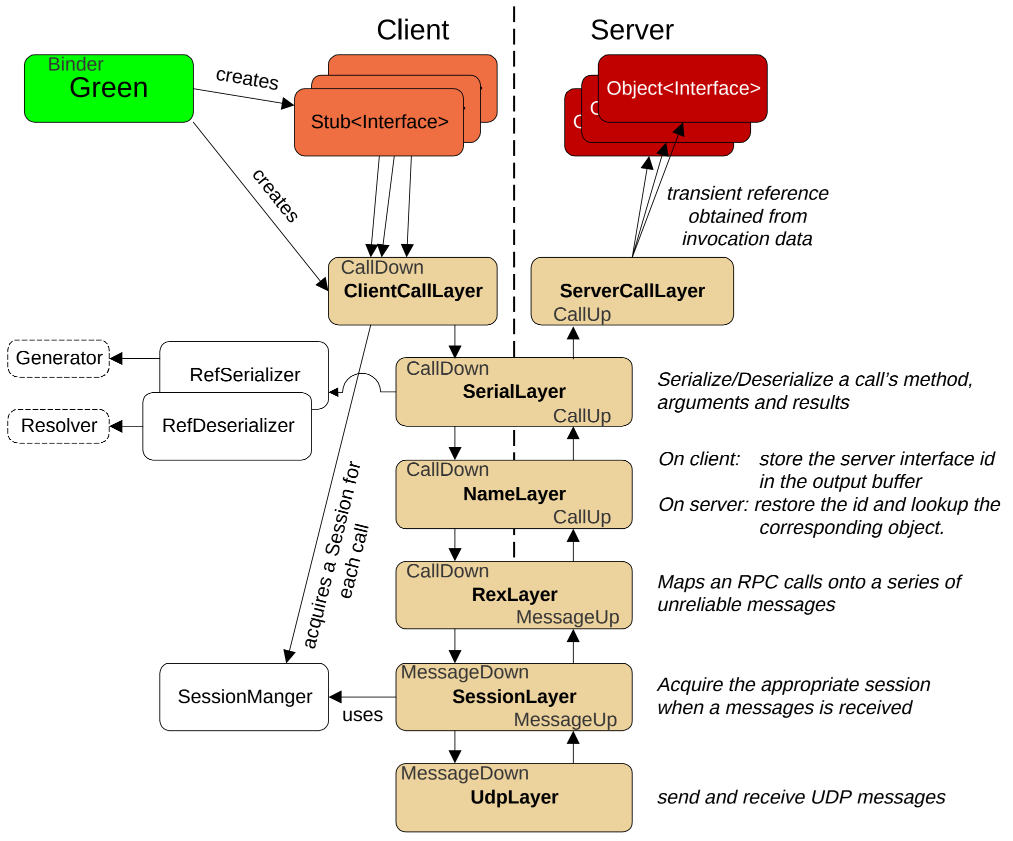

Client
Server
Binder
Green
creates
Stub<Interface>
Object<Interface>
creates
transient reference obtained from invocation data
CallDown
ClientCallLayer
ServerCallLayer
CallUp
Generator
RefSerializer
Resolver
RefDeserializer
CallDown
SerialLayer
CallUp
Serialize/Deserialize a call's method, arguments and results
CallDown
NameLayer
CallUp
On client: store the server interface id in the output buffer
On server: restore the id and lookup the corresponding object.
acquires a Session for each call
CallDown
RexLayer
MessageUp
Maps an RPC calls onto a series of unreliable messages
SessionManger
uses
MessageDown
SessionLayer
MessageUp
Acquire the appropriate session when a messages is received
MessageDown
UdpLayer
send and receive UDP messages

Figure 2 The Green Binder and Protocol Stack

4.2 An Example Protocol

Figure 2 gives an overview of the 'Green' protocol stack in the FlexiNet implementation. 'Green' provides simple remote method access using the REX RPC protocol over a UDP transport. It is an interesting example as it shows how aspects of call based, and message based exchanges are handled. Following the progress of a call from the client stub to the server object and back:

1. Initially a call is made on a client stub. The nature of this call will depend on the semantics of the interface, and the stub is responsible for converting this into a generic representation that may later be executed using Java Core Reflection. This de-couples the type of the object being invoked from the implementation of the protocol stack, and enables reuse of standard reflective components.

2. The stub will then call the top of the protocol stack. At this stage the arguments to the call, and the method class are available, and reflective classes may be called. For example, the arguments may be validated, or modified. There may be a number of reflective layers, for example to multiplex the call over a number of replicas, or perform other client-side processing. In the Green protocol, however, there is no reflective tinkering, and the call passes directly to the Client Call Layer.

3. The Client Call Layer acquires an appropriate session on which to perform the call. Sessions group a number of invocations that share the same service endpoint, to allow various optimisations. Other layers may utilise the session structure to cache information relating to a particular endpoint (for example encryption keys).

4. There may be additional reflective layers to further manipulate the call. After this manipulation, the call passes to the Serial Layer, which serialises (marshals) the method and parameters into an output buffer.

5. The Name Layer comes next and extracts de-multiplexing information from the name of the interface being referenced (e.g., an interface id), and stores this in the output buffer. The subpart of the name used to locate the peer layer in the server is then passed, together with the other call parameters, to the next layer down. This separates the different levels of multiplexing, keeping the system modular.

6. The REX layer acts as a gateway between reliable call and return method invocation, and unreliable one way messaging. The REX layer contains sufficient state to manage lost or duplicate messages, and utilises the session structure to map a series of message onto a number of invocations.

7. There may again be a number of additional (per message) layers (for example to encrypt or compress the outgoing message, before it reaches the session layer). The session layer stores information to allow the session to be re-established on the server. Using the mutex, the Green protocol stack ensures that only one call or message per session will be in progress above the session layer. This simplifies the coding of session related functions, and reduces the number of race conditions (such as duplicate messages or simultaneous messages and timeouts) that the programmer must deal with.

8. Finally, the UDP Layer sends the message as a single UDP packet to the server.

9. On the server, the message is received in the UDP Layer. A new thread is started to listen for further messages, and the thread that received the

message is sent up the stack to process the request. When it arrives in the REX layer, it is identified as a new request, and a timeout is set for acknowledgement (REX piggy-backs acknowledgements on replies). The message is converted to a call, and is passed up to the Name layer, which reads the interface id, and identifies the object being called. The call arguments and method are de-serialised in the next layer, and then the method is invoked on the service object by the Call Layer. The result of the invocation (normal or exception) unwinds the call stack: in the serialisation layer, the result is serialised, and in the REX layer, a message containing the result is constructed. The message is passed down to the Session Layer and treated in an identical way to any other outgoing message (whether it represents a request, reply or protocol message).

10. On the client, the REX layer eventually receives a reply message that it can pair with the original request. The original thread is then woken and unwinds up the stack. Finally, the result is returned to the client using standard Java means.

4.3 Binders

The discussion so far has described how a binding to a remote interface is represented, and how an invocation may be processed. In an invocation, it may be necessary to serialise references to local and remote interfaces. During deserialisation, proxies must be constructed to represent these exported references. This is the mechanism by which all bindings (other than an initial built-in reference to a trader) are constructed.

The object responsible for generating names for interfaces is called a ‘Generator’. The object responsible for resolving names is called a ‘Resolver’. The more familiar term, Binder, is used to refer to either generators or resolvers. A typical binder will both generate names and convert names generated by other (compatible) binders into the (stub, stack) pair previously described. A FlexiNet binder is therefore a factory for bindings.

In many systems, there is exactly one binder per process, however in FlexiNet we needed to support multiple binders and binding protocols, with possibly conflicting use of names. Each protocol stack therefore contains a reference to the generator and resolver to be used for generating and resolving the names of interfaces passed as arguments to invocations, or returned as results from invocations.

Typically, binders are arranged into a hierarchy, in order to factor out common functionality (such as the caching of previous bindings), and to allow a dynamic selection of the binder to perform a particular binding. An example binder graph is illustrated in Figure 3. This illustrates two ‘basic binders’, Green, as described above, and Red, which generates bindings using IIOP over TCP. There are three additional binders. Two are ‘Cache’ binders that store tables of previously resolved bindings. The third is a ‘Choice’ binder which dynamically chooses whether to use Green or Red based on the type of the interface being named, or the type of the name being resolved. For example, imagine that the choice binder has been initialised to always use

Green when generating names unless explicit QoS parameters are specified requesting that Red should be used (perhaps indicating that the interface will be passed to a CORBA client). When resolving names, Green or Red will be used as is appropriate.

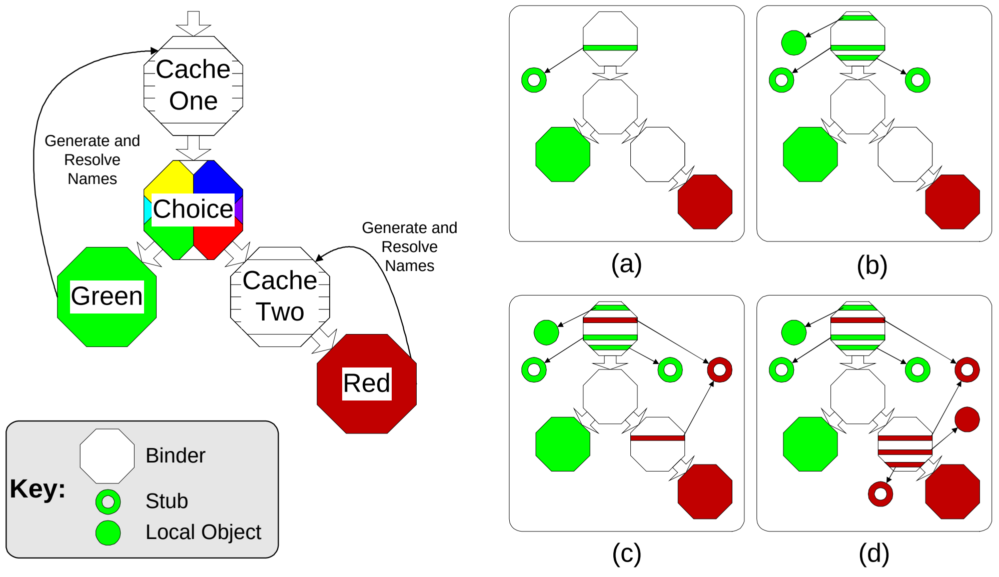


Figure 3 A Hierarchy of Binders

Figure 3(a-d) illustrates a possible execution path. In (a), the process has a single reference to a remote object, resolved using Green. Invocations may be made on this object, and in the process of this, stubs to additional local objects may be generated, and additional local objects may be named. By default Cache One and the Green binder will be used for this (b). If during one of these calls, a red (IOR) name needs to be resolved, Choice will select the Red binder to perform the resolution (c). When using this newly resolved interface, any local interfaces referenced will be named using the Red binder (d). This is essential, as Red uses the CORBA IOR name format, and the remote CORBA interface may not understand FlexiNet Green names.

An alternative arrangement of binders could lead to all interfaces being named by a tuple of a Green name and an Red name, allowing another process to choose which protocol to use. It is even possible to dynamically decide on the protocol to be used to name a particular interface. To illustrate this feature a ‘negotiation binder’ has been constructed that generates placeholder names for remote interfaces. When these are bound, a complex negotiation process is entered into, at the end of which, a binder is chosen to perform the actual binding. This approach is invaluable if different protocols have to be used depending upon the location of the client and the server. For example, to choose a specific security protocol based on the legalities of specific encryption technology or key lengths in the client’s domain.

To take this approach to an extreme, the binder graph could be augmented dynamically whenever a previously unknown protocol was encountered. This is perfectly feasible given Java’s class loading mechanisms. The real problem is one of controlling the number of different binders that might be required if such a scheme were to be adopted.

5 DISTRIBUTION TRANSPARENCIES

The RM-ODP framework identifies nine distribution transparencies. Of these, only two, *Access* and *Location* relate directly to remote invocation. In addition to these, FlexiNet protocols may provide varying degrees of *Failure*, *Replication* and *Security* transparency via meta-objects in the protocol stack. The remaining RM-ODP transparencies, namely, *Migration*, *Relocation*, *Persistence* and *Transaction,* cannot be tackled in this way. Instead, they require some notion of *encapsulation*, whereby all interactions with an object, or group of objects can be monitored and controlled. To achieve this FlexiNet implements the RM-ODP notion of a *cluster*. A cluster is the primitive RM-ODP engineering unit of encapsulation.

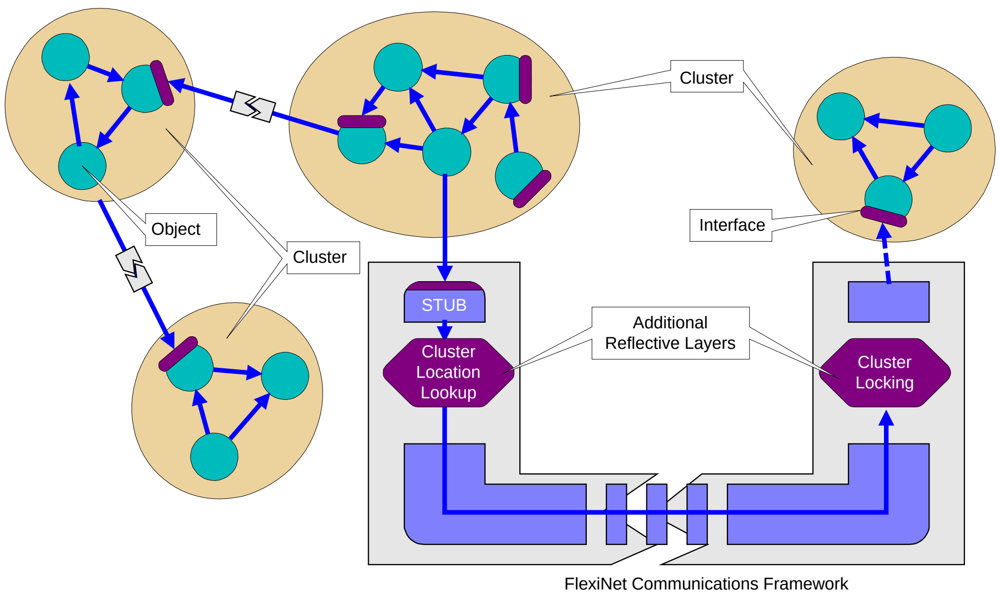


Figure 4 Encapsulation Using FlexiNet

Clusters are illustrated in Figure 4. All externally referenced interfaces in a cluster are accessed via a FlexiNet protocol stack. We arrange that when a thread in one cluster invokes a method in another cluster, we de-couple the threads so that the callee and caller cannot adversely affect one another by blocking or thread termination. Additionally each cluster is effectively given a separate Java security manager, and class loader. Thus each cluster becomes a 'virtual process' that is de-coupled from all other clusters in terms of name spaces privileges, code base and management. Clusters cannot examine the internals of each other, nor may arbitrary methods on objects in one cluster be called from another without mediation by the reflective layers in the protocol stack. These co-operate with a distinguished cluster management interface associated with the cluster to provide whatever kind of distribution transparency is appropriate. In the case of a mobile object the cluster manager arranges for the atomic transfer of the cluster from one location to another, and the protocol stacks include some kind of relocation function to track objects as they move.

Clusters are a container abstraction. By placing objects within a cluster, they may be given additional ‘non-functional’ behaviour. Cluster implementations have been built that provide migration, relocation and persistence transparencies.

Container abstractions are often associated with component-oriented systems. The core Java component abstraction is Java Beans. In particular the Enterprise Java Beans specification (EJBs) gives a template for components designed to run in a transactional container.

The FlexiNet cluster abstraction provides a higher degree of isolation than (standard) EJB containers, and relaxes some of the constraints placed on a component programmer. In particular a bean will ‘escape’ from an EJB container unless the following rules are followed:

- EJB methods must not perform thread operations. The current container/context is associated with the thread id. Any operations that change this will lead to erroneous behaviour.
- EJB methods must not pass references to (parts of) themselves. EJBs have a (single) public interface which is must be accessed via a proxy. If an EJB were to pass a reference to (an object within) itself, then this proxy would be bypassed.
- EJBeans share references to objects passed between each other on a local machine. This feature must be used with great care. If an EJB container decides to ‘rollback’ or recreate a bean due to transactional conflict, then the shared object is likely to be replaced by a copy. The exact behaviour will depend on the actions of the container.

For FlexiNet clusters, these issues are taken care off by the cluster mechanism.

The FlexiNet transactional framework is designed to use the EJB container model, rather than the cluster model. The motivation for this is primarily industry conformance. EJBs facilitate the reuse of standard third party components; if our transactional framework were built on a proprietary abstraction, then this advantage would be lost. Now that the EJB specification is more mature, it would be an interesting exercise to construct an EJB compliant cluster. However, if a bean were to take advantage of the less restricted environment that this would afford, it would cease to be portable to other EJB implementations.

5.1 Transactions

The trend in Internet applications is to move from browsing to real-time transactions with business critical information. Examples include online banking, order/entry, and customer service. Transactions have been used in online processing on mainframe systems for many years. However, there are some distinguishing characteristics of Internet applications, which change the way in which transactional applications are constructed and deployed.

Thin clients. In traditional client/server computing, an application-specific client needs to be pre-installed on the user machine to run an application. In Web-based Internet applications, the runtime components are downloaded from the Web site. Such a thin client model delivers two key benefits: universal access, and reduced installation and management costs.

Scale. Unlike traditional applications, Internet user communities can extend well beyond department or company. With these new "self-service" applications, access to a server becomes open to thousands of users all executing transactions simultaneously. This requires highly scaleable server architectures to support transactional applications.

Rapid development. Many corporations have started using Internet for publishing and collecting information, and intend to doing business over the Internet. Therefore, the technology for building Internet application systems must be very easy to use, develop and deploy.

All these characteristics of Internet applications requires system architectures that are scaleable, flexible and adaptable, whilst still easy to use, to develop and to deploy. “Three-tier” or “multi-tier” architectures are now emerging to address the needs of Internet applications in terms of scalability and dynamic access. In a three-tier architecture, most of an application’s logic is moved from the client to one or more servers in the middle tier. This provides a number of benefits. Server components can be replicated and distributed across many servers, to boost system availability. Server components can be easily modified to adapt to changing business rules and economic conditions, thus providing flexibility. Server components are also location independent, if they are built using distributed objects (e.g. CORBA), therefore system administrators can easily reconfigure system load.

In this new model, users find and launch applications on HTML pages at Web servers. Instead of simply loading a static page, a dynamic "applet" is downloaded to the individual’s browser. The applet bring with it protocols that allow the applet to communicate directly to application servers running in the middle tier. These servers access data from one or more databases, apply business rules, and return results to the client applet for display. This makes the middle tier the single most critical component of the emerging Internet application architecture from a developer’s point of view.

There are three popular architectures for building middle tier components: CORBA-style Object Request Brokers (ORBs), Transaction Processing Monitors, Web Application Servers. Although each has its strengths, none of them is ideally suited for the middle tier requirement of Internet transaction processing.

CORBA ORBs [OMG97a] have excellent multi-tier capabilities with strong distributed object invocation and related infrastructure services such as transactions and security. Unfortunately, the complexity of the overall solution and a lack of strong tool support limits their appeal to sophisticated developers. Additionally, most ORBs also have primitive server-side execution engines, limiting performance and scaleability.

TP monitors, on the other hand, have robust and mature execution engines that deliver excellent performance and scaleability. However, like ORBs, their overall complexity and proprietary APIs often make them difficult to use and expensive to install, administer, and maintain.

Web Application Server technology emerged in an attempt to transform Web servers to application servers. Web Application Servers are generally customer generated from one of several Web or site development tools. This strong tools focus leads to high developer productivity. On the flip side, scaleability is severely limited by the application server's direct tie to Web servers and the lack of non-HTTP protocols for application-to-application communication.

To address the need for a scaleable and easy-to-use middle tier, component-based transaction servers are emerging, such as Sybase's Jaguar CTS [SYBASE], and Microsoft Transaction Server [MICROSOFT]. They combine the best features of ORBs and TP monitors with component-based develop tools. This enables quick creation of scaleable applications. Component-based transaction servers offer built-in transaction management capabilities, and support distributed object invocation for multi-tier application communication. They also support rapid middle tier development and provide an execution environment for server components.

Our transaction architecture also aims to meet the needs of Internet transaction applications. Like Jaguar CTS and MTS, it is component-based, thus enables users to develop portable, customisable components, and assemble them into applications. It enables rapid application development and deployment using standard components and off-the-shelf tools. The architecture supports implicit transactions, removing from developers any concern for transaction management details. Any component installed in the server is a candidate for participation in a transaction. More importantly, no component in a transaction need concern itself with the behaviour of other components in regards to their effect on the transaction.

Unlike Jaguar CTS or MTS, our transaction architecture is also reflective, providing two additional features that are important for supporting Internet transaction processing. First, it allows the transaction infrastructure to be easily adapted to new application requirements and changing environments. For example, it allows application users to choose a particular concurrency control protocol for their application. Secondly, it allows programmers to provide application-specific information declaratively and separately from application code. This information can be used either at deployment time for configuring the transaction infrastructure to best suit the application component, or at execution time for improving system performance.

Our work started before any publication of Enterprise JavaBeans (EJB) [SUNc]. However, it turned out that they have similar goals: to provide implicit transactions thus removing from the developers any concern for transaction management details. We adjusted some of our design and implementation when the draft of EJB's specification was published to ensure that our transaction architecture fulfils EJB's specification. The final implementation of the architecture can be used as an EJB container to execute Enterprise JavaBeans.

The results of our work include a runtime execution environment, and a development toolkit. The execution environment consists of an underlying transaction system and an EJB container. The development toolkit provides a visual tool (`EnterpriseBeanBox`) for users to customise both the beans and runtime container. The container insulates the enterprise Bean from the specifics of an underlying system by providing a simple, standard API between the bean and container. The `EnterpriseBeanBox` is an extension of the BeanBox from the BDK [SUNd]. It maintains all the original functionality, but gains some new features to meet our special requirements.

5.2 Component-Based Software

In recent years, constructing applications through the assembly of re-usable software components has emerged as a highly productive way to develop custom applications.

The term component-based software is used to describe a software model, which specifies how to develop reusable software components and how these component objects can communicate with each other. A component is an encapsulated piece of code that can be combined with other components and with hand written code to rapidly produce a custom application. A component is designed to be used within another application, called a container, and designed to be reused and customised without access to its source code. A container provides an application context for components and provides management and control services to the component it stores.

In order to qualify as a component, the application code must provide a standard interface that enables other parts of the application to invoke its functions and to access and manipulate the data within the component. This is often termed "introspection" and enable the application developer to make full use of the component without requiring access to its source code. Components can be customised to suit the specific requirements of an application through a set of external property values. This is often called "reflection".

Server components are application components that run on a server. In a three-tier architecture, most of an application's logic is partitioned into separate server components to be deployed on a server system. A server component container provides a runtime environment to support the execution of server components. In our case we combine the robust runtime features of a traditional transaction processing monitor with the flexibility and reusability features of distributed components. The container provides the complex management services that are required to support high-volume business transactions, including multithreading, resource-pooling, and transaction co-ordination. The introspection and reflection facilities allow the container to take design time and runtime policy on how to couple its servers to application components.

5.3 Reflection and the Metaobject Protocol

Reflection [MAES87] is the capability of a computational system to reason about and act upon itself. Unlike conventional system, a reflective system allows users to perform computation on the system itself in the same manner as in the application, thus providing users with the ability to adjust the behaviour of the system to suit their particular needs.

In an object-oriented programming environment, reflection can be realised in the form of metaobjects that represent some internal information and implementation of the system. The interfaces of these metaobjects are called *metaobject protocols* (MOPs) [KICZALES91], because they allow application objects to communicate with metaobjects. Through MOPs, users can modify the systems' behaviour and implementation incrementally.

Using metaobject protocols, the actual behaviour of an application object is determined not only by itself, but also by the metaobject which it is associated with. The association can be thought of in terms of a binding between the application object and the metaobject. An application object can obtain the capability of a metaobject by binding to it. In this way, the functionality of an application is determined by its application objects, whilst the quality of application delivery is determined by the associated metaobjects. The quality of application delivery can be changed through alternative metaobjects without making changes to application objects. This makes it possible to provide system capabilities to an application program transparently and flexibly.

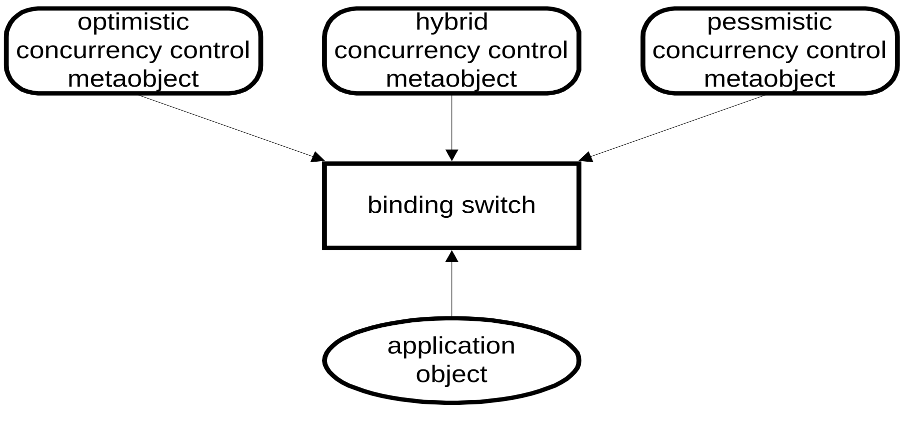

optimistic concurrency control metaobject
hybrid concurrency control metaobject
pessmistic concurrency control metaobject
binding switch
application object

Figure 5 The Metaobject Protocol Approach

For example as illustrated in Figure 5, an application object can become usable in a concurrent environment by binding to one of the concurrency control metaobjects. There is no need to make any change to the application object. A binding between an application object and a metaobject can be changed dynamically according to the run-time conditions. For example, when the conflict rate of concurrently accessing an application object is low, it would be better to bind it to an optimistic concurrency control metaobject. However, when the conflict rate becomes high, the binding can be switched to a pessimistic concurrency control metaobject. By monitoring its components and applications, the system can perform this switching automatically without disturbing application programs.

5.4 The Computation Model

The characteristics of Internet applications requires system architectures that must be scaleable, flexible and adaptive, whilst still be easy to use, to develop and to deploy. Our reflective component-based architecture meets the rigors of Internet applications by taking advantage of both the reflection technology and the component model.

To achieve the most benefits from the multi-tier architecture, server components should be implemented as shared servers. However, building a shared server is not an easy task. It is much harder than building a single-user application. Usually, shared servers need to support concurrent users, and they need to share system resources, such as threads, memory, and network connections. They also need to participant distributed transactions and enforce security policies.

It would be very hard for application developers, who are experts in business logic, but not necessarily in transaction monitor engineering, to address all these system issues. To solve this problem and to provide portability to application programs, our computation model allows a clear separation to be made between business logic and system issues. This enables application program to focus on application requirements without concern about the system issues. The separation also makes it possible for an application program to be executed in different system environment without making changes to its source code.

Because of the wide range of potential applications, with varying needs, it is impossible to provide a single monolithic application server infrastructure suitable for all applications. The implementation of an application server’s infrastructure must be flexible and adaptive so that it can be customised easily to cater for a particular application. To achieve this aim, our infrastructure represents alternative infrastructure choices as alternative metaobjects, as shown in Figure 6. Thus at deployment time, an application assembler can choose the most suitable infrastructure for his application by selecting the corresponding metaobject. The selected metaobject is then integrated with the application object to form a server component.

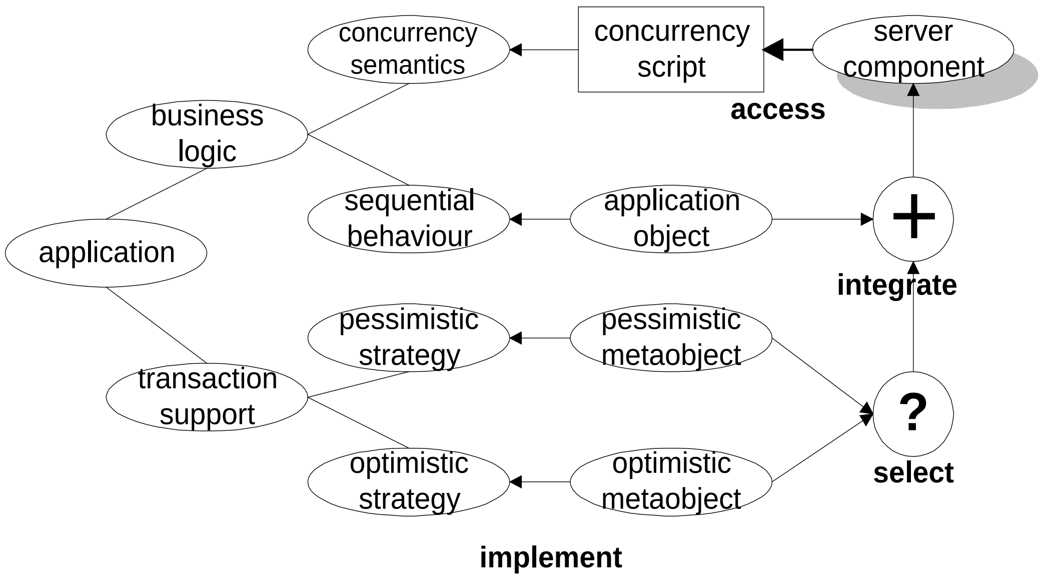


Figure 6 A Reflective Computational Model

A clear separation between the application program and the implementation of system issues is essential for making application component portable. However, without any information about the application, it would be hard for a system to provide a good quality of service to that application. Only when detailed knowledge about an application is available such as ordering constraints for consistency, it is possible for a system to optimise its behaviour and to improve its performance. We solve this dilemma by allowing application developers to specify application-specific information declaratively and separately from application programs. In such a way, we enable application information available to a system, but without increasing the burden of an application developer, nor losing the portability of an application program.

For a transaction system, the concurrency semantics of an application can be used to reduce delay due to blocking. Thus, we allow concurrency semantics to be represented in our model, but it is represented declaratively and separated from the sequential behaviour of an application. The sequential behaviour is implemented in application code; whilst the concurrency semantics are represented as a concurrency script. At runtime, the concurrency semantics would be used by the transaction strategies to schedule operations.

5.5 Concurrency Semantics

By taking into consideration type-specific semantics of operations, a transaction system can allow concurrent executions that would otherwise be forbidden if operations were simply characterised as *reads* and *writes*. The concurrent semantics of operations are usually represented by relationships between operations, such as commutativity [WEIHL85]. Given two operations on the same type, *p* and *q*, we say that they are commute if the result of executing *p* and then *q* on *d* is the same as the result of executing *q* and then *p* on *d*.

Consider, for example, a bank account class, *Account*. It has an associated set of operations: *credit* money to an account, *debit* money from an account, and *check* the balance of an account. In this example, two *credit* operations are commute, so do two *check* operations. Therefore, it is possible for two *credit* operations to be executed concurrently. However, this execution would not be permitted in a system that classified operations into *reads* and *writes* only. This shows a general phenomenon: by taking into consideration type-specific information, a system can permit greater concurrency than would otherwise be possible.

However, utilising concurrency semantics is not an easy task. It would makes the application program complicated, if the concurrency code is intertwined with the implementation of business logic. In order to avoid this, we allow application users to expressing concurrency semantics decaratively and separately from the implementation of the objects.

The concurrency semantics can be represented easily in pairs of operation names. For example, a pair *(p, q)* means that operation *p* and *q* are commutative. In judging whether two operations are commutative, one needs only to consider the logical relationship of the two operations, not the implementation of the operations. Our transaction system will ensure the atomicity of individual operations.

5.6 The Architecture

There are six kinds of entities in our architecture (Figure 7): the server component, application information script, server component container, client component, metaobjects, and underlying supporting system. The server components implement the business logic for an application.

A server component container provides certain system capabilities, such as multithreading, transaction and security. It also provides an application context, management and control service to the encapsulated server components. To provide flexibility and adaptability, a server component container represents implementation strategies in the form of metaobjects. A metaobject can be replaced by a new metaobject that implements the same functionality, but with a different strategy. In each server component container there are a number of “sockets” for plugging in metaobjects. Users can choose “off-the-shelf” metaobjects that are best suited to their applications at deploy time. They can also supply their own metaobjects. Metaobjects can be changed dynamically at runtime to cater for changing environment conditions.

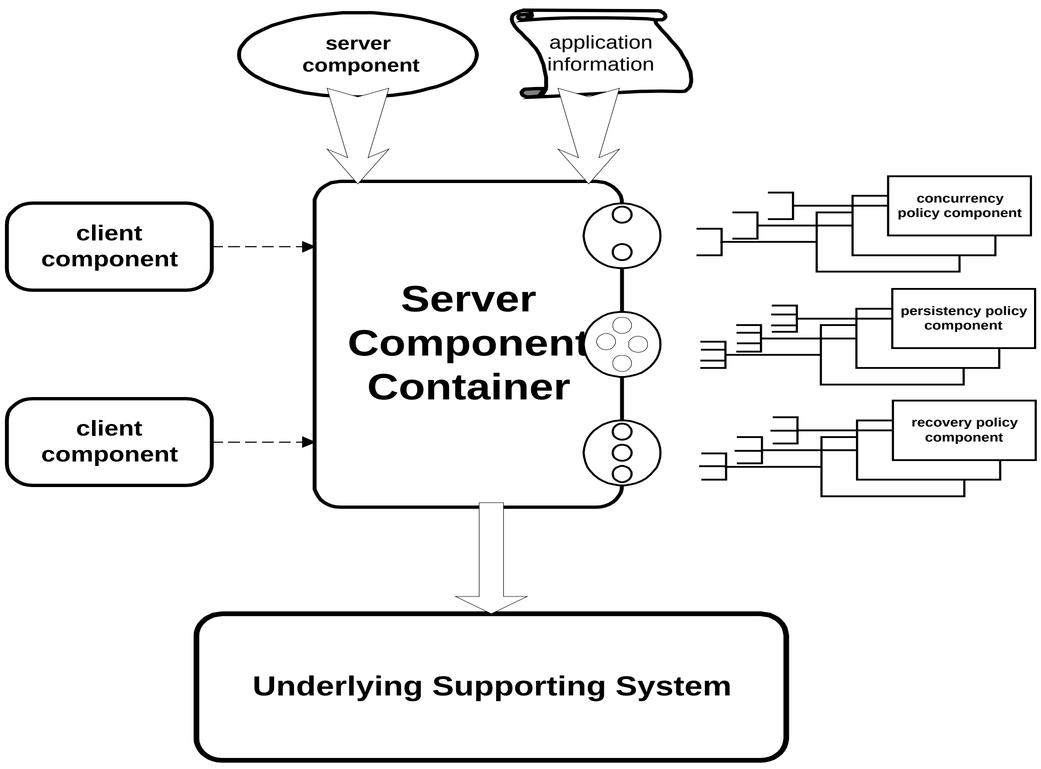

server component
application information
client component
client component
Server Component Container
concurrency policy component
persistency policy component
recovery policy component
Underlying Supporting System

Figure 7 The Reflective Component-Based Transactional Architecture

A server component container insulates server components from the underlying supporting system. The container automatically allocates system resources on behalf of the components and manages all interactions between the components and the underlying system. This ensures that the server components can be run in any system as long as it supports a compatible sever component container.

A server component container maintains control over a server component through a wrapper. A container provides an external representation of a server component. Client components do not directly interact with a server component, but with the external representation. This allows the container to intercept all operations made on the inside server components. Each time a client component invokes a method on a server component, the request goes through the container before being delegated to the target server component. The container can thereby implements system capabilities, such as concurrency control, security and transaction management transparently. The behaviour of the server component container are decided partly by the associated metaobjects.

Server components are built by using a component builder. Through the builder, users can manipulate and customise a server component through its property tables and customisation methods. Users can also assemble a server component with other components to create a new application. Furthermore, they can also attach component-specific information to the component; for example concurrency semantics, deployment policy or concurrency policy. This information may be used by the server component container.

5.7 Structure of the Server Component Container

A server component container manages a server component via intercepting invocations to the component. The interception is implemented through our reflection system. For each server component, a reflection object is generated which provides a client view of the server component. Clients access a server component through interacting with the corresponding reflection object.

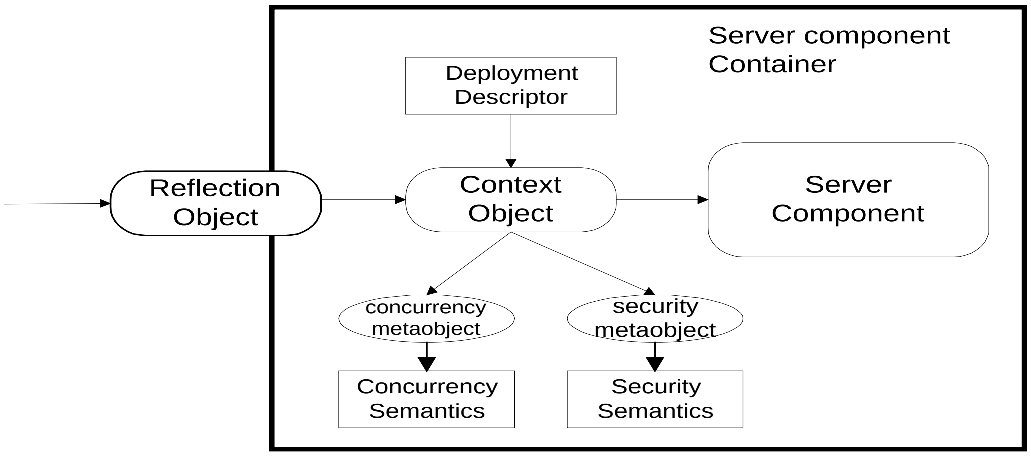


Figure 8 A Server Component Container

For each active server component, the server component container generates a context object to maintain its information, and a number of metaobjects to implement corresponding functionality, such as concurrency control and security checking. The reflection object will interact with the context object that in turn will interact with the corresponding metaobjects at particular points to enforce transaction and security rules (Figure 8).

To enable containers to utilising component-specific information to improve system performance, users can provide these information through scripts at assembly time (see chapter 12). The container will ensure this information is available to be used by corresponding metaobjects. For example, the concurrency control metaobject would use the concurrency semantics of a component to increase the degree of concurrency.

5.8 The Transaction Model

Our transaction model is based on the OMG's Object Transaction Service (OTS) specification [OMG94]. It is a well-defined transaction model, bringing the transaction paradigm and the object paradigm together. A major advantage of the model is that it enables every object to provide its own concurrency control and recovery, thus providing the possibility for an object to apply an individual concurrency control and recovery policy to cater for its specific requirements. However, this advantage is not exploited fully in the OMG's specification. The reflection functionality of our architecture provides the right tool for exploiting this advantage.

OTS supports distributed transactions that can span multiple databases on multiple systems co-ordinated by multiple transaction managers via a distributed two-phase-commitment protocol. Therefore, by using OTS our architecture ensures that a transaction of a server component can inter-operate with other component servers.

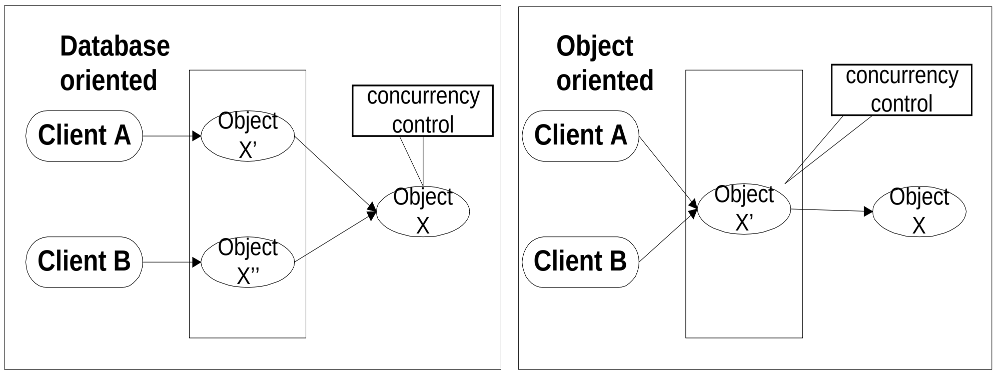


Figure 9 Database-Oriented and Object-Oriented Transactions

Our transaction model is object-oriented, rather than database-oriented. In a database-oriented model, the system component container focuses mainly on robust messaging. It is the database system that is responsible for concurrency control, recovery and persistence. This approach makes it easy to leverage existing database systems and transaction processing monitors to Internet applications. However, most database systems deal with concurrency based on file or records rather than objects. This makes impossible for them to utilise application semantics to improve concurrency control, and hence system performance. It also means that all components stored in a database system can only use the concurrency control method provided by the database system, whenever whether or not it is suitable for their applications.

Another drawback of the database-oriented model is that it keeps unnecessary copies of components in memory. For example, if two clients access a component *X* through a server concurrently, there would be two copies of *X* inside a container, each for one client (Figure 9). This wastes system resources and increases interactions to the database system.

By taking the object-oriented transaction approach, our architecture enables users to choose the most suitable concurrency control method for their application. The server component container automatically uses the selected concurrency control method to implement the transaction services. We also enable users to change the concurrency control method of a server component dynamically at runtime to cater for environment changes.

6 PERFORMANCE

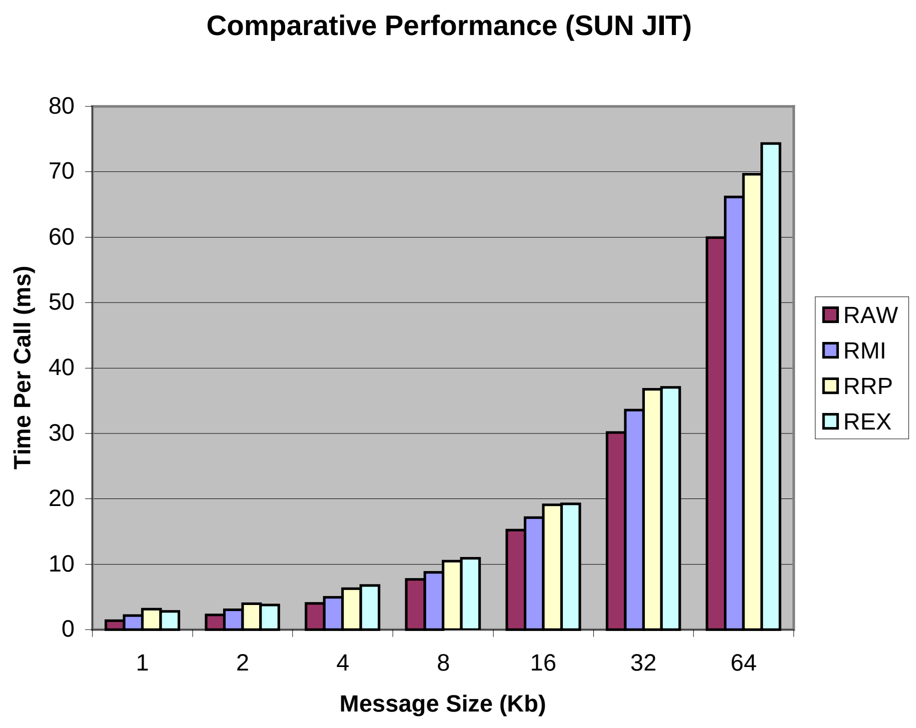
FlexiNet is a component based framework, and the protocols and abstractions that currently populate this framework were designed for modularity and reuse, rather than performance. For example, all the layers in a typical remote method invocation stack could be implemented as one module, in order to increase performance.

However, FlexiNet is fully resource controlled, and uses pools for resources such as buffers and threads, drawing on our earlier experience in C++ with DIMMA [HERBERT98]. The modularity is an advantage here, as different pool management policies may be 'slotted in' in order to trade off performance against resource usage.

Performance is notoriously difficult to measure. Many factors, such as a protocol's support for failure and simultaneous access, in addition to the actual reliability of the connection, and number of simultaneous clients will all effect the achieved performance. However, to give a simple indication of the 'raw' performance of FlexiNet compared to other protocols we ran a series of simple tests between two machines. For these tests, four protocols were used; Sun's RMI, FlexiNet using a TCP based protocol (RRP), FlexiNet using a UDP based protocol (REX) and a 'raw' TCP protocol. This latter protocol acts an indication of the inherent costs of a remote call. It uses simple Java TCP sockets and has a single threaded client and server. For all protocols, an array of bytes was used as the only argument to an invocation, and the invocation returned a void result. The results of running the protocols with different JVMs and different message sizes are shown in Figure 10. For the record, the machines used were Pentium Pro 200s, running NT Server 4 over a 10Mbit Ethernet. To reduce the effect of class loading, compiling and TCP flow control, 100 invocations were made on each connection prior to measuring the performance.

From the graphs the following points can be noted:

- The network time dominates the total time for all calls. None of the protocols is appreciably slower than a raw socket connection.
- For large messages, the UDP protocol was slower, this is due to the need to perform UDP fragmentation. This was particularly pronounced in the Microsoft JVM/JIT.
- The FlexiNet TCP protocol is around 15-20% slower than RMI on Sun's JVM/JIT, but around 10% faster on Microsoft's JVM/JIT. This suggests that Sun's RMI and JVM have been optimised to run well with each other.



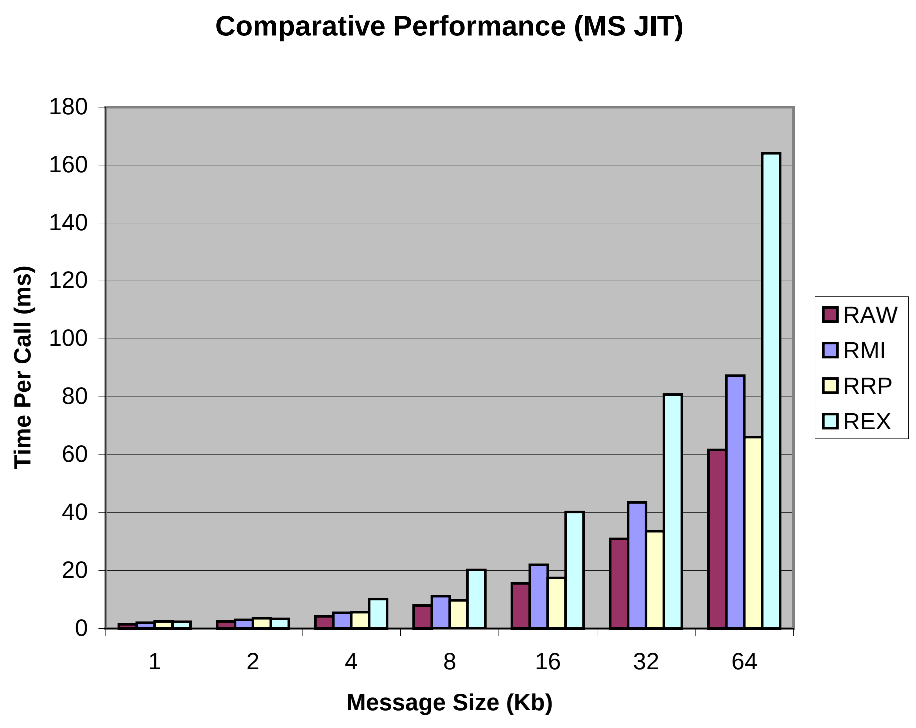


Figure 10 Comparative Protocol Performance

Unlike RMI, which relies on native methods and stub compilers in order to function, FlexiNet remains 100% pure Java, with no external tools required. This makes it highly portable across Java releases and JVM implementations. Future JIT or JVM performance increases should be fully reflected in FlexiNet's performance.

By tuning the combination of layers that make up a protocol, in addition the size and nature of thread and buffer pools, the performance tradeoffs can be tuned to suit the intended environment. Further work is investigating how at least some of the tradeoffs can be made automatically – by monitoring and estimating load and reliability factors. The structure of FlexiNet makes it particularly easy to add orchestration layers to collect this information.

7 SUMMARY

FlexiNet was designed to provide a platform which to perform code deployment and binding related experiments. As such, the emphasis was on modularity and flexible configuration. FlexiNet has made considerable use of language level introspection and has embraced reflective techniques. Not only does the resulting system highly modular, but it also performs as efficiently as other Java middleware offerings. The ability to support all of the ODP distribution transparencies has been an intellectual goal of many distributed system platforms, and the ability to construct these using FlexiNet serves as a powerful example of its extensibility.

8 FLEXINET TEAM

8.1 Organisation

FlexiNet was produced as part of Phase III of the ANSA project, between October 1996 and December 1998. During this time a number of people have worked on FlexiNet. On average, the team has been four to five people.

8.2 People

The following people have been involved with major parts of FlexiNet. In addition to the areas identified, there has been a great deal of discussion on general and specific architecture principles and engineering detail.

Andrew Herbert
: Director of Advanced Technology for Citrix Systems Inc and Chief Architect for the ANSA project as a whole.

Richard Hayton
: Designer and implementer of FlexiNet. Responsible for the majority of 'core' architecture and code.

Dave Otway
: Involved in early design of first binder stack. Worked on security additions to the 'Green' protocol, REX fragmentation, and RMP based messaging.

Laurence Jordan
: Designed and built the Dynamic Stub Generation system. Also wrote the 'Crimson' SSL binder.

Mathew Faupel
: Conducted early experiments including the 'negotiation based' binder. Wrote the IIOP Binder, and heavily involved with discussions on how to make FlexiNet amenable to other standard (badly behaved) protocols.

Will Harwood

Architect for the FollowMe ESPRIT project under which part of FlexiNet was designed. Did early work on the formal composition of components, and involved in the design of the security related aspects of FlexiNet, in particular security for mobile objects.

Mike Bursell

Worked on Mobile Objects in the FollowMe context.

Zhixue Wu

Designed and built the transactional framework.

Takanori Ugai (secondee from Fujitsu)

Worked on mobile object security, and SSL.

Peter Bagnall (secondee from BT Labs)

Wrote the Blueprints system.

Oyvind Haussen (secondee from Tromso University)

Conducted many early binding experiments. Involved in the early evolution of the binding system.

PART TWO: TECHNICAL OVERVIEW

9 BASICS

9.1 Introduction

The foundation of FlexiNet is *remote method invocation*. This is where a process running on one host obtains a reference to an object on a different host, and uses it as if it were a local reference. In particular, to invoke methods on the object. FlexiNet was designed for Java, and every effort is made to make remote method invocation similar to normal, local, method invocation. However remote method invocation does not function identically to local method invocation, for example there may be network failures, and there are therefore differences in the semantics of local and remote calls.

9.2 A Simple Example

Before considering the formal differences between local and remote invocation, consider a simple example. Figure 11 and Figure 12 define two runnable classes, which represent a server and client process respectively. The server is started first; and creates an instance of a `BankImpl` object, which it wishes to make this available to the client. To do this it must give the client a reference to the *interface* on the `BankImpl` object that the client should use. In this case, it is the `Bank` interface (Figure 13). For the moment we will ignore how this is done (shaded).

When the client obtains a reference to the bank, it may invoke any of methods in the Bank interface as if they were local. That is essentially all that a programmer needs to understand to use FlexiNet. For completeness of the Example, the definition of the Bank service (`BankImpl`) is show in Figure 14.

```
public class Server
{
  public static void main(String args[])
  {
    Bank bank = new BankImpl();
    FlexiNet.getTrader().put("Bank of Toytown",bank);
    System.out.println("Server Ready");
    Thread.currentThread().suspend(); // wait for calls
..}
}
```

Figure 11 An Example Server Application

```
public class Client
{
  public static void main(String args[])
  {
    Bank bank = (Bank) FlexiNet.getTrader()
                                  .get("Bank of Toytown");

    System.out.println("create an account with £80");
    bank.credit("Fred",80);

    try
    {
      System.out.println("withdraw 50 pounds");
      bank.debit("Fred",50);
    }
    catch (InsufficientFundsException e)
    {
      System.exit(1);
    }

    try
    {
      System.out.println("withdraw another 50 pounds");
      bank.debit("Fred",50);
    }
    catch (InsufficientFundsException e)
    {
      System.out.println("Overdrawn”);
    }

    System.out.println("check balance (should be 30)");
    System.out.println("balance = " +
                               bank.getBalance(“Fred”));
  }
}
```

Figure 12 A Client of the Bank Service

```
public interface Bank
{
  public void credit(String account,int amount);
  public void debit(String account,int amount)
            throws InsufficientFundsException;
  public int getBalance(String account);
}
```

Figure 13 An Example Service Interface

```
public class BankImpl implements Bank
{
  private Hashtable accounts = new Hashtable();

  public synchronized void credit(String account,int amount)
  {
    int balance = getBalance(account);

    accounts.put(account,new Integer(balance+amount));
  }

  public synchronized void debit(String account,int amount)
                           throws InsufficientFundsException
  {
    int balance = getBalance(account);
    if(balance < amount)
      throw new InsufficientFundsException();
    else
      accounts.put(account,new Integer(balance-amount));
  }

  public int getBalance(String account)
  {
    Integer i = (Integer) accounts.get(account);
    if(i==null)
      return 0;
    else
      return i.intValue();
  }

}
```

Figure 14 An Example Service Implementation

9.3 Specifying an Interface

As has already been hinted, FlexiNet only allows remote invocation of methods specified in an interface. We take the ODP [ISO95] view that interfaces are the natural points of access in a distributed object environment. This allows a server and client to evolve independently, with a well-specified ‘contract’ about how they interact.

There is another practical reason as well; clearly FlexiNet must provide some engineering object on the client that ‘looks like’ the server object, but that uses remote method invocation to implement the methods. The most natural way to provide this is to specify the methods in an interface, and have different implementations for the local and remote case.

FlexiNet currently only supports public interfaces with public methods.

9.4 Call Semantics

For the most part, a remote call may be considered like a local call. In particular, FlexiNet supports arbitrary nesting and multi-threading. However, local and remote calls *are* semantically different.

When a call is made on a remote interface, an essential decision is whether parameters should be passed *by reference* or *by value*. To retain the local method invocation semantics, all parameters should be passed by reference. This means that the callee receives a reference to each parameter object, and may shared use of this object with the caller.

Although this give clear semantics, this approach is neither possible nor desirable. It is not possible as FlexiNet can only support remote references to interfaces – parameters which are not of interface type cannot therefore be supported in a 'by reference' mode. The approach is not desirable for performance reasons. In general, the cost of a remote call is many times that of a local call. Better performance therefore results if small objects are copied onto a remote machine, rather than accessed by remote method invocation. In Java, most data items are actually objects. In the example of Figure 13, the 'account' parameter is an object. It is clearly reasonable to copy this from client to server, and this is what is done.

Most remote procedural call or remote method invocations systems employ a 'usually by value' approach, and FlexiNet is no exception. In Sun's RMI for example, all parameters are passed by value unless they are objects that extend `UnicastRemoteServer`. This approach was rejected in FlexiNet, as it does not allow an object to be passed by value in some cases, and by reference in others. It also does not allow distinct interfaces on objects to be passed, which was required in FlexiNet in order to follow the ODP architecture as closely as possible.

How them may be decide when to pass a parameter by reference, and when by value? Rather than examine the class of the parameter value, we examine the class of the *reference*. In Java, references are strongly typed and we can easily spot the difference between the two types of reference. This is illustrated in Figure 15.

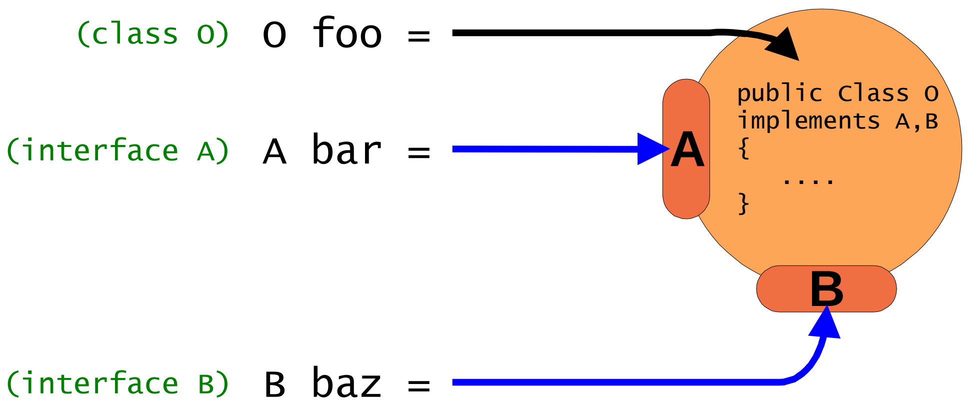


Figure 15 Distinguishing Interface References

9.4.1 Object Graphs

When passing a parameter by value, we must also deal with references to other objects and interfaces held in that object. We must therefore recurse through the graph of objects rooted at each parameter, and decide for each whether to pass it by reference or value. We use the same criteria in all cases; references to interfaces are passed by reference, and references to objects are passed by value. This is illustrated in Figure 16. If two parameters in a single call both refer to the same object (directly or indirectly) then only one copy is passed.

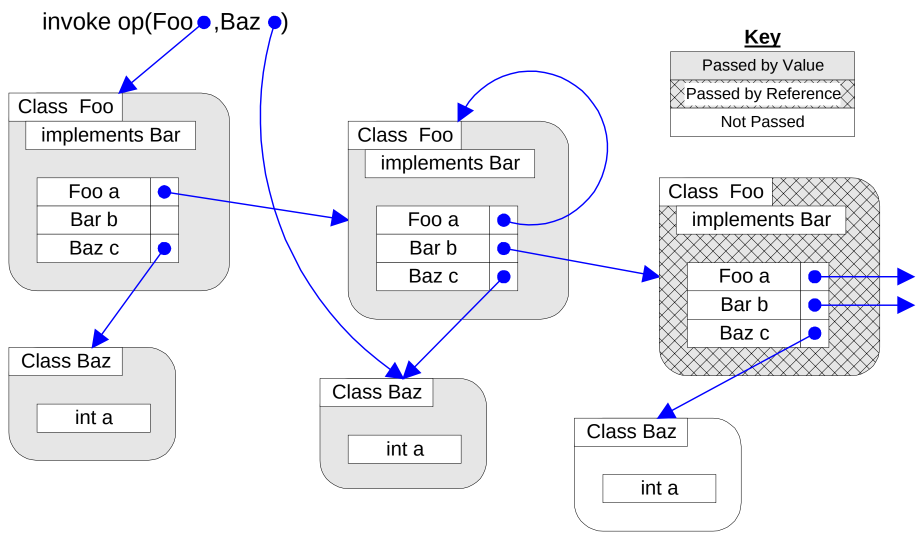


Figure 16 Passing Object Graphs by Value

9.4.2 Error Handling

A remote method invocation may raise any of the exceptions that might be raised if the call were made locally. In addition there may be a number of distribution related errors, such as network or server failure, or failure due to access control or authentication checks. These exceptions will be thrown as `RuntimeExceptions`, so that a programmer need not modify an invocation to catch them specifically. Of course, if the errors occur, and are not caught elsewhere in the program, them the client process will terminate. This approach is the same as that for other environment-related errors in Java, such as running out of memory. Strangely, Sun did not take this approach with its own remote method invocation system (RMI), and instead remote invocations must be ‘polluted’ with code to handle these special cases.

9.5 The Trivial Trader

The Trivial Trader is a simple name service provided to allow clients to locate advertised services. When a FlexiNet client or service starts up, it already has a reference to this service. A server may publish a service by entering a textual name for it into the trader. A client may then obtain a reference to the service by querying the Trader, supplying the same name. This is illustrated in the shaded portions of Figure 11 and Figure 12. Note that the current trader is extremely trivial. It does not attempt to determine if published services are still running, and only allows simple naming of them. It is also not persistent – if it crashes, all entries are lost. In a more complete environment, the trivial trader would be replaced with something more robust

To allow flexibility in the set-up of a FlexiNet environment, the trader is not pre-configured to run at a particular network address. In addition, unlike systems such as the RMIRegistry, it is only necessary to run one instance of the FlexiNet trader – not one instance per machine. The price to pay for this flexibility is that each process must be told the location of the trader. This is passed as a Java property value – typically by passing an additional parameter to the `java` program. The trader itself will check for the presence of this environment variable, and will start at the specified address if possible.

9.6 The FlexiNet Distribution

FlexiNet is distributed either as a single large zip file, or as a directory hierarchy on CD. In the following, it is assumed that this is copied/decompressed into a directory called *`FlexiNet`*.

The distribution consists of the following:

- A directory hierarchy matching the package hierarchy for FlexiNet. This is in the directory *`FlexiNet/`*`Packages`. It contains source code and class files. The class files have debugging information compiled in but disabled (see section 19.1). A description of the package hierarchy can be found in appendix I.
- A directory containing documentation. In particular, JavaDoc for FlexiNet classes, and a version of this document designed for on-line browsing. These may both be accessed using a web browser via *`FlexiNet/`*`Documentation/Index.html`.
- A Java archive (jar) of FlexiNet classes. These classes were compiled without debugging for performance. This may be found in *`FlexiNet/`*`lib/FlexiNet.jar`.
- A number of examples. These may be found in *`FlexiNet/`*`TestCode`. Chapter 43 gives an overview of these, and each sub-directory contains a `ReadMe.txt` file describing how to run the example it contains.

9.6.1 Installation

FlexiNet does not require ‘installation’ as such. The files should be copied to hard disc, and the `Package` directory put on the classpath. i.e.

```
    set CLASSPATH=%CLASSPATH%;FlexiNet\Packages   (windows)
or
    export CLASSPATH=$CLASSPATH:FlexiNet/Packages  (unix sh)
or
    setenv CLASSPATH=$CLASSPATH:FlexiNet/Packages (unix csh)
```

Alternatively, the FlexiNet jar may be put on the classpath:

```
    set CLASSPATH=%CLASSPATH%;FlexiNet\lib\FlexiNet.jar
```

Note. Either *FlexiNet*/Packages or FlexiNet.jar should be put on the class path, not both.

In addition to FlexiNet, a JDK1.1 compatible JVM must be installed. FlexiNet was developed using Sun's JVM (1.1.6), although others should function just as well.

9.6.2 Running the "Simple Bank" Example (Windows)

This may be found in *FlexiNet*\TestCode\trader\BankExample.

1. Start the FlexiNet trader

   ```
   java UK.co.ansa.flexinet.services.trivtrader.TrivTrader
   ```

   This will run the trader at an arbitrary address, and print the address to the screen.

2. Start the Bank Server

   In another window/machine, ensure the directory containing the example is on the classpath, and run the server, telling it where the trader is.

   ```
   cd FlexiNet\TestCode\trader\BankExample
   set CLASSPATH=%CLASSPATH%;.
   set FT=-Dflexinet.trader=xxxxxxx           (address from 1.)
   java %FT% Server
   ```

3. Start the Bank Client

   In another window/machine, ensure the directory containing the example is on the classpath, and run the client, telling it where the trader is.

   ```
   cd FlexiNet\TestCode\trader\BankExample
   set CLASSPATH=%CLASSPATH%;.
   set FT=-Dflexinet.trader=xxxxxxx           (address from 1.)
   java %FT% Client
   ```

9.6.3 Running the “Simple Bank” Example (Unix)

This is analogous to the Windows example above. The only differences are the different conventions for pathnames and setting environment variables.

1. Start the FlexiNet trader

   ```
   java UK.co.ansa.flexinet.services.trivtrader.TrivTrader
   ```

   This will run the trader at an arbitrary address, and print the address to the screen.

2. Start the Bank Server

   In another window/machine, ensure the directory containing the example is on the classpath, and run the server, telling it where the trader is.

   ```
   cd FlexiNet/TestCode/trader/BankExample
   export CLASSPATH=$CLASSPATH:.            (in csh, use setenv)
   set FT=”-Dflexinet.trader=xxxxxxx”       (address from 1.)
   java $FT Server
   ```

3. Start the Bank Client

   In another window/machine, ensure the directory containing the example is on the classpath, and run the client, telling it where the trader is.

   ```
   cd FlexiNet/TestCode/trader/BankExample
   set CLASSPATH=$CLASSPATH:.               (in csh, use setenv)
   set FT=”-Dflexinet.trader=xxxxxxx”       (address from 1.)
   java $FT Client
   ```

10 MOBILE OBJECTS

10.1 Introduction

Mobile Objects are an abstraction designed to support code and data mobility. In particular, they allows an executing application to 'jump' from one host to another. Obvious applications of this might be to support agency [BURSELL98], for load balancing or to prepare for network disconnection. The mobile object abstractions in FlexiNet are largely transparent, however there are some coding restrictions on mobile objects, and it is not in general possible to migrate an arbitrary application.

Mobile Objects are supported in FlexiNet as a specialisation of the RM-ODP *cluster* abstraction. A cluster is a collection of objects that are managed as a whole. A *mobile cluster* is a collection of objects that may be migrated from one host to another. A cluster may be thought of as a lightweight process and a mobile cluster as a process with the ability to jump from host to host. In practice it is a process-like abstraction that we wish to be mobile, not simply a single object. A detailed description of the Cluster and Mobile Cluster abstractions is given in part five, "Clusters and Capsules".

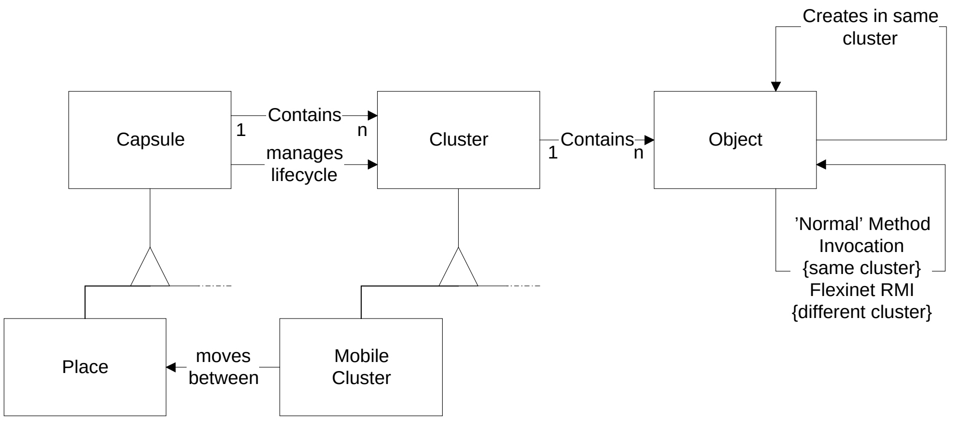


Figure 17 Clusters and Capsules

10.2 Mobility and Places

In practice, clusters move between hosts. At the software level, each location a cluster may reside is called a Capsule, and a Place is a specialisation of this that supports Mobile Clusters. There may be many different Place implementations that provide different facilities, or enforce different management policies. A vanilla Place implementation (`PlaceImp`) is provided as part of FlexiNet. Figure 17 summaries the relationship between objects, cluster and capsules.

10.3 Creating a Mobile Cluster

A cluster is in many respects like an applet. There is a distinguished class at the 'head' of the cluster, which is extended in order to specialise it for a particular implementation. For mobile clusters, this class is `MobileObject`. A subclass will normally override three of its methods as follows:

`<any interface>` `init(`*`<any>`*`,`*`<any>`*`….)`

This method is called upon construction of the first object within a cluster. The method may take any arguments and may return a reference to any interface. Typically this is a reference to be used by the creator to control the mobile object.

`void restart(Exception reason)`

This method is called upon the restart of a cluster; for example, after a successful move. The method is passed the reason for the restart.

`<any interface>` `copied()`

This method is called on a newly created copy of a cluster after the copy has been created (`init` is not called). It may return a reference to an interface, to allow its creator to reference it.

In order to create a new mobile cluster, the `createCluster` method is called on an existing Place (which may be remote). This will create a new empty cluster. A direct call of `createObject` on this cluster can then be used to create an application object within the cluster. This call will cause the construction of a new object of the identified class, and causes the `init` method to be called on this object. This may return a reference to an interface within the cluster. The `init` method is also free to create any internal threads required by the application class.

A simple Mobile Object is illustrated in Figure 18, and the construction of a new instance is shown in Figure 19. As the creation and initialisation of a Mobile Object is always the same, it is common to provide a static construction method in each mobile object class.

```
public class Robot extends MobileObject implements Command
{
  public String name;

  // Override init in MobileObject
  public Command init(String name)
  {
    this.name = name;

    return this;
  }

  // Override restart in MobileObject
  public void restart(Exception e)
  {
    System.out.println("Robot '"+name+"' restarting");
  }

  // implement Command.go
  public void go(Place p)
  {
    System.out.println("Robot '"+name+"' moving...");
    pendMove(p);
  }

  // implement Command.identify
  public String identify()
  {
    System.out.println("Robot '"+name+"' identifying...");
    return name;
  }

}
```

Figure 18 A Simple Mobile Object

```
// construct an empty cluster

Place p = …;
Cluster c = p.createCluster();

// create initialization arguments
Object[] arg = new Object[1];
arg[1] = “Metal Micky”;

// construct a new robot
Command r = (Command) c.createObject(Robot.class,args);
```

Figure 19 Constructing a New Robot

10.4 Moving

Mobile clusters are *autonomous* in that only a thread within a cluster may directly cause it to move. In order to move, the cluster must enter a state where a consistent copy of it can be taken. The majority of this is handled by the infrastructure, but some co-operation is required by the application code. There are two versions of the move method.

`void syncMove(Place p)`

This indicates a synchronous (immediate) move. Any further calls made on interfaces exported by the cluster are blocked until the move completes. The move does not take place until all threads running within the cluster have exited. In particular, other calls made on the cluster that are executing must exit, and the application code must shutdown any internal threads. Once all threads and calls have exited, then the move will take place. The thread that invokes `syncMove` will never return – unless the move fails, in which case it will throw an exception.

`pendMove(Place p)`

This flags a pending move. It is treated identically to `syncMove`, except that the thread calling `pendMove` returns immediately. When the move actually takes place, an error will lead to a call to `restart` with a `MovedFailedException` as the argument.

Once a move has completed, the `restart` method is called on the mobile object. This, like `init`, is free to create any internal threads.

10.4.1 Copying

Copying a mobile cluster is very similar to moving it. A thread within the cluster invokes the `copy` method. Calls into the cluster are blocked, and when all threads are exited, the cluster is copied to the new location, and the `copied` method is invoked on the new object. This may return an interface reference, which is passed to the calling thread as the return value of the `copy` operation. By default, the `copied` method calls restart, with a new `Copied` exception as the only argument. In the default case, it returns a null result.

10.5 Communications

Within a cluster, objects may communicate freely using standard Java method invocation. Between clusters, communication takes place using FlexiNet remote method invocation. No special distinction need be made by a programmer between communications with a stationary service, and a mobile object. This conforms to the 'sea of objects' abstraction.

10.6 Constraints on Mobile Objects

Mobile clusters are arbitrary Java programs. However, there are some stylistic constraints.

Thread Management
: A cluster may create or destroy threads and thread groups at will. However, the application programmer is responsible for shutting down all threads prior to movement.

Static Fields and Methods
: It is possible to configure FlexiNet to give each cluster a separate view of static state. However in general, static methods should be used with care by mobile code, as this constitute a backdoor means by which mobile object may communicate with each other. This circumvents the mechanisms used to track threads, and can result in unexpected behaviour.

Serialisation
: Classes used to construct a mobile cluster must be serialisable using the FlexiNet serialiser. In general terms, this means that they must provide a public no-args constructor, and contain only public or transient data members. In JDK 1.2 this latter restriction will be removed. Details of FlexiNet serialisation are given in Chapter 26.

11 PERSISTENT OBJECTS

11.1 Introduction

Persistent Objects are a cluster-based abstraction that allows clients to access persistent objects transparently. The persistent object itself implements one or more application specific interfaces. The client is given a reference to these, and may access the object as if it were a 'normal' object. The infrastructure actually stores the object on disc, and transparently reads the object in and writes it back before and after each method invocation.

Like mobile objects, persistent objects are supported by a specialisation of the *cluster* abstraction. Persistent clusters are called *storables* and are stored in Capsules called *Stores*. This is analogous to the Mobile Cluster, Place relationship (Figure 20).

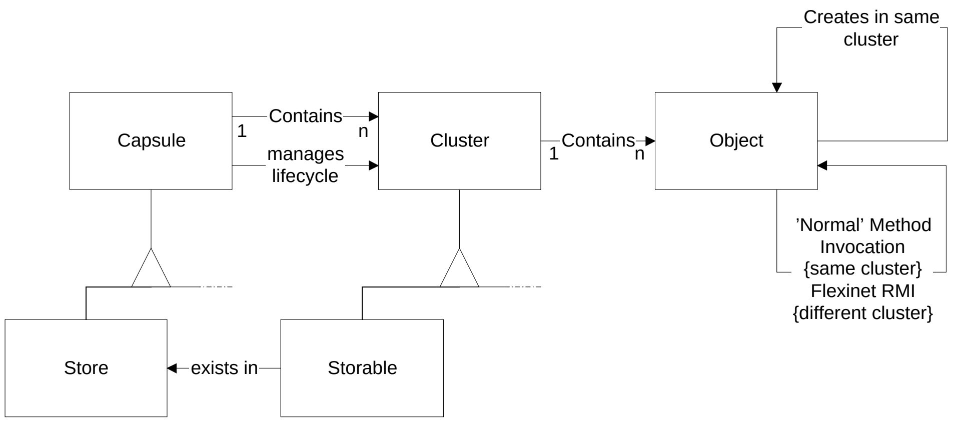


Figure 20 Storable Clusters

11.2 Creating a Persistent Object

Unlike mobile objects, persistent objects are given 'meaningful' names, to allow them to be managed (for example deleted), and to allow a client to re-obtain a reference to them after a crash.

The creation of a persistent object (storable) is analogous to the creation of a mobile cluster – except that the creation interface is on the store's root directory object, rather than on the store itself (Figure 21).

```
// construct an empty cluster

Store s = …;
Directory root = s.getRootDirectory();
Cluster c = root.newStorable("example account");

// create initialization arguments
Object[] arg = new Object[1];
arg[1] = "Richard";

// construct a new storable Account
Account a = (Account) c.createObject(Account.class,arg);
```

Figure 21 Creating a Storable

11.3 Managing Persistent Objects

In addition to the 'transparent' interface to the stored object itself, the store implementation also provides various management interfaces. The `StoreManager` interface provides facilities for the creation and destruction of Stores themselves. Within a store, the Directory interface provides methods to enumerate Storables by name, and delete them. It is envisaged that the `StoreManager` interface will be extended to set policies on a per-store basis (for example access rights, or maximum size); and the `Directory` interface will be extended to provide more user-oriented management, for example facilities to examine the size of each `Storable`.

11.4 Communications

Storable clusters act like mobile clusters. Within a cluster, objects may communicate freely using standard Java method invocation. Between clusters communication takes place using FlexiNet remote method invocation. No special distinction needs to be made by a programmer, that the interface the are accessing is actually on a remote, stored object.

11.5 Constraints on Persistent Objects

Persistent clusters have some stylistic constraints placed upon them. Like Mobile Clusters, they have the same constraints with respect to thread management, static fields and methods and serialisation. In addition, each storable must be read from disc (deserialised) before each invocation and then written back (serialised) afterwards. Storables should not therefore have any internal threads – unless these are extremely short lived. Storables may invoke nested invocations on other Storables (or other objects) but there are not facilities to manage unwinding after a nested failure (Incomplete invocations are simply discarded). For complex scenarios, the transaction abstraction should be used instead.

12 TRANSACTIONS

12.1 Introduction

The FlexiNet transactional framework is an implementation and extension of the Enterprise JavaBeans specification. It was designed in parallel with the rest of the FlexiNet. It uses the FlexiNet infrastructure to support the runtime execution environment, and uses a visual component builder interface for the assembly of applications out of 'bean' components.

The FlexiNet Transaction Framework supports standard Enterprise JavaBeans. A full specification of Enterprise JavaBeans can be found at [SUNc].

The transaction architecture aims to meet the needs of transactional Java applications. Our work started before any publication of Enterprise JavaBeans (EJB). However, it turned out that they have similar goals: to provide implicit transactions thus removing from the developers any concern for transaction management details. We adjusted some of our design and implementation when the draft EJB specification was published to ensure that our transaction architecture fulfils the EJB specification. The final implementation of the architecture can be used as an EJB container to execute Enterprise JavaBeans.

Like EJB, the FlexiNet transactional framework is component-based, thus enables users to assemble portable, customisable components into a transactional infrastructure. Moreover, the architecture is also reflective. Therefore, it provides two additional features that are benefits of supporting transaction processing in Java.

- It allows the transaction infrastructure to be easily adapted to new application requirements and changing environments. For example, it allows application authors to choose a particular concurrency control protocol for their application.
- It allows programmers to provide application-specific information declaratively and separately from application code. This information can be used either at deployment time for configuring the transaction infrastructure to best suit the application component, or at execution time for improving system performance.

The results of our work include a runtime execution environment, and a development toolkit. The execution environment consists of an underlying transaction system and an EJB container. The development toolkit provides a visual tool (`EnterpriseBeanBox`) for users to customise both the beans and runtime container. The container insulates the enterprise bean from the specifics of an underlying system by providing a simple, standard API between the bean and container. The `EnterpriseBeanBox` is an extension of the `BeanBox` from the Bean Development Kit (BDK). It maintains all the original functionality, but gains some new features to meet our special requirements.

In this chapter, we describe the process of developing an EJB application, particularly how to use the `EnterpriseBeanBox` to deploy a bean in a container. We also explain how to demarcate the boundaries of a transaction, and how to implement a concurrency control protocol that can be used by the container. The description is in terms of a simple transactional bank account example, and this example is provided in the FlexiNet distribution.

12.2 Enterprise Bean Development

The first phase of developing an enterprise application is to program enterprise beans. An enterprise bean is a portable, reusable, and container-independent software component. An enterprise bean implements a business task, or a business entity. Each bean includes its Java classes, its remote and home interfaces, its deployment descriptor, and its environment properties. The enterprise Beans must conform to Enterprise JavaBeans component contract to ensure that they can be installed into any compliant EJB container.

An example bean, `AccountBean`, is shown in Figure 22.

The `AccountBean` class defines three basic operations: `credit` for putting money into an account, `debit` for withdrawing money from an account, and `balance` for checking the balance of an account. In addition, the `getInitialBalance` and `setInitialBalance` pair define a simple JavaBean property. A JavaBean property is a single public attribute. A simple property represents a single value. It can be customised by a user without accessing the source code.

The activation, deactivation and removal of an enterprise bean are managed automatically by the bean container. However, if a bean would like to perform some external actions when these events occur, these actions can be defined in `ejbActivate`, `ejbPassivate`, and `ejbRemove` respectively. The container will ensure the corresponding operation will be invoked when such an event occurs. The `setSessionContext` method stores the reference to the context object to an instance variable. This method is called after the instance creation.

```
public class AccountBean implements SessionBean
{
  private double currentBalance;

  public AccountBean( ) { }

  public AccountBean(String name)
  {
    this.name = new String(name);
  }

  public double getInitialBalance()
  {
    return currentBalance;
  }

  public void setInitialBalance(double ibal)
  {
    currentBalance = ibal;
  }

  public void credit(double value)
  {
    double bal = currentBalance;
    // pause to demonstrate race condition
    try {Thread.currentThread().sleep(5000);
    } catch (Throwable thr) {};
    currentBalance = bal+value;
  }

  public void debit(double value) throwsOverdrawException
  {
    if (currentBalance < value)
      throw new OverdrawException("No enough money");
    currentBalance = currentBalance-value;
  }

  public double balance( )
  {
    return currentBalance;
  }

  // methods and fields for EJB use
  public void ejbActivate(){ }
  public void ejbPassivate(){ }
  public void ejbRemove(){ }

  protected SessionContext sessionContext;

  public void setSessionContext(SessionContext ctx)
  {
    sessionContext = ctx;
  }
}
```

Figure 22 An Example Enterprise JavaBean

12.3 Application Assembly

An application assembler is a domain expert who compose applications that use enterprise Beans. The application assembler works with the enterprise bean's client view contract. Although the application assembler must be familiar with the functionality provided by the enterprise bean's remote and home interfaces, but he does not have to have any knowledge of the enterprise bean's implementation.

The output of the application assembler can be new enterprise Beans, or applications that are not enterprise Beans (for example, servlets, applets, or scripts). The assembler may also write a GUI for the applications.

In a three-tier architecture, the client is very thin and only responsible for data presentation and the user interface. Following on from the simple bank example of chapter 9, we define a simple transactional bank service. In this example, the client program focuses on how to provide a user-friendly interface for performing following actions: credit money in an account, withdraw money from an account, and check the balance of the account. The larger part of the client program is concerned with the GUI. The user interface is shown in Figure 23.

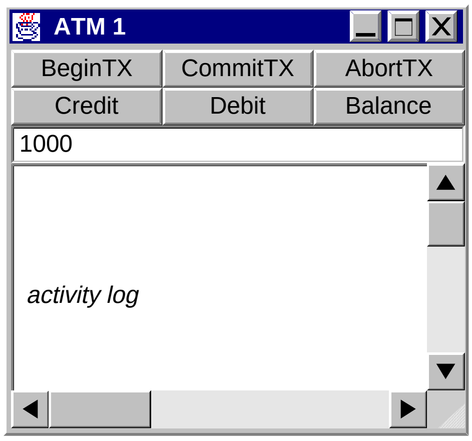


Figure 23 The UI for the Example Application

Another issue for client program is the declaration of transaction boundaries. Our transaction framework is a high-level component framework that attempts to hide system complexity from the application developer. We assume most application developers and application users do not need to access transaction management explicitly. The burden of managing transactions is shifted to the container providers. Our transaction framework implements the necessary low-level transaction protocols, such as the two-phase commit protocol between transaction manager and database systems, transaction context propagation, and distributed two-phase commit.

However, the clients have to control transaction scopes, that is, to tell the system when a transaction starts and when it ends. Clients use the `javax.jts.UserTransaction` interface for this purpose. This package defines the application-level demarcation interface. The application programmer demarcates transaction boundaries with `begin` and `commit/rollback` calls. This is illustrated in Figure 24. The transaction framework ensures that all the actions performed inside the boundaries are transactional.

```
import javax.jts.UserTransaction;
……
UserTransaction tx =
     (UserTransaction)new AnsaUserTransaction();
tx.begin( );
……                   //do work
tx.commit( );

tx.begin( );
……                   //do work
tx.rollback( );
```

Figure 24 Demarcating Transactional Boundaries

Application assembly can be performed before or after the deployment of the bean into an operational environment.

12.4 Deployment

A bean deployer is an expert in transactional infrastructure responsible for deploying enterprise Beans into a container. A deployer typically uses the `EnterpriseBeanBox` to adapt enterprise Beans to a specific operational environment.

First, a deployer can decide the transaction attribute for an enterprise bean. There are six possible values:

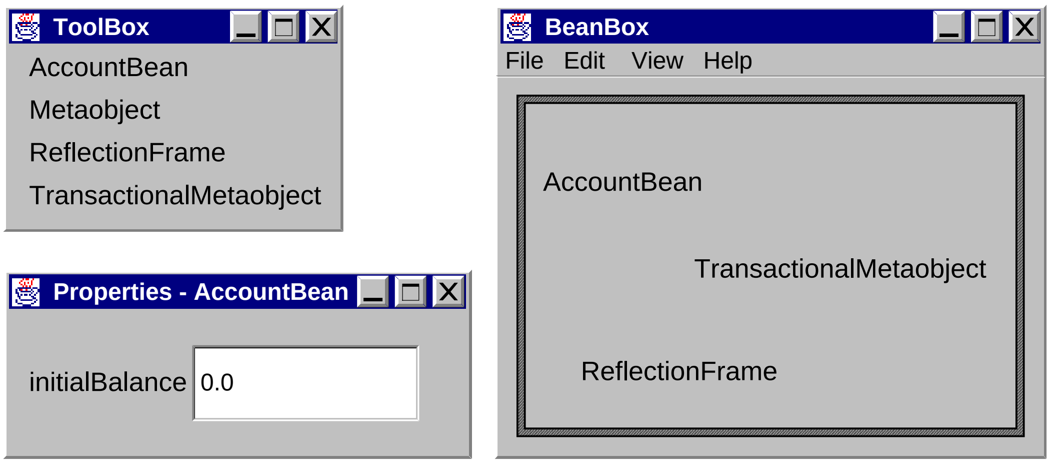
- TX_NOT_SUPPORTED: a container must always invoke an enterprise bean that has this transaction attribute without a transaction scope.
- TX_BEAN_MANAGED: an enterprise bean that has this transaction attribute can use the `UserTransaction` interface to demarcate transaction boundaries.
- TX_REQUIRED: if a client invoke an enterprise bean that has this transaction attribute while the client is associated with a transaction context, the containers invokes the enterprise bean's method in the client's transaction context. Otherwise, the container automatically starts a new transaction before delegating the method call to the enterprise bean.
- TX_SUPPORTS: an enterprise bean that has this transaction attribute is invoked in the client's transaction scope. If the client does not have a transaction scope, the enterprise bean is also invoked without a transaction scope.
- TX_REQUIRES_NEW: an enterprise bean that has this transaction attribute is always invoked in the scope of a new transaction.

- TX_MANDATORY: an enterprise bean that has this transaction attribute is always invoked in the scope of the client's transaction. If the client attempts to invoke the enterprise bean without a transaction context, the container throws the `TransactionRequired` exception to the client.

Furthermore, we allow a deployer to customise the behaviour of the container. A deployer can choose which concurrency control protocol should be used for an enterprise bean. The concurrency control protocol can be provided by the container provider, or implemented by third-parities. The design of concurrency control mechanisms is outline in section 12.7.

Finally, we allow a deployer to specify the concurrency semantics of an enterprise bean. This information can be used by the container to optimise the concurrency control decisions.

A deployer uses the `EnterpriseBeanBox` to input these deployment decisions. When this is started, the *BeanBox, ToolBox*, and *PropertySheet* windows appear on the screen. To instantiate a bean in click on the desired bean in the ToolBox and then click in the `BeanBox` area.



initialBalance 0.0

Figure 25 EntepriseBeanBox Main Windows

To deploy an enterprise bean, three components must be instantiated: the application bean, (For example the `AccountBean`), the reflection frame bean `ReflectionFrame`, and the meta-object bean `TransactionalMetaobject`. Figure 25 shows these three beans are instantiated in the BeanBox.

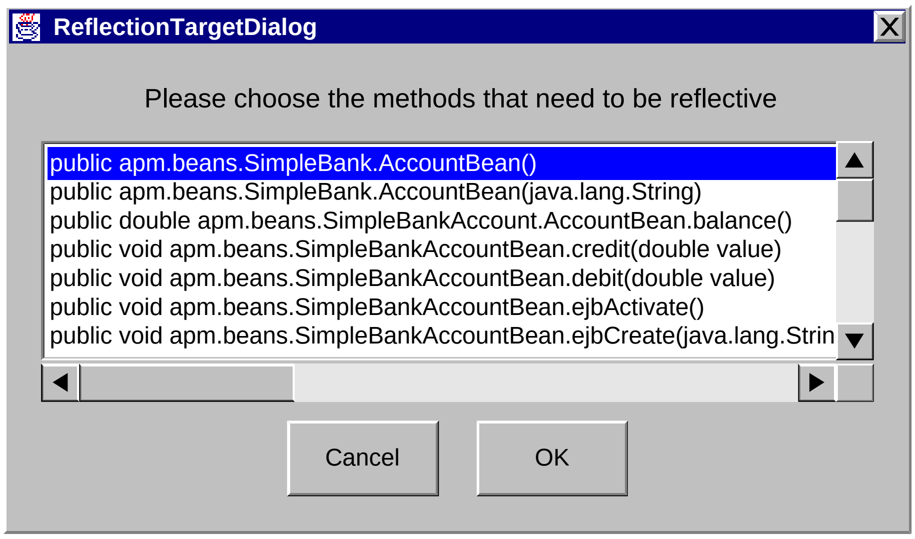
In order to deploy the beans, the following steps must be followed:

- Instantiate the `AccountBean`, `ReflectionFrame`, and `TransactionalMetaobject` beans.

- Select the `ReflectionFrame` bean

- Select the Edit→Reflection→Application menu.
  The `EntepriseBeanBox` positions a line under the mouse pointer that can be used to connect to the `AccountBean` to the `ReflectionFrame` bean.

- When the beans are connected, the `EntepriseBeanBox` responds with a dialog box that displays all the public methods of the `AccountBean`. This is shown in Figure 26.

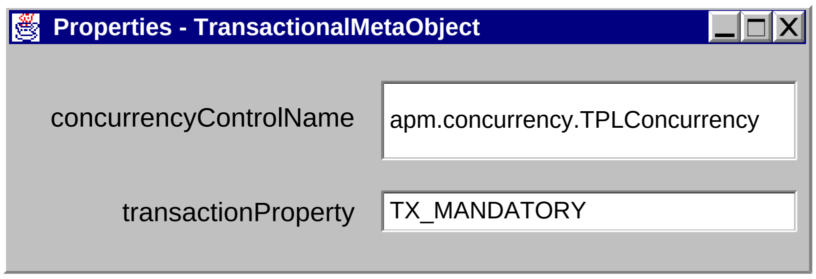


Please choose the methods that need to be reflective

```
public apm.beans.SimpleBank.AccountBean()
public apm.beans.SimpleBank.AccountBean(java.lang.String)
public double apm.beans.SimpleBankAccount.AccountBean.balance()
public void apm.beans.SimpleBankAccountBean.credit(double value)
public void apm.beans.SimpleBankAccountBean.debit(double value)
public void apm.beans.SimpleBankAccountBean.ejbActivate()
public void apm.beans.SimpleBankAccountBean.ejbCreate(java.lang.Strin
```

Figure 26 Marking Methods as Reflective

- Select all the methods to be exported in the box, and push the *OK* button.
- Select the *ReflectionFrame* bean again
- Select the *Edit→Reflection→meta-object* pulldown menu. The EntepriseBeanBox positions a line under the mouse cursor that should be connected to the *TransactionalMetaobject*.
- Select a transaction property in the property box from the pull-down menu. (Figure 27)
- Input the class name of the desired concurrency control protocol in the *ConcurrencyControlName* area.

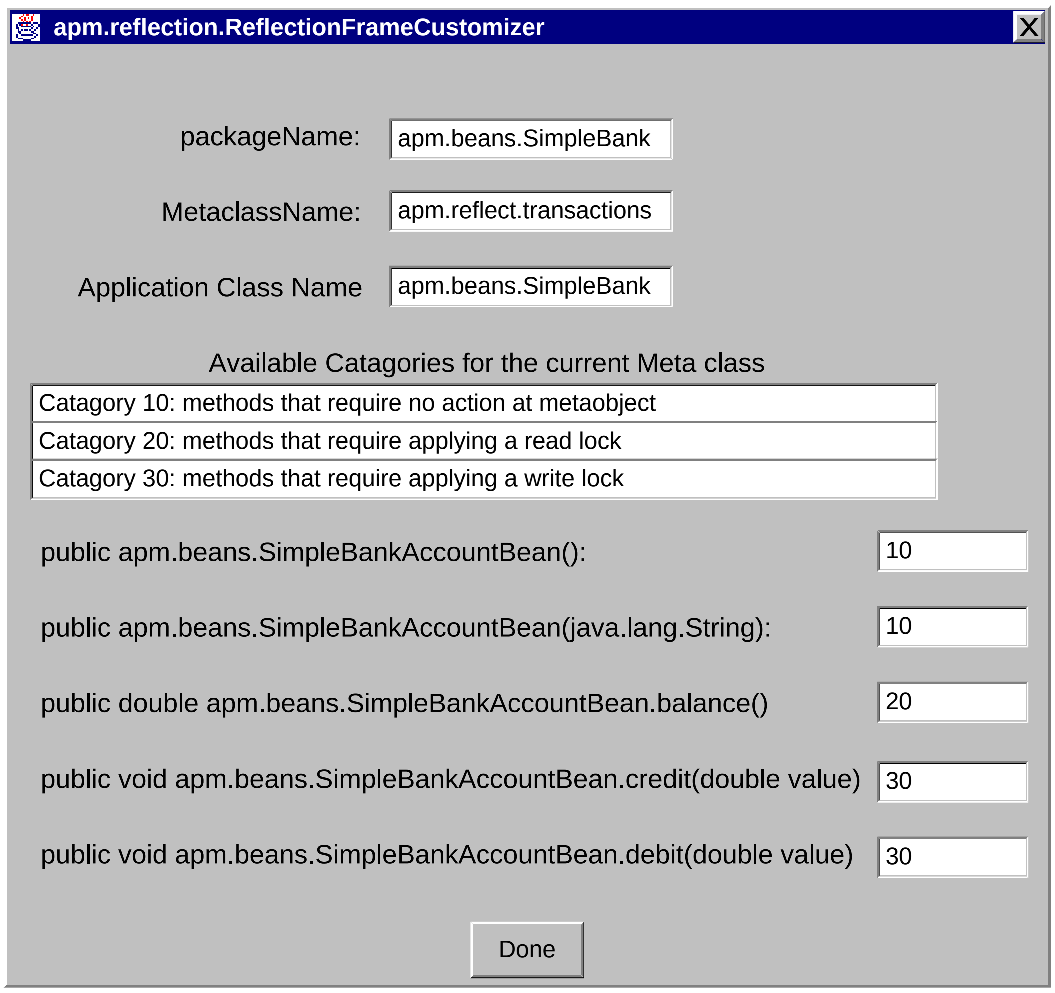


concurrencyControlName: apm.concurrency.TPLConcurrency

transactionProperty: TX_MANDATORY

Figure 27 Transactional Meta-Object Properties

- Select the ReflectionFrame bean again
- Select the *Edit→Customise* pulldown menu. The BeanBox responds with a dialog box. It displays the information input so far. More importantly, it allows input of the concurrency semantics of the enterprise bean by giving a category number to each of the operations chosen earlier. The definitions of categories are displayed in the dialog box as well. (Figure 28)


apm.reflection.ReflectionFrameCustomizer

packageName: apm.beans.SimpleBank

MetaclassName: apm.reflect.transactions

Application Class Name apm.beans.SimpleBank

Available Catagories for the current Meta class

Catagory 10: methods that require no action at metaobject
Catagory 20: methods that require applying a read lock
Catagory 30: methods that require applying a write lock

public apm.beans.SimpleBankAccountBean(): 10

public apm.beans.SimpleBankAccountBean(java.lang.String): 10

public double apm.beans.SimpleBankAccountBean.balance() 20

public void apm.beans.SimpleBankAccountBean.credit(double value) 30

public void apm.beans.SimpleBankAccountBean.debit(double value) 30

Figure 28 Customising the Reflection Frame

- Input a category number for each operation in the corresponding text area according to the category definitions.
- Push the *done* button
- Select the File→GenerateReflectionJar pull-down menu

The results of the deployment process consist of a number of jars. The `accountBean.jar` represents the customised `AccountBean` class and its associated classes. The `Reflection.jar` includes all the classes required to run the `accountBean` in the container, namely the two interfaces for the client to access the bean, and two classes that implement the two interfaces. These interfaces and classes are generated and compiled by the `EntepriseBeanBox` automatically.

12.5 Starting an Application

The results of the deployment process are a number of Java archives (jars). In order for the class loader of the container can find them, they should be put under a same directory. The name of the directory will be passed as a parameter of the program so that the class loader can find the related jars.

- Start the FlexiNet trader:
  ```
  java
  UK.co.ansa.flexinet.services.trivtrader.TrivTrader
  ```
- Start the transaction server:
  ```
  java -Dflexinet.trader=xxx

  org.omg.CosTransactions.ApmTransactionFactoryImpl
  ```
  Note: `xxx` is the address of the trader.
- Start an application server:
  ```
  java -Dflexinet.trader=xxx
       UK.co.ansa.transaction.container.NewAnsaContainer
       $(JARHOME) Reflections.jar
       beans.SimpleBank.AnsaAccountBeanHome TxServer
  ```
- Start a client:
  ```
  java -Dflexinet.trader=xxx
         applications.newSimpleBank.Bank
  ```

12.6 Using the Example Application

The simple bank example provides two ATM machines for accessing one bank account. One can credit money in, withdraw money from the account, or check the balance of the account via either of them. This example intends to demonstrate the capability of our transaction framework for providing proper concurrency control for any enterprise bean transparently.

There is no code for dealing with concurrency at all in the `AccountBean` implementation. However, as the transaction property for `AccountBean` was set to TX_MANDATORY in the deployment stage, then the container will ensure data consistency of the `AccountBean` even in a environment where these is concurrent access. As an example:

- Credit `1000` pounds into the account via *ATM1*: type in `1000` in the text area, then push the *credit* button.
- Check the balance of the account via both *ATM1* and *ATM2*: push the balance button in either of them. The balance will appear on the display area. It should be `1000` in both machines.
- Credit another `1000` via *ATM1*, and immediately withdraw `500` via *ATM2*.
- Check the balance again. The balance should be `1500`.

Note that although a credit and debit operation were performed simultaneously, the container and the transaction system ensure the result is the same as these two actions are performed sequentially.

Now, go to the *EnterpriseBeanBox* and change the transaction property for AccountBean into TX_NOT_SUPPORT. Repeat the above actions, and the final balance of the account will not be 1500, but 2000. This is because the container will not provide any concurrency control for a bean whose transaction property is TX_NOT_SUPPORT. Actually, this is the original behaviour defined by the bean. In the previous round of test, it is the container within which the bean was executed that makes the bean can maintain its consistence in a concurrent environment.

From this demonstration, it is clear that the FlexiNet transaction framework can provide transaction properties to application programs transparently. This allows application developers to totally focus on the business logic without deep concern about transaction issues.

12.7 Concurrency Control Protocol

The transaction framework can use third-party protocols to provide concurrency control so that the application deployer can choose the most suitable concurrency control protocol for the application.

A class intended to implement a concurrency control protocol must extend the Concurrency class as shown in Figure 29. When an invocation to a bean inside the container is received, the container will call the beforeAccess method before delegating the invocation to the target bean. The beforeAccess method will decide whether and when the invocation should allow go ahead, and take action to ensure transaction properties. For example, in a class implementing the typical 2-phase-locking protocol, the beforeAccess method allows the invocation to go ahead only when an appropriate lock can be granted for the invocation. It will also make some state backups to ensure the bean can return to the previous consistent state, if the current transaction is aborted for any reason.

Similarly, the container will call the afterAccess method after the invocation finished but before passing the result to the original caller. Normally, little needs to be done inside the afterAccess because none of the resources, such as locks and backups, can be released until the end of the transaction.

A concurrency control class also needs to implement methods for participating in the *2-phase-commit* process of the transaction: prepare, commit, commit_one_phase, and rollback. The prepare method can respond an invocation in several ways. If no modification has been done by the transaction on the associated bean, the prepare method can return VoteReadOnly. If all the data needed to commit the transaction can be written (or have already been written) to stable storage, it can return VoteCommit. It should VoteRollback in any other circumstances, even if it has no knowledge about the transaction.

The commit method should commit all changes made as part of the transaction to the associated bean. Normally, it also clears up all the resources held on behalf of the transaction, such as `locks` and `backups`. Similarly, the `rollback` method should rollback all the changes made as part of the transaction to the associated bean.

The `commit_one_phase` method is used when there is only one bean is involved in a transaction, thus there is no need for the first phase.

```
public class Concurrency implements Serializable
{
  public Concurrency()
  {
    this.appObject = null;
  }

  public Concurrency(Object appObject)
  {
    this.appObject = appObject;
  }

  public void beforeAccess(int catId,
              Coordinator coordinator)
              throws LockException, InterruptedException
  { }

  public void afterAccess(Method appMethod,
              Coordinator coordinator)
  { }

  public int prepare(Coordinator coordinator)
  { return 0;}

  public synchronized void commit(Coordinator coordinator)
              throws NotPrepared, HeuristicRollback,
              HeuristicMixed,HeuristicHazard
  { }

  public void rollback(Coordinator coordinator)
              throws HeuristicCommit, HeuristicMixed,
              HeuristicHazard
  { }

  public void commit_one_phase(Coordinator coordinator)
  { }

  protected Object appObject;
}
```

Figure 29 Concurrency Interface

PART THREE: ARCHITECTURE

13 INTRODUCTION

13.1 Introduction

FlexiNet was designed ‘from the ground up’. Whilst we wished to support interoperability with existing systems, and conform to, or extend, existing standards, these were secondary goals. FlexiNet grew out of dissatisfaction with current offerings, and a clean slate provided the opportunity to build a coherent architecture.

13.2 Platform

FlexiNet was developed to be *Java specific*. This allowed us to leverage the facilities provided by Java to provide a clean architecture and straightforward API. In particular facilities such as *objects by value*, *subclassing of arguments* and *dynamic late linking* are used in our architecture to simplify many aspects of the design, and to help make FlexiNet extensible.

FlexiNet also makes considerable use of Java’s *strong typing* and *introspection* facilities. FlexiNet *could* be reengineering on other languages that did not have these facilities, but it would lose some of its architectural integrity, and internal type-safety. If FlexiNet were reengineering in, say, C++, then it is likely that *subclassing of arguments* and related features would have to be omitted.

FlexiNet was developed on NT 4.0 and Solaris using Sun’s JDK1.1. Work started with JDK1.1.1 and by the end of the project JDK1.1.7 was being used. Some pieces of code are no longer compatible with early version of JDK1.1. In particular parts of the Information Space will not compile on JDK1.1.3 or below. Microsoft compilers and JVMs have also been used, but a large number of inconsistencies were found between the different compiler vendors and versions (particularly with newer features such as inner classes). FlexiNet will run under most Microsoft JVMs, although parts of it will not compile using (some versions of) the Microsoft compiler.

13.3 Principles

Where possible we reused existing concepts and principles:

ODP Reference Model

> The computational model from RM-ODP was uses as a basis for the programmer interface. Additionally, a number of RM-ODP engineering model concepts were reflected into the computational model. The ODP notions of interfaces and objects was used, and ODP clusters were implemented to provide encapsulation. We strayed from the architecture in many details. In particular, ODP interfaces are state-full. FlexiNet (and Java) interfaces are not. ODP clusters provide a fixed set of capabilities, and have strong requirements on their contents. We weakened these requirements, and generalised the cluster concept.

Java Language

> We attempted to keep the FlexiNet API and remote object semantics as close to normal Java as possible. In particular, we use Java interface classes rather than IDL files, as this is more natural for a Java programmer.

Build for Change

> FlexiNet was designed to be constantly upgraded and changed. We attempted to minimise the amount of 'global knowledge' and interdependencies between components, so that parts could be replaced, or new components added.

Multiple Everything

> FlexiNet was designed to be component based, and to allow more than one instance of any component to co-exist. This is important, for example if a client has to speak two versions of a protocol. FlexiNet make very little use of 'static' data, as this is intrinsically restrictive.

Reflective Implementation

> FlexiNet attempts to use its own mechanisms internally wherever possible. For example a FlexiNet name, passed to identify an interface, is an ordinary object, and treated as such when serialised, deserialised or otherwise manipulated. This approach allows us to change the specification of one component (for example a name) with minimum impact on other components.

Details of these principles, and others, are described in the following chapters.

14 NAMING

14.1 Introduction

The primary purpose of middleware platform such as FlexiNet is to support the interaction between a client and a service. *Names* are used in FlexiNet to identify interface on services, and the *resolution* of a name provides a client with a reference to a service that they may use. The generation and resolution of names is therefore a crucial part of the FlexiNet framework.

A FlexiNet `Name` may represent any interface. This might be a concrete interface on a real object, or a more abstract interface which has zero or more concrete implementations, for example an interface to a naming service that is implemented by a number of distributed servers. Although this later case is useful, the most usual use of names is to represent concrete interfaces.

There are two stages in the establishment of a client's reference to a (concrete) interface. First, a *generator* is used to generate a *name* for the interface. By choosing an appropriate generator, a service can make coarse grain decisions about the *protocol* used for communication between client and server. For example, one generator might produce names that relate to a UDP based protocol that is efficient over unreliable networks, whilst another might produce names relating to an encrypted TCP based protocol.

Once a client has obtained a name they use a *resolver* to resolve the name to a *proxy* for the original interface. Different resolvers are capable of resolving name that use different protocols. A proxy is a local object that stands for a remote one.

For a particular protocol, the information contained in a name may vary. It must contain sufficient information to allow the resolver to construct a proxy *bound* to the original interface. It may also contain hints about how this binding should be created, for example what performance trade-offs should be made, or what action should be taken in the case of communication failure. Taken to an extreme, the name could contain all the code required in order to create the binding. In this extreme, the resolver is not required at all. This case is particularly useful, as it allows a client to resolve names for protocols that it has not previously been configured to understand.

14.2 Multiple Generators and Resolvers

The majority of interfaces within a particular service are likely to be named using the same generator, as a ‘general purpose’ generator will usually suffice. However, there is frequently the need for a service to use more than one generator; for example a service may wish to use one protocol for ‘normal’ communication and a second protocol when communicating securely. It might also wish to use special-purpose protocols to aid the debugging of a particular interface, or for interfaces that lead to poor performance when accessed remotely using the default protocol. For example, a read-only service may wish to use a protocol that caches previous requests on the client machine.

Equally, a client may need to resolve names generated by many different generators. This will typically be because the client has been passed references to interfaces on services using different default protocols, or services using special purpose protocols.

To support the use of multiple generators and resolvers, the concept of a *Binder Graph* is introduced. The purpose of the binder graph is three fold.

1. During name generator, it provides an ordered list of generators that are given the opportunity to name the interface in turn. By ordering the generators within the list, a server can indicate naming policy preferences.

2. During resolution, a node in the graph chooses the appropriate resolver to resolve a particular name. The graph may be augmented with additional resolvers dynamically, as a side effect of resolution.

3. The binder graph performs caching and housekeeping, to allow individual generators and resolvers to be as simple as possible, and to ensure that interfaces are consistently named and resolved.

14.3 Binder Graphs

Quite complex binder graphs can result from the composition of many generators and resolvers. A typical graph is shown in Figure 30. This shows three generators; X and Y are special-purpose generators that only generate names for a small proportion of possible interfaces. Green is a general-purpose generator and will generate a name for an arbitrary interface. The Null Generator terminates the list of generators and is used to raise an exception if a request is made that is not handled by X, Y or Green.

The graph also shows three resolvers, `SmartChoice` is itself a resolver, this stores in its database two other resolvers, Green and Magenta, which resolve names which use particular protocols.

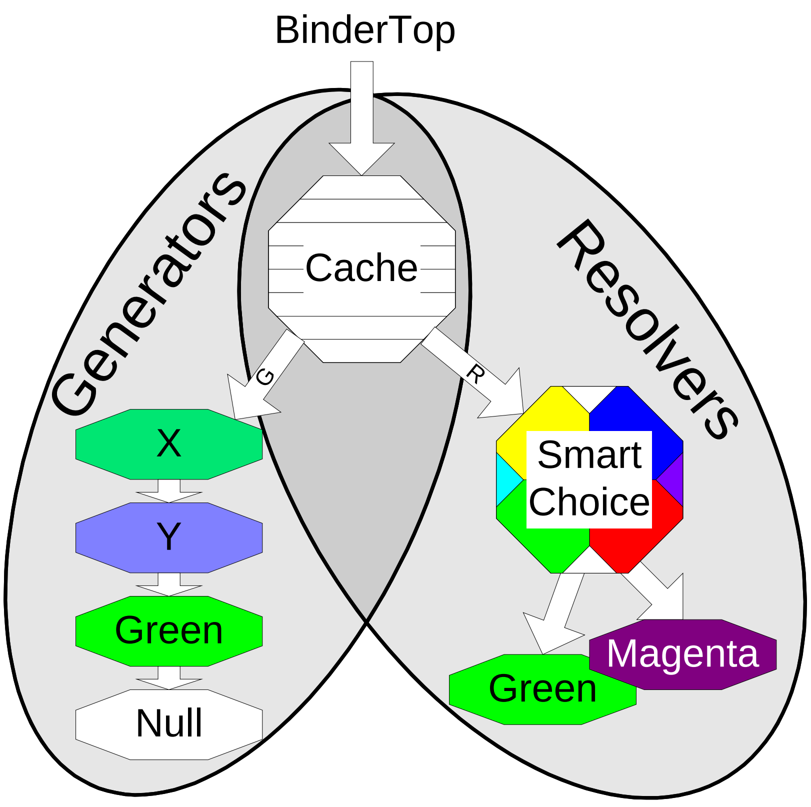


Figure 30 An Example Binder Graph

The root of the graph is 'Cache', which is both a Generator and a Resolver. This performs caching of names generated and resolved by the other nodes in the graph. The root of the graph is commonly referred to as 'BinderTop' and many FlexiNet components require a reference to this, as it represents a single access point for the generation and resolution of names. In particular, any generator or resolver that needs to pass an interface *by reference* will use BinderTop to generate and resolver names for those interfaces.

14.3.1 More Complex Binder Graphs

Some protocols have special requirements on the range of other protocols that may be used to name interfaces that they pass by reference. For example, the IIOP binder must conform to an imposed standard for the transmission of interface references. If the default binder graph were used, the IIOP binder might pass a name generated by some other protocol – which would breach the IIOP standard. To overcome this problem, we may design a more complex binder graph, shown in Figure 31. The second cache in this graph is used purely for IIOP names (IORs). This ensures that an IOR is generated for an interface when required, even if some other name has already been generated. This illustrates how the binder graph concept can allow complex binding requirements to be met through the reuse of basic components.

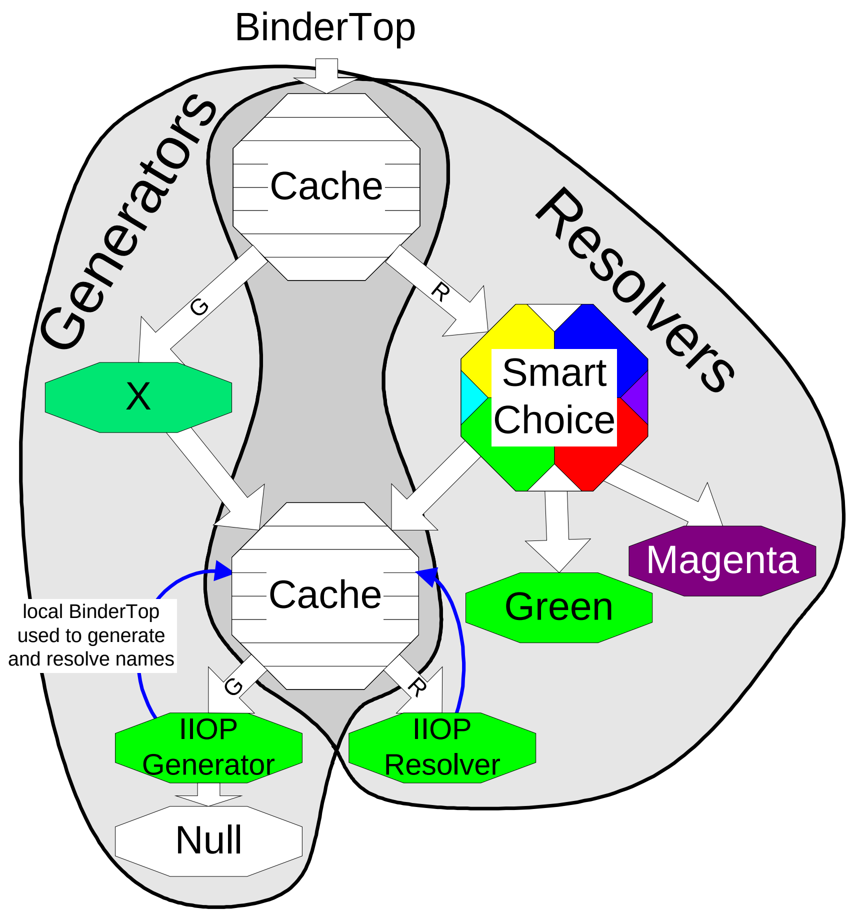


Figure 31 A Special-Purpose Binder Graph to Support IIOP

14.3.2 Smart Choice

The function of the `SmartChoice` resolver is to choose the appropriate protocol specific resolver to be used to resolve a given name. Rather than do this directly, `SmartChoice` calls a resolve method *on the name itself*. The name may then examine the available choice of resolvers, and pick an appropriate one. In extreme cases, the name may create a new resolver, and augment the binding graph. It may even perform resolution itself, completely removing the need for a protocol specific resolver. This is particularly useful for ‘one off’ protocols that will not be required when resolving other names.

14.4 Naming Proxy Objects

When a name is resolved, the resulting object will be either a ‘real’ object or a ‘proxy’ object. The distinguishing characteristic of a proxy object is that it is not itself named (and cannot be named). Instead, an attempt to name the proxy object will return the name of the interface that the proxy represents. The reason for this is simple, if client A is passed a reference to a service object S (internally a name for S); and then passes this reference to a second client, B, then B should obtain the name for S, not the name for the intermediate object. This is illustrated in Figure 32.

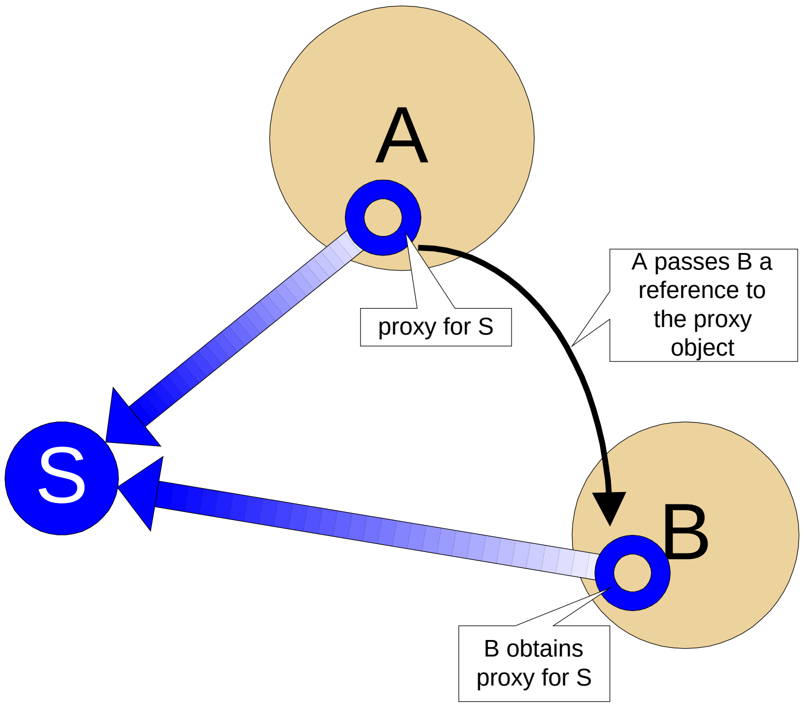


Figure 32 Passing References to Proxies

This subtlety is handled by the Cache. When naming an interface on an object, if that object is a proxy object, then the object is asked for its name directly.

A corollary of this is that proxy objects may implement only one (nameable) interface. It is of course possible for them to implement other, engineering, interfaces, but these cannot be passed by reference using FlexiNet.

14.5 Names as Objects

Names are themselves Java objects. This allows different naming classes to be used to implement different strategies for resolution. In particular a name may be able to resolve itself *without* the use of a resolver. This feature is used when creating smart and generic proxies, as described in section 17.3. A further possibility is for a name to dynamically load or create an appropriate resolver and add it to the Smart Choice's database.

As names are always passed by value, care must be taken that they do not become inappropriately large. For example, it is perfectly reasonable for a naming *class* to contain a blueprint for the construction of a binder (See chapter 29), but it would generally be an unreasonable overhead if this blueprint were carried in each name instance.

As names are objects, a client receiving a name is no more (or less) likely to be willing to load and execute the name's class as it would be to load and execute any other classes passed to it. The ad-hoc invention of naming classes should be avoided, if these must be used in open environments where class loading may be restricted for security reasons.

14.6 Generator Interface

The Generator interface has a total of five methods.

- `Name generateName(Object obj,Class cls,FlexiProps qos)`
  This generates a name for the specified interface that meets the required Quality of Service (QoS) constraints. `cls` is an interface class, and `obj` is the object that implements the interface. This may be a proxy object.

  The third parameter, `qos` is normally null, but may be used to specifiy constraints on the type or format of the name returned. QoS parameters are given as a set of `FlexiProps` (see section 29.1). In general, a generator should only generate a name if it both understands, and meets the requirements of, *every* specified constraint.
  When QoS is not specified, the Cache binder at the head of the binder graph will remember each name generated, so a generator below it in the graph will only be asked to name an interface once. However, when using QoS parameters, a programmer may require that *different* names are generated for the same interface, in response to different calls to `generateName` so that each QoS instance can be individually identified. For this reason, `Cache` does not remember names generated when QoS is specified. If names meeting a particular QoS *do* required caching, then it is up to the particular Generator to perform this function.

- `boolean grantName(Object obj,Class cls,Name name,`
  `FlexiProps qos)`
  Grant the specified name to the specified interface with the specified QoS. This is used to start up services at specific addresses.

- `void dropNames(Object obj,Class cls)`
  Remove all knowledge of all names generated for the specified interface. Subsequent calls on these names should fail as if the name has never existed. The generator must not subsequently reuse any of these names to represent a different interface.

- `String stringifyName(Name name) throws BadName`
  Return a stringified form of the specified name. This is intended to be parsed back into a `Name` using `resolver.parseName`. The method `Name.toString` simply provides a human readable description of the name, and may not contain sufficient information to re-parse. Section 14.8.1 describes the format for stringified names.

- `boolean addGenerator(Generator g)`
  Each generator (notionally) contains a list of alternate generators to be used to handle calls that this generator is unable (or unwilling) to

handle. This called adds the specified generator to the head front of the list.

The (first) alternate generator is used in the following circumstances:

- To generate names for interface that this generator does not wish to name.
- To generate names when this generator cannot meet the specified QoS.
- To grant names that this generator cannot grant.
- To stringify names that this generator cannot stringify.

The list of generators is terminated with a special `NullGenerator` which returns exception or failure conditions whenever called. In addition to this, when `dropNames` is called on a generator; the generator should call `dropNames` on the nested generator, in addition to performing any local processing.

14.7 Resolver Interface

The resolver interface has just three methods, as follows:

- `boolean resolvesProtocol(String protocol)`
  Return true if the resolver is capable of resolving names in the specified protocol. A resolver must normally be capable of resolving all valid names in a protocol it resolves. If, in practice, a number of resolvers are required to manage different partitions of a namespace, then these should be wrapped and presented to the system as a single resolver.

- `Object resolveName(Name name,Class cls,FlexiProps qos)`
  `throws BadName`
  Resolve the specified name and return an object implementing the interface it represents using the specified QoS (normally null). This object will ordinarily be a proxy object, or the object that the name was originally generated for - if that object is local. `BadName` should be thrown if resolution fails. Note, there is no requirement for the resolver to validate that communication with the named interface is actually possible. The `cls` parameter is the class of the interface that the name refers to. This parameter is specified to remove the requirement for names to include their interface type. This is always available from the context of the resolution.

- As with generating names, the Cache at the head of the binder graph will remember which names have been resolved, and a resolver will not be asked to resolve the same name twice. However if QoS is specified, the resolver must perform any caching that is required.

- `Name parseName(String name) throws BadName`
  Parse a name previously generated by `Generator.stringifyName`. Throw `BadName` if this cannot be accomplished.

14.8 Name Interface

Name classes all implement the abstract class `Name`. `Name` is actually specified as an interface, as this gives developers more freedom in the implementation of concrete Naming classes. `Name` has two methods

-
  ```
  Object resolve(Class iface,BinderDB ctxt,
                 FlexiProps qos) throws BadName
  ```
  Resolve the name to an interface of class `iface` with the specified QoS constraint. To aid resolution the binding context, `ctxt`, is supplied. This is essentially a small database of resolvers and other useful information. Typical implementations will lookup the appropriate resolver within the context, and then call `resolveName` on it.

- `String getProtocol()`
  Return the protocol that the name uses. This is used within resolvers for type checking.

14.8.1 Stringified Name Format

It is possible to represent most name classes in a stringified format. This is intended primarily to aid debugging. In the normal case, Names should be passed between JVMs, or stored, using FlexiNet serialisation. However, to allow clients to locate their first reference to the distributed system, a secondary mechanism is useful. This is usually a stringified reference to a naming service. It is important to note that some names may not have a stringified form.

The format of a stringified name is a protocol name followed by a colon, and then a protocol specific string. Protocol names are strings made of any character other than colon and open and close parenthesis. To allow new protocols to be created by different organisations without fear of name-clash, each protocol name should be proceeded by the standard Java package name prefix for the creating organisation. For example “UK.co.ansa.flexinet.rrp”. An exception is made for Citrix Systems (Cambridge), as the originator of FlexiNet, who may use simple short names.

The protocol specific string usually (but not exclusively) comprises of a number of sections contained in parenthesis. These correspond to the structure of the name object they represent. As names may contain other name-like objects, these is typically recursively structured. For example the name of an interface on a cluster, using the request reply protocol is:

```
rrp:((123.45.56.78:1000)(12345678:12345678))(4)
```

This has two sections, the cluster address and the interface identity. The cluster address section is itself composed of two parts: the TCP endpoint and the cluster identity.

14.8.2 Semantics of Names

Names are used to represent interfaces, however they are not strictly identifiers for interfaces – two different names may be used to represent the same interface. Names can be compared for equality, however no conclusion can be made as to whether two unequal names represent the same interface.

Two distinct names may act identically – either absolutely or when viewed by a particular client. Names are not considered as secret. They are passed between processes in plain text (unless the protocol used for communication is inherently encrypted). They may also be copied.

Each name relates to a *protocol*. The protocol is effectively the *class* of a name. In practice, a protocol is identified by a Java string. The protocol “self” is used to identify names that resolve themselves. A resolver for any other protocol must resolve all correct names in that protocol. (This removes the need for a one to many protocol to resolver mapping).

15 GENERIC MIDDLEWARE

15.1 Introduction

Many of the facilities provided in a middleware platform such as FlexiNet are *generic*, in that they do not depend on the type of the interfaces, object or data involved. However some features, such as type checking and marshalling (serialisation) *are* type specific. This has lead to one of three paths being taken in the design of middleware:

1) Type checking and marshalling are avoiding, for simplicity, and/or to allow generic code to be used. Most messaging and group communications systems have taken this approach. The obvious disadvantage is that type safety, and utility, is lost.

2) Type specific code is generate on a case-by-case basis. As the type dependant code is application specific, this requires the automatic generation of middleware code (for example the use of stub compilers). Typically, this leads to heavyweight stubs that are protocol specific. Systems employing this technique tend to support few protocols, because of the need to generate different stubs for each.

3) The type information is encoded and passed as additional parameters. This adds to the complexity of the middleware system. CORBA ORBs employ a mixture of (2) and (3).

In Java, there is a fourth possibility. Java is strongly typed, and we may cast an object to a generic type (`Object`) and back to a specific type in a type safe way. In addition, the introspection facilities of JDK 1.1 provide sufficient information to allow marshalling to take place in generic code. This has allowed FlexiNet to be designed the following features:

1) Stubs are only used to cast invocations to a generic form. They are completely protocol independent. This has lead to stubs that are simple and universal. An ‘on-the-fly’ stub generator has been constructed to build these stubs. This would not have been a cost-effective approach with heavyweight or protocol-specific stubs.

2) All other middleware code is generic, but type safe.

3) Generic data may be passed as objects, rather than bytes. This allows Java to enforce the type safety, and makes debugging considerably easier.

4) Methods may also be passes generically, and are type checked by Java.

15.2 Stubs

The purpose of a FlexiNet stub is twofold. Its primary function is to convert an invocation form a type specific form to a generic form. This is represented by an Invocation object. The stub's secondary function is to store naming information to allow it several stubs to multiplex calls over the same communications stack.

An simplified example stub is illustrated in Figure 33. A complete example can be found in chapter 27.

```
public Class Foo_Stub extends FlexiStub implements Foo
{
  private Object body;
  private Name   name;
  static WrappedMethod barMethod = …;

  // method from proxied interface
  public Baz bar(String s,int i) throws myException;
  {
    Object[] arg = new Object[2]; // convert arguments to
    arg[0] = s;                   // object array
    arg[1] = new Integer(i);

    // construct new invocation object
    Invocation inv = new Invocation(barMethod,name,arg);

    inv.invoke(body); // invoke invocation on body

    Object rc = inv.getReturnValue();

    if(inv.isExceptionalResult())
    {
     // cast to approriate exception type and re-throw
      if(rc instanceof myExcepetion)
        throw (myException) rc;
      else if (rc instanceof RuntimeException)
        throw (RuntimeException) rc;
      else
        throw new FlexiNetRuntimeException(rc);
    }
    return (Bar) rc; // return normal result
  }
}
```

Figure 33 Example Stub

15.3 Invocation Class

The `Invocation` class is used to represent an invocation. It contains information about the target object and method, the arguments and (after being invoked) the return value or exception raised. For the most part, `Invocation` is a record class, with get and set methods. It also has a number of other methods, in particular invoke(..) which will execute the stored invocation on the target object. The methods and constructors are listed below.

```
Invocation()
```
Construct a new invocation object.

```
Invocation(WrappedMethod method,Object target,Object args[])
```
Construct a new invocation object from a (method, target, arguments) tuple. The method is given in a wrapped form, which allows efficient access to the method's signature for serialisation.

```
Object getTarget()
void   setTarget(Object target)
```
Get/set the target object upon which the invocation will ultimately be invoked. The target may also be a *representation* of the ultimate destination (for example a name for it). This feature is typically used in remote method invocation, where an invocation object on a client contains the *name* of the target object on the server.

```
Method getMethod()
void   setMethod(Method m)
```
Get/set the method to be invoked.

```
void setMethod(String name,String sig)
void setMethodClass(Class ifaceClass)
```
Set the method to be invoked in from it's signature, and the interface class that implements it. This is used as an efficient way of constructing an invocation from a serial form.

```
String  getMethodName()
String  getMethodSig()
Class[] getMethodParameterTypes()
Class   getMethodReturnType()
```
Get various attributes of the method. These functions are provided as more efficient access than the corresponding methods on `java.lang.reflect.Method`. In addition, they may be used before the method is fully resolved.

```
Object[] getArguments()
void     setArguments(Object[] o)
```
Get/set the array of arguments to passed when the invocation is invoked.

```
Object getReturnValue()
void   setReturnValue(Object o)
void   setExceptionalReturn(Throwable t)
```

Get/set the return value of the invocation. Normally these are set automatically, as a result of a call to `invoke()`. The method `setExceptionalReturn` should be used when the invocation raises an exception.

```
boolean isExceptionalReturn()
```

Determine if the invocation threw an exception.

```
void invoke() throws BadCallException
```

Perform the invocation on the stored target. If target implements the `GenericCall` interface, then `target.invoke()` is called, with this invocation as an argument. This may throw a `BadCallException`. Otherwise, if target implements the stored method, this is invoked using Java core reflection. If `target` implements neither of these, a `BadCallException` is thrown.

```
void invoke(Object target) throws BadCallException
```

A second form of invoke which takes an explicit target.

15.3.1 Engineering Support

Invocation objects are typically used to represent invocations that will ultimately be invoked on a remote machine. To aid the engineering of protocol stacks, Invocation objects may store references to a number of engineering objects relating to the reflection, or remote execution of the invocation. These are accessed via a number of additional methods as follows:

```
Session getSession()
void    setSession(Session s)
```

Get/set the session associated with this invocation. Sessions are engineering objects used to orchestrate communication with a remote machine. Sessions are described in section 16.5.

```
InputBuffer getInputBuffer()
void        setInputBuffer(InputBuffer x)
```

Get/Set the input buffer related to this invocation. On the client, the input buffer contains the serialised result, on the server it contains the serialised invocation.

```
OutputBuffer getOutputBuffer()
void         setOutputBuffer(OutputBuffer x)
```

Get/Set the output buffer related to the invocation.

```
void recycleInputBuffer()
void recycleOutputBuffer()
```

Discard and recycle the input or output buffer. This method is called once the buffer is no longer needed.

```
void   push(Class cls,Object obj)
Object pop(Class cls)
```

Invocation objects maintain a stack of ‘additional arguments’ or ‘additional results’ that may be used to pass extra information between engineering object on the client and sever. The push and pop methods take an additional argument, the *expected* class of the object being pushed/popped. This must be the same for a pair of push/pop operations. It is used to optimise the serialisation of the object’s class. See section 26.7 for details.

```
int getStackSize()
```

Return the size of the stack of additional arguments/results.

```
ObjList getRawStack()
void setStack(int noitems,DeSerializer deserializer)
```

A low level engineering interface to get/set the stack as a whole. Normally only used by the serialisation layer in remote invocation stacks.

Different fields within an Invocation may be valid at different stages of resolution or execution of the invocation. When the stub creates an Invocation, only the method, target and arguments are set. On return, the stub expects the result to be set. Other fields may be used to process the invocation or may be left unused.

15.4 Discussion

The invocation class is a general way of representing an invocation, and may be used in a purely local context. In particular a stub and invocation object may be used as a mechanism for achieving local method reflection. This is discussed in section 17.5.

In the design of the `Invocation` class, care was taken that invocations could be efficiently invoked on stub objects. This is important as client-side reflection is often used to wrap a remote invocation (so called Smart-Proxies). To achieve this, all FlexiNet stubs implement the `GenericCall` interface. When an invocation is invoked on an object implementing `GenericCall`, this corresponds to a simple and direct method invocation. Composition of stubs used for reflection and remote execution therefore gives both a high degree of abstraction, and high performance. Smart Proxies are described in section 17.3.

16 BINDING

16.1 Introduction

Binding is the process of linking a client proxy to a service object. It is undertaken by two complementary components. On the server, a *generator* generates a name for an interface. This name is passed to the client, and is then resolved, usually by a matching *resolver*. (See chapter 14). The term binder is used to describe both of these components, and in general, one object will be responsible for both generation and resolution.

Beyond this definition, the implementer of a generator or resolver has few restrictions. However, the majority of binders for remote communication have a standard form, and this is described in this chapter. It should be noted that not all binders will conform to this, and in particular, `Cache` and `SmartChoice` are two binders that do not.

16.2 Protocol Stacks

On the client, a stub is usually connected to the top of a *protocol stack*. This is a chain of *layers* which ultimately is connected to a network endpoint of some form (typically a socket or set of sockets). On the server, there is an analogous arrangement, with the top of the stack connected to the application level service object(s). This is illustrated in Figure 34.

The protocol stack covers much more than is usually regarded as a 'protocol'. In addition to the manipulation of network packets, the stack will be responsible for resource management, and any other invocation related functions, such as authentication, access control, auditing, locking, transactions, failure tolerance etc. The specifics covered by a particular stack are dependant on the class and configuration of the binder used to create it.

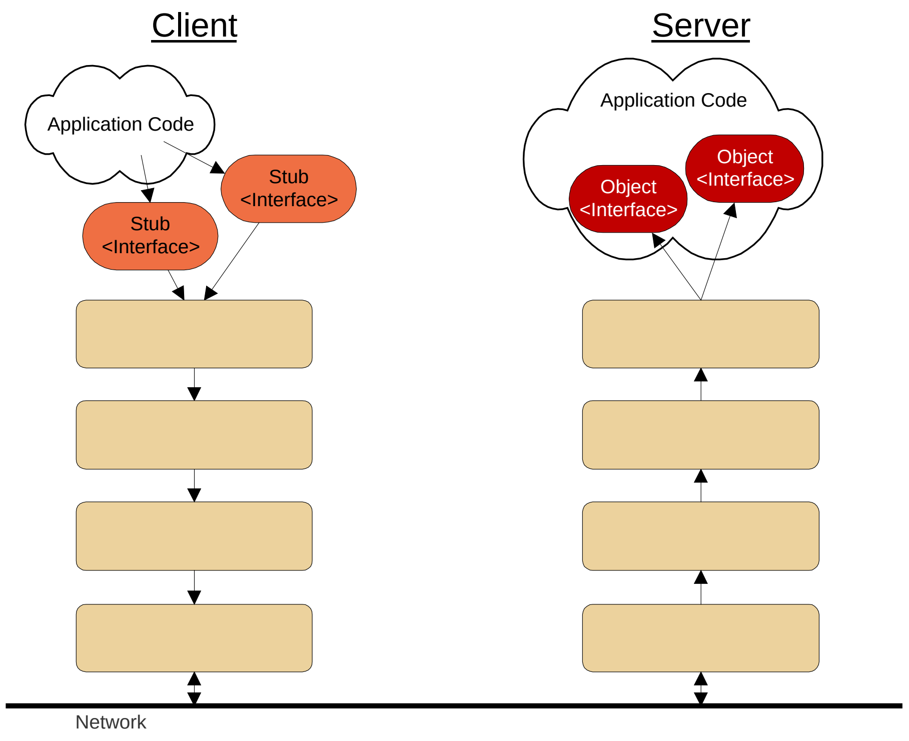


Figure 34 An Abstract Protocol Stack

16.3 Binders as Stack Factories

The primary purpose of a binder is to generate and/or resolve names. In a generator, when a name is generated, sufficient infrastructure must be constructed to allow invocations arriving off the wire to be recognised as invocations on the named interface. In general a generator must therefore create a new protocol stack associating some network endpoint with the named interface.

Similarly, in a resolver, when a name is resolved, the created stub must be associated with the top of a binding stack that will ultimately create a network connection with the peer on the server. This will again require the construction of a new protocol stack, or the reuse of an existing one.

16.3.1 Multiplexing

It would be extremely inefficient if every named interface was associated with a different network endpoint, and required a different protocol stack. For this reason, most binders can allow some degree of multiplexing through the stack, so that some or all of the layers are shared. In fact, for the majority of protocols the entire protocol stack can be shared. Exceptions to this rule are generally protocols that have strong requirements for unshared sockets – for example multicast or SSL protocols.

In the majority of binders, there is therefore a single protocol stack used for both the client (resolver) and server (generator). This stack is created upon the construction of the binder, or on the first call to generate or resolve. Upon subsequent calls to `generateName`, the binder configures a multiplexing layer within the stack to read and resolve the identity of the callee object. Upon calls to `resolveName`, a stub is created containing the name, and linked to the (shared) top of stack.

16.4 Generic Call

When designing the various layers that make up a FlexiNet protocol stack, the intention was to encourage reuse. For this reason, each 'invocation' layer corresponds to a standard interface. Logically, this interface should be `GenericCall`, as each layer gives an invocation abstraction to the layer above it (on the client) or below it (on the server). However, as a single layer object is often used to manage both client calls (down the stack) and server calls (up the stack) this interface cannot be used – as a Java class cannot implement an interface twice. To overcome this dilemma, two new interfaces are defined; `CallUp` and `CallDown` – these are simply distinguished synonyms for `GenericCall`.

The order of the layers may be varied, providing that the appropriate information is available in the `Invocation` object in order for a layer to perform its task.

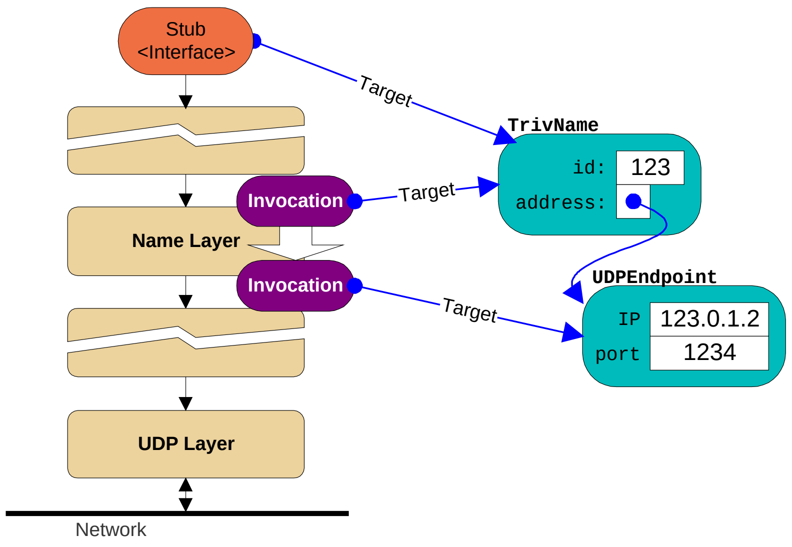


Figure 35 Decomposition of a Name

16.4.1 Decomposition of Names

In order to allow multiplexing layers to be reused in different protocol stacks, it was essential that they were not tied to strongly to the class of name being used. The approach taken is to construct names from a nested series of components which may be decomposed as the protocol stack is traversed. Figure 35 gives an example name and stack fragment. When the Name layer is called on the client, the 'target' field of `Invocation` contains the full name of the interface. The name layer extracts the interface id, and *overwrites* the target field of `Invocation` with the address from the `TrivName`. It is therefore not concerned with the format of this part of the name, be it a `UDPEndpoint`, `TCPEndpoint` or other addressing information.

Similarly, when the `UDPLayer` is called, the target field of `Invocation` contains a `UDPEndpoint`. The `UDPLayer` is not concerned with what form of multiplexing preceded it, or how the `UDPEndpoint` was incorporated into a full name.

The naming information recorded in an invocation object will vary from a full name to an arbitrary object containing addressing information. To preserve some type checking, the tag interface `NameFragment` is used to indicate an object used for naming.

16.4.2 Addresses

In general a FlexiNet name may represent an abstract interface or a group of interfaces. However at some point during invocation, the target interface for the invocation (or a sub-invocation) will be identified. Addressing information for this interface is stored in an object that implements the `Address` interface. This has a single operation that is used by the protocol stack to determine the low level multiplexing endpoint. For example, the service socket that the client must connect to. This endpoint is significant, as concurrent invocations to the same endpoint must be managed, to ensure that requests and replies are correctly matched.

16.5 Sessions

If a series of calls are made between a particular client and server, then there may be scope to develop a 'shared model' of part of the environment, and reduce the per-call overhead. We call such an abstraction a 'session'. Simple examples of this include piggy-backing reply acknowledgements onto further requests, and co-ordinated failure management. More complex examples might include building shared dictionaries of short-codes for commonly used class or interface names, or establishing 'session keys' for security.

In some systems this role is taken by a *connection* object, for example a TCP connection. However this is inadvisable, as the trade-offs related to connection duration and session duration are quite different. For example, on a dial up line, maintaining a connection may be prohibitively expensive – however, maintaining a *session* would lead to more efficient calls, and hence a reduction in cost.

In a FlexiNet client stack, each invocation that has been resolved to a particular `Address` is associated with a session object. The session object has a number of standard fields, and in addition acts as a dictionary for additional (key,value) pairs. On the server, in incoming invocation is associated with a session as soon as the client's claimed identity can be determined.

The session object may be used to cache information between calls, and to pass additional information between layers in a single call. The lifetime of sessions is maintained by the RPC Layer (as it is closely tied to the RPC protocol), and in general a session may be destroyed at any time when a call is not actually in process. Sessions should therefore be used to *cache* information, rather than to store it permanently. For most protocols under moderate load, sessions are expected to survive for a least a few minutes.

16.6 Concurrency

Probably the biggest complication in binder engineering is support for concurrency. Unsurprisingly, it is also one of the biggest causes of lurking bugs. Problems tend to occur with race conditions between incoming or outgoing messages and timeouts for resends or session/connection termination. In FlexiNet, a simple concurrency control model has been designed, to reduce the incidence of bugs. As the performance of a protocol is closely linked to the threading policy, the model does not dictate a particular policy, but makes it easier to manage concurrency regardless of the threading policy.

The essential idea is that within a particular region of the protocol stack, we use a mutex to ensure that there will be at most one call in progress *on a particular session*. As calls on different sessions are independent, this removes concurrency concerns from the majority of layers. In general only the layers responsible for gaining the mutex need be concerned with concurrency at all. The mutex is engineered as part of the session object. It is taken in three different circumstances.

- When a session is associated with an outgoing call. This is managed by the `ClientCallLayer` (or an equivalent).
- When a session is associated with an incoming message. This is managed by the `SessionLayer` (or an equivalent).
- When a timeout occurs, and some action needs to be taken; for example the destruction of the session, or a retransmission. This is handled in the RPC Layer.

In each of these cases, care is taken to deal with all possible race conditions. This removes the majority of the concurrency control complexity, and leaves other layers free to manipulate sessions and handle multiple calls. This is illustrated by Figure 36.

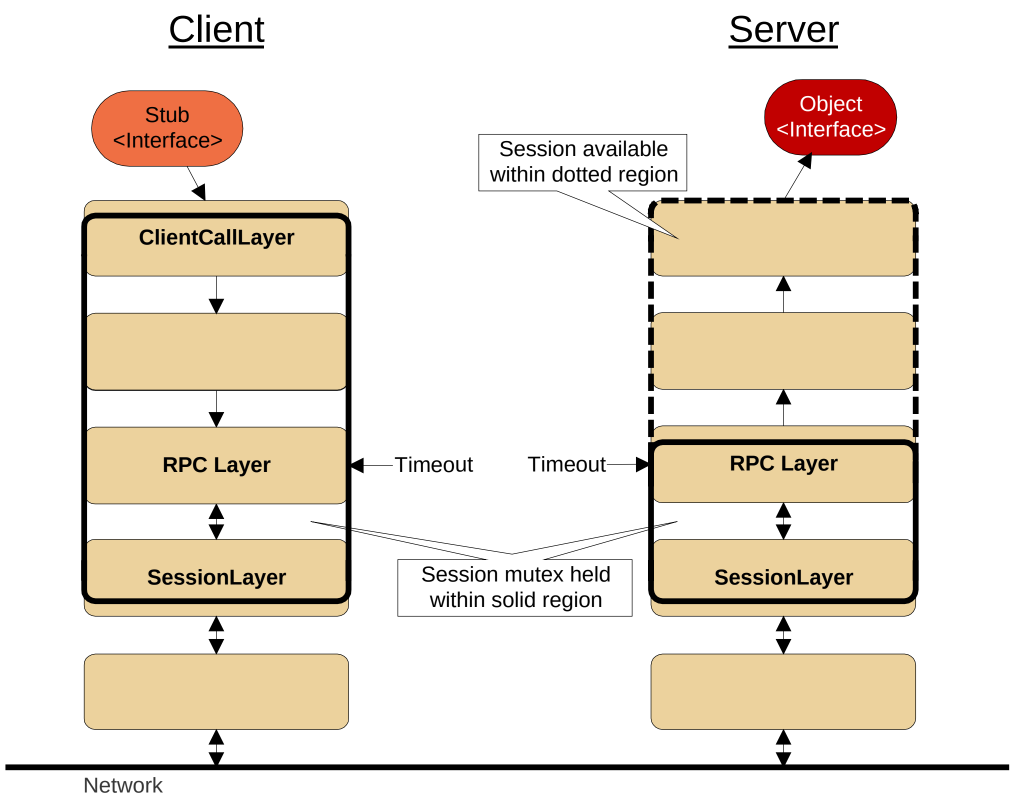


Figure 36 Session Mutex

Let us consider the flow of an invocation in detail. When an invocation is made on a client stub, it will pass down the stack until it reaches the `ClientCallLayer`. Up until this point, a session has not been allocated to the invocation, and there may be many similar invocations in progress at once. However, the calls are independent, and any shared state will be contained within a particular layer (which must perform its own concurrency control).

In the `ClientCallLayer`, a session is obtained from the `SessionManager` before proceeding. In the standard implementation of `SessionManager`, this never blocks, and an existing session is returned, or a new one created if no suitable ones are available. Generally, there will only be a small number of sessions in use between a particular client and server. Once the session has been obtained, the mutex is taken. There is some careful coding here, as there is a potential race condition between obtaining a session and acquiring the mutex. The chance of race condition is low, and a rapid retry loop is used to handle it. In some message based protocols, the attempt to obtain the mutex may cause the thread to block. This can only occur if a protocol message is received on the session during the race period. If this occurs there will be a short delay – but long term delays cannot result.

Layers below the `ClientCallLayer` may safely use the session without concern for concurrency issues. The session mutex may be held on the client for the duration of the call, or it may be temporarily released within the RPC Layer whilst the thread is blocked awaiting a reply. The details of this are dependent on the RPC Protocol implemented by the particular RPC Layer. This must be aware of, and deal with, any race conditions.

Typically, the RPC Layer will manipulate a protocol state transition table, and store the current state for each session, within the session object. This may be safely manipulated within the mutex-region and used to manage race conditions.

On the server, a received call is associated with a new or previous session as soon as the client's peer session can be determined. Typically this is the first thing to be read from the incoming message. (In some connection-oriented protocols, the session is tied to the connection, so the session mutex is held even before the message is read). Again, the chance of delay due to blocking is low. This may only occur if the session itself it timed-out, which is an unlikely event. If timeout *does* occur, the call is discarded, and it is left to the client to retry. In this rare case, a new session will be allocated (possibly after a protocol specific session establishment exchange).

The session mutex is held at least until the RPC Layer. In message based protocols, the mutex is released before the call proceeds further up the stack. This is to allow the call to be processed in parallel with further protocol messages. For example in the Rex protocol, the client may *probe* the server to check that it is still processing the request. Although the mutex is released, the session may still be used by subsequent protocol layers, as the RPC Layer must preserve the following invariant. *There will be at most one server-side call in progress above the `RPCLayer`, per session, and the session will not be destroyed until it completes.* As the session may be simultaneously be used below and above the RPC Layer, care must be taken that state stored in the Session object is not inappropriately shared between these layers. It is up to the layer designers to ensure this. (The Invocation object provides an alternative place for shared state).

This approach to concurrency control has prove particularly useful when adding low level protocol layers, such as the Rex Fragmentation layer (See section 20.3.5), as the strong guarantees it provides considerably simplified the design.

16.7 Resources

There are a number of resource classes used within a protocol stack. In particular threads, buffers and sessions need to be allocated and destroyed. An issue here is maintaining the separation between layers. For example, on the client, a high layer must create an output buffer for a request, but this output buffer will ultimately be used by the bottom layer in the stack. The naive approach would have been to define a single class for each resource type, so that layers could be freely mixed. This was rejected, as it would impose a 'one size fits all' strategy for different protocols. Instead, we use resource factories to provide an *abstract resource creation* interface. In the example, the high client layer would contain a reference to an output buffer factory which it would use to create output buffers on demand. The binder could then choose an appropriate factory to best meet the needs of the bottom layer.

This topic is returned to in chapter 28 where resource management is considered in more detail.

16.8 Binder Configuration – Blueprints

Most binders construct a protocol stack of a number of layers, and link these layers to each other. In addition they must inform the layers of resource factories and helper objects. There is generally a degree of flexibility in the configuration of a binder – for example there may be alternatives for some layers, that provide a different degree of multiplexing, or an alternative network protocol. There is also typically a choice of resource factories, or policies for the reuse of resources. Finally, simple parameters may be altered, such as the allowed degree of concurrency, or the port to listen on. The configuration of such a potentially complex system is non-trivial. A 'blueprint' system has been designed which allows a binder to be specified and checked for consistency. This is described in chapter 29.

17 REFLECTION

17.1 Introduction

Classic reflection may be considered as a 'look aside' on some aspect of the execution of a system. FlexiNet is concerned with method reflection. In a classic system, when a method is invoked on an object, the call is *reflected* to a meta object which may examine and modify the invocation before allowing it to proceed. Similarly, the result is reflected to the meta-object, which may modify it. Typical uses of reflection include auditing of requests, or controlling concurrent accesses to an object (as the meta-object may block and/or reorder requests).

In some senses, an RPC system can be considered as reflective. A client invokes a method on an object, and this is instead passed to a meta-object (The RPC system) which manages the invocation. The use of generics within FlexiNet encourages this approach.

17.2 Reflection in Protocol Stacks

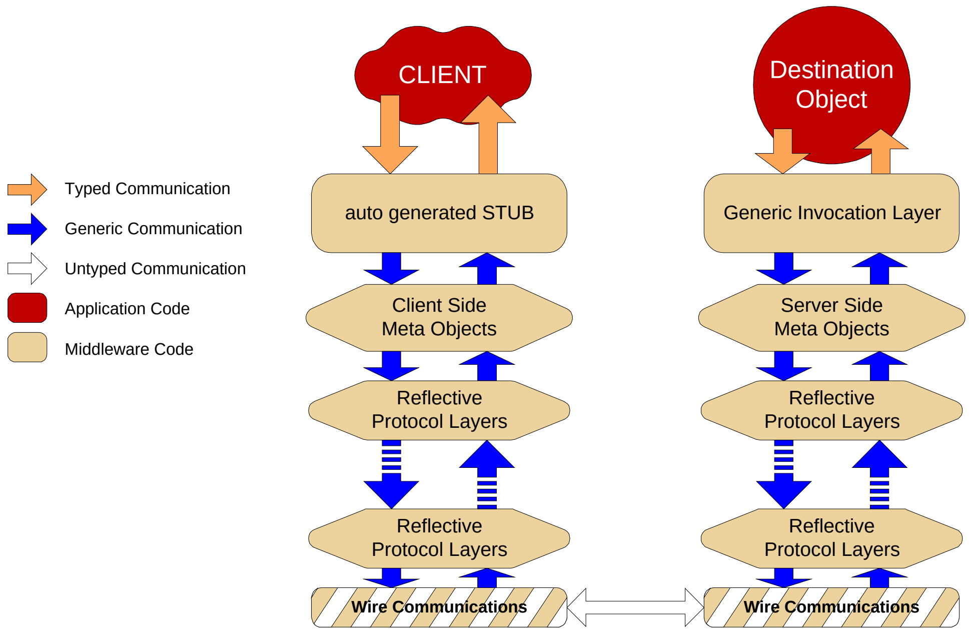


Figure 37 A Reflective Protocol Stack

The function of a stub in FlexiNet is very similar to the function of a reflective object in a reflective object system: it converts a call from a type specific, to a generic form. We may therefore consider each layer in a protocol stack as a reflective layer, and indeed high level layers on either the client or server may perform exactly the same functions as meta-objects in a Classic reflective object system (Figure 37). This form of reflection is most appropriate for protocol-specific reflection as opposed to application or class specific reflection – this is because all objects named using a protocol will generally have the same reflective layers.

The design of protocol stacks, and binders to create them, is a complex task. Frequently, a programmer wishing to utilise reflection will not be skilled in the construction of binders, and in addition may wish to use the same reflective layers over multiple 'standard' protocols. A more appropriate interface to reflection for many applications is therefore the use of *generic smart proxies*. This will be described in section 17.4, after the simpler non-generic case is considered.

17.3 Smart Proxies

Smart Proxy is a term used to refer to code that is loaded onto a client in place of the standard (dumb) proxy created by a stub-stack pair. Typically, a smart proxy is used when some 'intelligence' is required on the client side of a invocation, for example to perform caching. Such intelligence is usually service specific, and although it *could* take place within a custom protocol stack, a lightweight approach is more appropriate.

In FlexiNet, the use of smart proxies is initiated by the server, and the client need not be aware that a particular interface is handled by a smart proxy. In fact, the name resolution process described in section 14.3.2 is all that is required in order to implement smart proxy-like behaviour. However, in order to make it easy to write smart proxies without knowledge of the resolution process, an additional API in the form of the `SmartProxy` superclass has been defined.

17.3.1 Mechanism

The Smart Proxy implementation makes use of the fact that `Name.resolve` may return *any* object implementing the required interface. We may therefore define specific naming classes that encapsulate proxy objects, which may be returned from calls to `resolve`.

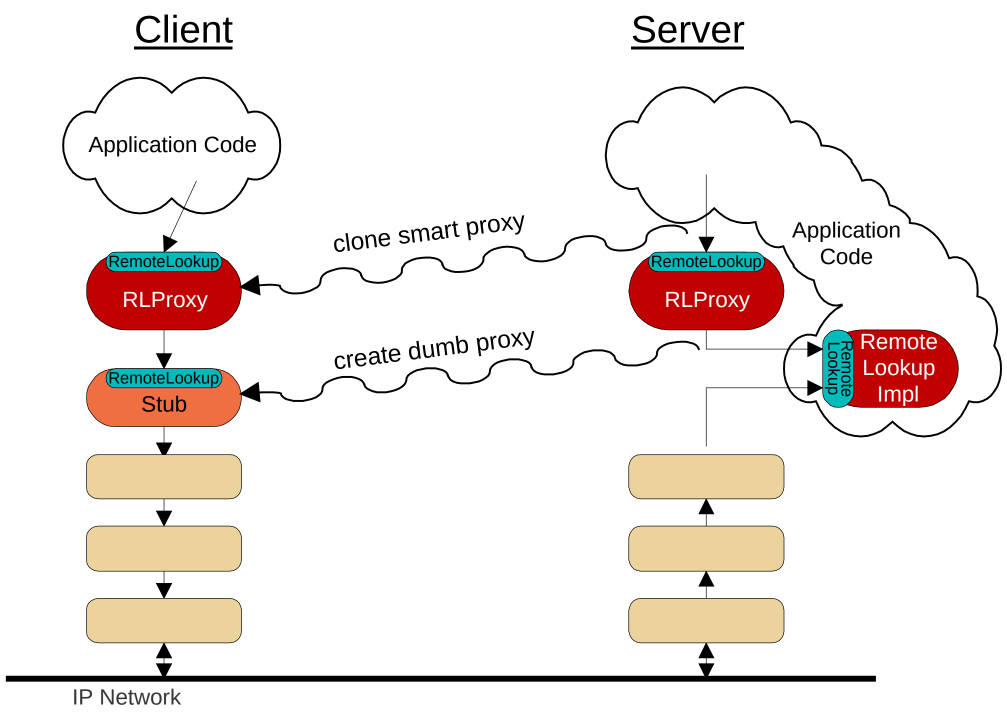


Figure 38 The Object Graph for an Example Smart Proxy

A secondary issue is how a service arranges that (a name for) a Smart Proxy is passed in preference to an ordinary reference. The straightforward approach is to define a Generator to spot cases where Smart Proxies should be used, and generate a name for an appropriate smart proxy. This approach is reasonable, however it assumes that the designer of a Smart Proxy has a reasonable understanding of the naming architecture. It is also relatively complex for a 'simplifying' abstraction.

To avoid these issues, a lateral approach was required. We note that, by definition, a service is aware that it is using Smart Proxies. We can use this fact to give straightforward semantics to the use of Smart Proxies. We arrange that reference to smart proxies themselves are the only special cases. A server may therefore create a proxy locally, and then pass a reference to this (rather than to the service itself). FlexiNet will spot this, and create a smart proxy on the client to match. The proxy itself will normally contain a reference to the 'real' service. This will be treated as an ordinary reference and a dumb proxy will be created accordingly.

This approach provides a simple and straightforward means for a programmer to indicate that a smart proxy should be used, and does not require the any additional management APIs.

The object graph resulting from the use of a smart proxy is illustrated in Figure 38. The definition of the proxy itself is given in the following section.

17.3.2 SmartProxy Class

The abstract class `SmartProxy` has been created to manage the complexities of name generation and resolution. A programmer wishing to use a smart proxy, extends this abstract class by adding the appropriate methods for the proxied interface. Generally, they will also wish to add a reference to the real service interface. An example caching proxy for a read-only service is given in Figure 39. This corresponds to the example object graph in Figure 38.

The `SmartProxy` class works by extending the `Name` and `ProxyObject` interfaces. As it is a `ProxyObject`, it will be directly asked for the name it represents during serialisation, rather than having a name generated for it. (Section 26.10.2). It returns itself as a suitable name. During deserialisation, the `resolve` method will be called on the smart proxy (acting as a `Name`), it will then return itself as a suitable proxy.

```
public class RLProxy extends SmartProxy implements RemoteLookup
{
  // reference to ‘real’ service
  public RemoteLookup service;
  // local cache of values. Service is read only
  // so no invalidation protocol is needed.
  transient Hashtable cache;

  public RLProxy() { this(null); }

  public RLProxy(RemoteLookup s)
  {
    service = s;
    cache = new Hashtable();
  }

  // RemoteLookup.get(key)
   public Object get(String key)
  {
    // first look in cache
    Object value=cache.get(key);
    if(value==null)
      {
        // not in cache, lookup in service
        value = service.get(key);
        if(value!=null) cache.put(key,value); // store in cache
      }
    return value;
  }
}
```

Figure 39 Example Smart Proxy

17.4 Generic Proxies

It is often useful to combine the flexibility of binder reflection, with the ease of programming of smart proxies. The *generic proxy* abstraction is designed to meet this requirement. A generic proxy is an object that sits between a standard stub and the top of the communication stack. It therefore acts like an additional layer to the stack, and provides an environment for reflection, but it is not tied to a particular binding protocol, and may be used with many binders.

Like Smart Proxies, Generic Proxies are actually FlexiNet names, however the generation and resolution process is different. As Generic Proxies are type independent, the technique used for generating Smart Proxies is not applicable. Instead, a special `Generator` must be constructed to generate Generic Proxies of a particular class. Like any generator, this may choose whether to generate a name for a particular interface, or whether to pass it to the next generator in the list (See section 14.3).

The designer of a generic proxy generator needs to understand a little about the use of FlexiNet names. It is not possible to simply construct a proxy containing a direct reference to the service object – as serialisation of this object will lead to that reference been replaced by another proxy, leading to unbounded recursion. Instead, the proxy should contain a `Name` for the interface, which may be generated by calling the next generator in the list (thus avoiding recursion).

In general, a generic proxy may contain many names. The abstract classes `GenericProxy` and `GenericProxyGenerator` are provided to help with their construction. The majority of generic proxies contain only one name – the name of the 'real' server object. To further simplify the construction of such proxies, the abstract class `SimpleGenericProxy` is provided. Figure 40 gives the Smart Proxy example from Figure 39 as a generic proxy using the `GenericProxy` class, and Figure 41 gives the same example using `SimpleGenericProxy`. The two examples result in an identical object graph, shown in Figure 42.

Invocations are made by the client to a stub object, which then calls the `invoke` method on the generic proxy. (See section 15.3). The proxy then manipulates the call and if appropriate, performs a remote invocation by calling a second stub object. It should be noted that these nested calls are handled by the stub's `GenericCall` interface, and have a low overhead.

```
public class ROProxy extends GenericProxy implements Name
{
  public Name serviceName; // the name of the real service

  transient Object service;// (a dumb proxy for)
                           // the real service
  transient Hashtable cache;

  public ROProxy(Name n)
  {
    serviceName = n;
  }

  public ROProxy() {}

  // Name.resolve - called on client on creation
  public Object resolve(Class iface,BinderDB ctxt,
                      FlexiProps qos) throws BadName
  {
    cache = new Hashtable();
    service  = serviceName.resolve(iface,ctxt,qos);
    return super.resolve(iface,ctxt,qos);
  }

  public void invoke(Invocation i) throws BadCallException
  {
    // assume method=get(string)->object (by construction)
    String key = (String) i.getArguments()[0];

    Object value=cache.get(key);
    if(value==null)
     {
       i.invoke(service); // perform remote invocation
       if(!i.isExceptionalReturn())
        {
          value = i.getReturnValue();
          if(value!=null) cache.put(key,value);
        }
     }
    else
      i.setReturnValue(value);
```

Figure 40 Example Use of GenericProxy

```
public class ROProxy2 extends SimpleGenericProxy
{
  private transient Hashtable cache;

  // for GPGenerator
  public ROProxy2(Name n)
  {
    super(n);
    cache = new Hashtable();
  }

  // for serialization
  public ROProxy2()
  {
    cache = new Hashtable();
  }

  public void invoke(Invocation i) throws BadCallException
  {
    // assume method=get(string)->object (by construction)
    String key = (String) i.getArguments()[0];

    Object value=cache.get(key);
    if(value==null)
      {
        // perform the invocation usng super.invoke
        super.invoke(i);
        if(!i.isExceptionalReturn())
          {
            value = i.getReturnValue();
            if(value!=null) cache.put(key,value);
          }
     }
    else
      {
        i.setReturnValue(value);
      }
  }
}
```

Figure 41 Example Use of SimpleGenericProxy

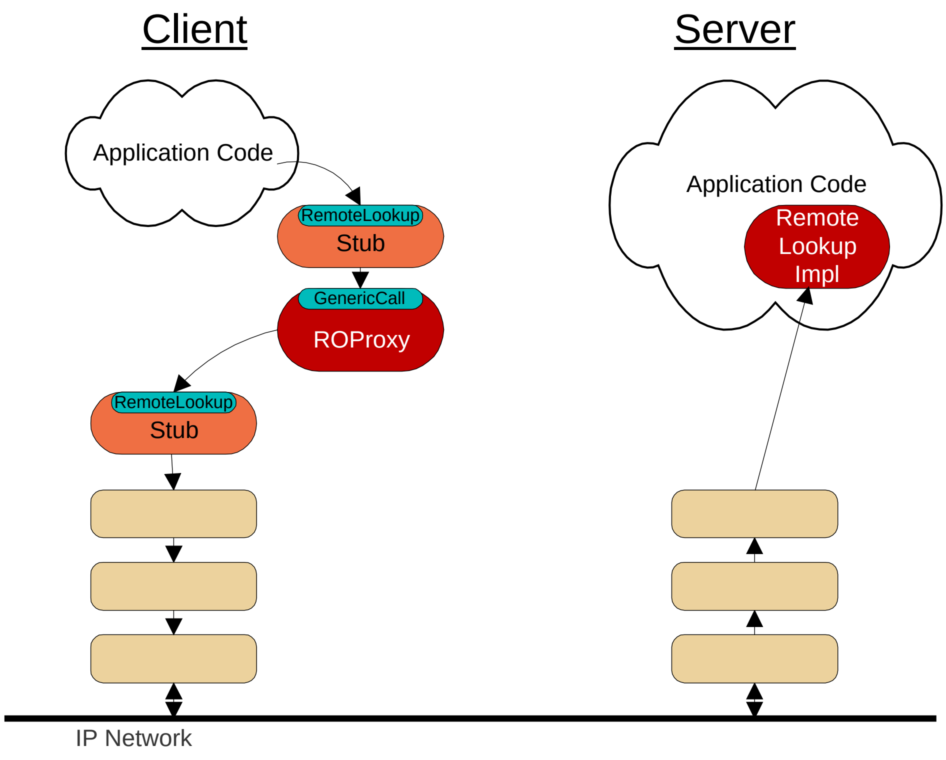


Figure 42 Use of Generic Proxies

```
public class  RLGenerator extends GenericProxyGenerator
{
  public Name generateName(Object obj,Class cls,FlexiProps qos)
  {
    // only generate names for a particular class
    if(cls==RemoteLookup.class)
      {
        // create a skeleton to wrap the object on the server
        RLSkeleton skel = new RLSkeleton(obj);

        // generate a name this skeleton
        Name basename = generateBaseName(skel,cls,qos);

        // return a proxy that references the skeleton
        return new RLProxy(basename);
      }
    else
      return generateBaseName(obj,cls,qos);
  }
}
```

Figure 43 An Example Generator that Creates a Proxy/Skeleton Pair

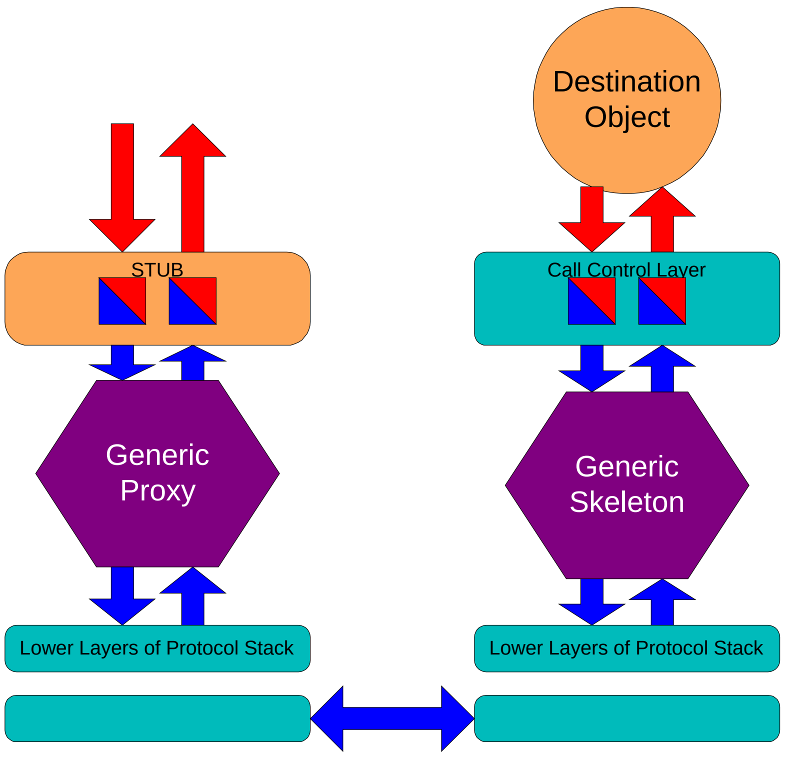


Figure 44 Using Proxies and Skeletons for Reflection

17.4.1 Generic Skeleton

FlexiNet does not normally require server-side stubs (or Skeletons as they are often called). However, when using generic proxies, it is often useful to add a reflective layer to the top of the server stack. This can easily be arranged in the Generic Proxy Generator, and an example generator to do this is shown in Figure 43.

Together a generic proxy/skeleton pair can be used to augment an existing protocol with an additional 'meta' layer on each side. For transaction processing in FlexiNet, for example, this technique is used to pass the transaction Id from client to server, as an additional parameter in each call. Figure 44 illustrates.

17.5 Local Reflection

The generic call interface may also be used when building meta-objects for local reflection. For local reflection a *reflective object* is required that implements the service interface, and hands off invocation to a *meta-object* that provides a wrapper around the *service-object*. In FlexiNet terms, this corresponds to a proxy object (which is a simple stub), a meta-object (which implements the `GenericCall` interface), and the unmodified service-object. This is illustrated by example in Figure 45.

Application Class to be reflected
```
  Class FooImpl implements Foo
  {
    …
  }
```

Meta-Class
```
  class MetaAudit implements GenericCall
  {
    Object realObject;

    public MetaAudit(Object o)
    {
      realObject = o;
    }

    public void invoke(Invocation i)
                    throws BadCallException
    {
      System.out.println(“Calling: ” + i.getMethodName());
      i.invoke(realObject);
    }
  }
```

Application code using reflected object
```
  GenericCall g = new MetaAudit(new FooImpl());
  Foo reflectedFoo = (Foo) FlexiNet.reflect(g,Foo.class);

  // may now use reflectedFoo
```

Figure 45 Local Reflection

18 RESOURCE MANAGEMENT

18.1 Introduction

Resource management is concerned with controlling the population of different kinds of resource. Without a resource management abstraction, resources are simply created and destroyed as required. By recycling or pre-allocating resources, we can amortise the cost of creation and garbage collection. By limiting the population of different types of resource we can prevent or control overload. In particular, during overload, the additional cost relating to allocation or garbage collection will exasipate the problem. In addition, if some resources are bounded (for example total CPU, or memory) then controlling the resources explicitly can allow graceful degradation of service (for example a server may refuse to allow additional clients to connect when overloaded).

The resource management abstractions introduced in this section are used for a secondary purpose. As they abstract the creation of resource, they may be used for *abstract resource management*. This is where a component creates a resource without having to know its concrete type. For example in the Serial Layer, a Serializer resource must be created in order to serialise new requests. As this creation is abstracted by a serialiser factory, the Serial Layer can be configured to use different serialisation policies by changing the factory that it uses. Abstract resource management is a key feature of FlexiNet, and is a pre-requisite for the reuse of many protocol layers.

18.2 Factories

The simplest resource management abstraction is a Factory. A factory is an object that constructs resources of a particular class on demand. This has two advantages over the naive use of a constructor.

- The callee need not know the concrete class of the object being created
- There can be many factory instances for the same class, and each can maintain different 'static' state.

An example Factory class is shown in Figure 46.

```
public class MyTicketFactory implements TicketFactory
{
  private int nextNumber;

  class FooBarTicket implements Ticket
  {
    int no;
  }

  // abstract constructor for Ticket
  public synchronized Ticket getTicket()
  {
    FooBarTicket t = new FooBarTicket();

    t.no = nextNumber++;

    return t;
  }
}
```

Figure 46 An Example Resource Factory

18.3 Pools

A pool is a factory that recycles. In addition to creation mechanisms, the pool has a 'recycle' method that can be used to return used objects so that they can be recycled for later use.

18.3.1 Issues with using Pools

Originally, FlexiNet made heavy use of pools, however in retrospect, this was a poor decision.

The primary problem with the use of pools was the difficulty in debugging. By managing some resources within pools, it was essential that the resources be correctly allocated *and freed* (i.e. returned to the pool). If a resource was returned to the pool early (when some other object still contained a reference to it), the resource might be reallocated, and two objects would then be unwittingly sharing the same resource. Equally, if a resource was never returned to the pool, then this would result in a 'leaked resource' and the pool would run dry. In addition, as resources are, in general, arbitrary objects, they contain a number of fields that are set to default values on construction. When a pool reallocated a resource, it was essential that all fields were set back to the correct values. This code had to be hand-crafted, and a mistakes were hard to spot.

The use of pools accounted for the vast majority of lurking bugs that lead to unpredictable failures in FlexiNet (generally when resource usage got high). We had abandoned the Java garbage collector, and were paying the price.

The solution to these problems was to abandon the use of pools almost completely. We re-evaluated the benefit of using resource pools on a case by case basis. For all cases bar one, it was decided that the recycling nature of pools caused more harm than good. As Java is implemented entirely on a heap, and all transient objects must be created and freed from the heap, there was little utility in treating some object classes as 'special' and (in effect) maintaining a private heap. The only exception to this was in the recycling of 'packets' – simple fixed sized byte arrays used in the basic buffer implementation. This special case is discussed in section 28.3 after basic buffers have been described.

The secondary use of pools, the accounting of the number of resource in existence *is* of benefit – but only for resource classes where it is meaningful to control the population. For each case we had to trade off the added code complexity of the need to recycle resources, with the benefit of accounting. For most cases, we decided to use a simple Factory abstraction instead.

18.3.2 Managing Numbers of Resources

When controlling the creation and destruction of a resource by the use of a pool abstraction, it is possible to control the overall number of resource instances in existence. This may be used, for example, to ensure that a fixed number exist, or that there is an upper (or lower) bound on the total number. More subtle policies include controlling the bounds on the number of *available* resources rather than the total, or policies that do not control the number, but audit it, either for debugging, or for performance profiling.

In FlexiNet, each `Pool` contains a reference to a `PoolManager` responsible for managing the number of resources in existence. Whilst the pool implementation understands the concrete type of the resource being managed, and its specific features with respect to construction, destruction and reinitialisation; the resource manager only understands *abstract* resources, and is responsible solely for maintaining a set of instances, and informing the pool when a new resource should be created, or an old one destroyed

Whenever a resource is requested from the pool, the pool passes the request to the Pool Manager. This may refuse the request, block the request until a resource is available, return a resource from the pool, or indicate that a new resource should be created. In the latter case, the pool constructs the resource, and returns it to the client.

When a resource is returned to the pool, the pool passes the resource to the pool manager, which may store it, or request its destruction. In the later case, the pool is responsible for any 'tidying' prior to dropping the object to allow it to be garbage collected.

A number of different resource manager implementations have been built:

`RecyclePoolManager`
: This maintains an upper bound on the total number of resources held within the pool. When a resource is requested, one is returned if the pool is not empty. If the pool is empty, then a new resource is created if less that the upper bound exist, if not the callee blocks/fails (there is a blocking and a non-blocking interface). Returned resources are stored and never freed.

`CachePoolManager`
: This is the usual pool manager used for buffers. It maintains an upper bound on the number of *available* resources. On a request, a resource is returned from the pool if available, and a new one is created if not. When a resource is returned, it is stored in the pool if the pool is not full (upper bound reached).

`DebugPoolManager`
: This audits the use of resources and checks for resources allocated in one pool and erroneously returned to another.

`NullPoolManager`
: If no pool manager is specified, resource are created on demand, and destroyed on return (i.e. the pool acts like a factory).

18.4 Resources

There are several different resources used within FlexiNet that need to be controlled.

Buffers
: For input and output of remote messages. For performance, it is useful to reuse buffers, as this reduces the garbage collection and heap fragmentation overhead. The number of buffers kept in reserve will effect both performance and memory overhead. Buffers themselves are created by buffer pools. The standard buffer implementation (Basic Buffers) uses a pool to store the *packets* that make up the buffer.

Sessions
: The number of sessions active at any one time is directly related to the number of clients that a service can support. Attempting to support to many clients will lead to starvation of other resources. The `SessionManager` is responsible for the construction of sessions. It uses a `SessionFactory` to create new instances.

Threads
: Typically a protocol will have one or more threads active and ready to receive a request. If there is no thread waiting when a request arrives, then this will lead to a delay. There is a `ThreadResource` abstraction for pooled threads, however most protocols integrate the thread management into the protocol

management, as this can give additional performance in this critical area.

Other Resources

There are a number of other object classes that are treated as resources, and are allocated using Factories or Pools. This is primarily to aid the construction of reusable components.

PART FOUR: ENGINEERING COMPONENTS

19 INTRODUCTION

19.1 Debug

Before diving into the detailed design of the various engineering components within FlexiNet, it is worth understanding the design and use of the FlexiNet tracing system. This may be used to selectively trace parts of FlexiNet, and is extremely helpful when trying to understand their function, or when tracking down bugs.

The tracing system is implemented in the static class Debug. This has one primary method, `trace(…,string)` which is called thousands of times throughout the FlexiNet code. Debug will select which of these trace statements should be output to the screen, based on a configuration script read at start up. As the cost of calling trace() is high – even if the string is not output, a stylised calling convention is used. Code wishing to call trace, should first check if the static variable `Debug.on` is true. For example

```
if(Debug.on)
  Debug.trace(this,“About to call foo(“+arg+”)”);
```

The variable `Debug.on` is set in one of two ways. For a final deployment, it should be is set to ‘`public static final boolean on=false`’. By recompiling FlexiNet, this will *compile out* all `Debug.trace()` statements within the code. For development use, `Debug.on` is set to ‘`public static boolean on=false`’. This will cause debug statements to be compiled in, but stepped past. The debug system will check the configuration file, and if some debugging is enabled, the `Debug.on` will be set to true.

There is therefore a performance cost associated with:

a) Compiling in the trace statements (as they must be stepped past)

b) Enabling *any* tracing, as all trace statements will be called, though only selectively printed. Enabling tracing typically slows FlexiNet to around 50% of its original speed.

19.1.1 Debug Interface

The Debug system has twelve public methods. All are static.

`void trace(Object obj,String msg)`
`void trace(Class  cls,String msg)`
`void trace(Object obj,String msg,Throwable t)`

> If tracing is enabled for the specified class, output the message `msg`, together with the optional exception (The exception back trace is also output). Calls to trace with an object, rather than class parameter are for ease, and correspond to `trace(obj.getClass(),msg)`.

`void traceX(Object obj,String msg)`
`void traceX(Object obj,String msg,Throwable t)`

> As `trace`, but only trace if *extended tracing* is enabled. Extended tracing is identical to 'normal' tracing, and is used to indicate trace statements that will produce a large amount of output.

`boolean tracing(Object obj)`
`boolean tracing(Class cls)`
`boolean tracingX(Object obj)`

> Test if tracing (or detailed tracing) is enabled for a particular class/object. This may be used by code wishing to avoid an expensive call to `trace`(...) that would be discarded – for example the message may have to be constructed from many pieces.

`void help(String msg)`

> Output a message that relates to an unexpected event (for example a class not found), that is likely to be an application bug. Typically a help message will be produced whenever the 'correct' behaviour would be a silent failure. They do not necessarily correspond to errors. Help messages may be enabled independently of ordinary tracing. Help messages are a candidate for internationalisation (see section 19.2) although this is little used in the current implementation.

`void bug(String s)`

> Report a bug in FlexiNet. A standard message together with the specified string is output. The program then terminates.

`void assert(boolean flag,String s)`

> Validate that flag is true, if false call `bug(s)`.

`synchronized String getThreadID(Thread thread)`

> In trace statements, an identifier for the current thread is output. This method allows the application to obtain an identifier of the same format. It is used when debugging thread synchronisation.

19.1.2 Configuring Debug: The .debug file

The `.debug` file is used to configure which classes should have tracing enabled for them. It consists of a list of statements, each of which enables or disables tracing of a particular class, a particular package or a sub-package.

The order of the statements is unimportant; each class will be traced according to the most closely binding trace statement.

For example, a typical .debug file might be.

```
# Tracing for serialization
-UK.co.ansa.flexinet
+UK.co.ansa.flexinet.basecomms.serialize.ref
-UK.co.ansa.flexinet.basecomms.serialize.ref.RefSerializer
+X_UK.co.ansa.flexinet.serialize.ref.RefDeSerializer
+UK.co.ansa.flexinet.basecomms.layers.CallLayer
+UK.co.ansa.flexinet.basecomms.layers.ClientCallLayer
```

- The first line is a comment. Blank lines, and lines beginning '#' are ignored. All other lines should start '-' or '+'.
- The second line disables tracing for all classes in packages beginning "`UK.co.ansa.flexinet`". This is the default action, and this line is redundant.
- The third line enables tracing for all classes in the package "`UK.co.ansa.flexinet.basecomms.serialize.ref`"
- The fourth line disables tracing for a specific class.
- The fifth line starts "`+X_`". Lines like this enable *extended* tracing for a class or package. Extended tracing is typically used when a large amount of trace code is generated. This line enables extended tracing for `RefDeSerializer`. This will produce detailed information about each object that is deserialised.
- The final two lines enables tracing of the `CallLayer` and `ClientCallLayer` classes. These are often worth tracing is they are typically the top layers of the server and client stack respectively, and output trace statements that show when a call enters and leaves the FlexiNet system.

19.2 Internationalisation

The internationalisation system can be used to 'translate' strings stored within a static class into different languages. It has been designed to make it straightforward for a programmer to define and use such strings, and efficient for them to be accessed.

The basic approach is to have a 'Text' class that extends the FlexiNet `Language` class, and provides a number of public static strings that may be efficiently used by the application code. For example:

```
if(status==1)
  System.out.println(Text.call_failed_unknown);
```

The `Language` class ensures that the all such fields are initialised to appropriate values for the current locale. To do this the programmer provides a number of language-specific sub-classes of `Text` within the same package, which define values for each of the strings. These classes are named after the display language for the locale, for example `French.class`, `English.class`, `German.class` etc.

Strings in a particular language may also be accessed directly, for example:

```
System.out.println(French.failed_to_contact_server);
```

Both of these mechanisms are extremely efficient, and require only three byte codes (as opposed to two for access to a string literal). In addition, as the keywords used to identify the strings are Java field names, the java compiler will spot if any are mistyped, and report an error.

In addition to these mechanisms, an auxiliary mechanism is provided for applications that need to deal with several different languages, and that need to pass locale information as variables. This may be performed as follows:

```
Errors l = new German();
System.out.println(l.get(Errors.not_multicast));
```

This mechanism is less efficient, but is required in some circumstances.

The implementation of the `Language` class is straightforward. Upon initialisation of a 'Text' subclass it determines the current locale and locates the language-specific subclasses. If locale information is unavailable, then the subclass can supply an explicit default. The fields in the subclass are then examined, and for each, the appropriate string is determined from the language-specific class. There is a useful optimisation here for the native language of the programmer; language-specific strings need only be provided if they cannot be generated automatically from the keyword. For example, the field `not_multicast` will be set to "not multicast" if no specific alternative is given in the language-specific class.

An example text class is given in Figure 47. English and French language version of this is shown in Figure 48.

```
public class Errors extends Language
{
    static
    {
      // supply default language if locale is unknown
      init( Errors.class, "English");
    }

    // strings for localization
    public static String
       not_multicast,
       call_failed_unknown,
       failed_to_contact_server;
}
```

Figure 47 An Example 'Text' Class

```
public class English extends Errors
{
 public final static String
  not_multicast       ="is not a multicast address",
  call_failed_unknown ="call failed remotely (unspecified)";
}

public class French extends Errors
{
 public final static String
  not_multicast=
           "n’est pas une adresse de diffusion restreinte",
 call_failed_unknown=
           "appel distant echoue’ (non specifie’)",
 failed_to_contact_server=
           "echoue’ a contacter le serveur";
}
```

Figure 48 English and French Language Version of Errors

20 REX BINDER (GREEN)

20.1 Introduction

‘Green’ was the first binder written in FlexiNet. FlexiNet binders are generally named after colours, as giving meaningful names that distinguish a binder from future implementations is difficult. For example the candidate name for Green was ‘RexBinder’ – but since then at least four other Rex based binders have been written.

Green supports remote method invocation between client and server. The server may export many interfaces on the same or different objects, and the client has access to these via FlexiNet stubs. A process may be both a client and a server, and there may be multiple simultaneous calls and arbitrary nesting of calls.

The protocol that Green is uses is the “Remote Execution Protocol” (REX) which was first designed for use in ANSAware in 1987. In this incarnation, REX is implemented over UDP messaging and has been simplified from the original protocol by the separation of messaging and fragmentation.

The original version of Green did not support fragmentation, although this has since been added. The exercise of adding fragmentation to an established protocol was a useful validation of FlexiNet’s concurrency control scheme (See section 16.5).

20.1.1 Rex Protocol

The REX protocol is an RPC protocol that was designed with the following key features:

- It is resilient to loss of any message or messages, including protocol messages.
- Under moderate client-server interaction, only two messages are required per invocation; a request and a reply.
- It can support both ‘small and fast’ and ‘large and slow’ invocation
- Server failure is detectable – even if it occurs mid-invocation.

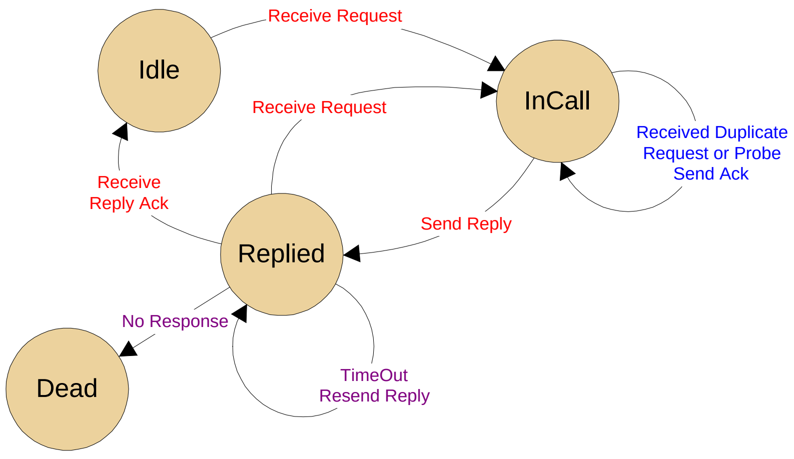


Figure 49 REX Server-Side State Transition

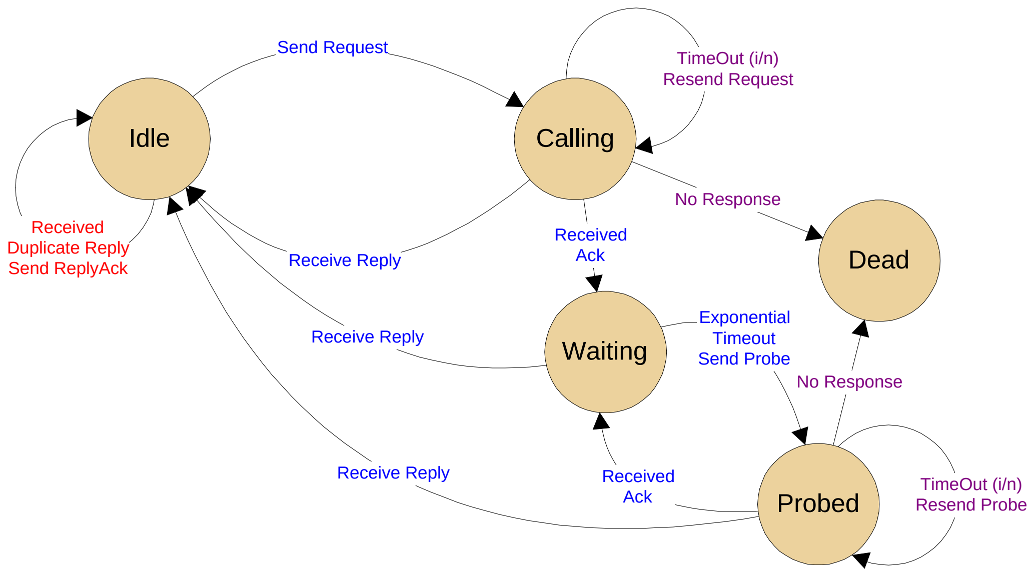


Figure 50 REX Client-Side State Transition

REX server-side state transition is illustrated in Figure 49, and the client side state transition is illustrated in Figure 50. Under normal operation, the client sends are request and then waits for a reply. If the reply is not forthcoming, the client resends the request a number of times (and eventually gives up). If the server receives a duplicate request, it sends an acknowledgement to the client, to inform that it is processing the request. From then on, the client need only periodically 'probe' the server to check that it is still alive. A simple sub-protocol deals with re-sending and acknowledgement of probes.

When the server has completed the request, it sends a 'reply' message. The client does not acknowledge this but under moderate load, the client will perform a subsequent request at the server, and this will be treated as an implicit reply acknowledgement. If no acknowledgement is forthcoming, the server will resend the reply periodically, until it receives an implicit or explicit reply acknowledgement.

The entire state transition is performed on a per-session basis. This allows a client to perform invocations in parallel with each other, and this is simply treated as if the invocations came from different clients. There is scope for the sharing of probes and acknowledgements between sessions, but this optimisation is unnecessary, as these messages are small and rarely sent.

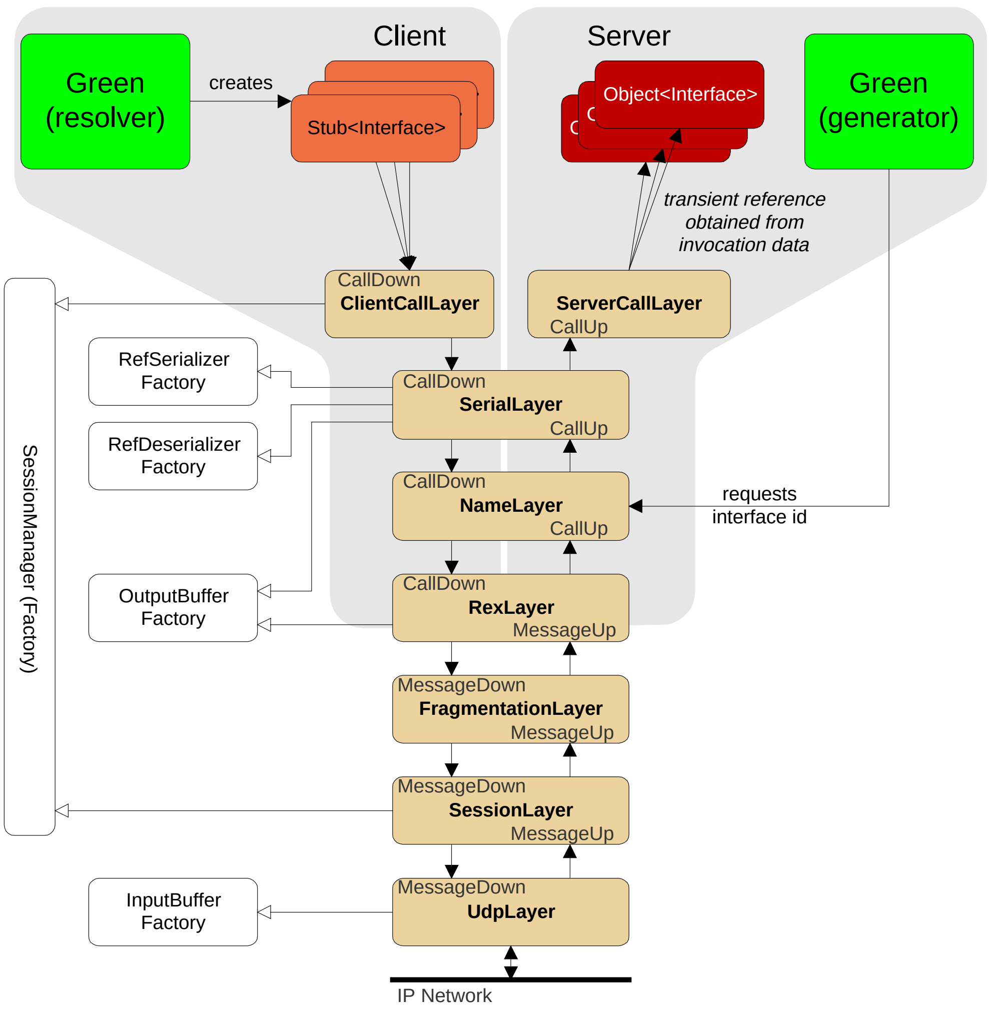


Figure 51 The Green Binder and Protocol Stack

20.2 Basic Operation

Upon initialisation, the REX binder creates a single protocol stack that is used for all invocations – whether client or server side. This is illustrated in Figure 51. In addition to the stack, there are a small number of shared resources – factories for input and output buffers, and a Session Manager.

20.2.1 Name Generation

All exported names are `TrivNames` – pairs of a `UDPEndpoint` and an integer interface ID. In order to generate a name, the binder calls the

`TrivNameLayer` to generate an interface ID, and then pairs this with the shared UDP endpoint (See Figure 52).

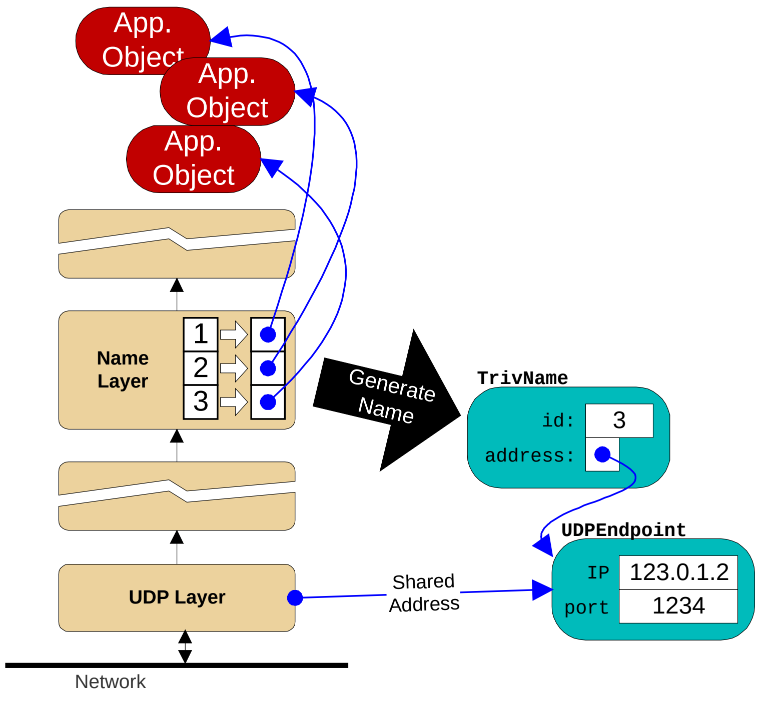


Figure 52 Creating a Name

20.2.2 Name Resolution

To resolve a name, a new Stub is created, and initialised with a reference to the top of stack, and the name being resolved.

20.3 Client Side Call Processing

In this section we walk through the process of a call down the stack. In subsequent sections we consider how a message is received off the wire, and how requests are handled on the server.

20.3.1 Call Layer (ClientCallLayer)

On the client, an invocation is made on a stub, and passed as an Invocation object to the top layer, the `ClientCallLayer`. This obtains a session from the `SessionManager`. Any free session to the correct port on the server host will suffice. The Call Layer is not aware of the details of the protocol being used, but may identify the session endpoint by calling `getSessionEndpointIdentifier` on the target address stored in the Invocation object.

The `SessionManager` manages two types of sessions, “Up sessions” are used for data travelling from the wire up the stack, and “Down Sessions” are used for client calls proceeding down the stack. In this case, a Down Session is required. One is removed from a pool managed by the session manager, and returned to the client thread. If no session is available, a new one is created. On completion of the call, the session will be returned to the pool, and may be reused by the same (or a different) thread wishing to make a subsequent, non-overlapping call to the same service endpoint (port). After receipt from the session manager, the session is locked, and validated for race conditions.

20.3.2 Serial Layer

The serial layer obtains an output buffer from the factory to write the invocation in to. It then serialises the method name, signature, and parameters. When the call competes, it is responsible for deserialising the result (exception or normal). If the result is exceptional, it is *not* the responsibility of this layer to throw the exception. It is simply recorded in the invocation object. In general, a layer should only re-throw exceptions that its peer layer generated.

20.3.3 Name Layer (TrivNameLayer)

The Name Layer examines the address of the interface being invoked. This must be a `TrivName` (as Green only resolves `TrivNames`). It extracts the interface id and writes this to the output buffer. Output buffers are segmented, so the Name Layer does not need to be aware of the size/position of data written by the serial layer or other layers (see Section 28.2). The Name Layer then overwrites the target field in the Invocation with the address portion of the `TrivName`. In this way it need not be aware of the type of protocol above which it is multiplexing, and lower layers need not be aware that multiplexing has taken place. (See section 16.4.1).

20.3.4 RPC Layer (RexLayer)

The RPC Layer is responsible for managing method invocations over asynchronous messages.

To process a client call, the `RexLayer` first writes a header into the output buffer to indicate that the message is (the next) request, and then sends it to the layer below. Once sent, the message is stored in the associated session. The session is then added to a timer queue, so that the message may be resent if no reply if forthcoming. The client thread waits at this point is not woken until a matching reply has been received.

For simplicity we will gloss over the details of the Rex protocol. Let us instead follow the progress of an outgoing message from the `RexLayer` to the wire. This will be the same for requests, replies and protocol messages. For messages, the Invocation class is inappropriate. Instead just the buffer and session are passed.

20.3.5 Fragmentation Layer

Fragmentation is handled at this point. This will be described in section 20.6.

20.3.6 Session Layer

The session layer writes an identifier for this session (and the peer server session if known) into the buffer. As the session is no longer available, the layer below is passed the output buffer and address.

20.3.7 UDP Layer

The `UDPLayer` simply sends the request to the given address, and then returns void. The stack rapidly unwinds back to the RPC Layer.

20.4 Receipt of a message

Messages are received off the wire symmetrically on client or server. It is only at the RPC Layer that it is determined if they represent a request, reply or handshake. In this section, we walk through the progress an incoming message makes.

20.4.1 UDP Layer

Messages are received off the wire in the `UDPLayer`. This contains a pool of threads waiting to listen for incoming messages, but only one actually listening at a time. When this thread receives a message, it stores the bytes into an `InputBuffer` (obtained from the input buffer factory) and passes it up to the next layer.

In its simplest mode of operation, before passing the message up the stack, it unblocks/creates a new listener thread. However, for efficiency, it may be more appropriate to wait until the message has been serviced, and then allow the original thread to continue listening. This will depend on the time take to service the message – which is dependant on the type and context of the message. Rather than burdening the UDP layer with this knowledge. a 'KickThread' interface is provided, which allows a layer further up the stack to inform the UDP layer that the processing of a message will take some time, and that it would be better to start a new listener. This is used in Rex with fragmentation and is discussed in section 20.6.

20.4.2 Session Layer

When an incoming message reaches the `SessionLayer`, it must be associated with the appropriate session. First, the Session Layer reads the session identifier from the input buffer, it then calls the Session Manager and asks for a matching ‘Up Session’. There are actually two sessions identified. The session on this machine, and the session in the peer machine. For all calls other than initial requests, both values are known. For an initial request, the client specifies that the server session is unknown, and the server must allocate a new session. There is a possibility of an initial request being duplicated, so the SessionManager must handle this, and return the same session to both requests. Duplicate suppression of messages is itself performed by the RPC layer.

20.4.3 RPC Layer

When a message is received by the `RexLayer`, there are several possible courses of action, depending on the type of the message. Protocol messages generated by the peer layer may lead to a stored message being resent or a timer modified. Replies that match a stored request will lead to the requesting thread being woken and handed the reply. We will go through the third case, the receipt of a ‘request’ message in more detail.

The state of the session associated with the request is first updated to indicate that a request is in progress. Only one request is allowed at a time for each session. The session is then unlocked is to allow further protocol messages to be received relating to this call, whilst the call is still in progress. In Java, locking is block-structured, so in practice this involves unwinding the call stack back to the session layer, to unlock the session, and then calling directly back to the RexLayer. The `Continuation` interface is used to abstract this as cleanly as possible.

Once the session is unlocked, the call proceeds up the stack. When the call returns, the output buffer stored in the Invocation has a header appended to it, indicating that it represents a reply; and is then sent as an outgoing message. The `RexLayer` also deals with some server side errors by marking replies as normal, error, or extended error. The latter two resulting from the call up the stack returning exceptionally. This will always be due to a system exception, not an application level exception. At this point, passing error information is troublesome, as we are below the serial layer. A simple protocol is defined for allowing Exception classes to provided ‘extended information’, to allow them to be recreated on the client machine.

20.5 Server Side Call Handing (above Rex)

On the server, a call is effectively initiated by the `RexLayer`, and ultimately unwinds back to the `RexLayer`. The `RexLayer` is then responsible for converting the result into a reply message.

20.5.1 Naming Layer (TrivNameLayer)

The Name Layer reads the identifier of the interface being called from the input buffer, and looks this up in the dictionary it maintains. It then sets the invocation target to the corresponding application object. In addition to this, the `TrivNameLayer` also retrieves the stored class of the exported *interface*. This is also stored in the Invocation for later validation that the method being invoked is one that was exported.

20.5.2 Serial Layer

The serial layer reads the method name and signature and converts this to a real method which is stored in the `Invocation` object (At this point conformance with the exported interface is checked). It then reads the arguments and deserialises them to real objects. The call is then passed up the stack. When it returns, the result (exception or normal) is serialised.

20.5.3 Call Layer

This is the top layer of the server-side stack, and by this point the Invocation object is complete. This layer invokes `Invocation.invoke`. The invocation object, in turn, invokes the stored method and arguments on the target object. If the target implements the `GenericCall` interface, then this is used to perform the invocation. Failing this, Java core reflection is used.

20.6 Fragmentation Approach

Fragmentation was added to Green long after the basic protocol was written. Unlike previous Rex implementations, fragmentation was written as a separate layer 'slotted in' below the Rex Layer. This approach was taken based on experience with the initial ANSAware implementation, where the protocols for fragmentation and RPC were intertwined, which lead to considerable additional complexity.

20.6.1 Basic Concepts

Each (large) message is broken in to a number of fragments, and each fragment is given a fragment number to allow the fragments to be reassembled, even if they are reordered, or some are lost. A buffer implementation was designed to be used with fragments, that stores buffers as a linked list of segments, where each segment is the size of a single UDP packet.

On message output, the calling thread sits in a loop and outputs each fragment in turn.

On input, the fragmentation layer assembles fragments in a structure stored in the session. Normally, an up-call will lead to a fragment being added to the structure, and the call then returns. When the last fragment of a message is received, the calling thread is then used to call up to the next layer, with the complete message.

20.6.2 Locking Issues

There are several potential areas for 'feature interaction' between the fragmentation layer and other layers. In particular, there is a potential race condition between fragment assembly and protocol messages. The fragmentation layer may be assembling a message at the same time that the Rex layer above it sends a new outgoing protocol message.

The session locking strategy used in FlexiNet takes care of most of the problems. As the session is locked whenever the Fragmentation Layer is called, that layer may safely process messages, and use the session, without being concerned with thread activity. One remaining problem is if the fragmentation layer has to deal with *very* large messages. With these, the time taken to fragment or assemble may be large enough to interfere with the normal processing of acknowledgements by the Rex protocol. This might lead to Rex unnecessarily re-sending messages, and in an extreme case would lead to protocol failure (when a large message leads to the re-send of another large message, and a downwards spiral). To prevent this problem, the Fragmentation layer briefly releases the session lock during sending. This allows any incoming protocol messages a chance to get through to the Rex Layer.

20.6.3 Fine Tuning

There are a number of ‘fine tunes’ to optimise the use of fragments.

Thread Kick

When processing a large message, the Fragmentation layer will quickly deal with an incoming fragment (unless it is the last fragment). It can then return to the UDP layer and the same thread can be used to listen for the next incoming message. This is far more efficient that immediately spawning a second listener in the UDP layer, as this will typically lead to a thread switch per fragment.

Rex Acknowledgements

Vanilla Rex does not acknowledge requests, instead the client resends the request if the reply does not arrive quickly. This approach leads to fewer messages in the usual case that RPCs are handled quickly. However for slow RPCs, there is an overhead of a resent request, and a acknowledgement. For short requests, this overhead is two messages and the ‘normal case’ optimisation is worthwhile. However for fragmented requests, this approach is suspect, particularly as large requests often equate to slow RPCs. For this reason, the Rex layer has been modified to detect requests corresponding to fragmented messages, and to immediately acknowledge them.

Missed Fragments

In the initial implementation, the Fragmentation Layer would simply drop a message it failed to assemble. This is clearly an expensive approach if message lost is likely. The optimisation is to send a nack indicating the *first* fragment that is lost. The peer machine then resends all fragments from the lost fragment onwards. This approach is taken, as typical network behaviour is to drop no messages, or to drop a burst of messages. Statistically, this approach sends less messages than a 'nack single fragment' approach, and is considerably simpler.

Fragmentation Timeout

If the last fragment of a multi-fragment message is not received, the fragmentation layer will not spot this (it will wait indefinitely). A solution would be to add a timer into the Fragmentation Layer, to spot this special case. This approach was rejected as the overhead of timer management was large compared to the likelihood of this fragment being lost. Instead we rely on the Rex layer to timeout and re-send the entire message. For use in a particularly unreliable network, this decision may require re-evaluating.

20.7 Fragmentation Engineering

The fragmentation layer relies on the fact that `BasicOutputBuffer` stores long messages in a linked list of packets. The binder supplies the `BasicoutputBufferFactory` with a suitable packet size.

The fragmentation layer uses a two-byte fragment segment in each buffer containing the fragment number. These start at one and the last fragment is marked by negating its fragment number. Single fragment messages have a fragment number of minus one.

20.7.1 Outgoing messages

The down method transmits each packet in a multi-packet buffer as a separate fragment. Between each fragment it releases the session lock by waiting on the session for a microsecond, so as to enable the session to process incoming messages.

If the receiving fragmentation layer notices a gap in its sequence of fragment numbers, it will send a NAK fragment. A NAK fragment is marked by setting the two most significant bits of the fragment number to `10` and the least significant 14 bits to the number of the first missing fragment.

when a transmitting fragmentation layer receives a NAK fragment, it rewinds its output buffer to the requested fragment and resumes transmission of the message from that fragment.

20.7.2 Incoming messages

Incoming messages are processed by the lower layers one fragment at a time in single packet buffers. The fragmentation layer assembles the fragments of a multi-fragment message from multiple single-packet input buffers into a single multi-packet input buffer. The partially assembled input buffer is stored in the session object.

If the fragmentation layer notices a gap in its sequence of fragment numbers, it will send a NAK fragment for the first missing fragment. It will discard any duplicate fragments.

While assembling a multi-fragment message, the fragmentation layer returns almost immediately to the UDP layer, which can then listen on the same thread for the next fragment and avoid an unnecessary thread switch. However, when the fragmentation layer receives the last (or only) fragment of a message it needs the current thread for an indeterminate time to deliver the message to the layers above it. Therefore, before delivering the message, it calls the `kickListenerThread` method on the UDP layer to allocate another thread to listen for incoming messages.

20.8 Summary

Green was the first FlexiNet protocol and provides a complete and efficient RPC implementation. However, it is no longer used as the default FlexiNet binder. This is because a TCP based binder can be made more efficient. On a level playing field, Green *ought* to out perform TCP binders such as Magenta. The reason it does not is primarily because:

a) UDP is less well supported, and less heavily optimised, than TCP in many operating systems.

b) REX over Java performs error detection/correction in Java. RRP over TCP performs error detection/correction via the TCP libraries which are native (and probably Kernel) code. This is far faster.

c) Green is more susceptible to poor thread implementations than is Magenta.

21 RRP BINDER (MAGENTA)

21.1 Introduction

RRP (Request Reply Protocol) is a simple but efficient RPC protocol designed specifically for FlexiNet. It is designed to be an ‘ideal’ protocol for Java, and is considerably simpler than the implementation of a standard such as IIOP.

RRP is a TCP based protocol that supports a request-reply abstraction. The protocol itself passes only messages, and pairs requests with replies. It does not understand objects, interfaces or other ‘high level’ concepts. It makes use of the session abstraction, and for each session uses a TCP connection for low level communication. The TCP connection may be timed out or closed independently from the life of the session. The `RRPLayer` provides the same abstraction as the `RexLayer` and all layers below it within the Green protocol.

21.2 Magenta

To a first approximation, Magenta is a Green binder, but with the RPC, session, fragmentation and UDP layers replaced with a single RPC-to-wire layer, `RRPLayer`. A simple substitution of these layers would give a valid binder, however Magenta has a couple of additional features.

In part five, we will introduce the notion of ‘clusters’ within FlexiNet. These lead to names requiring two levels of indirection: A request received off the wire is first de-multiplexed to a cluster, and then layer de-multiplexed to an interface within a cluster.

The Magenta binder is capable of resolving names produced by a RRP binder for use with clusters. It therefore has additional layers to deal with the additional levels of multiplexing. In all, Magenta can deal with three type of name.

1) `TrivName`s consisting of an ID and `TCPEndpoint`
   These names are generated by Magenta binders on ‘standard’ systems.

2) `TrivName`s consisting of an ID and `ClusterAddress`. Where the `ClusterAddress` in turn consists of a `ClusterID` and `TCPEndpoint`. These names are generated by Magenta binders on ‘cluster’ systems.

3) MobileNames consisting of a (reference to a) MobileNamer, ID and ClusterAddress. These names are generated by Magenta binders on 'mobile cluster' systems. (see Chapter 35).

These effects are seen in three areas of the binder. Firstly, there is an additional layer akin to the TrivNameLayer to write the ClusterID part of the name. Secondly, there is a layer to deal with rebinding MobileNames that refer to interfaces that have moved. Finally, the binder functions responsible for generating, resolving, parsing and stringifying names must deal with the additional cases. The Magenta binder is illustrated in Figure 53.

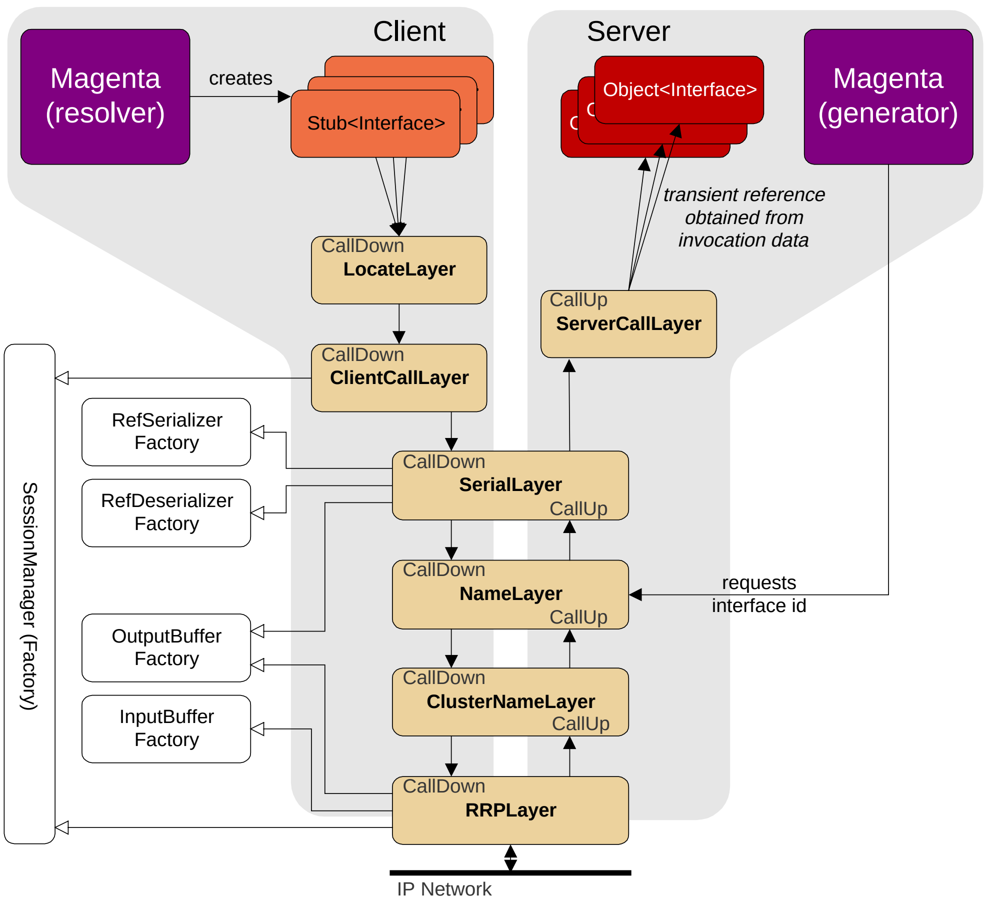


Figure 53 The Magenta Binder Stack

21.3 Cluster Name Layer (SingleClusterMuxLayer)

This layer is responsible for managing the ClusterID associated with the names of interfaces in remote or local clusters. The Magenta binder itself does not normally support local clusters, although it can be configured to support a single local cluster. This is typically used to support a service that can be (manually) restarted on a different machine. In either configuration, Magenta can resolve names generated by Magenta-compatible binders that do support clusters.

When a call is sent down to this layer of the Client, if the `Invocation` target is a cluster address (class `GNameFragment`), then the cluster ID is extracted and written into the output buffer. The `Invocation` target is then amended to contain the capsule address portion of the cluster address. For calls to non-cluster addresses, this layer does nothing and garbage will be passed as the cluster id.

On the server, incoming calls reaching this layer are normally passed up unmolested. If Magenta has been configured to support a single cluster, then the identity of the cluster being called is read from the input buffer, and checked against the identity of the local cluster, which is stored in this layer. An `UnKnownNameException` is thrown if these do not match.

21.4 Locate Layer

The `LocateLayer` is a client side only layer that deals only with `MobileNames`. For calls on names of other classes, it passes calls through with no effect. For calls to `MobileNames`, the Locate Layer uses the `getAddress` method on `MobileName` to get the current address of the interface. If a call using this fails because the implementing cluster has moved, then the thread loops, calls `getAddress` again, and retries. With each call to `getAddress`, the `effort` parameter is increased, to indicate that the previous result was outdated, and the `MobileName` should 'try harder' – for example, it might ignore a local cache and communicate with an authoritative name service. The Locate Layer gives up if `MobileName.getAddress` throws an exception, or returns the same address as the previous attempt, suggesting that continuation would be futile.

This implementation is imperfect – a `MobileName` referring to an interface on a cluster moving back and forth between two locations might correctly return an address that is invalid by the time it is used. It may then become valid again, by the time that the next call to `getAddress` is made. This would lead to a failure, although the interface remains contactable. It would be possible to produce more optimistic version of Locate Layer that tried for longer before failing, although for a 'perfect' solution, some form of synchronisation would be required.

21.5 RRPLayer

The `RRPLayer` implements the RRP protocol. We will consider its function as a client and server separately.

21.5.1 As a Client

When a client request arrives at the `RRPLayer`, it will already be associated with an `RRPSession`. From this, the layer may determine the TCP connection that the session uses (There is a one-to-one session/connection mapping). For the first call, the connection will be null, and a new one will be created by binding to the server address.

When the connection is first established. A handshake is undertaken, whereby the client sends an identifier for the client session and an identifier for the server session (or `null` if not known). The server responds with the identity of the server session, which the client stores in the session for subsequent use if the connection is later broken.

The client writes the request buffer to the socket (along with its length) and then blocks attempting to read the return status. When reading and writing to a TCP socket, care must be taken that unnecessary flushes are not caused. For this reason, the length of the buffer is added to the beginning of it, and then the whole written as a block. Similarly, when reading, as few individual reads as necessary are made.

When the reply returns, the client thread unblocks, and reads a length and status byte. It then reads the reply which will be the return value or an RRP error status, which is converted to a `BadCallException`. Before returning up the client stack, a timer is set to timeout the TCP connection.

If the TCP connection timer fires, the connection is torn down, and a further timer set for the session destruction. Note that neither timer can fire whilst a call is being processed – because the session itself is locked. The timer implementation ensures this by delaying the signalling of a timeout until it can acquire the session lock (See section 28.4)

21.5.2 As a Server

As a server, the `RRPLayer` has a number of listener threads that are blocked waiting for new connections. Only one thread actually listens. When a new connection arrives, a second thread is unblocked/created to handle the next connection, and the original thread calls `processClient`. When this ultimately returns, the thread is either returned to the listener pool, or destroyed – depending on the concurrency parameters set when the layer was initialised.

To process a client, the server first reads the client's session identity from the wire, and uses this in a call to the Session Manager to obtain the appropriate server session. It immediately sends a handshake to the client indicating the matching server session id. This exchange only takes place when a new connection is made.

The server then locks the session semi-permanently, as it will be the only thread processing requests on using this session. The session will eventually be unlocked when the connection is broken. The server thread sits in a loop reading client requests, sending them up the stack, and then writing replies. Care is again taken with minimising the number of flushes sent by the TCP system. Before the first read, the server sets the socket read timeout to be equal to the desired connection timeout. If this fires, or the connection is closed by the client, the server sets a timer for the session destruction, unlocks the session, and the processClient method returns.

21.6 Discussion

The RRP protocol has been designed and implemented with certain threading characteristics. On a server, one thread is required for each client with an open connection. This decision is partly due to the nature of Java threads – they are generally cheap, and there is no socket level ‘select’ command. It is also partly due to the desire to minimise thread switching. In RRP there need by no thread switch during an RPC on either client or server. This is particularly important as in current Java implementations, thread switching is very slow (~0.25ms on a Pentium Pro 200 running NT).

An important feature of RRP is that only one request is ever in progress on a particular connection. This removes the need for ‘listener’ threads found in IIOP and other protocols. However it does mean that a pathological client/server pair that nests many RPC between each other will lead to the creation of a large number of connections. This is considered to be a rare case, and will lead to inefficiency rather than failure (assuming there is no bound on the number of connections placed by the operating system).

A similar approach was independently developed in the OmiOrb system, from Olivetti Research Labs [ORL98], for use in their IIOP protocol. We rejected this approach for IIOP as we feel it is not in the spirit (though arguably within the law) of the IIOP standard.

22 IIOP BINDER

22.1 Introduction

IIOP is an ‘interoperability protocol’ supported by all CORBA ORBs. FlexiNet supports IIOP to allow FlexiNet clients to access CORBA services, and vice versa. As IIOP is a standard protocol, we may not modify it to better fit FlexiNet (or Java) concepts. It therefore poses the challenge of supporting a ‘foreign’ protocol. We do not attempt to make FlexiNet a CORBA ORB, rather we wish applications written using FlexiNet to interoperate with CORBA using FlexiNet coding style and concepts as much as possible. We must therefore translate between FlexiNet concepts such as *Interface, Name, Object* and CORBA concepts such as *Object, IOR* and *Data type*.

A secondary goal of the IIOP binder was to determine how well FlexiNet met its *flexibility* goal with respect to third party protocols that are based on ‘foreign’ assumptions and concepts.

The standards used for the IIOP binder work were the relevant OMG specifications that were current at the time of development: [OMG97b], [OMG98a], [OMG98b], [OMG98c]. In order to support the use of IIOP as a alternative protocol to REX or RRP, we need to support the serialisation of *all* Java types. To do this, we made use of the ‘Objects by Value’ and ‘Java to IDL’ mapping specifications, which had not reached their final forms at the time of implementation.

Some familiarity with IIOP and the CORBA standards is assumed of the readers of this chapter.

22.2 Basic Operation

Upon initialisation, the IIOP binder creates a single protocol stack that is used for all invocations – whether client or server side. This is illustrated in Figure 54. In addition to the stack and standard shared resources, there is also an `IDLMapper`, which is responsible for the mapping between Java and CDR types. This is described in section 22.6.1.

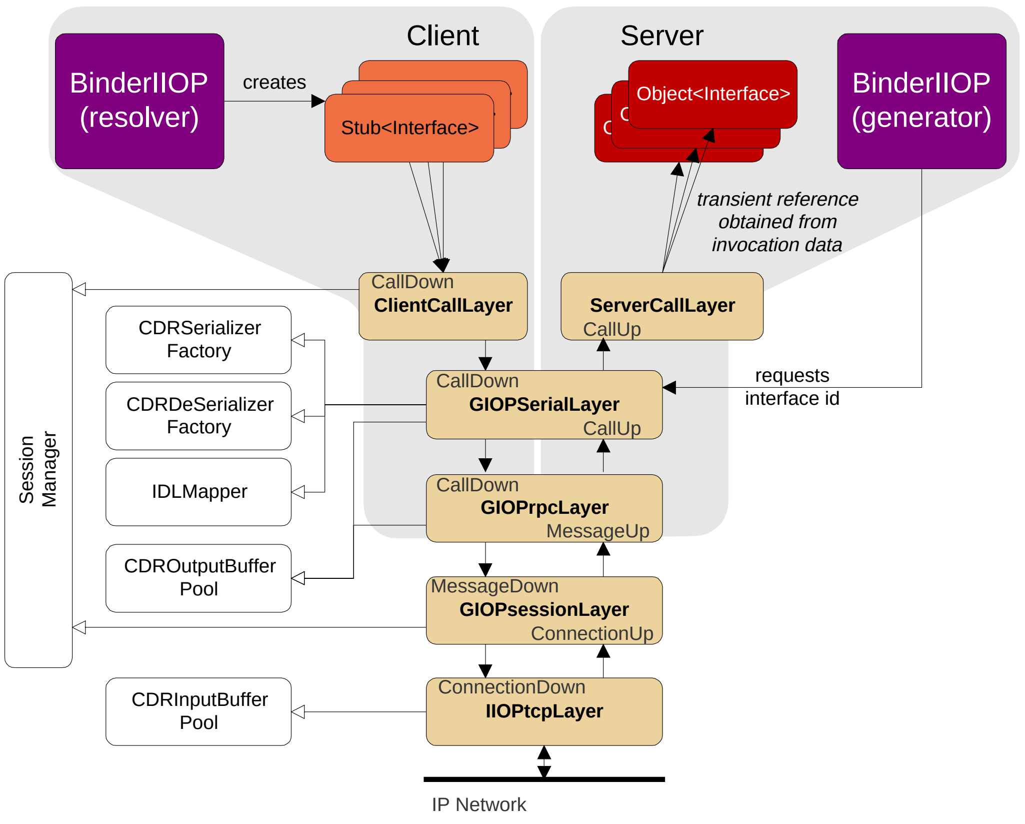


Figure 54 The IIOP Binder Stack

IIOP is actually a generic interoperability protocol (GIOP) running over a TCP/IP transport. The majority of the stack is therefore GIOP specific, with only the transport layer being IIOP specific. `BinderIIOP` itself is a concrete subclass of `BinderGIOP`, and is only responsible for transport specific features, the other layers being initialised by `BinderGIOP`.

22.2.1 Name Generation

Names in GIOP are represented by standard objects called IORs (Interoperability references), whose structure is defined by the CORBA standard. They contain details about the referenced object and addressing information for connecting to the object via one or more possible protocols. The information is held in an encapsulated form, i.e. it has been converted to its wire format and stored in a byte array. Furthermore, an IOR can contain multiple addresses for a given object.

In order to treat IORs as FlexiNet names, they are encapsulated into `GIOPnames`, which implement the standard FlexiNet name interface. `IIOPname` is a subclass of this used for IORs using the IIOP protocol.

When a name is passed in a ‘standard’ FlexiNet protocol, the name may be an object representing the union of a set of alternative names (typically alternatives using different protocols). The same facility is provided in IORs using the support provided for arbitrary extensions through Tagged Components [OMG98c]. This may be used to allow FlexiNet processes to pass IORs containing both IIOP and (non-IIOP) FlexiNet names. When an IOR is deserialised by FlexiNet, non-IIOP FlexiNet names are resolved in preference to IIOP names. To engineer this, `BinderIIOP` may be initialised with a reference to a secondary generator which is used to generate FlexiNet specific names. This is typically a reference to `BinderTop`. `BinderIIOP` detects and handles the special case that it is called recursively via `BinderTop`.

Thus, a standard IOR can be exported from a FlexiNet system to an arbitrary third party. If that IOR is ever imported back into a FlexiNet system, the FlexiNet-specific name can be extracted and the preferred internal protocol can be used between the two FlexiNet systems instead of IIOP.

22.2.2 Name Resolution

`BinderIIOP` can resolve both `IIOPnames` (generated as described above), and raw IORs (assuming that the IOR contains an IIOP address). Support for raw IORs was added to circumvent wrapping and unwrapping during deserialisation. In retrospect, this ugliness was probably unnecessary.

When resolving an IOR or `IIOPname`, `BinderIIOP` first checks whether it contains an embedded FlexiNet name (as described above). If it does, that name is extracted and passed to the secondary resolver to be resolved in whatever manner it chooses. If this is not the case, a new Stub is created and initialised with a reference to the top of the IIOP stack, and the name being resolved.

22.3 IIOP Connection Model

Before describing the function of the IIOP binder stack in detail, it is worth pausing to consider the computational model of IIOP with respect to messages and connections. Unfortunately, IIOP is a little confused in this respect, which makes for some ‘interesting’ engineering.

There are two basic approaches to RPC handling. Message-based and Stream-based. In a message-based protocol, such as REX or RRP, the invocation is marshalled into a buffer, which is then sent to the server. This reads the buffer *and then* processes it. With a stream-based approach, the client opens a connection and then starts to marshal the invocation into it. This may be sent to the server is parallel with the marshalling. This removes the need for the client to hold the entire marshalled message in memory at once. On the server, the message may be unmarshalled in parallel with reading it from the client. This removes the need for the server to store the entire marshalled message, and allows for improved performance if the four activities (client marshal, client write, server read, and server unmarshall) can take place in parallel. In either scheme (message or stream based) the ‘high level’ approach may be mapped to a low level message or connection based transport.

It is an issue of some contention as to whether the stream-based approach warrants the extra complexity. The memory use advantage only applies to very large invocations, and any performance gain may disappear if many different invocations take place simultaneously (as they provide alternative scope for parallelization).

In many respects IIOP is stream-based; however the standard insists that the client indicate the total size of a request, prior to sending it. This prevents a client from sending a part-marshalled request, and the advantages of a stream-based approach on the client are lost. On the server, a stream-based approach would still be appropriate. Unfortunately, in a later revision of IIOP, the notion of 'fragmented messages' were added, to alleviate the problem of the client having to assemble an entire request prior to sending it. Whilst this is helpful on the client; it complicates a server-side stream-based implementation, because it must use the 'fragment length' field to aid reassembly. This 'strongly encourages' the server to use a message-based approach. (A better approach, in the authors opinion, would have been to remove the length field completely – and possibly replace it with a transport-specific framing layer).

IIOP, as it is today, is therefore effectively a message-based protocol, as it has the standard hallmarks of sized messages, fragmentation and reassembly. However, as a message based protocol it has some failings. The length itself is buried within the marshalled data, and is not easy to recover. This makes it difficult to split an IIOP protocol stack into standard read-assemble-process layers. In the FlexiNet implementation of the IIOP protocol stack, we took a message-based approach, and accept some 'knitting' of the layers as required to identify the length in a message as it is read from the wire. This was relatively straightforward as the other FlexiNet binders are all message-based, and it is in keeping with the trend of IIOP development, which appears to favour this approach.

22.4 Client Side Call Processing

In this section, we walk through the process of a call down the stack. In subsequent sections, we consider how a message is received off the wire, and how requests are handled on the server.

22.4.1 Call Layer (ClientCallLayer)

The IIOP protocol uses the same `ClientCallLayer` as the Green protocol, and it behaves identically. See section 20.3.1.

22.4.2 Serial Layer (GIOPserialLayer)

This layer subsumes the function of both naming and serialisation (as the two are intertwined in IIOP). It obtains an output buffer and then serialises the parameters of a GIOP Request message, namely:

- service context data (a standard block of information providing encoding format details)
- message id (which doubles as a session id)
- a flag indicating if the invocation is one way
- the target object id (extracted from the name of the destination object)
- the method name
- a callee principal string (always a default value in our implementation)
- method parameters
- Any context values provided by the IDL Mapper (none are provided by the current mappers).

When the invocation completes, if it was not a one way call, this layer is responsible for deserialising the result (exception or normal).

22.4.3 RPC Layer (GIOPrpcLayer)

This layer deals with all the messages involved in the GIOP protocol. For invocations, this means formatting the header of the buffer received from the serial layer to make it a GIOP Request message and then passing it down to the session layer. If a reply is expected to the request, the calling thread will then wait for notification that the response has arrived (or that the connection has been closed).

22.4.4 Session Layer (GIOPsessionLayer)

The session layer ensures that a connection exists over which to send the message, caches the identifier of the connection and then passes the message to the TCP layer to send to the remote host. It then returns to the RPC layer, which will wait if a response is expected.

22.4.5 TCP Layer (IIOPtcpLayer)

This simply sends the request on the given connection, and then returns. The stack rapidly unwinds back to the RPC layer.

22.5 Receipt of a message

Messages are received off the wire symmetrically on client or server. It is only at the RPC layer that their type is examined and appropriate action taken. In this section we walk through the progress an incoming message makes.

22.5.1 TCP Layer

Messages are received off the wire in the `IIOPtcpLayer`. The layer contains a pool of threads waiting to listen for incoming messages, but only one actually listening at a time. When this thread receives a message, it stores the bytes into an Input Buffer and passes it up to the next layer. Before passing the message up the stack, it unblocks or creates a new listener thread.

`TcpLayer` is a general purpose transport layer for TCP based protocols which multiplex many messages over a single connection. In such protocols, it must be possible to read one message, and then simultaneously process this, and read the next message. `IIOPtcpLayer` is a thin veneer over the standard `TCPLayer`, to deal with the problems of extracting the length from IIOP messages.

22.5.2 GIOPsessionLayer

When an incoming message reaches the `GIOPsessionLayer`, it is associated with the appropriate session. The GIOP protocol does not incorporate the concept of sessions, but we simulate them as they are useful abstraction when designing and refining a binder. To implement sessions in GIOP, we use the message id field within request-response message pairs. So, if a response message is received, its id is used to find the session associated with the previously sent request message. In other cases, either the last session associated with the source endpoint is used, or if there is none, a new one is allocated. Having found an appropriate session, the message is then passed to the RPC layer for interpretation.

22.5.3 GIOPrpcLayer

The GIOP protocol contains a number of messages in addition to the pair used to encode an RPC request and response. The RPC layer is responsible for handling all of these, which include such things as checks for object existence, connection termination, and fragmentation of large messages. If the message is a response to a previous RPC request, then control is handed over to the waiting request thread via a thread rendezvous, and processing continues as described above. In the case of a protocol error or closed connection, any waiting thread is woken up and informed of the problem. Requests are passed straight up to the RPC layer with the active thread retaining control. On return, any result buffer is formatted as a Response message and returned to the originator.

22.5.4 GIOPserialLayer

A request is dealt with by deserialising the received buffer contents and constructing an Invocation object, which is handed to the Call layer. The buffer is returned to the pool at this point.

On return from the Call layer, if the request was not a one-way request, the results of the invocation are serialised into a newly allocated buffer and sent down to the RPC layer.

22.5.5 Call Layer

This is the top layer of the server-side stack and works identically under IIOP as for other protocols such as Green or Magenta.

22.6 Implementation Issues

IIOP is not a native FlexiNet, nor even Java, protocol. In addition it is ‘committee designed’ which has lead to a number of inconsistencies and difficulties.

22.6.1 IDL/Java mapping

A standard FlexiNet serialiser is responsible from converting an arbitrary object graph into a sequence of primitive Java types, which are then written to an output buffer which handles the mapping of primitive types onto bytes.

With IIOP, primitive types are CDR types. These have a different format and range from their Java equivalents, and in particular the mapping is not one-to-one. For example, a Java String is used to represent both narrow and wide CDR strings. To complicate matters, the mapping from Java *classes* to CDR types is non-trivial, and there may be many different mappings. These are described in the relevant specifications for IDL to Java mapping [OMG97b], Java to IDL mapping [OMG98b], and objects by value [OMG98a].

Standard implementations of Java-based ORBs solve this problem by compiling IDL into stubs that contain hard coded routines to correctly encode and decode the values. FlexiNet however was designed to do without pre-compiled stubs by making use of the reflective capabilities of Java to obtain the necessary information about data types. We did not want to sacrifice the flexibility and ease of use that this approach offers.

The problem was solved by defining IDL mapping objects which the IIOP stack consults to determine the correct mapping between Java and CDR data types. A default mapper is associated with the stack on creation, but can be overridden during name resolution on a per name basis. The mapper objects implement the `IDLmapper` interface, behind which they can provide any policy they choose for mapping between types.

Two implementations of mappers are provided with the FlexiNet distribution:

`IDLmapperBasic`
: This implements a basic mapping that conforms to the IDL to Java mapping [OMG97b] and is sufficient for encoding all standard CORBA data structures

`IDLmapperObjByValue`

This implements a mapping which conforms to the Java to IDL [OMG98b] and Objects by Value [OMG98a] specifications. This allows all Java objects to be serialised, and allows the IIOP Binder to be used as a standard FlexiNet binder.

These two implementations use 'educated guesswork' to determine which mapping is intended. Though this works well, clearly it will fail if the same Java type is mapped to two different CDR types within a single interface, e.g. if an interface contains both 'wide' and 'narrow' strings. Ultimately, guaranteed correct encoding and decoding of method calls on an interface can only be done through reference to the IDL definition of the interface. Although not provided with FlexiNet at present, an IDL mapper class could be developed that read information from an interface repository to provide fully correct encoding and decoding.

22.6.2 Method types

A related issue is that of method information that is only available from the IDL definitions, e.g. whether or not the method is one-way, and whether parameters are `in`, `out` or `inout`. As with type mapping, the IDL mapper class provides the answers to these choices, and the mappers provided with the distribution use educated guesswork to avoid reliance on IDL. They assume that there are no one-way methods. `Out` and `inout` parameters can be distinguished from `in` parameters as they are represented by special Holder classes. If the holder object's value is null on the first call to the method, it is assumed to be an `out` parameter else it is taken to be an `inout`. An IDL mapper object that made use of IDL compiler generated information could be written to provide guaranteed correct answers.

22.6.3 Serialisation

There are three stages to serialising a Java invocation as a GIOP message; each of these stages is handled by a distinct class.

- On serialisation, the `IDLmapper` determines the equivalent IDL signature for a Java method.
- The parameters of that method are then serialised by an instance of the `CDRSerializer` class; this decomposes complex IDL types into the fundamental types supported by the CORBA Data Representation (CDR) and writes instances of those types into a `CDROutputBuffer`.
- The `CDROutputBuffer` handles the mapping of CDR types into a byte representation within the buffer, including recording a byte ordering flag.

Conversely, on deserialisation, the `CDRInputBuffer` receives the raw bytes from the wire and handles byte ordering transparently to interpret them as CDR data types. A `CDRDeSerializer` then constructs these types into instances of compound IDL types and then these are mapped to Java types by an `IDLmapper` again. This is illustrated in Figure 55.

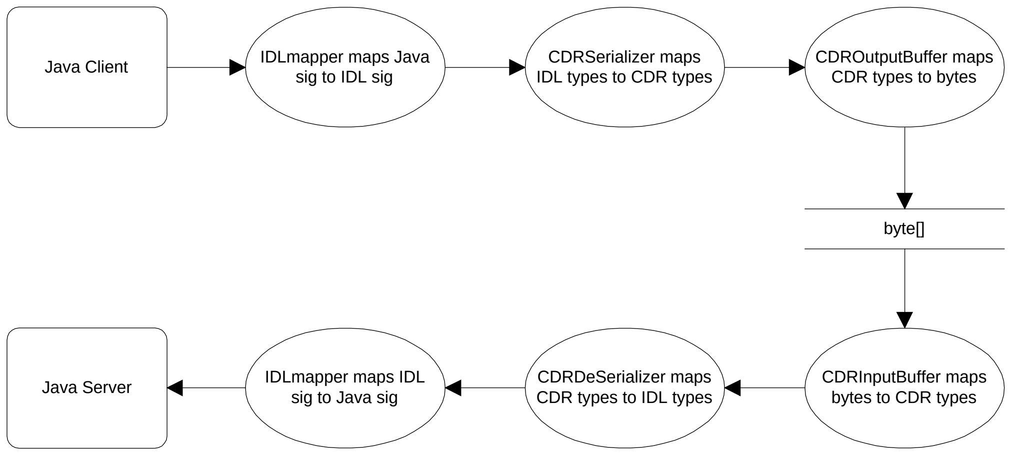

Java Client
IDLmapper maps Java sig to IDL sig
CDRSerializer maps IDL types to CDR types
CDROutputBuffer maps CDR types to bytes
byte[]
CDRInputBuffer maps bytes to CDR types
CDRDeSerializer maps CDR types to IDL types
IDLmapper maps IDL sig to Java sig
Java Server

Figure 55 Serialisation and Deserialisation of GIOP Messages

22.6.4 Pass by value

The FlexiNet IIOP implementation (optionally) supports 'Objects by Value'. This is still not an official part of the CORBA specification. The extensions covered in the Objects by Value RFP [OMG98a] were used to implement this feature. These extensions involve modifications to certain standard OMG classes such as `TCKind` and `InputStream`, which in turn means that we had to modify the CORBA Java classes that are included in JDK1.2. These modified classes are shipped as part of the FlexiNet distribution, and must be used instead of the standard JDK1.2 CORBA classes.

22.6.5 Sessions

Sessions in FlexiNet are used to hold state information for at least the lifetime of a request-reply pair and possibly longer. Other FlexiNet protocols rely on a session identifier being sent in a request which is then returned in the reply and used to associate the correct session with that reply. The format of the GIOP message however is dictated to us by the standard, and it does not include a session identifier field.

A solution of sorts was engineered by using the message id field that is present in all of the request-reply pair messages in GIOP to double as a session id. As the value sent in the request is guaranteed to be returned by the server in the associated reply, we can happily use this value to select the correct session on the client side when a response is received. Server sessions though cannot be maintained over multiple calls as there are no guarantees about the message id sent by clients, which results in less efficient usage of sessions on the server side than in other FlexiNet protocols.

22.6.6 Message format

The protocols designed by the FlexiNet team have fixed size segments within the message formats used, which allows layers in the protocol stack to encode and decode their required information in isolation from other layers in the stack. This in turn facilitates he assembly of arbitrary sets of layers to form protocol stacks to meet specific requirements without recourse to modifying the layers themselves. IIOP was not designed with this approach in mind, and consequently the IIOP message format contains many instances of variable length data. Furthermore, the total length of the IIOP message is not the first value encoded in that message (as it is for FlexiNet protocols).

This message format has two consequences: firstly, the TCP layer offers a stream interface upwards, delegating the responsibility to work out how many bytes to read to higher layers. This is to support true 'stream-based' protocols, as well as to aid the modularity of the IIOP implementation. Secondly, there is some 'blurring' of the layers, although this has been kept to a minimum. In particular, the session layer has to understand a little of the format of the part of the message handled by the RPC layer in order to extract the message id that it uses as a session handle; and the input buffer class has to understand the message header format in order to read the length of the message.

22.6.7 Message Fragmentation

Version 1.1 of the GIOP protocol supports the fragmentation of large messages into smaller pieces. The FlexiNet implementation of the GIOP protocol does not make use of this feature when sending messages itself, however the GIOPmessageLayer understands fragmentation and correctly reassembles fragmented messages before further interpretation.

22.6.8 Threading issues

The GIOP standard states that traffic to more than one object can pass over the same connection between two hosts. Furthermore, if there is more than one request outstanding on a connection, no guarantee is given that the responses will be in the same order as the requests.

These features of the protocol have two effects on the threading model used by the IIOP implementation. First, on the client side, the thread handling a client request does not wait for the response (because the first response on that connection might not necessarily be for it). Instead it waits in the message layer, and a dedicated pool of receiver threads handles the connection. Once the correct thread to handle the response is identified, control is handed over via a rendezvous. Secondly, as a request can be received on a connection while another is still outstanding, the IIOP stack always has a thread listening on the connection. When a message is received, a new listener thread is allocated from a pool while the previous listener handles the message.

This approach is relatively costly, but necessary to stay within the bounds of the IIOP standard. A better approach (to the design of IIOP) would have been to allow the client to dictate the terms of use of each connection it established. This would allow clients with sufficient resources to choose a much simpler, and more efficient, model (for example that used in RRP) whilst allowing the full flexibility for clients with limited socket resources.

23 SSL BINDER (CRIMSON)

23.1 Introduction

SSL (Secure Socket Layer) is a security protocol, originated by Netscape Communications Corporation, which provides authentication, encryption, and integrity checks. The Crimson binder uses iSaSiLk, an implementation of SSL version 3 [IAIK]. As SSL works with TCP connections, the Crimson binder is a direct descendant of the Magenta binder. In fact the default behaviour of the Crimson binder is not to use SSL at all, and behave in every way as if it were a Magenta binder.

23.2 Crimson

The primary distinguishing feature of Crimson is that it uses a `ConfigurableSocketFactory`, which is capable of creating SSL sockets and plain sockets, whereas Magenta used the standard `SocketFactory` which produces plain sockets only. As SSL is highly configurable, the configuration data is passed to the socket factory as an extra `FlexiProps` argument in the socket creation methods `getSocket` and `getServerSocket`. The Crimson binder constructor also takes a `FlexiProps` configuration argument, and if this contains an “ssl” property, then this will be used to configure sockets when they are created.

If Crimson is constructed in this way, the configuration data is passed to the components that use the socket factory to create sockets, namely the session factory (client side) and the RPC layer (server side). If the Crimson configuration has no “ssl” property, then Crimson acts as a Magenta binder.

23.3 Security Algorithm Selection

The first thing that happens when a client SSL socket attempts to connect to a server SSL socket is algorithm negotiation. The client sends lists of algorithm names identifying possible configurations for a connection.

The first list specifies candidate compression methods. In iSaSiLk the only option is “no compression”.

The second list specifies security algorithms. The client *requires* that the server choose one item from the list. It *prefers* the server to choose an item at the front of the list. If the server is unable or unwilling to connect the client using an algorithm from the list, then it will close the connection, and an exception will be raised on the client.

Each item on the list is a security algorithm triple. This identifies three things

- a key exchange algorithm
- an encryption algorithm (or "no encryption")
- a message authentication code (signature) algorithm

The primary function of SSL configuration is specifying which triple are suitable for a given connection, and providing sufficient additional information for the selected algorithms (for example Keys, Certificates etc.)

23.4 SSL Configuration

There are three categories of property used to configure the Crimson binder. These relate to *algorithm selection, client/server identification* and *trust*. They will be described in the following three sections. In general, some properties in each category must be specified for each connection, although a particular binder instance usually treats all connections in a uniform way.

23.4.1 Algorithm Selection

The client and server must both enumerate which algorithms they are willing to use, and give an order of preference. Two sets of properties are used for this, one for a server's security configuration, and one for client's. The structure of the two sets is the same.

- `client.algorithm_sets`
  This gives a simple enumeration of acceptable algorithm triples. The properties should be a list of algorithm triples, specified as a space concatenated list of strings or an array of strings. Each string should be of the form K+C+M, where K is the key exchange algorithm, C the encryption algorithm, and M the message authentication algorithm.

  Every available algorithm triple listed in this property is included as an acceptable algorithm triple. A client will be willing to use any of these triples to talk to the server.
- `server.algorithm_sets`
  An analogous property for servers. A server will be willing to allow a client to connect using any of these triples.
- `client.key_exchange_algorithms`
  `server.key_exchange_algorithms`

- `client.cipher_algorithms`
  `server.cipher_algorithms`
- `client.mac_algorithms`
  `server.mac_algorithms`
  These properties provide an alternative means of specifying acceptable algorithm triples. They may be used instead of, or as well as, the `algorithm_sets` properties. Each of these properties should be a list of algorithm names, specified as a space concatenated list of strings or an array of strings. Names that do not correspond to algorithms of the appropriate type are ignored.

  Every available algorithm triple formed by choosing one algorithm from each set is considered to be an acceptable algorithm triple.
- `client.algorithm_sets.order`
  `server.algorithm_sets.order`
- `client.key_exchange_algorithms.order`
  `server.key_exchange_algorithms.order`
- `client.cipher_algorithms.order`
  `server.cipher_algorithms.order`
- `client.mac_algorithms.order`
  `server.mac_algorithms.order`
  These properties are used to specified the order of preference of the algorithm triples defined using the preceding properties. Each property should be a list of algorithm names, specified as a space concatenated list of strings or an array of strings.

  If two triples are given a conflicting ordering using these properties, then the algorithm sets property takes precedence, followed by key exchange, cipher, and message authentication code algorithms, in that order.

23.5 Identification

Once the server socket has successfully chosen a mutually available cipher suite, it sends an SSL certificate to the client socket. An SSL certificate comprises a sequence of X509 certificates. The first of these contains the server's public key, the servers name, plus some additional information, and a signature generated using a certificate authority's private key. The next certificate in the chain should be the certificate for the same certificate authority, and contain the authority's public key, which can be used to validate the signature in the server's certificate. Similarly this first authority certificate is signed. The signature on an authority's certificate may have been generated by yet another authority's private key, in which case the corresponding certificate should be the next in the chain. The alternative, for the last in the chain, is for the certificate to be self signed, so that the signature can be checked using the public key from the same certificate.

The properties used to supply certificates to the server and client have identical structure.

- `certificate_directory`
  Root directory for files used for certificates and temporary RSA key pairs. Properties which contain file names for these types of stored object can be given relative to this directory, or be given as absolute paths.

- `client.key_and_certificate.RSA,`
  `server.key_and_certificate.RSA,`
  `client.key_and_certificate.DSA,`
  `server.key_and_certificate.DSA,`
  `client.key_and_certificate.DH,`
  `server.key_and_certificate.DH`

  When supplied, these properties can be either key and certificate pair objects, or the names of files where key and certificate pairs are stored.

- `client.password.RSA,`
  `server.password.RSA,`
  `client.password.DSA,`
  `server.password.DSA,`
  `client.password.DH,`
  `server.password.DH`
  The passwords to use to decrypt the private keys of key and certificate pairs read from file.

23.5.1 Trust

These properties may be used to define which clients may connect to a service interface, or to restrict the identity of the service to which a client is connecting. As SSL certificates are issued by certification authorities then the acceptable/required certification authorities must also be specified.

When the SSL certificate is received by the client, then the configuration set up by the Configurable Socket Factory will always check that the chain of X509 certificates is signed as previously described. In addition, it can also be configured to check that the chain contains a certificate for a known certificate authority. This option is controlled by the properties:

- `client.require_trusted_root`
  A boolean property.

- `client.authority_certificates`
  Either a string containing file names concatenated with separating spaces, or an array of X509 certificate objects.

  In addition, a server has the option of requiring an SSL certificate from the client. This is set up by using the configuration properties:

- `server.require_certificate,`
  `server.require_trusted_root`
  Boolean properties.
- `server.authority_certificates`
  Either a string containing file names concatenated with separating spaces, or an array of X509 certificate objects.

All of the configuration and initialisation of the SSL connection happens when server sockets are created and when client sockets are connected. There is no interaction with the client or server applications.

23.6 Supported Algorithms

The algorithms currently supported by iSaSiLk are:

- Key exchange
  - RSA
  - RSA_EXPORT
  - DH_DSS (Diffie Hellman with public key from a certificate signed using a DSA authority certificate),
  - DH_DSS_EXPORT
  - DHE_DSS (Diffie Hellman with temporary keys, DSA signed, using a DSA user certificate)
  - DHE_DSS_EXPORT
  - DH_RSA (Diffie Hellman with public key from a certificate signed using a RSA authority certificate)
  - DH_RSA_EXPORT
  - DH_anon (Diffie Hellman with temporary keys, unsigned)
  - DH_anon_EXPORT

  The export versions restrict the use of DH, and RSA keys to fewer that 512 bits when used for encryption.

  If an ephemeral or anonymous Diffie Hellman algorithm is selected, then temporary Diffie Hellman parameters are required by the server. If any of these algorithms are available after having processed the cipher suite properties for a server socket, then the Configurable Socket Factory will automatically generate the parameters.

  If RSA_EXPORT is selected, or RSA with NULL encryption, and the server certificate contains a public key longer than 512 bits, then the server requires a temporary RSA key pair that satisfies the length restriction. This has to be set using the properties (server only):

  - `server.temp_key`
    This should either be the name of a file containing an RSA key pair, or an explicit RSA key pair.

- `server.temp_password`
  This is only required if `server.temp_key` contains a file name, in which case it should be the password to use to decrypt the private key in that file.

- Encryption
  - NULL (no encryption)
  - RC4 (40 bit key)
  - RC4/CBC (with cipher block chaining, still 40 bit key)
  - IDEA/CBC, DES/CBC (40 bit key)
  - 3DES/CBC (Triple DES with CBC, two 40 bit keys).
- Message authentication
  - MD5
  - SHA

Not all of the combinations are included is SSL version 3, for example the only combination which includes IDEA is with RSA and SHA. See the iSaSiLk documentation for details.

23.7 Generating Certificates

The Java application `CertificateGUI` has been written to provide a reasonably convenient way of producing both authority and user certificates for use with the Crimson binder. The following sections describe how to use the interface to generate the certificate files for a SSL certificate comprising a chain of three X509 certificates.

The following sections detail the various windows that form the user interface of the program.

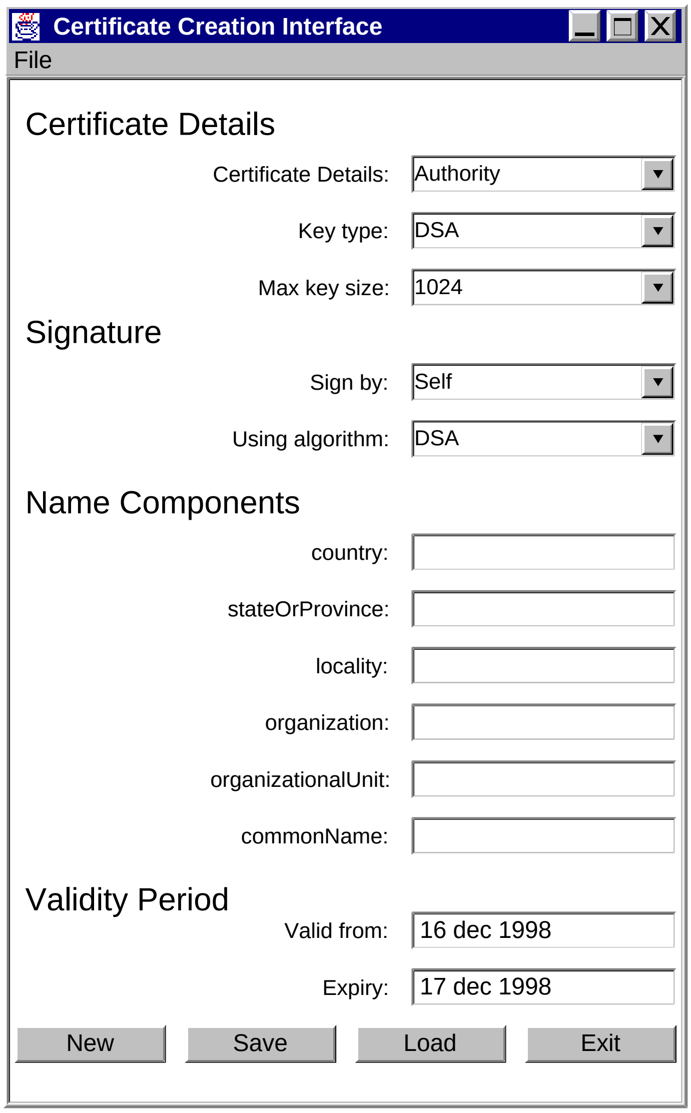
23.7.1 Main window

The main window (Figure 56) is divided into four data entry areas, plus a row of buttons and a status line.

Certificate Details

This area contains three choice items: Certificate type, Key type, and Maximum key size.

An SSL certificate comprises a sequence of X509 certificates, starting with a *user* certificate, followed by one or more *authority* certificates. The certificate type choice is used to distinguish these two X509 certificate roles.


Certificate Creation Interface

File

Certificate Details

Certificate Details:	Authority
Key type:	DSA
Max key size:	1024

Signature

Sign by:	Self
Using algorithm:	DSA

Name Components

country:	
stateOrProvince:	
locality:	
organization:	
organizationalUnit:	
commonName:	

Validity Period

Valid from:	16 dec 1998
Expiry:	17 dec 1998

Figure 56 CertificateGUI Main Window

Three key types are supported: DSA, RSA, and DH. As DH keys are only used for encryption, and authority certificates are used to sign other certificates, it is not sensible to generate an authority certificate that has a DH key type. The program detects nonsensical combinations when the generation of a new certificate is requested, and pops up an error dialog which details the problem detected.

Two key size options are provided. These always relate to the size of the public key of the key pair and are included so that users can comply with legislation. The actual key length is randomly chosen to be of a length which does not exceed the stated size, nor be significantly shorter. This has the effect of strengthening the algorithm over fixed key length choices by allowing more possible keys to be used.

Signature

This area contains two choices. The "sign by" choice provides a selection of the known private keys which can be used to sign the new certificate. In the case of a root authority certificate this can be the private key corresponding to the public key on the certificate itself (this is listed as "self", and is the only choice until an authority certificate file is loaded, or an authority certificate is generated.

The second choice item is for the signature algorithm to use, comprising two DSA based algorithms and three RSA algorithms. If a DSA key and an RSA based algorithm (or vice-versa) are selected, then no certificate is possible, again detected when certificate generation is requested.

Name Components

This area allows the entry of the most common name components used in X509 certificates, chosen from a vast collection of esoteric possibilities provided by the x509 specification. At least one of the fields must be non-empty to avoid an error dialog.

Validity Period

Although it is possible to generate certificates which expire at a date prior to their "valid from" date, or at a date in the past, this is regarded as being an error.

“New” Button

This generates a new key pair and produces a new X509 certificate holding the public key of the pair with signature, name, and validity period as entered into the areas as detailed above. The key generation can be quite slow, and whilst this is being done the status line (below the buttons) explains that this is happening. Once the objects have been successfully created, an dialog pops up asking for a single name to use to refer to them by. This is used as the base name for files when they are saved.

“Save” Button
“Load” Button

These open the load or save dialogs respectively. These operations are also available from the file menu.

“Exit” Button

This exits the program, after prompting for confirmation.

23.7.2 The Save Dialog

This dialog contains a directory area at the top. Under this is a list of the objects that have been created but not saved to file. For each authority certificate there will be two entries; the x509 certificate by itself (for use in checking a certificate chain for a trusted root), and a pair comprising a private key and SSL certificate (for use in signing certificates in a subsequent session). A user certificate just has the pair of the private key and SSL certificate.

The Save button initiates the saving of the selected item (if any) to file. When a key and certificate object is saved, the private key component is encrypted prior to writing the file, and to do this a password entry dialog is popped up. The private key encryption dialog has two text fields, for the password and its repeat, plus a choice of the encryption algorithm to use.

23.7.3 The Load Dialog

The directory area of the Load dialog is identical to that in the Save dialog. Underneath this is a list of the files which may be loaded. This will only list those files with names of the form `*name*_key_and_cert.der` that contain a key and SSL certificate pair, and where the main X509 certificate is not currently loaded.

The Load button initiates the loading of a key and certificate from the selected file. The private key read from file is assumed to be encrypted, and a private key decryption dialog will appear. If the correct password is entered, the key will be decrypted and the pair loaded.

23.8 Generating Temporary RSA Key Pairs

A secondary program, `TempKeyGUI`, is provided for generating temporary RSA keys. This is much simpler. The main window has a choice of maximum key size, which would normally be selected as 512 bits, as this is the commonly used maximum size for temporary RSA keys. The New button generates a new pair of keys, and then pops up a Save dialog. This dialog has a directory area identical to that in the certificate generating program, followed by a file name field, a password and password repeat field plus a choice of algorithms to use to encrypt the private key of the pair.

24 RMP BINDER (BLACK)

24.1 Introduction

Black supports reliable message passing between a group of member objects. Messages are transmitted by methods with fully typed arguments and a `void` result. Message methods can be grouped into interfaces just like call methods and a group may export many interfaces. Black is currently unfinished. In particular it reuses some components that should be customised for efficiency.

A group *member* is both a client and a server. A member sends a message to the group by invoking the corresponding method on the FlexiNet stub for its interface. Control is returned to the sender as soon as the message has been transmitted and only a local runtime exception may be thrown. The protocol delivers the message to each member of the group (including the sender) and its method is then executed by each member. There are no reply messages (not even null ones or exceptions).

The protocol that Black uses is based on a subset of the “Reliable Multicast Protocol” (RMP) which is described in [RM] and which in turn is based on the Chang and Maxemchuk protocol [CM84]. The protocol characteristics and state machine are described in section 24.4.2.

The RMP protocol controls the QoS (reliability, ordering and resilience) of message delivery. Black only implements a subset of the RMP QoS options. RMP provides unreliable and reliable delivery; Black only provides reliable delivery. RMP provides unordered, source ordered, and totally ordered delivery; Black only provides totally ordered delivery. Both provide K resilient, majority resilient and totally resilient delivery. (K resilient means the message has been acknowledged by K nodes before the protocol delivers it to the application.)

The Black binder and protocol stack are based on the Green ones. The top three layers (`Call`, `Serial` and `Name`) of the Black stack are the same as those used in Green. The `MessageLayer` is a heavily cut down form of `RexLayer`, which cuts out all the, acknowledgements, probes, timeouts and retransmissions. This is because the lower layers provide reliable delivery.

The `SessionLayer` is the same as used in Green. The `SessionManager` and Session could probably be cut down.

The FragmentationLayer is inserted between the MessageLayer and the SessionLayer. Currently the fragmentation layer from Green is used, but this supports NACKs for reliability, and this is redundant when used over the RMP protocol.

In a well-structured world, RexLayer would be a subclass of MessageLayer, and the Green FragmentationLayer, SessionManager and Session would be subclasses of the Black ones.

The Reliable Multicast Protocol is implemented in an extra RmpLayer inserted beneath the SessionLayer. In addition, UdpLayer has been extended to form a MulticastUdpLayer.

24.2 Basic Operation

Upon initialisation, the Black binder creates a single protocol stack that is used for all invocations – whether client or server side. This is illustrated in Figure 57.

In addition to the stack, there are a small number of shared resources – factories for input and output buffers, and a Session Factory.

At the moment only a single group can exist. When multiple groups are implemented, the RmpLayer will multiplex a number of MulticastUdpLayers using data structures similar to sessions.

24.2.1 Name Generation

All exported names are TrivNames – pairs of a UDPEndpoint and an integer interface ID. In order to generate a name, the binder calls the TrivNameLayer to generate an interface ID, and then pairs this with a new UDPEndpoint containing the group's multicast IP address and UDP port .

24.2.2 Name Resolution

To resolve a name, a new Stub is created, and initialised with a reference to the top of stack, and the name being resolved.

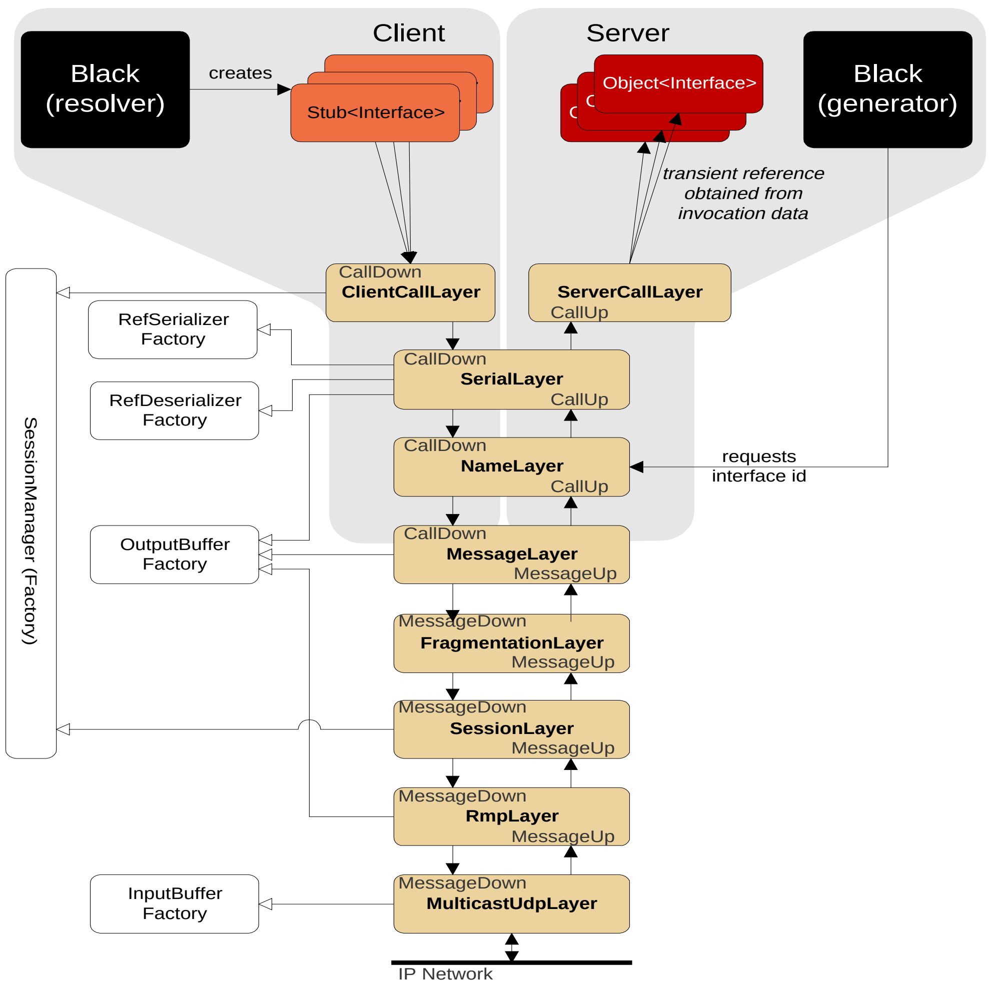


Figure 57 The Black Binder and Protocol Stack

24.3 Client Side Call Processing

In this section we walk through the process of a call down the stack. In subsequent sections we consider how a message is received off the wire, and how requests are handled on the server.

24.3.1 Call Layer, Serial Layer, Name Layer

Identical to Green (sections 20.3.1-20.3.3).

24.3.2 Message Layer

This layer is responsible for managing method invocations over asynchronous messages.

`MessageLayer` is a cut down `RexLayer` (section 20.3.4). Basically, it converts Invocations into messages. All timeouts, queues, retransmissions and protocol messages (ACKs and Probes) have been removed. Sequencing has been left in for now as a check on the `RmpLayer` ordering. Request reply pairing has been left in for when non-member call invocations are added to the `RmpLayer`.

Message invocations (i.e. with a void result) return up the stack to the caller as soon as the message has been transmitted.

24.3.3 Fragmentation Layer

Identical to Green (section 20.3.5), but the NAK protocol is redundant and can be removed.

24.3.4 Session Layer

Identical to Green (section 20.3.6).

24.3.5 Group Layer (RmpLayer)

This is an additional layer not present in the Green protocol.

Outgoing messages are typed as 'Data', source ordered, tagged with the sending host's IP address, sent on down the stack and queued for possible retransmission.

24.3.6 UDP Layer (MulticastUdpLayer)

For outgoing messages, the function is identical to Green (section 20.3.7), except that the message is sent to a multicast address.

24.4 Receipt of a Message

Messages are received off the wire symmetrically on client or server. It is only at the `RmpLayer` that they are determined to be application or protocol messages. In this section we walk through the progress an incoming message makes.

24.4.1 UDP Layer (MulticastUdpLayer)

Identical to Green (section 20.4.1), except that the layer listens on the group's multicast IP address and UDP port rather than the host's IP address and process's UDP port.

24.4.2 Group Layer (RmpLayer)

This additional layer is a massive change from Green (compared to this, every other change has just been minor tinkering).

The basic principle of the protocol is: a sender multicasts a message to all members of the group (including itself) and it is positively acknowledged by a single multicast ACK sent by a designated member known as the token holder (Figure 58). The ACK is also used to pass the token round all members of the group in turn so that each member can order, deliver and discard its messages with the required QoS.

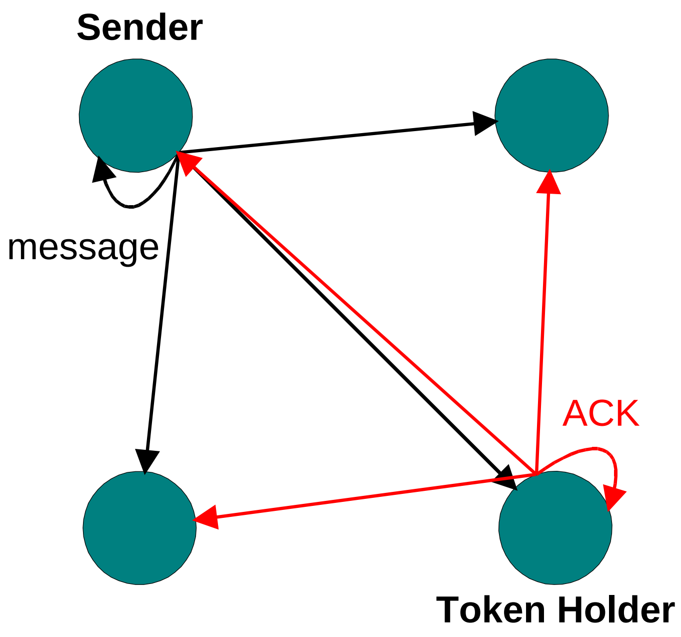


Figure 58 Basic Operation of RMP

But because UDP messages may be lost, its a little more complicated than that…

A sender adds its own UDP address and a local source order to a message and multicasts it to all members of the group. All (or most) of the group members receive the message and queue it in their holding queue. If the token holder receives the message, it assigns a total order to the message and multicasts an ACK message containing the senders UDP address, sender's source order, group's total order and the next token holder's UDP address.

The message sender periodically retransmits the message until it receives the ACK for its message. The token holder periodically retransmits the ACK until it receives an ACK with a higher total order or a ‘confirm’ message for the ACK.

When a member receives the ACK for a message, it places it in its ordering queue along with a slot for the message. If it finds the message in its holding queue it orders the message by transferring it from the holding queue to its slot in the ordering queue, otherwise it multicasts a NAK requesting a copy of the missing message. If a member sees a gap in the sequence of ACKs it multicasts a NAK requesting a copy of the missing ACK.

After a member has received K-1 [K <= group membership] ACKs after the ACK for a message then the member can deliver that message up the protocol stack for local execution. This is illustrated in Figure 59.

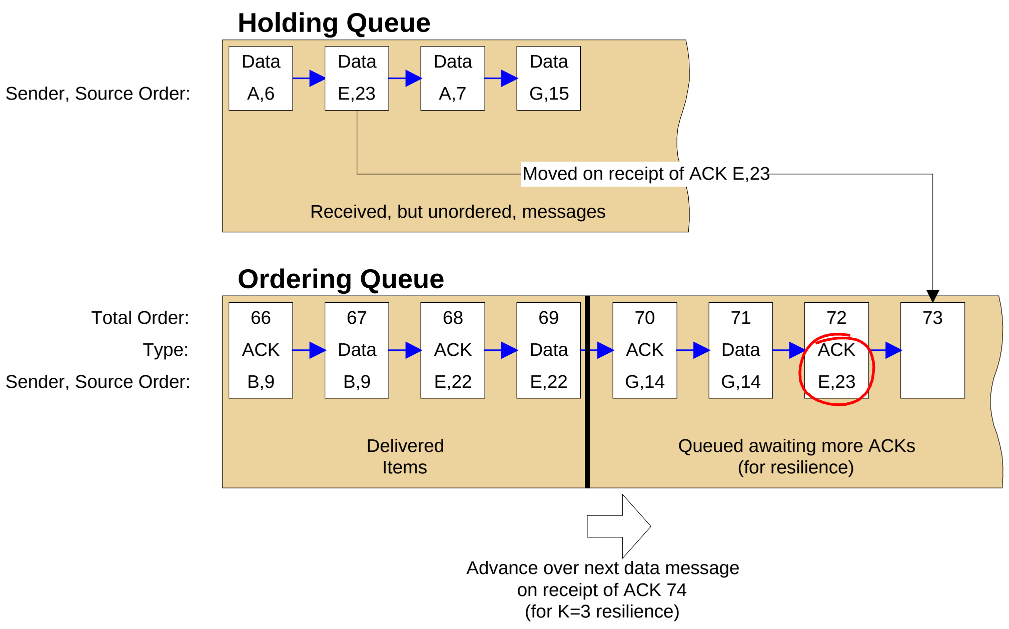


Figure 59 RMP Message Queues

Each member keeps each message in its ordering queue until the token has been passed round every member of the group since the message was ACKed, it can then discard it.

A member may not accept the token until it is in possession of all messages up to and including the message acknowledged by the ACK passing it the token. It must send out a NAK for each missing message and wait until it has received them all before becoming the token holder.

If a token holder, or member retrieving messages prior to becoming the token holder, receives a retransmitted ACK it should send a confirm message to the sender.

NACKs are replied to with a multicast of the request message or ACK by the token holder, because it must have obtained copies of all outstanding messages before accepting the token. They may also be replied to by the previous token holder, if is still waiting for an ACK or confirm message from the new token holder.

If messages are not sent fast enough to ensure timely circulation of the token, then a null token is transmitted round one complete circuit on a heartbeat time out.

Orderly group membership changes are initiated by Join and Leave messages which are replied to by the token holder with the new membership list in a List message. The List message is ordered in the same sequence as data messages so that the membership change takes place between the same data messages in each group member.

When the token gets lost, a reformation protocol is invoked to discover the remaining members, find the highest total order and elect a new token holder.

For further details of the membership and reformation protocols see [RMP].

24.4.3 Session Layer

Identical to Green (section 20.4.2).

24.4.4 Fragmentation Layer

Identical to Green, but the NAK protocol is redundant and can be removed.

24.4.5 Message Layer

No protocol messages are generated or received by the `MessageLayer`.

Messages are processed as in the `RexLayer` except that exceptions from message invocations are discarded.

24.5 Server Side Call Handing (above RPC)

Identical to Green (section 20.5).

25 OTHER BINDERS

25.1 Yellow – REX over TCP

Rex is a reliable protocol designed to run over an unreliable message passing system (such as UDP). However, it can also be run over TCP. This may be required for machines with no UDP support – or for access to services across a firewall – where TCP is a better supported protocol. The Yellow binder is identical to Green, except that it uses a `TcpDatagramLayer` instead of a UdpLayer. This layer 'hides' TCP connections to give a datagram interface similar to UDP.

Yellow was created both as a test to determine how easy it was to construct a new binder out of existing pieces (`TcpDatagramLayer` being the only new piece) and to provide TCP support at a time when Green was the only other binder. This was a half way point in the development of the first SSL-base binder, Rose.

Yellow is less efficient than Magenta – as the protocol management (the Rex Layer) is unaware of the connections, so cannot manage them effectively. Apart from specialist use, Magenta should usually be used in preference to Yellow.

25.2 Blue – REX over UDP with Mobility

This is a REX-based binder designed for use with clusters and capsules. It is implemented as Cluster-Capsule binder pair. It is described in the context of clusters in chapter 32.

25.3 Negotiation Binder

This was part of an early piece of work on negotiated binders (so early, the colour name convention hadn't kicked in).

25.4 Burgundy – Magenta using Blueprints

This is a version of Magenta that is constructed using the Blueprint system (chapter 29). Other than initialisation routines, its structure is identical to that of Magenta.

25.5 Other Binders Created During Development

A number of binders were created that have subsequently been replaced or that have otherwise become extinct. An incomplete list is as follows:

- Rose – REX over TCP with SSL
  This was created during early SSL experiments – before a more suitable TCP transport was available. It was removed because Crimson (RRP with SSL) is a better alternative.
- Lemon – REX over TCP with SSL and Mobility
  The mobility-aware version of Rose. This has likewise been replaced with an RRP version. In addition, the design of clusters evolved after this binder was designed, and Lemon was never updated.
- Purple – Same domain pseudo-binder
  This was a piece of binder 'magic' that was required to in order to initialise clusters. After a reengineering of this cluster abstraction, this became obsolete.

26 OBJECT SERIALISATION

26.1 Introduction

Serialisation is the process of converting a graph of objects in to a serial, or byte array, form. This is generally required either to pass the objects to another JVM (for example in a remote method invocation) or to store the objects persistently to disc. Serialisation is similar to *marshalling*. The difference is that serialisation is an object-oriented process – and in each object may be a *sub-class* of the class expected.

26.2 Serialising Interface References

When considering a serialisation system for use in a middleware platform, we must consider one important special case. When serialising a graph of objects, we must be able to distinguish between objects to be passed as data (by value) and objects that should be passed by reference.

In FlexiNet, we wanted to support the ODP abstractions as closely as possible, and so took the decision that references to objects should be passed by value, and references to interfaces should be passed by reference. This is illustrated in Figure 60. To determine the difference between a reference to an object and a reference to an interface, the *class* of the reference must be examined.

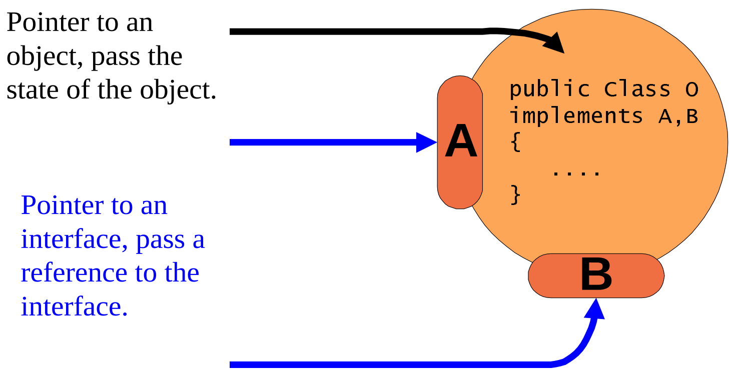

Pointer to an object, pass the state of the object.
Pointer to an interface, pass a reference to the interface.
A
B
public Class O
implements A,B
{
....
}

Figure 60 Determining Whether to Pass by Reference or Value

26.3 Java.io.Serializable

JDK 1.1 shipped with a ‘built in’ approach to serialisation. Although it would have been straightforward for FlexiNet to use this, unfortunately it was considered too limited. In particular, the Sun’s serialiser cannot be extended to deal with special cases on a *by use* basis. This is exactly what was required in FlexiNet to spot the difference between a reference to an *interface* on an object, and a reference to the object itself.

Sun’s serialiser does support extension, but only on a *per class* basis. Each class may provided special serialisation methods to serialise it and its subclasses. This is used in Sun’s RMI system, and is why RMI server objects must extend a specific superclass.

26.4 FlexiNet Serialisation Approach

Rather than producing a serialisation system which specifically distinguished between object and interface references, we decided to design a serialiser that could deal with *by use* special cases in general. The *per class* specialisation provided by Sun’s serialiser is a special case of this.

Every time a new use of an object is detected (i.e. every time an object is referred to by a reference of a different class), then a serialiser method is invoked to determine if this use should be treated as a special case, In the default serialiser (`BasicSerializer`) no uses are special, however by subclassing `BasicSerializer` we can provided for different special cases.

26.5 Contextual Information

When serialisation, there is usually a *context* associating the serialiser and deserializer. That is to say that the serial form does not have to be fully self-describing – the deserializer can determine some information about the data from context in which it is used.

For example, in a method invocation, the server expects each parameter to be a subclass of the corresponding parameter class in the method signature. If any of these classes is ‘Final’, then no typing information for that parameter need be included in the serialiser data. If a parameter is of the expected class (as opposed to a subclass), then a flag may be serialised instead of a full class name.

FlexiNet’s `writeObject` method therefore takes an additional parameter, the expected class of the object, as determined from context. This is used both to reduce the amount of contextual data written to the serial form, and to determine the way in which the object is being referenced.

26.6 Object Graphs

The basic operation of a serialiser is to walk over a rooted object graph. For each object the following procedure is undertaken

To serialise (object,context):

- If the context is a primitive type (boolean, char, int etc.), write the value it represents. Exit.
- If the object is null, write a flag to indicate this.Exit.
- Lookup (object,context) in the dictionary. If has been serialised before, write the dictionary index of the previous use. Exit.
- Add this (object,context) to the dictionary. Remember its index (`i`).
- Determine if this use is special. If it is, write `i`, and a flag to indicate which special case. Then use custom per-case code to write the appropriate state. Exit.
- If the object is not special, the context has had no effect. Lookup (object,null) in the dictionary. If it is found at index `j`:
  - Replace (object,context)->`i` with (object,context)->`j`
  - Write `j`
  - Exit.
- Add (object, null)->`i` to the dictionary. Write `i`.
- Write the true class of the object (see section 26.7)
- Write the state of the object in 'slices' from its ultimate superclass down to its concrete class.

  To serialise a slice of object
  - Check for special methods for serialising this slice, use if available. Exit. A per-slice special method might, for example serialise an object of class `Foo`, but then allow standard techniques to serialise additional fields added to a subclass of `Foo`. This is currently not used in FlexiNet.
  - For each field, write the data for that field. If the field contains an object, or an array of objects, write those objects by recursively calling serialize(object,context). The context is the type of the field. The fields may be read (and set) using Java core reflection.

Note.

The absence of some flags in the serialised byte stream must be detectable. In the current implementation, this is done by ensuring that each flag written is distinct from all the other values that may also be written at that point in the stream. In particular

- The null-object flag is a impossible dictionary index
- The special case flags are all distinct from the first item written to identify a class.

26.7 Class Serialisation

Classes must be serialised on two different occasions. Firstly, an actual class may be passed as the 'object' parameter in a call to `writeObject`. Secondly, classes must be serialised to identify the actual class of objects written to the stream. This second case is much more common.

There are many issues relating to class serialisation. Firstly, it is inappropriate to serialise the bytecodes making up the actual class. These may be very large, and in general, all classes imported by the serialised class would also have to be passed. As the recipient is likely to have loaded (or have better access to) some or all of these classes, this would be extremely wasteful. For this reason alone, the naive approach is rejected.

Class serialisation is therefore concerned with supplying identification information, so that a recipient can determine which class to load, and where the byte-codes for that class, and classes it imports, can be located.

There are several parts to this information

- What is the fully qualified name of the class?
- Which version of the class should be used?
- Where can the class be obtained?
- Which versions of imported classes should be used? and where may the classes be obtained?
- What is the certification to the authenticity and authorship of the class?

In addition to this information, there is an issue as to whether the recipient will be willing or able to actually load and execute the class. They may be unwilling due to security policy, or unable because the class is not accessible to them, or because they have already loaded a different class with the same name. (This second limitation is removed by use of the FlexiNet Class Repository).

There are many approaches to serialising classes, and for this reason, the FlexiNet serialiser may be parameterised with a `ClassSerializer` that is solely responsible for serialising classes. Two `ClassSerializers` have been written, a trivial one described in section 26.7.1 and a more complex one that interfaces to the FlexiNet Class Repository described in section 39.4.1.

Although serialising classes is in general a complex issue. In the majority of cases, the class to be serialised is exactly that class that is expected from the context of the call. For example, the method `op(Foo f)` will probably be passed an object of class `Foo` more often that an object of a subclass of `Foo`. In this special case, all that is required is for the serialisation of a flag to indicated that the class is as expected; the recipient will have already loaded this class in when loading the interface class. A second common optimisation is to spot subsequent references to a class previously serialised, and to serialise the index of the class within a dictionary. This optimisation is particularly signification when a number of interface references are serialised together – as they will tend to make use of the same naming classes.

26.7.1 ClassNameSerializer

This is a trivial class serialiser. It assumes that the peer deserializer has access to a class of the same name and version. In addition, it assumes that the peer uses its own means for validating the authenticity and authorship of the class.

The approach is straightforward

- If the Class is the *expected class* then a flag is written. Exit.
- The class is looked up in a dictionary. If it has an entry, the index is written. Exit.
- The class is added to the dictionary, and the index is written.
- The fully qualified class name is written.

26.8 Basic Serialiser

The superclass for all existing FlexiNet serialisers is called `BasicSerializer`, this implements the Serializer interface, which is used by other parts of FlexiNet to abstract the particular serialiser being used. Similarly, the peer deserialiser is called `BasicDeSerializer`. The 'basic' serialiser performs object-graph walking with no special cases. Other serialisers override the 'special' method to provided additional functionality.

26.9 Sun Compatible Serialiser (SCSerializer)

This is a subclass of BasicSerializer that spots per-class special cases in a manner compatible with Sun's `Externalizable` interface. This allows FlexiNet's serialiser to serialise application objects with special serialisation methods conforming to this standard interface. However, it should be used with a modicum of care, as the `Externalizable` interface does not pass context – and any per-use special cases will therefore not be spotted until the first level of recursion.

26.10 Reference Serialiser (RefSerializer)

This is the primary serialiser used by FlexiNet. It is a subclass of `SCSerializer`, so will deal with `Externalizable` special cases in addition to its own.

`RefSerializer` is designed to spot references to interfaces, and serialise these 'by reference'. To achieve this, it looks for two special cases.

26.10.1 References to Interfaces

References to interface are detected because the class of the *reference* is an interface class. The special case is handled by generating a name for the interface, and writing the following information to the output buffer.

- A flag to indicate that this is a special case
- The class of the interface being referenced
- The name generated to represent the interface (*care!*)

The interface name must be written with a modicum of care. It should be written as a 'value' object, not a reference. This is an issue with Smart Proxies, where the name will also be a proxy (which would trip the second special case). The solution is to use a pseudo-interface as a flag in the call to `writeObject`. This flag is spotted by the `RefSerializer` during the recursive call, and no special case is raised.

26.10.2 References to Proxy Objects

Proxy objects are used within FlexiNet to represent a 'view' on to an interface – typically to allow a client to have a reference to an interface on a remote object. If a reference to a proxy object is serialised, the name of the real interface should therefore be serialised instead. For the most part, this could be dealt with as a reference to an interface, as described above. However adding this special case allows us to deal with two problems described in the next section. When a proxy object is serialised, the following information is written.

- A flag to indicate that this is a special case
- The class of the interface represented by the proxy
- The name of the interface represented by the proxy

This information is identical to that written for the first special case, so the deserialiser does not have to distinguish between them.

26.11 Automatic Widening of Interface References

Java is an object oriented programming language, and frequently objects are *narrowed* in order to be manipulated generically, and then *widened* upon retrieval.

An example of this is use of the standard `Hashtable` class.

```
Foo f = new FooImpl();

Object o = (Object f) // narrowing
Hashtable.put("myfoo",o);
…
o = hashtable.get("myfoo");
```

```
f = (Foo) o; // widening
```

This approach is equally applicable in a distributed context, for example if an interface is stored in a Trader. However, there are two important issues. Firstly, if an interface is cast to be of type `Object`, then the `RefSerializer` will be unable to detect that the intention is to pass a reference not a value. The second issue is that widening of interface references is generally disallowed, as it is a security risk (a client gaining a reference to one interface on an object could illicitly convert it to a reference to a second interface).

These problems are tackled together by using a proxy. As the proxy stores a reference to a particular interface (of explicit type) then it is irrelevant how the proxy is cast. The `RefSerializer` will know both that it is to be passed by reference, and the class and name of the interface being represented. To use this feature, the programmer must *tag* that a narrowed interface may be widened by a client.

For example

```
 Foo f = new FooImpl();

 Object o = FlexiNet.tag(f,Foo.class) // narrow and tag
 remoteHash.put("myfoo",o);
…
 o = remoteHash.get("myfoo"); // same or different client
 f = (Foo) o; // no special action on widening
```

In practice, what actually occurs is that the *tag* operation creates a local proxy for the identified interface. This proxy contains explicit information about the class of the interface that it represents. This used by FlexiNet when it is serialised. This `tag` method provides a general mechanism for specifying constraints on exported interfaces. More examples are given in section 41.3.

To aid in the type safety of services that wish to handle references to 'any' interface, the class `Iface` is defined as the 'most general' interface class (in the same way that `Object` is the most general object class). A programmer may use this to indicate that an interface is required. This does not require that application interfaces actually extend `Iface` – this is handled by the proxy class used by the engineering – it is simply an annotation to encourage type safety.

26.12 StubSerializer

This serialiser was designed to be used when serialising application code in order to migrate it. Here the distinction between pass by reference and pass by value is different than from 'normal' FlexiNet use. We require that all objects are passed by value unless they are actually proxies to remote interfaces – in which case the names of the interface should be passed, and the proxies recreated when the state is deserialised.

To do this, `StubSerializer` overwrites special() with a method that looks for proxy objects. Proxies to remote objects are dealt with in an identical fashion to `RefSerializer`, however proxies to local objects are treated as 'normal' objects, so that they may be recreated in an identical fashion on the destination machine. A special case is that auto-generated stubs are not serialisable by ordinary means, so proxies for local objects which are themselves auto-generated, must be serialised in a special way.

`StubSerializer` produces output which is compatible with that output from `RefSerializer`. There is therefore no `StubDeSerializer` – `RefDeSerializer` is used instead.

26.13 Discussion

The decision to write a new serialiser for FlexiNet was a hard one. Sun's serialiser provides a standard that application programmers are familiar with, and in particular, classes that required special serialisation are much more likely to be provided for under Sun's system that ours. However Sun's approach is less flexible. In addition to requiring that 'server' objects extend a specific base class, it is not possible to change the action of the serialiser without changing the application classes. Suns serialiser cannot therefore be used to migrate application code in the same way as FlexiNet's. This is one of the primary reasons which FlexiNet's "Mobile Object Workbench" is currently a world beater.

There are two important limitations to the current FlexiNet system. Firstly, as we use a pure Java approach (as opposed to Sun) we must use Java core reflection to read and set fields on objects been serialised. In JDK 1.1, we are therefore restricted to serialising `public` fields. In JDK 1.2, it is possible to change the security privileges of the serialiser, and this issue will be resolved.

The second limitation is that `readObject` and `writeObject` methods provided by some classes under Sun's `Serializable` interface cannot be used by FlexiNet. This is because they take as a parameter a reference to Sun's serialisation system, and we cannot override this. We believe Sun appreciates the restriction this places on system designers, as they have provided `Externalizable` as an alternative interface. We can, and do, make use of this.

27 STUB GENERATION

27.1 Introduction

The role of a FlexiNet stub is to provide an auto-generated proxy object. This has a collection of methods, each of which, when invoked, cause the identical invocation to occur on some other, usually remote, object. Furthermore, the result value of such an invocation is passed back and returned by the stub method, or if an exception is raised, then this is similarly passed back and re-raised by the stub.

It is normal practice in systems which provide this sort of capability to include a language for describing the methods which the stub should have, and some sort of compiler which produces the stub from its definition. In FlexiNet the stub description language is replaced by a Java interface class, and the stub generator is a specialised compiler which constructs bytecode for stub classes. The bytecode, when loaded into a Java program, produces a class that implements the interface from which it was compiled. The stub generator is usually linked with the application, and generates stubs dynamically 'on demand'.

Each instance of a stub class contains a `body` object that embodies a connection to the 'real' service object that the stub is a proxy for. This is set when the stub is generated, and (generally) does not change.

27.2 Stub Generation

The construction of stub classes from interfaces is made possible by two features of the Java system. First, Java provides introspection facilities. These allow FlexiNet to analyse the interface class to obtain descriptions of all of the methods in the interface, along with their argument types, return types, and exceptions

Secondly, Java has user defined class loaders, which can load bytecode as an array of bytes to produce a class. In FlexiNet, just such array of bytes is produced by the stub generator, and as a result the stub class can be defined as well as loaded at runtime. This removes the need for an external, build time, compilation tool, as found in CORBA and Java RMI.

In FlexiNet terms, each stub method constructs an `Invocation` object to represent the invocation taking place. The invocation object must contain the following information

- The name and signature of the method being invoked. This is stored in an instance of the WrappedMethod class. This is simply a wrapper for java.lang.reflect.Method that allow more efficient access to some attributes.
- The target upon which the method is being invoked. This may actually be the name of the remote interface rather than an explicit java reference to the service object.
- The parameters being invoked on the method. These are passed as an array of objects.

In addition to this, each stub stores a reference to a *body* which it passes the newly constructed invocation to. This is the actual service object, or more typically, the top of a communications stack.

Invocation objects must be created dynamically, as one instance is required per invocation. However the method, body and name may be shared between invocations, and do not change during the lifetime of the stub. The method objects (one per method) may therefore be created and stored in the stub during construction. The body and name are created externally to the stub generator, but are also set at construction time.

Whenever a stub method is invoked, a new `Invocation` is constructed, containing the `WrappedMethod` object corresponding to the stub method, the name, and the arguments of the call as an array of objects. The `Invocation` object provides an `invoke` method which initiates communication. When communication has completed, the `Invocation` object also contains a result value to be returned or thrown by the stub.

This gives a the basic structure for a stub method:

- Construct an array of the arguments to the method call, and construct an invocation object containing it.
- Call invoke.
- Retrieve and analyse the result, to return or throw.

The bytecode for each method starts by constructing the argument array and initialising it with the arguments as passed to the stub method by the application. As not all Java values are objects, the bytecode must convert primitive types into corresponding object types. For object types a simple assignment will suffice.

Following the code for the construction of the `Invocation` object, comes the code to call the `invoke` method with the body as its argument. There is a possibility that this call will fail and raise a `BadCallException`, and the stub catches this exception and converts it to a runtime exception before re-throwing.

The final stage is to analyse the result. First, a method is invoked on the Invocation object to determine if the invocation raised an exception. For a non-exceptional return, the result must be converted from object to primitive type (if appropriate) by calling an type-specific method on the wrapper object. The return value is then returned as the result of the method.

For the case where an exception is to be raised, there are three possibilities.

- The return value may be one of the listed exceptions for the method (as obtained from the interface class)
- The return value is a `RuntimeException`
- The return value is an internal FlexiNet exception that should be converted to a general purpose runtime exception.

The bytecode generator creates bytecodes for a cascade of 'if' clauses to deal with these cases. An example of the source code equivalent to a stub method is illustrated in Figure 61.

The generated stub data is loaded using a specialised class loader. This is used because default Java class loading is from the file system, and we wish to avoid writing on-the-fly generated classes to temporary files.

As there may be many different (standard) class loaders within an JVM, there are potentially many different name-spaces for Java classes. For example, the Class Repository (chapter 39) makes use of this feature. The stub class loading system therefore must maintain (at worst) one stub class loader for each ordinary class loader. If fewer stub class loaders were used, then this might violate the separation between the different class name spaces. The stub class loading system keeps an table of stub class loaders, indexed by the class loader that the interface class was loaded with. In this way it can dynamically generate stub class loaders on demand, and ensure that the minimum number is created. The stub class loader loads the minimum number of classes, and in particular refers back to the (standard) class loader used to load the interface class in order to load any application classes that are required to support the interface. This approach also ensures that the stub can be safely cast to be of the interface type, and that the application need never refer to the stub type explicitly.

```
public Class Foo_Stub extends FlexiStub implements Foo
{
  private Object body;
  private Name   name;

  static WrappedMethod barMethod = new WrappedMethod("bar",
               "(Ljava.lang.String;i)Lbaz;");

  // method from proxied interface
  public Baz bar(String s,int i) throws
                             myException1,myException2;
  {

    Object[] arg = new Object[2]; // convert arguments to
    arg[0] = s;                   // object array
    arg[1] = new Integer(i);

    // construct new invocation object
    Invocation inv = new Invocation(barMethod,name,arg);

    try
    {
      inv.invoke(body); // invoke invocation on body
    }
    catch (BadCallException engineeringExc)
    {
      // engineering exception – rethrow
      throw new FlexiNetRuntimeException(engineeringExc);
    }

    Object rc = inv.getReturnValue();

    if(inv.isExceptionalResult())
    {
     // cast to approriate exception type and re-throw
      if(rc instanceof myExcepetion1)
        throw (myException1) rc;
      else if (rc instanceof myException2)
        throw (myException2) rc;
      else if (rc instanceof RuntimeException)
        throw (RuntimeException) rc;
      else
        throw new FlexiNetRuntimeException(rc);
    }
    return (Bar) rc; // return normal result
  }
}
```

Figure 61 An Example Stub Method

The stub generator has been designed to simplify the construction of other stub and proxy classes using an interface as a template. However, this remains a complex process, not least because the current bytecode generator only produces the subset of Java bytecodes that was required for our stubs.

Despite the fact that we only use a subset of set of Java bytecodes. The API for bytecode generation is still too extensive to list here. The generated form of the bytecode data, an object of class `ClassData`, has a print method that can be used to produce a file containing the class in human readable form. A utility Java program `PrintClass` is also provided. This reads in a class and outputs it in the same human readable format as the print method. These can be used to spot differences between generated an equivalent compiled classes.

28 RESOURCE IMPLEMENTATIONS

28.1 Introduction

This chapter contains a description of some of the resource and utility types defined in FlexiNet.

28.2 Buffers

FlexiNet, like most middleware platforms, supports a buffer abstraction into which outgoing messages are written, and from which incoming messages are read. This is used in preference to directly reading/writing 'to the wire' partly to aid code separation, and partly because block writes are usually faster at the network/operating system interface.

FlexiNet is unusual in that there is a high degree of separation between the different layers of a protocol stack. It is important that different layers that wish to write into an output buffer are co-ordinated – so that one does not write over data written by another. The approach taken is to use *segmented buffers*. These are buffers which are logically split into a number of segments. Each layer is then informed of the segment(s) it should use, and the layers are therefore kept independent. This is illustrated in Figure 62.

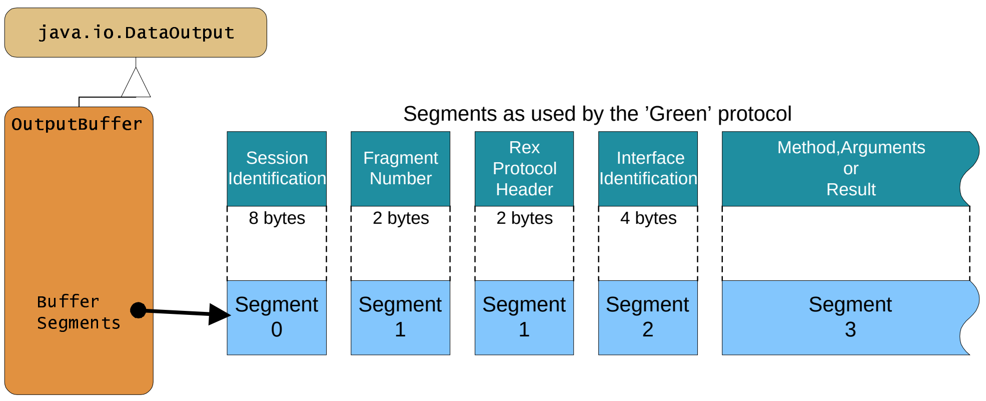


Figure 62 Buffer Segments

FlexiNet distinguished between *input buffers* and *output buffers*. Output Buffers implement the `DataOutput` interface, and primitive types may be written directly to them (the buffer performs appropriate marshalling).

Similarly, Input Buffers implement `DataInput`. All components use buffers via the abstract interfaces `InputBuffer` and `OutputBuffer`. This allows them to be reused with different buffer implementations, that perform different forms of marshalling, or that have different implementations for extending buffers.

There are two primary buffer implementations. Basic Buffers are a 'native' buffer implementation and are used by the majority of protocols. CDR Buffers are used in the GIOP protocols. They use CDR marshalling formats when reading/writing primitive types. They are also based on an earlier buffer implementation, which is less efficient at extending buffers.

28.3 Basic Buffers

The basic buffer implementation is finely tuned for performance. It uses a 'chain of packets' abstraction to store the data internally. New packets may be cheaply appended to the chain in order to extend the buffer. For datagram oriented protocols, each packet within the buffer corresponds to a packet to be sent on the wire. The buffer implementation can reserve a space at the front of each packet to allow per-packet header information to be written.

`BasicBuffers` are managed by Buffer Pools. As pooled resource, they contain a `recycle` method. However, they are not themselves reused. Instead, when a buffer is recycled, the packets contained within it are stored by the pool manager, and the buffer itself is discarded. This reduces the possible introduction of bugs that was found with arbitrary recycling of objects within other pool implementations. (see section 18.3.1). The following steps are taken to ensure safe recycling:

- References to packets are encapsulated by buffers and pools. Only these classes need be correct to ensure that a packet is not erroneously shared between two buffers.
- When a buffer is recycled, it is crippled so that it no longer contains a reference to its packets. If a buffer is erroneously used after being recycled, this will lead to an immediate failure rather than a lingering bug.
- The `finalize` method for buffers recycles them. This ensures that buffers are eventually recycled, even if the programmer forgets.
- Packets contain only bytes, not references to other objects. This reduces the impact of a buggy reinitialisation routine. We define that a new packet contains arbitrary data, so the need for reinitialisation is avoided completely.

To ensure that buffers can be rapidly allocated, we minimise their complexity. In particular, individual buffers do not contain information about buffer segments; they simply contain a pointer to the current position within the buffer. When a call is made to change the current segment, the buffer pool is consulted, and returns the segment's offset. Basic buffers can be optionally be compiled with additional bounds checking code that validate that reads/writes to a segment do not overrun. This checking is by default only performed when debugging of FlexiNet as a whole is enabled.

28.4 Priority Queues

Most protocols require some form of queue abstraction. In particular, many protocols require a *timer queue* which is used to resend messages over unreliable transports, or timeout connections or sessions. The `PriorityQueue` class was designed to perform efficient queuing of items ordered by key. A subclass of this, `TimerQueue` is used in Green, Magenta and most other protocols.

The priority queue manages an ordered queue of items, each of which is associated with a key. The item with the lowest value is stored at the front of the queue. The implementation is tuned to make the addition and removal of items as efficient as possible, regardless of their position within the queue. Internally, a heap structure is used to maintain the ordering relationship between items. The heap is implemented as an array of (references to) handle objects. Each handle contains a reference to the queued object, the key value and a back reference to the object's position within the heap. This allows for efficient removal from the heap, without recourse to searching. This is illustrated in Figure 63.

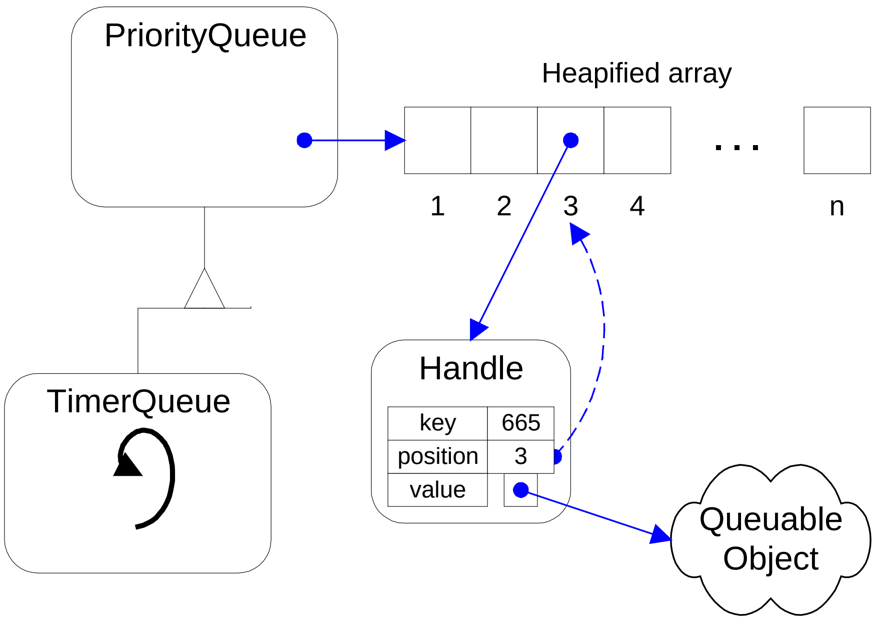


Figure 63 Priority Queue Implementation

Each object stored in the queue must implement the `Queueable` interface. This contains the `dequeued` method, which is used by sub-classes of `PriorityQueue` to inform an object that it has been removed from the queue.

An important feature of the priority queue abstraction is its interaction with synchronisation mechanisms. Frequently, a dequeued item must itself be locked prior to some action being taken. The thread performing the dequeuing would therefore lock the queue (to dequeue) and then lock the object (during processing of the dequeue event). Equally, an object that needs to be queued is often locked, this leads to acquisition of the locks in the opposite order. This could lead to deadlock.

To avoid this, the queue implementation ‘juggles’ its locks when dequeuing an object so that obtains the locks in the order object, queue. This removes the possibility of deadlock. The priority queue also manages the race condition of an object being simultaneously removed from a queue manually, and being de-queued (for example due to timeout). This allows straightforward use of priority queues.

For example if a mutex is held on an object, and a timeout occurs leading to the object being dequeued, the callback will not occur until the mutex is released and may be attained by the timer thread. If in the meantime the item is removed from the queue, the callback will not occur at all.

There is one issue here: in Java, there is no way to ascertain if a lock is held on an object without attempting to gain the lock (and possibly blocking). In the case of the timer queue, this may result in the callback on *other* objects being delayed, because the object with the lowest timeout is locked for a long period of time. For current protocols, this limitation is not significant, as mutexs are only held briefly; however this must be considered if the timer queue is used for other purposes.

28.5 Cache

Various parts of FlexiNet perform some form of caching. An abstract cache interface has been defined to allow different caching policies to be used interchangeably. The cache interface is shown in Figure 64. It is essentially a key-value lookup.

```
public interface Cache
{
  public Object put(Object key,Object value,int weight);

  public Object put(Object key,Object value);

  public Object get(Object key);

  public void resize(int size);

  public int getCurrentSize();

  public int getMaxSize();
}
```

Figure 64 The Cache Interface

There are currently three cache implementations, two of which are trivial.

- Bottomless Bucket
  This is a simple hashtable. It never discards anything from the cache.
- Null Cache
  This does not store anything at all.
- Blocking Hashtable
  This is a useful cache abstraction. It is essentially a hashtable, but has and additional method, `getb` (get blocking). This will retrieve an item from the cache, but if it is not present, will mark the callee as the 'handler' for this key. Any subsequent calls to get or `getb` *with the same key* will block until the handler has performed a `put` operation. This may be used to ensure that only one thread performs an expensive lookup if a cache lookup fails. The handler may put a null value into the cache to indicate that it has failed. The next thread to attempt a `getb` will become the new handler.

29 BLUEPRINTS

29.1 Introduction

FlexiNet is highly configurable, and a mechanism is required for the specification of complex component interdependencies. The `FlexiProps` system was designed to fit this purpose. This is a simple extension of the Java property scheme that allows arbitrary (non-string) properties. FlexiProps is still used for the specification of some properties, in particular for specifying quality of service constraints. However, it was found to be lacking when used to specify complex compound objects, such as binder protocol stacks. For this reason, the 'blueprint' system was introduced.

An important use of the blueprint system is to allow a client to specify modification to a standard template used in binder initialisation. For example, a client may wish to produce a variant of the Magenta binder that uses a different threading policy, or that uses a different buffer pool abstraction. The type checking facilities in blueprints ensure that the modification do not lead to an inconsistent graph.

29.2 Blueprints

Blueprints are a mechanism for specifying an arbitrary object graph and then co-ordinating the construction of the graph. Each node in the graph represents an object that is to be constructed, or a property value. The blueprint maintains a set of constraints on the class that the object might have, and the possible values of each property.

The blueprint system will validate a blueprint by asking each class represented within the graph to add constraints on the values of other nodes that it depends on. For example, a node representing a UDP messaging layer may place a constraint on the node representing the buffer property indicating the maximum size for a buffer.

The blueprint system will iterate over a graph until it has found a set of candidate values for each node that meet all constraints. It can then co-ordinate the construction of the object graph matching this blueprint.

The utility of Blueprints comes from the fact that only the top node within the graph need be specified, and this will add additional nodes and constraints, recursively specifying the full blueprint. A programmer may specialise this template be predefining constraints on any particular node within the graph, without having to be aware of the details about how this node relates to other nodes. If the programmer specifies that a non-default class should be used for a particular node, then this class will be used when the object graph is ultimately created, and when the classes are interrogated to add additional constraints.

Blueprints are used for protocol stack specification and construction. A common requirement is to use a default stack (as specified by an existing binder) with small modifications, such as a change in buffering policy, or a different serialisation factory. In these cases, blueprints allows straightforward specification with a guarantee the blueprints will report any conflicts, if the new graph is inconsistent, or one component is not compatible with another.

In principle all that is required is that each node has constraints set on its value, and a proof system could then be used to choose appropriate values. However in practice, such a scheme would be far to inefficient (and in general intractable). To make the approach practicable, for each node both *constraints* and *suggestions* are stored. For example, a messaging layer might constrain that the buffer size is less than 8K and suggest that 4K is a suitable size. This suggestion will be used if it meets any constraints placed on the property by other nodes, and if not other suggestions override it.

29.3 Blueprint Structure

A Blueprint graph is held internally as a tree structure with 'symbolic links'. Each node in the graph represents an object to be created or a property value, which is simply stored. On each node are a number of *suggestions* for a possible value, and a number of *constraints* on what the value may be. Once the blueprint has been resolved completely, each node will also contain a *value*. The object that the root node specifies is built or composed from the components specified in the lower parts of the tree. Blueprints are hierarchical for this reason, the object that is required is built from sub-components represented by the lower parts of the tree, which are themselves built from components is yet lower parts.

Each link in the tree is given a meaningful name, and a 'dot separated' pathname is used to identify a particular node. The root of the tree is an exception, and is unnamed. In Figure 65, node 5 is referred to as "a.d", while 6 which has a symbolic link from 2 may be referred to either as "b.f" or "a.e". The structure of a Blueprint defines scopes for each of the nodes. For example in Figure 65, all the nodes are visible in the root's scope. However only nodes 4,5 and 6 are in node 2's scope, and 3 is not accessible at all. The link "e" from 2 to 6 must be established from a scope that contains both 2 and 6, in this case only the creator of the Blueprint or node 1 may do that. It would be impossible for node 2 since prior to the link being established 6 is outside its scope.

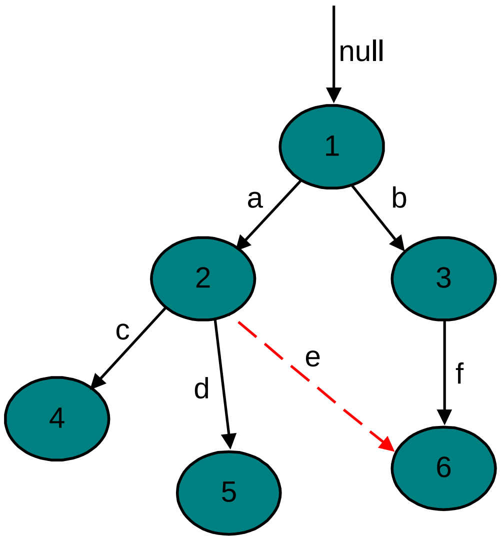


Figure 65 A Blueprint Graph

Internally, the graph below any node forms a valid blueprint, and refers to its descendants by relative names. For example, node 2 refers to node 6 by the name “c”.

Each node in the Blueprint may contain a number of suggestions as to its eventual value, and a number of constraints on that value. The constraints must be mutually compatible, that is, it must be theoretically possible to find a value that satisfies all the constraints. If a constraint is added which breaches this requirement the Blueprint will throw an exception, and fail to apply the constraint.

The order in which suggestions are made is significant. When there are two or more suggestions on a node, they will be considered in the order in which they were made. The reason for this is to ensure that the outer scopes take precedence in normal usage. This is discussed when Resolution is examined in detail.

29.4 Writing Classes for use in Blueprints

For a class to be a valid suggestion for a blueprint node, it must implement the `UniformCreate` interface and a number of static methods. The static methods are not specified in the interface, as Java does not allow this.

`public static void setRequirements(Blueprint)`

This allows a class to add suggestions and constraints within its scope. The method is passed the blueprint node that will eventually be initialised to be an object of this class. From this it can access all nodes within that node’s scope. Typical usage would be to constrain nodes that will later be used to set the object’s internal state.

`public static Object createUninitialised()`

This is used by the blueprints system to create an object graph, once the correct object class for each node has been ascertained. It simply creates an instance of the class. It is distinct from the no-arg constructor as it creates an object that is uninitialised.

`public int initialise(Blueprint)`

Initialises the object, obtaining any state needed from the supplied Blueprint. The argument is the same as for `setRequirements`. Any other initialisation that is needed should also be done at this point.

29.5 Use

To construct an object graph, a blueprint node for the root object is first created which suggests the (only) possible value for the root object. The creator may then add suggestions or constraints to the blueprint, or child nodes. There are three phases to the initialisation of an object graph from a blueprint.

- Firstly, the blueprint is resolved. To do this a suggestion is chosen for each node, and `setRequirements` is called for each node in the graph that corresponds to an object to be created. As each requirement is added, it is checked for consistency. The resolution algorithm will back off and try different suggestions until it finds a consistent set.
- Secondly, the graph is walked and `createUnitialised` called for each node. This constructs an uninitialised object graph.
- Finally, the object in each node is initialised.

Resolved blueprints may be stored rather than used to immediately create an object graph. This is useful, as resolution may be costly for complex object graphs. Blueprints have a utility constructor allowing a graph to be read from a previously saved state. A code fragment for constructing a binder using blueprints is shown in Figure 66.

```
// create a blueprint for an object of class Burgundy
Blueprint bp=new Blueprint(Burgundy.class,Burgundy.class);

// suggest that a special purpose serial layer is used
// This suggestion takes precidence over default suggestions
// as it is made in the outermost context
bp.suggest(“serial”,CompressingSerialLayer.class);

// Built the object graph and read the result
bp.construct();
Burgundy burgundy = (Burgundy) bp.get(null);
```

Figure 66 Using Blueprints

29.6 Resolution

Resolution is a complex process, and is in fact NP-complete. In practice, most solutions are found trivially, but the potential to store previously resolved graphs is important for difficult cases.

The resolver uses a backtracking algorithm which walks the Blueprint tree in a top-down manner (root first). At each node it picks the first untried suggestion and tests it against the current constraints. If the suggestion fails, it tries the next one, and so on, until it runs out of suggestions or a suggestion is accepted. In the case of running out of suggestions it returns a failure.

Once a suggestion is accepted, the node tries to resolve all the nodes below it (without following links). Should any of these fail, then the current suggestion at this node is deemed a failure and the next suggestion is taken.

If all the children are successfully resolved then the current suggestion is rechecked against the current constraints, since it is possible that the children have imposed new constraints. If all this is successful then the value for this node is set to the current suggestion.

If the node in question is the root of the Blueprint tree this results in a return from the resolution method, otherwise the node calls its parent to report successful completion. This call rather than a return allows the parent to fail (due to constraints imposed by its children), and unwind the stack to find the latest untried suggestion. In fact, all successful suggestions cause a call rather than a return. Eventually the root gets a call of this type, and so long as it's satisfied, the stack can unwind leaving the resolved tree.

In effect, the way the algorithm works is to do both the non-deterministic solution positing, and testing in a single phase. This means that when a suggestion is rejected, all possible sub-solutions that are as yet unresolved are rejected. In this way, the number of tests can be reduced. This in no way mitigates the fact that the problem is NP-complete, but it does mean that the number of solutions to be tested is reduced from the worst case. In principle this could be reduced further, by resolving those parts of the tree with fewest suggestions first, thus making the length of backtrack shorter when it does happen.

One bug in the algorithm has been found. If suggestions are made on a node that has already successfully resolved and a later node fails (implicitly one that is not in its scope of the resolved node), then the new suggestion will never be tried. This means that a possible configuration has been missed. Problems can only arise if suggestions are made over a link. As this feature is not currently used this is not presently a problem. This limitation could (and should) be rectified in a later revision of blueprints.

29.7 Instantiation and Initialisation

Instantiation is a very simple process. The blueprint tree is walked, and all for all nodes representing an object to be created, the `createUnitialised` method is called.

Initialisation is somewhat more complex. Again the blueprint tree is walked and for each node where an object has been created in the previous pass, the `initialise` method is called on the object, passing it the corresponding node as argument. It is possible that during initialisation of this object, it will need to make invocations on other objects within the blueprint graph that have not themselves been initialised. If an object is invoked as is not (sufficiently) initialised to handle the call, it should raise an `UnitialisedException`. An initialisation method may catch this and return a status flag to the blueprint system to indicate that it, itself, has been unable to complete initialisation. The blueprint system will continue initialising other objects, and will eventually return to this object and attempt to complete initialisation. It this way blueprints may create object graphs, where some parts of the graph are mutually dependent on each other.

The initialise method actually has three possible return values

- Complete
  This object is initialised. Do not call `initialise` again
- Progress
  This object is not yet fully initialised, but has made progress. It might be worth initialising other objects that might depend on this one.
- Stall
  This object is not yet fully initialised and has made no progress since the last call to `initialise`. If all objects are stalled, then no progress can be made, and failure is reported.

Note that it is the responsibility of the objects themselves to either behave idempotently with respect to initialise calls, or to record their previous return value, so as to prevent repeating previous set-up. Classes must correctly report progress or stall. A class that always returns Progress regardless of its real set-up status will cause an infinite loop and is an error.

The potential failure mode of the system is when there are two classes which require each other's services for set-up, and cannot offer those services until they themselves are completely initialised. In this case both will stall causing a failure.

29.8 Relationship to JavaBeans

The JavaBeans API suggest standard mechanisms for the creation of objects, and for the specification of properties. When beans are composed, one bean may veto parameter changes suggested by another bean (or by a visual tool).

In many respects, JavaBeans is an alternative system to Blueprints, all be it more primitive. If the two approaches were merged, then a commonly understood abstraction (Beans) could be used within FlexiNet, and the visual tools associated with beans could be used to aid the prototyping or construction of new binder stacks. An additional advantage is that the adoption of the bean style guide would give a strong relationship between property values and fields within an object. Given this relationship, much of the code responsible for setting requirements for a given class could be inferred from the names and types of the object's fields. This would make the design of a blueprint-compatible class more straightforward.

A merge of the two technologies is worthy of further consideration.

PART FIVE:
CLUSTERS AND CAPSULES

30 CONCEPTS

30.1 Distribution Transparencies

The RM-ODP standard identifies nine distribution transparencies. Of these, only two, *Access* and *Location* relate directly to remote invocation. In addition to this, some FlexiNet protocols may provide some degree of *Failure*, *Replication* and *Security* transparency. The other transparencies, *Migration*, *Relocation*, *Persistence* and *Transaction* cannot be tackled on a per-invocation basis. Instead, they require some notion of *encapsulation*, whereby all interactions with an object, or group of objects can be intercepted and managed.

The RM-ODP standard introduces the notion of a *cluster* as a unit of encapsulation. We have implemented this concept within the FlexiNet framework. Using this construct, FlexiNet may be used to support mobile objects (requiring Migration and Relocation transparencies), persistent objects (requiring Persistence and Failure transparency) and transactional objects (requiring Transactional transparency). These are discussed in chapters 36, 1, and 1 respectively.

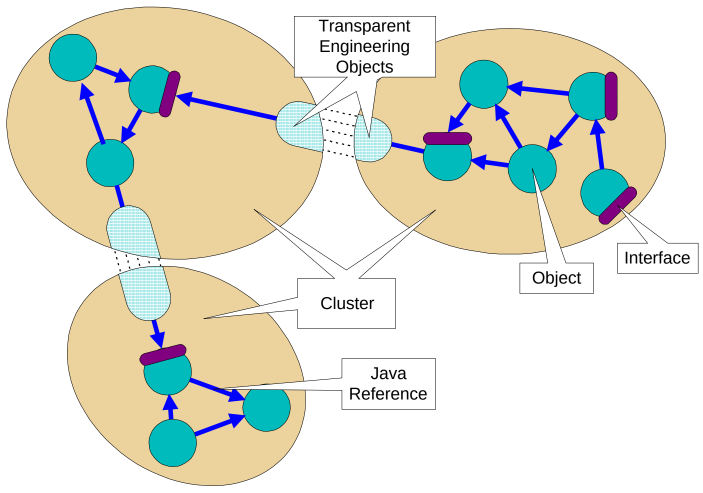


Figure 67 Encapsulated Objects

30.2 Encapsulation

What makes a set of objects encapsulated? We define this as a sub-graph of the entire object graph, such that all references into and out of the sub-graph (or *cluster*) are implemented by FlexiNet interface references. This is illustrated in Figure 67.

If we never pass references across FlexiNet interface references, then it is impossible for two clusters to join, so once encapsulated, a cluster remains encapsulated. In FlexiNet, when communication takes place across an interface reference, objects are passed by value, and interfaces are passed by constructing a proxy object in the destination cluster – not the sharing of a Java reference. In the design of a FlexiNet clustering mechanism, we are therefore only concerned with the initial construction of clusters. The normal FlexiNet invocation model will ensure that *once a cluster, always a cluster.*

30.3 Strong Encapsulation

One important application for Clusters is to support mutually distrustful pieces of code. When used in this way, Clusters provide a 'virtual JVM' environment, whereby different clusters are isolated from each other, to some degree. We term the techniques used to achieve this "strong encapsulation", in particular we arrange that:

- Each cluster contains its own thread group, and is a manager for those threads. Whenever a call is made from one cluster to another, then a new thread is created within the callee, to service the request. In this way, callee failure or blocking will lead to caller timeout (a 'safe' failure) and the caller has no control over the calling thread, and cannot block or kill it. The callee and caller are therefore isolated from each other.
- Each cluster has its own view on the class namespace. Classes loaded by one cluster cannot restrict the possible classes loaded by another. Each cluster also has a distinct set of static methods and data, so they cannot interfere with each other.
- Each cluster has its own security manager, and may have different security policies applied to it.

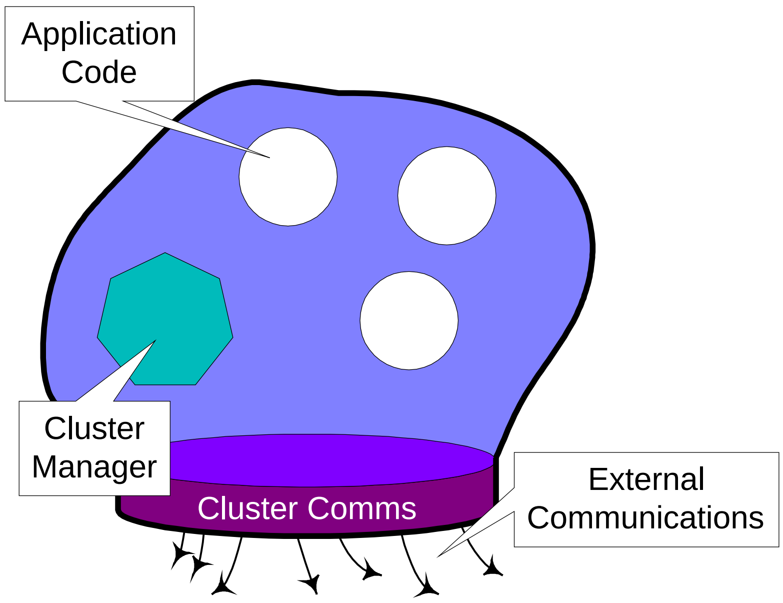


Figure 68 Cluster Components

30.4 Cluster Components

We distinguish between three components that together make up a cluster. These are illustrated in Figure 68.

- Cluster Manager
  Each cluster contains a distinguished *management object* that is used to orchestrate access to the cluster from outside agencies, and to control access to system resource from within the cluster. Different cluster managers may provide different facilities, for example, the *mobile cluster manager* provides an additional interface to support a mobility abstraction (see chapter 36).

- Cluster Comms
  The Cluster Comms component is the FlexiNet infrastructure that surrounds the Cluster and enforces the encapsulation. There may be different Cluster Comms implementations, that correspond to support for different communication protocols.

- Application Code
  The third component of a cluster comprises of the objects that make up the application cluster itself – the other two components being infrastructure components. A *null cluster* consists of a cluster manager and cluster comms, with no application code component.

30.5 Capsules and Nucleus

To support the cluster abstraction, there are two additional computational objects. A *capsule* is both a container and a factory for clusters. A particular

JVM may support several capsules in order to support different types of cluster, or for other reasons (for example one Capsule per user). In engineering terms, a capsule is simply a cluster with a 'more powerful' cluster manager – just as a factory is also an object.

A cluster (or capsule) is created by constructing an appropriate Cluster Manager, and them *wrapping* it with a newly created Cluster Comms. This associates the Cluster Manager with a new communications environment, and separates it from the rest of the process. This is illustrated in Figure 69. As cluster factories, capsules are responsible both for the creation of Cluster Managers, and for wrapping them. Capsules may also create other capsules.

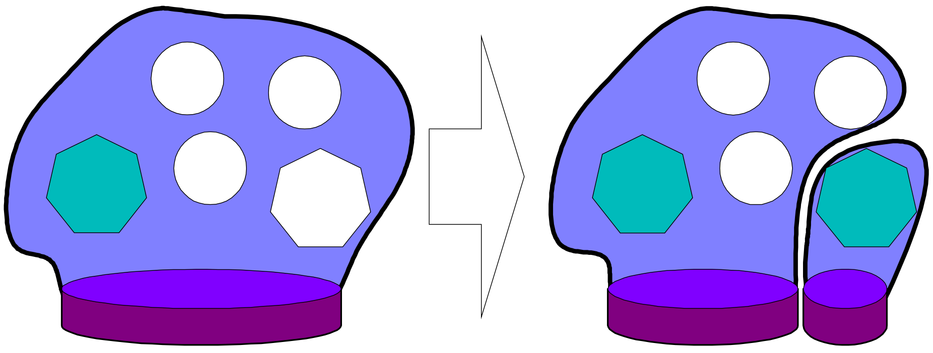


Figure 69 Wrapping a Cluster Manager to Create a New Cluster

The first capsule within a JVM is a special case, as there is no factory capsule to create it. The *Nucleus* is provided for this purpose. It provides a `wrapCapsule` interface used to wrap the first capsule created within a JVM. This is unusual in that only associates the capsule with a FlexiNet environment, it does not separate it from the calling thread – as that thread is not within an existing capsule. In effect, `Nucleus.wrap` wraps the entire JVM as a single capsule.

31 SYSTEMS ARCHITECTURE

31.1 Aims

In addition to the support of the Cluster and Capsule abstraction, there are a number of additional, engineering aims of the implementation. These where determined after an initial trial with an ad-hoc implementation.

- Configurable Communications
  Applications using FlexiNet are generally unconcerned with the selection of generators and resolvers made available to them. This has allowed the development of a 'plug-and-play' test bench, whereby a command line argument can be used to change the default binding protocol. This has been useful for both protocol testing, and to allow controlled integration of services in an evolving platform. As the cluster abstraction requires more intimate knowledge of the binding process, it is tempting to closely link these two disparate pieces of engineering, and indeed the initial cluster implementation was single-protocol. An important aim of the full design, was to allow the same sort of binder configuration as is found in non-cluster FlexiNet applications.

- Distributed Capsules
  Capsules are factories for clusters, and in the current implementation, clusters must be created within the same JVM as the capsule. However, for scalability it may later be required that clusters are created on a 'farm' of JVMs managed by the capsule. In the initial implementation, the cluster creation interface included the construction of an application object. In a distributed implementation, this would lead to the capsule having to load application classes – a potential bottleneck. The new interface was designed so that clusters are created *empty* – the burden of creating application classes is therefore moved from the capsule JVM to the cluster JVM.

- Extensible Cluster Management
  'Vanilla' clusters are of relatively little use. To provide a useful cluster abstraction, it is necessary to specialise some or all of the cluster manager, capsule or communications infrastructure. The system must be architected it such a way as to make this possible.

- Simplicity
  The engineering of Clusters is necessarily complex. However, as it is envisaged that programmers will wish to produce there own specialised Cluster abstractions, is it important that the internal

interfaces that such a programmer would need to understand are kept as straightforward as possible.

31.2 Cluster Identity and Address

A major aim with the design of the cluster abstraction was to support clusters that might migrate from one machine to another. This migration might be due to mobility of an executing cluster, or simply due to the restarting of a failed cluster. To aid in the construction of this abstraction, two concepts are introduced. The cluster identity is a pseudo-random identifier that uniquely identifies a cluster within a particular capsule, and *is expected* to be globally unique. The global uniqueness of a cluster identity cannot be guaranteed for two reasons. Firstly as it is a random number, there is always a possibility of clash, and secondly a malicious program may intentionally create clusters with the same identity. A cluster will normally retain its identity after it (logically) moves, however in the case of a clash, the cluster will be assigned a new identity. Cluster identities are therefore only guaranteed to be unique within a capsule, but the expected uniqueness property may be used in optimising the design of certain services (such as the relocation service).

The second concept that is introduced is a Cluster Address. This is a tuple of a cluster identity and a protocol specific address. It is effectively a name for the cluster as a whole. The names of interfaces *within* a cluster are always tuples of a cluster address and protocol specific inter-cluster multiplexing information. It is therefore possible to re-map an interface name from one cluster to a replica of the cluster, by substituting one cluster address for the other.

31.3 Components and Interfaces

A cluster comprises of two components, for each of these, there is a corresponding factory component within the Capsule.

- The Cluster Manager is created by the Capsule Manager
- The Cluster Comms is created by the Capsule Comms. This is used to ‘wrap’ the cluster manager and isolate it from the rest of the system. Once wrapped, objects within the cluster may only be accessed via the Cluster Comms.

This is illustrated in Figure 70.

The Capsule Manager is also responsible for managing clusters it has created via calls to the Cluster Manager on (a subclass of) the `ClusterManager` interface. In turn, the Cluster Manager may control its own Cluster Comms via calls to the `ClusterComms` interface.

During an invocation, the Cluster Comms will communicate with the Cluster Manager to inform it of a call’s progress, and to allow the manager to control interaction with the cluster.

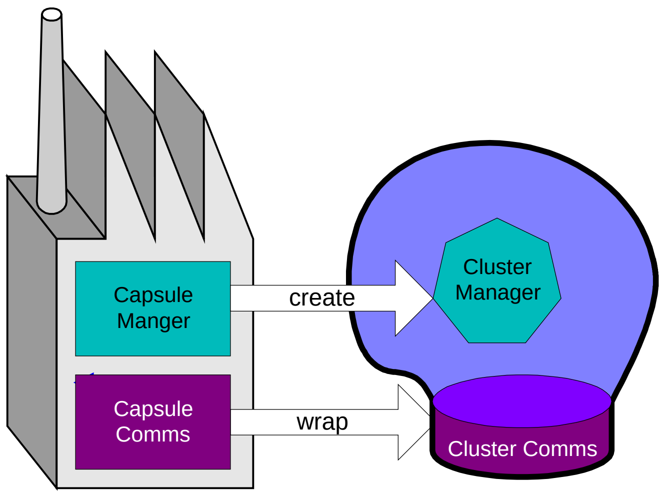


Figure 70 Creating a Cluster

In addition to these interfaces, there is one other standard interface. The cluster manager implements (a sub class of) `Cluster`. This provides a high level management interface, to allow application code to control the use of the cluster. The primary method on this interface is `createObject`, which is used to instantiate a cluster with application code.

The standard interfaces are illustrated in Figure 71.

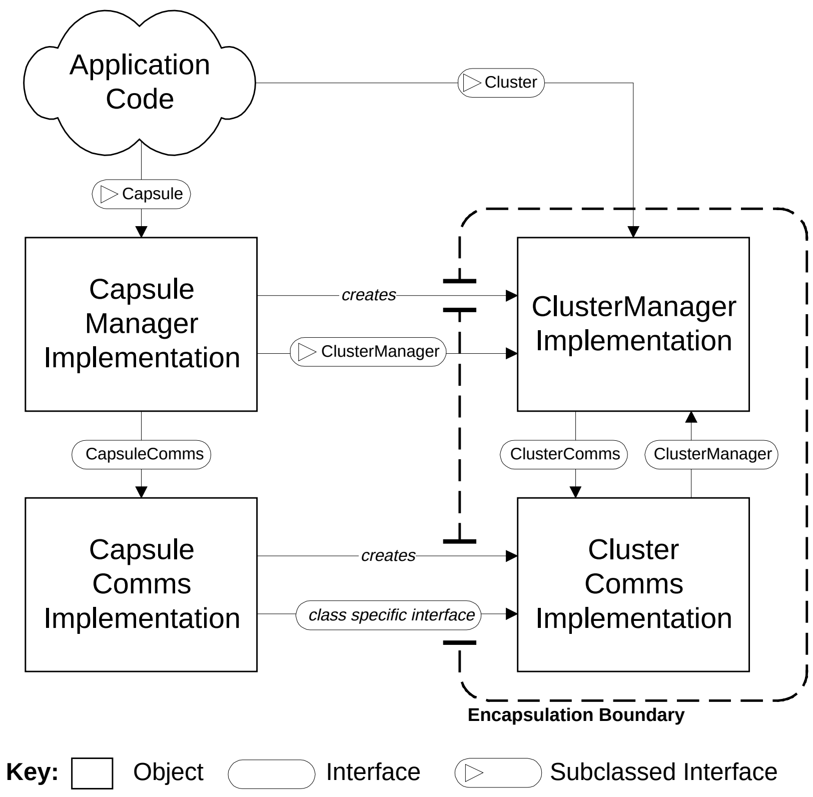


Figure 71 Standard Cluster Interfaces

31.3.1 ClusterManager

This is the 'most basic' cluster manager interface. Cluster managers designed for particular types of cluster are likely to extend this. This interface is made available to the capsule that created the cluster, as the cluster owner, and to the cluster's Cluster Comms. A cluster manager may also provide additional public interfaces, or interfaces for use by threads within the cluster.

The first method is called by the Capsule Comms during wrapping:

`public void setComms(ClusterComms comms)`

Set a reference to the local (per-cluster) communication sub-system. An explicit reference is required as a management interface to the FlexiNet generators and binders used in the cluster.

The next two methods are called by the Cluster Comms to orchestrate access to the cluster:

`public boolean startCall()`

A callback to inform the cluster manager that a client is attempting to call an interface within the cluster. The manager may block this call until the cluster is in a consistent state, or may return `false` to indicate that the cluster is closing down.

`public void endCall()`

A callback to inform the manager that a client has finished an invocation into the cluster.

The final method is used by Capsule Manager to obtain a reference to the public cluster management interface. Typically, the `ClusterManager` interface will be extended with additional methods for the Capsule Manager to use:

`public Cluster getClusterIface()`

Return the public interface to the cluster (if there is one). This interface will typically be a sub-class of Cluster.

Note. In the current implementation, there are two additional calls on `ClusterManager`. `GetThreadGroup` is used to determine the per-cluster thread group. This call (or an equivalent) should actually reside within Cluster Comms, to allow it to support weakly encapsulated clusters that do not have their own thread group. The second call, `setMobileNamer` is used to configure clusters that support mobile naming. This topic will be discussed in a later section.

31.3.2 ClusterComms

This interface is used by a Cluster Manager to configure and control its Cluster Comms. It has a number of methods which are mainly concerned with restarting a cluster from a previously captured state.

`ClusterAddress getClusterAddress()`

Return the address of the cluster as a whole. This may be used to later recreate a cluster at the same address, or to translate names issued by a cluster with one address to names issued by a second, equivalent, cluster.

`Object getExports()`

Return an object which encapsulates information about all the interfaces exported from this cluster. This may be stored, and later used to recreate a cluster with the same exports.

`boolean setExports(Object exp)`

The dual of `getExports`; set the interfaces exported by this cluster, to those identified in exp. When setting exports, the names of exported interfaces will be translated to the equivalent name in the new cluster. A client with a reference to an interface to the previous cluster may therefore convert it into a reference to the equivalent interface in the new cluster, by mapping the cluster address embodied in the name to the new cluster's address. The details of this a protocol specific.

`void dontExport(Object obj,Class cls)`

Inform the Cluster Comms that a particular interface is to be discluded from the set of exports returned by `getExports`. This is typically used to note that an interface is an engineering interface, and should not be recreated if the cluster were rebuild from a stored state.

`Name nameIface(Object obj,Class cls)`

Generate a name for an interface explicitly. This name may be stored and later used in a call to `nameIface(obj,cls,name)` in a new cluster.

`boolean nameIface(Object obj,Class cls,Name name)`

Give an interface a specific name. This is of specialist use, and is part of the process required to restart a cluster at a previous address. The name should have been returned by a previous call to `nameIface`.

`boolean redirectToCluster(MobileClusterName newCluster)`

This call informs the cluster comms that calls to the cluster should be redirected to a different cluster. This is typically used in mobile clusters, as the final action of a cluster manager, once a replica cluster has been created in a remote capsule.

```
Object getClusterByValueReference(Object originalRef,
                                  Class ifaceClass)
```

This call is used to obtain a reference to a remote cluster that is associated with a `StubSerializer`, rather than an ordinary serialiser. This is used to pass the contents of the local cluster 'by value' to the remote cluster.

```
Serializer   getClusterByValueSerializer(DataOutput output)
DeSerializer getClusterByValueDeSerializer(DataInput input)
```

These three calls are used to get engineering objects capable of reading or writing the cluster state. They are used when writing the cluster state out to a file.

```
Binder getBinderTop()
```

Return a reference to the top of the binding stack. Used only during wrapping.

```
void setMobileNamer(MobileNamer namer)
```

Indicate that names generated for interfaces should be 'mobile' in that they can be indirected via the relocation service 'namer'. This is used to allow a reference to be transparently moved from one cluster instance to another.

```
void destroy()
```

Destroy the communications system and prevent future incoming or outgoing calls.

```
void setNameHandler(NameHandler handler)
```

A hook into the binding system, set a handler to be used to catch `NameNotFound` exceptions and dynamically create service objects to service the failed request.

31.3.3 Capsule

This interface is used by application code to create new clusters. Particular implementations may extend this to provide additional per-capsule management functions

```
public Cluster createCluster()
```

Create a new cluster. This is the most commonly used call. A cluster will be created with the default types of Cluster Manager and Cluster Comms for this capsule.

```
public Cluster createHintedCluster(MobileNamer namer,
                                   GlobalID idHint)
```

Generate a new cluster using the specified information as a hint. `namer` is the mobile namer to use to name interfaces generated by this cluster. `idHint` is a suggested identity to give the cluster. This call is used when recreating a cluster from another cluster, or from a stored state. If the suggestions are taken, then the relocation service will be able to store the mapping more efficiently than if the suggestion is ignored.

```
public Cluster createNamedCluster(MobileClusterName name)
                                 throws BadName
```

Create a cluster of a given name. This is used to recreate a cluster at an old address after a crash. It is rarely required as the relocation service may be used to re-map references to a cluster with a different name. This will throw `BadName` if the name is unsuitable.

31.3.4 Cluster

The Cluster interface has a single operation used to create new objects within the cluster. A particular cluster implementation may only allow this method to be called once.

```
public Iface createObject(Class cls,Object[] args)
                         throws InstantiationException
```

Create an object of class `cls` within the cluster. If the object has an `init` method matching the specified arguments, then this will be called after the object has been constructed. This may return an interface, which will be passed to the client.

31.3.5 CapsuleComms

The `CapsuleComms` interface is an interface onto the Capsule Comms for use by a Capsule object. Typically, there will be one capsule comms per capsule, although it is perfectly reasonable to multiplex several capsules above a Capsule Comms.

```
public Object wrap(ClusterManager clusterManager,
                   Class          managerIfaceClass,
                   GlobalID       idHint,
                   ClusterAddress requiredAddress,
                   ClusterComms   calleeClusterComms)
                   throws BadName
```

Wrap a cluster manager to construct a new cluster. The parameter `managerIfaceClass` specifies the subclass of `ClusterManager` that should be returned as a wrapped interface to the cluster manager. `idHint` provides a suggestion as to the identity to be associated with the cluster (if non-null) and `requiredAddress` specifies the address the cluster should use (if non-null). The final parameter, `calleeClusterComms` is used to wrap the management interface before returning it. This is used when ‘spawning’ one cluster form another (as shown in Figure 69).

```
public Object wrap(ClusterManager clusterManager,
                   Class          managerIfaceClass,
                   String         address,
                   ClusterComms   calleeClusterComms)
                   throws BadName
```

A second form of wrap that takes a stringified address.
Deprecated – used by information space only.

```
public void addNameHandler(NameHandler nameHandler)
public void removeNameHandler(NameHandler nameHandler)
```

These methods are used to dynamically load clusters on demand. They specify name handlers to be used to resolve the names of clusters that a client believes are local to this capsule, but do not in fact exist.

```
public String getBootstrap()
```

Return a bootstrap string for this capsule comms, that may be used to reinitialise it after a crash.

31.4 Design of Cluster Comms

The cluster comms sub-system is essentially a per-cluster instance of the FlexiNet binding system. The Cluster Comms object itself is simply a wrapper for a binder graph. Resolvers within this graph may be 'standard' resolver classes, however Generators must provide some additional management facilities, and subclass the interface `ClusterGenerator` to do this. The additional facilities are required for managing the lifecycle of the cluster, in particularly, destroying it, and instantiating it from a stored state.

The `ClusterGenerator` interface extends Generator, and provides five additional methods:

```
public ClusterAddress getClusterAddress()
```

Return the address of this cluster. In the current implementation, each cluster may have only one (leaf) generator, and hence one cluster address. In principle a cluster could have a number of generators.

```
public Name mapName(Name originalName)
```

Convert a name generated by the corresponding generator in a previous incarnation of this cluster to a name within this cluster. This is performed by removing the embedded cluster address, and replacing it with the current one.

```
public boolean redirectToCluster(MobileClusterName newName)
```

Redirect any further calls into the cluster from names generated by this generator, to the named cluster. The 'newName' is a tuple of a cluster address and a reference to the relocation service that clients will contact when attempting to locate this cluster. Different generators may use different algorithms to achieve this, but the normal behaviour is to pass the redirection information on to the relocation service, and then refuse all incomming calls.

`public void destroy()`

Destroy the generator, and refuse all incoming calls. If called after redirectToCluster, then the redirection should continue to take place.

`public void setMobileNamer(MobileNamer namer)`

Set the relocation service (mobile namer) to be embedded in names generated by this generator. A client with a ‘mobile name’ of this form will contact the relocation service to determine whether the cluster has been migrated. In general, the relocation service is only contacted after an ordinary invocation has failed. However, for some clients or protocols, the relocation service may be contacted in advance (For example if the client has reason to believe that the cluster has migrated).

31.4.1 Shared Cluster Communications

If a capsule supports a large number of clusters, then it is unreasonable for each to have a completely separate communications system. In practice, this may lead to each having a listener socket and listener thread. To reduce the overhead, generators and resolvers for use in clusters are usually split into two parts; a shared per-Capsule Comms part, and a private, per cluster part. This is illustrated in Figure 72.

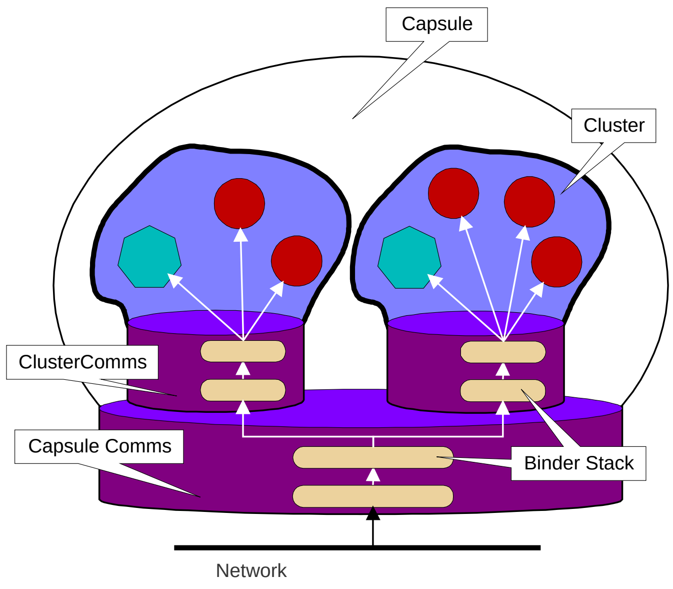


Figure 72 Sharing Communications Resources Between Clusters

The per-cluster part of a generator or resolver implements the `ClusterGenerator` or `Resolver` interface, as might be expected. A standard interface is also defined for the bottom, shared part. There is a high degree of independence between the two parts, and this allows one top part to be fitted over different bottom parts. The top part is typically involved with high level functions, such as interface naming, serialisation, access control etc, and the bottom part is involved with protocol specific functions, such as call control, error handing, multiplexing, threading and the like. The bottom part of a resolver implements the `CapsuleResolver` interface. The bottom part of a generator implements `CapsuleBinder` (as it may act as both a resolver and generator). These interfaces are explained in the following two sections.

31.4.2 ClusterResolver

This interface provides low level support for a per-cluster Resolver. Many of the calls are equally applicable to supporting a Generator. It has five methods.

`public int getNoBufferSegments()`

Cluster binders make use of segmented buffers in an identical way to standard binders. The per-cluster part of a binder needs to know how many of the buffer's segments are used by the per-capsule part. The per-capsule part claims segments 0 – (n-1) and the rest are used by the per-cluster part.

`public SessionManager getSessionManager()`

Return a reference to the session manager. The type of session manager is dependent on the wire protocol, and so is handled by the per-capsule part of the binder. However, the top part of the binding stack requires a reference to this, so it may obtain sessions for outgoing calls.

`public OutputBufferFactory getOutputBufferFactory()`

Similarly, the buffer type is protocol-specific, and is therefore managed by the per-capsule part of the binder. The per-cluster part requires this to obtain output buffers for serialisation.

`public CallDown getCallDown()`

Return a reference to the top of the shared part of the binder.

```
public Address parseClusterAddress(String address)
                                   throws BadName
```

Parse a cluster address.

31.4.3 ClusterBinder

This interface extends `ClusterResolver` and provides additional support for per-cluster generators that are split and multiplexed over a shared per-capsule part.

`ClusterAddress getClusterAddress(Object clusterRep)`

Return the address of a cluster. `ClusterRep` is the representative of the cluster, from the point of view of the per-Capsule portion of the binder. It is actually the object towards which incoming calls should be directed. This may be used both to determine the name of a cluster, and to cause the naming of a newly created cluster.

`ClusterAddress getClusterAddress(Object obj,GlobalID hint)`

A form of getClusterAddress that supplies a cluster id as a hint. This is used to obtain an address for a newly created cluster, and to specify a hint that it should reuse a previous cluster identity if possible.

```
boolean grantClusterAddress(Object obj,
                            ClusterAddress address)
```

Assign a specific address to a cluster. This will return false it the address is unsuitable.

`void dropCluster(GlobalID id)`

Remove all knowledge of the specified cluster. Calls made to this cluster in future should fail. The binder should not reuse this id in future, as clients of the old cluster might be unable to detect that the original cluster had been destroyed. An exception to this is if a call to `grantClusterAddress` or `getClusterAddress` is made with this id as a hint.

```
String stingifyClusterAddress(ClusterAddress address)
                              throws BadName
```

Stringify a cluster address.

`void addNameHandler(NameHandler nameh)`

Add a name hander. The name handler(s) will be called whenever an incoming call contains addressing information relating to a cluster that does not exist. The name handler may cause the cluster to be created (or, for example, recovered from disk), and the binder will then try again.

`void removeNameHandler(NameHandler nameh)`

Remove a name handler.

31.5 Generating a Cluster

This section gives a brief overview of the process of creating a cluster. It is illustrated by Figure 73.

a) a `createCluster` call is made on an existing Capsule Manager. This creates a Cluster Manager of an appropriate type.

b) The Capsule Manager calls its Capsule Comms. The Capsule Comms has a stored reference to the shared capsule binders and resolvers, and uses these to create new cluster binders and resolvers for the cluster. There are several different capsule comms implementations, and each implementation will produce a different set of binders and resolvers. These are analogous to FlexiNet testbenches (see Chapter 42).

c) The Capsule Comms creates a binder graph and a cluster comms to isolate the details of the communication system from the cluster manager. The cluster manager and cluster comms are associated with each other.

d) The capsule comms uses the binders to name the cluster manager interface from *within* the cluster, and then resolve the cluster manager interface from *outside* the cluster. This results in an encapsulated interface to the cluster. This interface is used by the Capsule Manager to externally control the cluster, in particular to obtain a public management interface to it. This may be used to create objects within the cluster.

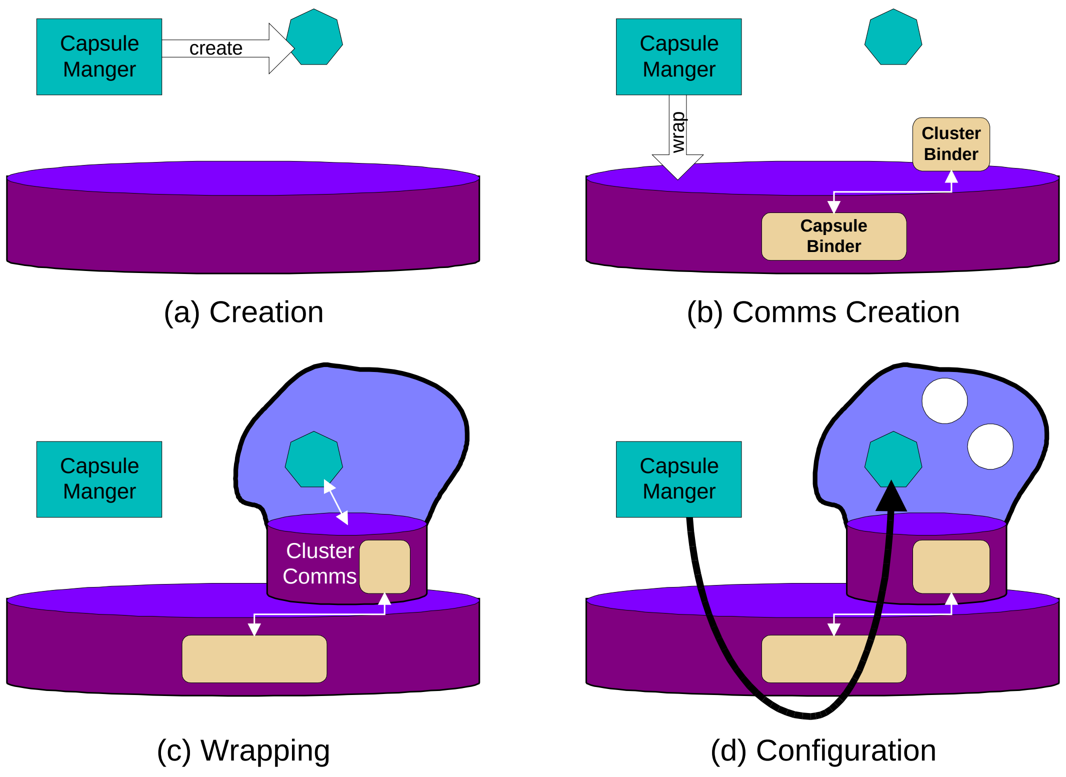


Figure 73 Stages in the Creation of a Cluster

32 A CLUSTER BINDER (BLUE)

32.1 Introduction

Blue is the archetypal cluster binder. It is both a generator and resolver, and assumes the provision of a `CapsuleBinder` to deliver invocations into the cluster, and to process outgoing invocations. In fact, Blue is the *only* cluster binder that has been built thus far. It has been deployed over a number of different capsule binders, and has been used with a number of different Cluster Managers, to provide different styles of cluster management.

The Blue Cluster binder has five layers, illustrated in Figure 74.

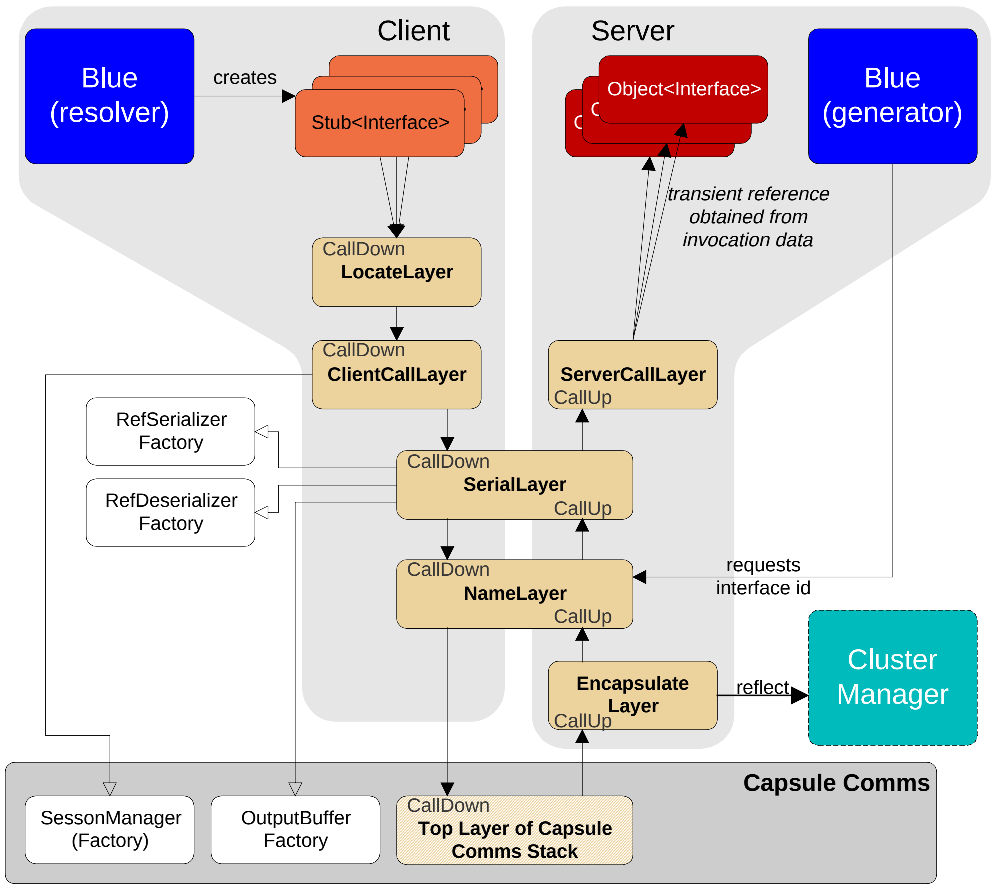


Figure 74 Blue Binder Stack

The function of most of these layers has been explained in previous chapters. In this chapter, we cover the remaining layers

32.1.1 Encapsulate Layer

This layer represents the encapsulation boundary around the cluster. Before an incoming invocation may proceed up the stack, the following must be validated:

- The cluster is ready and willing to receive the invocation. This is checked by a call to `startCall` in the `ClusterManager` interface. (Section 31.3.1).
- The invocation is handed over to a thread within the cluster's thread group. This is to support strong encapsulation. (See section 30.3).

These two operations must be are performed atomically. This allows a cluster manager to pause a cluster by first locking the cluster (i.e. blocking all calls to `startCall`), and then waiting until all executing threads exit. If the operations were not performed atomically, then there would be a possibility of a cluster manager checking for no thread activity, and then a thread being started by the Encapsulate Layer. An alternative approach would be to explicitly count the number of calls to `startCall` and `endCall`, and use this as a measure of activity, instead of the thread count. This approach would be advantageous in clusters that did not otherwise require strong encapsulation.

Note. The design of the `ClusterManager` could be improved with respect to strong encapsulation. Ideally, thread groups and activity monitoring should be a function of the Encapsulation Layer. This would allow different forms of encapsulation. This has not been updated in the current implementation, due to the complex locking strategies within the Information Space, which interact with cluster locking.

33 A CAPSULE BINDER (MAGENTA)

33.1 Introduction

Magenta is a capsule binder based on the RRP protocol. When used together with ‘Blue’ it generates and resolves names that are compatible with the Magenta binder. The binding stack produced by the Magenta Capsule Binder is shown in Figure 75.

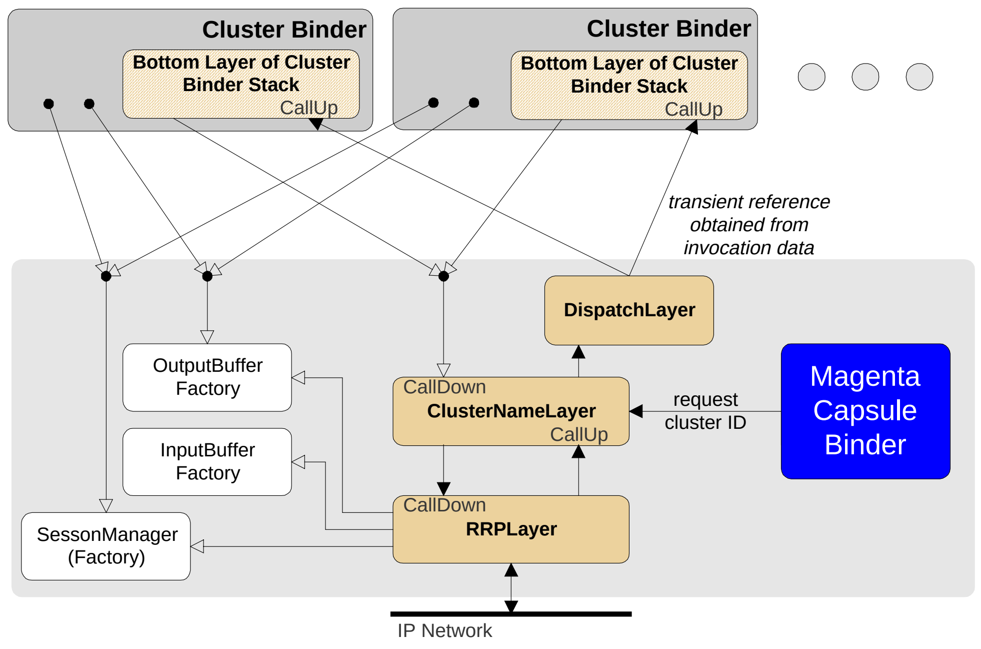


Figure 75 Magenta Capsule Binder

33.2 Binder Layers

The Magenta Capsule Binder has three layers.

`RRPLayer`

This is identical to the RRPLayer in the Magenta binder.

Cluster Name Layer (`GnameMuxLayer`)

This is a naming layer which is similar to `TrivNameLayer`, but which uses large (128bit) identifiers to discriminate between the multiplexed endpoints, rather than small integers. This is more appropriate for naming clusters.

A second difference is that it stores a mapping from identifiers to objects, rather than from identifiers to *interfaces on* objects. This simplification is possible as clusters are always accessed through the same middleware interface.

`DispatchLayer`

A form of `CallLayer` to deal with the fact that protocol layers have CallUp interfaces, not GenericCall. It simply calls `invocation.getTarget().callUp(invocation)`

34 ENCAPSULATION ISSUES

34.1 Introduction

Java is not a pure object oriented programming language. That is to say it is possible to access system objects and resource *without* first obtaining an object reference. This is a problem, as the encapsulation mechanism used in FlexiNet works by controlling the propagation of object references.

In particular, threads may access static methods and static fields. This circumvents the encapsulation mechanisms, as static fields and methods are accessible from all clusters. This problem is expatiated by the fact that most of the Java system resources (files, windowing, sockets etc) are accessed in this way.

A further issues is the use of callbacks. Java is sloppy in its handling of threads. If a system object makes an asynchronous callback to an application object, then this will generally take place in an arbitrary thread. As we use thread groups for protection and encapsulation, this is a dangerous loophole. This problem is particularly rife in the Abstract Windowing Toolkit.

34.2 Static Methods and Data

We divide the issue of access to static methods and data into two cases; access to system classes, and access to application classes. Access to system classes can be handled on a case-by-case basis. Access to application classes clearly cannot.

34.2.1 Access to System Classes

Public static data in system classes is rare. The use of public static methods is much more common. Thankfully, most of these calls are security checked – by a call to the security manager. We can arrange that each cluster (effectively) has its own security manager, and can therefore control the use of static methods to some degree (see section 34.5). Making this a watertight solution, particular in the case of future Java system classes, is a difficult task.

34.2.2 Access to Application Classes

There is more opportunity to control the use of application classes. We can arrange that different clusters use different class loaders, and thus obtain completely independent instances of classes. The use of static methods/fields therefore ceases to be a problem. This solution cannot be used for system classes (in general) as these classes must be loaded in order to get the basic FlexiNet system running. By this stage, it is too late to control the class loaders used. The potential cost of loading classes many times over is high, and for this reason we allow classes to be tagged to indicate that they may be safely shared without breaching the encapsulation rules. Currently, this tagging is a manual process, although an automated conservative checker could be built. The class loading approach and architecture is outlined in chapter 39.

34.3 Asynchronous Callbacks

Asynchronous callbacks allow a system thread to run application code within a cluster. The thread auditing used to count the number of threads will then fail, and the per-cluster security manager will also fail, as this relies on the thread id to determine the callee cluster. This is a dangerous problem, and in general must be dealt with on a case by case basis.

34.3.1 Abstract Windowing Toolkit

The AWT uses a great deal of asynchronous callbacks to inform the application of user interaction (mouse movement, button clicks etc). These call backs are made by an essentially arbitrary thread. A further issue is that the AWT creates a number of background threads, and it does this using the thread group of the first thread to call particular AWT methods. This means that threads may get created within the cluster thread group that are nothing to do with it, and that can never be destroyed. This upsets the thread auditing, and in particular, mobile object become unable to move.

To solve these problems, a wrapper for the AWT is provided. This is of two parts. A demon is started prior to the use of clusters. This 'tickles' the AWT to cause it to create its background threads. These are therefore created in a safe, non-cluster thread group. The demon then exits. The second part is a wrapper for the AWT event system. This is used to catch AWT events, and then re-signal them within the cluster's thread group. Correct use of this wrapper avoids all the AWT related encapsulation issues. The per-cluster security manager may be used to ensure that a cluster does not access the AWT event system directly. The event wrapper is illustrated in Figure 76.

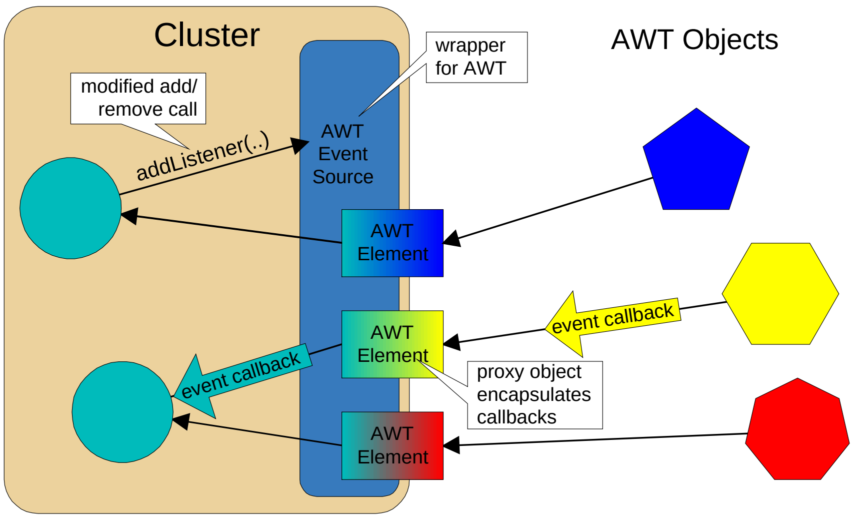


Figure 76 AWT Wrapping

34.3.2 Finalize Methods

Before a Java object is garbage collected, its `finalize` method is called. This may be used by a malicious or erroneous program to ‘capture’ the finalizer thread and use it for its own purposes, leading to all of the problems outlined above. The current implementation ignores this issue, but a simple fix is to give the finalizer thread low privileges, removing the utility in this attack. A cluster could still use this to upset the thread auditing, but not in a way that would be to their advantage.

34.4 Resource Usage

One issue that clustering does not address is resource usage. A thread in one cluster may use up all the available memory or CPU resource. There is no way of preventing this without operating system support. Strongly encapsulated clusters are *lightweight* processes. They do not have the resource controls that full processes do. However, cluster can be run in different real processes, and two correct cluster will communicate identically (albeit more slowly) if they are running on separate processes than in the same process. In a distributed application running distrustful clusters, this approach should be used to limit the risk associated with running potentially malicious code.

34.5 Cluster Security Manager

The cluster security manager is illustrated in Figure 77. 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.

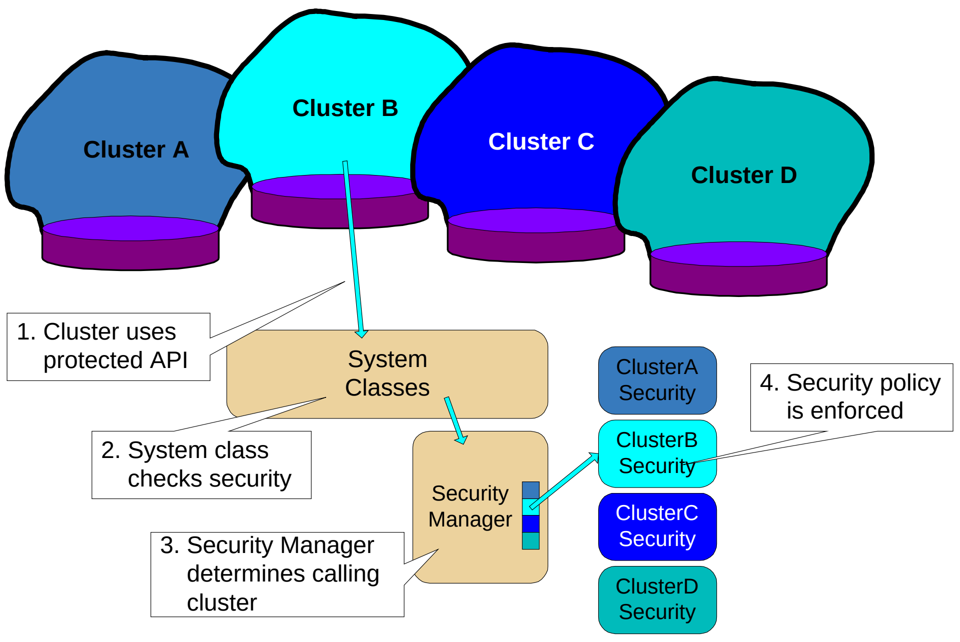

Cluster A
Cluster B
Cluster C
Cluster D
1. Cluster uses protected API
System Classes
2. System class checks security
Security Manager
3. Security Manager determines calling cluster
ClusterA Security
ClusterB Security
ClusterC Security
ClusterD Security
4. Security policy is enforced

Figure 77 Per-Cluster Security Managers

35 MOBILE NAMING

35.1 Introduction

In general, clusters support *mobile names*. These are names which used to identify cluster that may migrate from network address to another. This may occur, for example, if a host crashes and a service is restarted on a different host. Mobile names contain two pieces of information; the last known address of the service (a FlexiNet name) and a reference to a 'relocation' service that can provide current addressing information.

In general, either of these pieces of information may be omitted. If the last known address is not specified, then the relocation service is always contacted. If the relocation service is not specified then a 'well known' service is used. In the current implementation both the last known address and relocation service are always specified in mobile names.

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.

35.2 Interface

The relocation service can be implemented in a number of ways. The interface to it is straightforward and has the following methods:

```
MobileClusterName resolve(GlobalID clusterID)
        throws NotFoundException
```

Determine the last known address for a specified cluster. This call returns a structure consisting of the (new) cluster id, the last known address and the mobile namer to be contacted in future. This is to allow a mobile name to be moved from one instance of the relocation service to another, and allows a cluster to be assigned a new identity (to deal with clashes).

```
boolean remapName(GlobalID clusterID,
                MobileClusterName newName)
```

Inform the relocation service that a particular cluster has a new current address. This may also specify a new mobile namer that has taken over management of this cluster, and a new identity for the cluster, to be used in future.

`void dropName(GlobalID clusterID)`

Discard all knowledge about a particular cluster. This is usually used when the cluster has been destroyed.

Generally, the identity assigned to a cluster will not change over its lifetime. This allows the relocation service to optimise storage of information about a cluster. 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.

35.3 A Toy Relocation Service

The current implementation of FlexiNet relies on a 'toy' relocation service that trivially implements the above interface. It is not crash resilient, and in some obscure cases will leak memory. More importantly, it is not secure. Malicious clients may cause errors be creating several clusters with the same identifier.

The implementation is based on a simple hashtable, which maps cluster ids to mobile cluster names. For clusters which have never been assigned a new identity, and that still use the original instance of the relocation service, then this stores 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 new relocation service, but has not changed identity (or has done both at once), then one entry is stored in each instance of the relocation service. At worst, clients will have to contact each of these in turn before correctly locating a cluster. However, if moving between relocation services is rare (as is expected) then clients will generally have references to the correct relocation service.

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.

The relocation service 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.

35.4 Failings of the Toy Approach

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 instance 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 has the following features. This is outlined in chapter 47. The first three features are also provided by the toy implementation.

1. We arrange that a client can locate the appropriate relocation service for a cluster rapidly, without having to search.
2. We allow naming records to be moved between relocation services 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 loop hole, removes the reliance on a statistical probability (that randomly choosen 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 new instance of the service can be archived and never need updating. We allow these records to be ultimately deleted.

PART SIX: MANAGED OBJECTS

36 MOBILE OBJECTS

36.1 Introduction

A ‘mobile object’ abstraction was added to FlexiNet as part of the work on the “FollowMe” project. As such, this abstraction was tuned towards the needs of autonomous mobile agent systems [HAYTON98].

The mobile object abstraction provides a means for a cluster to move from one capsule to another. This movement is initiated by a thread within the cluster, and is co-ordinated by a specialised cluster manager. When a cluster moves, it is the responsibility of the cluster itself to cleanly shut down any executing threads. These may then be restarted in the new location.

Over and above the abstractions provided by ‘vanilla’ clusters, and mobile naming, all that is required of a mobile cluster implementation is a procedure for co-ordinating a move, and a mechanism to migrate a cluster’s state.

36.2 Orchestrating Mobility

If a cluster is to be migrated from one host to another, then the move must be *atomic*. That is to say, there must be no processing of the cluster between the point at which the cluster state is recorded, and the point at which the cluster state is recreated on the new host. In addition, from the moment when the cluster is recreated, the old version must be discarded.

To ensure this, we must halt all processing within the cluster prior to attempting a move. As it is not possible to pause a thread, and then restart it on another machine, in order to suspend processing, we must halt all threads. As mobile cluster use strong encapsulation, threads are created within a cluster in two circumstances: when a thread within the cluster explicitly creates another thread, and when an invocation enters the cluster in order to process a method invocation from an external client. We must wait for all of these threads to terminate before processing a move request.

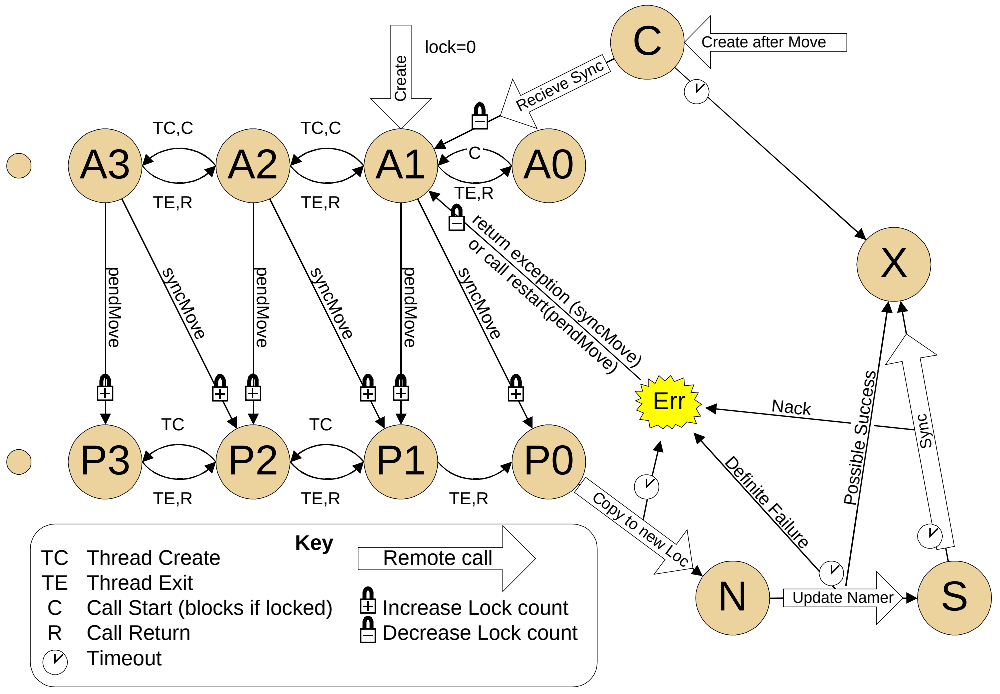


Figure 78 Thread States for a Mobile Cluster

The mobility API provides an interface to aid a cluster in shutting down all its threads. When a move is requested, the mobile cluster manager blocks all future incoming calls, to prevent new threads being created for that reason. It then monitors the number of threads within the cluster, and waits until all of these exit. The thread state transition diagram is shown in Figure 78. During normal operation, the cluster moves between states A_0 and A_n, which represent *active* states with *i* threads. Once a move has been requested, the cluster is transferred to a pending (P) state. In pending states, new incoming invocations are blocked, and threads may only be created explicitly. The cluster manager will wait until there are no active threads within the cluster before performing the move. It is the responsibility of the cluster to ensure that all threads terminate.

The other states in Figure 78 are related to the co-ordination of cluster migration. They ensure that the move is *fail safe*.

36.2.1 Initiating a move

In order to move, a thread within the cluster invokes a `pendMove` or `syncMove` call. Both of these request a move 'as soon as possible', the difference being whether the move is handled by the current thread (`syncMove`) or a newly created thread (`pendMove`). When a move call is invoked, the object enters a pending state. This obtains a lock preventing external invocations from entering the cluster.

The cluster will move between P states as threads are created and destroyed. When it enters state P_0 it will undergo a series of transitions that result in the creation of a new cluster in a different capsule. The original cluster will then be discarded (it enters state X). If an error occurs during this process and it can be safely inferred that the new cluster has not been created, then this cluster 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 cluster 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).

The cluster is started in state A_1 by a call to `restart`. Prior to this, the lock is released in order to unblock any pending invocations. If a cluster wishes to restart in a locked state, for example in order to allow it to initialise transient state, then it may obtain an addition lock prior to calling `pendMove` or `syncMove`.

36.3 Capturing Cluster State

The process of capturing a cluster’s state is in principle straightforward. A reference to the ‘root’ cluster object is passed by value. This leads to the serialisation of all of the cluster’s state, which will be transparently recreated on the destination machine. There are however a number of subtleties that must be considered.

- It is possible for a cluster to export an interface that is not reachable from the root object. To ensure that these parts of the cluster are also passed, a ‘super root’ object is constructed that contains a table of exported interfaces. This table is maintained by a specialised Cache binder, `ClusterCache`.
- Interfaces exported from the cluster must be served by the new cluster. The mobile namer will ensure that invocations are directed to the new cluster, and the `ClusterCache` must re-export each of the interfaces, when the cluster is recreated to ensure that the invocation reaches the correct object. This is done by granting the previously allocated names to the replica objects in the new cluster.
- Interfaces exported by the `ClusterManager` should not be re-exported, as the `ClusterManager` is not itself copied, and its role is taken by an equivalent abstraction in the destination cluster (which may enforce a different management policy). The `ClusterCache` maintains a list of such exceptions.
- The normal serialiser, `RefSerializer` distinguished whether to pass by reference or value based on the distinction between interface and reference types. For cluster migration, a better distinction is between parts of the cluster (objects) and references to external interfaces (proxies). A different serialiser, `StubSerializer`, is therefore used.

36.4 Copying a Cluster

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.

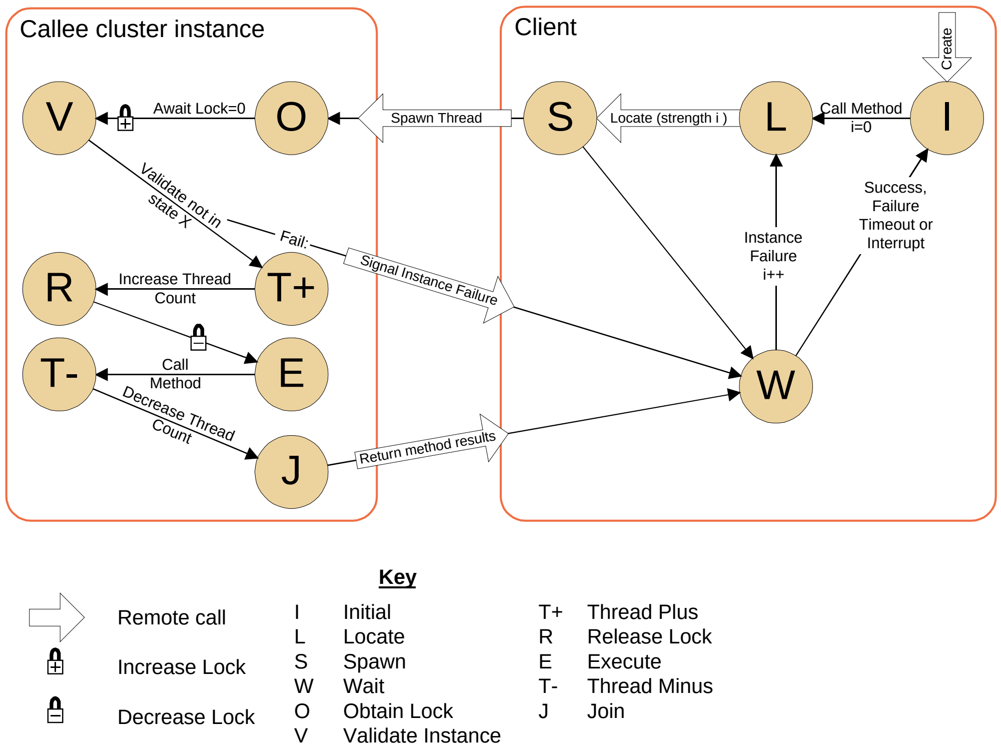


Figure 79 Method Invocation State Transition

36.5 Method Invocation

When a thread from outside a (mobile) cluster attempts to invoke a method exported by an object in inside a cluster, it must block if the callee cluster is in the process of moving. The state diagram in Figure 79 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. This is significant if the callee and caller represent components of mutually distrustful systems.

37 PERSISTENT OBJECTS

37.1 Introduction

This work was conducted as the 'Information Space' workpackage of the FollowMe project. The architectural goal of the Information Space is to extend the range of transparencies available through FlexiNet by providing persistence transparency (robust objects).

37.2 Incremental Approach

The approach taken was to produce an architecture for Information Spaces which could be implemented incrementally. Persistence transparency is the core requirement, with failure recovery, mobility and replication transparencies to be implemented later. Additional facilities, which may be required, include customised concurrency control, customised serialisation of objects and transactional capabilities.

37.2.1 Evolving a design

A simple approach to storing objects is to abstract a file system as an object store service. Clients can be given a remote interface reference to this service and send objects *by value* to the store, giving each object a name. The objects can later be retrieved by specifying the same name. An abstract interface to such a service is shown in Figure 80.

```
public interface CopyStore
{
  public void copyInto(String name, Object obj);
  public Object lookupCopy(String name);
  public void remove(String name);
  public String[] listCopyNames();
  public Object[] listCopies();
}
```

Figure 80 A Simple Interface to an Object Store

This has uses, but does not provide transparent persistence or help multiple clients to share the storable objects. A client has to copy the object out to examine or change it, then copy it back in. If multiple clients do this then some client's versions may be overwritten.

For a higher-level, more object-oriented approach, the stored objects should be treated as first class objects. They should be able to offer services to clients through exported interfaces, in exactly the same way that standard (non-persistent) objects do. A Java interface allowing such 'storables' to be created is shown in Figure 81.

```
public interface StorableStore
{
  public Iface newStorable(Class cls, Object[] args);
  public void delete(Iface storable)
}
```

Figure 81 A 'white box' Store Interface

The `newStorable` method of this interface creates an object that is transparently persistent. The interface to the storable returned to the caller can be used as a normal reference. However, behind the scenes, the infrastructure arranges that the effects of each invocation are stored before the result is returned to the client.

37.3 Storables and Clusters

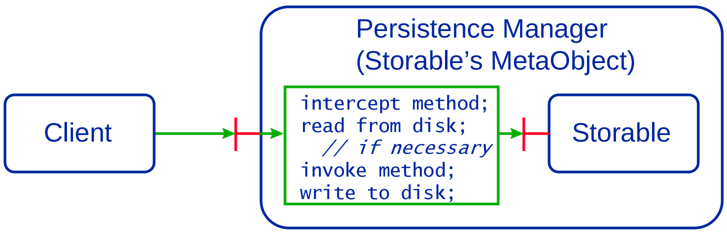


Figure 82. Encapsulated Storable

As clients must use the store's interface to manufacture storable objects, and clients are returned an interface reference to the new storable, a desirable isolation between clients and storables is achieved. Clients are not given object references to the storables, allowing the implementation of the storable's interface to be fully encapsulated. This is exactly the separation provided by the Cluster abstraction.

Conceptually, the store interposes a persistence manager between the client and the storable. The persistence manager reflects the storable's interface, interposing persistence behaviour before and after invocations. The persistence behaviour recovers the storable from disk if necessary, applies the invocation, saves the storable back to disk, then returns results to the client (Figure 82).

In practice, the Store's persistence manager is realised as a `ClusterManager`. This acts as a meta-object for the destination object.

37.4 Managing Persistence and Failure

When a storable is created, the client is returned an interface reference to the storable, which is sufficient to allow it to be used. However, if the client were to crash before persistently recording this interface reference (for example in another storable) then it would be impossible for a reference to the storable to ever be re-obtained (for example to delete it).

In addition, the administrator of the physical storage may wish to browse the storables and recover disk space. To support both these requirements, each storable is associating with a 'meaningful' name supplied by the creator. This allows recovery after failure, and allows client to supply names that are meaningful to a (human) administrator.

To aid with the management of storable names, a directory interface is introduced. Directories form a tree structure and each store is associated with a directory tree. The `copyIn` and `newStorable` methods are associated with the directory, rather than the store, allowing additional parameters to be passed to specify meaningful names for the created storable. A `newDirectory` method is also provide for creating sub-directories (Figure 83).

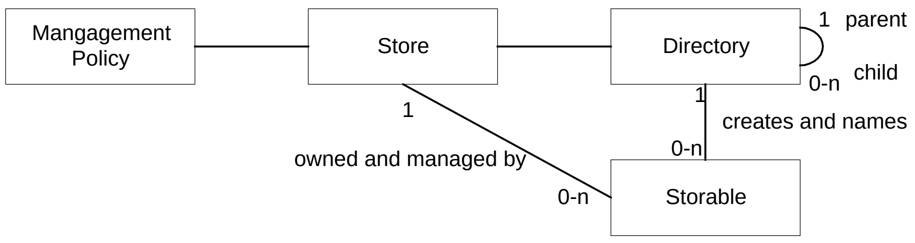


Figure 83 Class Diagram of Stores, Directories and Storable Objects.

37.5 Managing Stores

A store as described above is the owner and manager for all the storables in its directory tree. The Store is a logical grouping of storables, and different stores will be configured with different management policies for their storables. There may be a requirement for many stores to use the same physical storage area (disk).

A physical storage area, such as a physical file system or database may be shared by many applications, agent systems or agents. To allow applications to share storage areas, a management layer needs to be introduced (Figure 84).

```
public interface StoreManager
{
  public Store newStore(String name) throws ISException;
  public Iface newStore(String name, Class cls,
                        Object[] args)
                        throws InstantiationException,
                        ISException;
  public void remove(String name);

  public String[] listNames() throws ISException;
  public Iface[]  listStores() throws ISException;
  public Iface    lookup(String name) throws ISException;
}
```

Figure 84 StoreManager Interface

A `StoreManager` provides a name space of `Stores`, and is responsible for a region of physical storage. The physical storage may be a disk or a database (Figure 85). The `newStore(name)` method creates a new store with the default implementation and management policy. The `newStore(name, class, args)` method allows custom stores to be created with different behaviours.

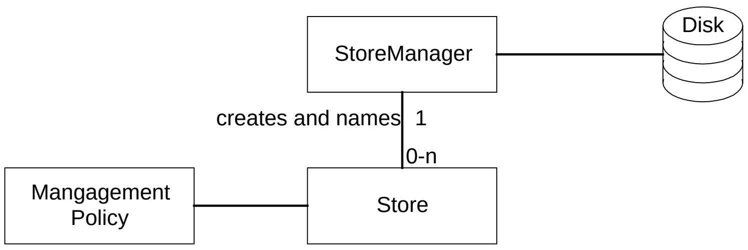


Figure 85 Class Diagram of StoreManagers and Stores

37.6 Interfaces

The key external interfaces to the Information Space are:

`StoreFactory`

This offers `newStore` and `remove` methods for creating and destroying stores. A store is a receptacle for storable objects, typically objects that are to be managed as a collection (for example objects belonging to one user).

`Store`

This encapsulates a set of storables. Storables are stored objects (or groups of objects). Each storable is associated with a directory entry, primarily for management purposes. A storable may be accessed via its directory entry or via an interface reference returned upon its creation. The `Store` interface has only one significant method, `getRootDirectory`. All stores and directories are reachable from this.

`Directory`

This offers methods for creating and destroying storables. Its methods can be grouped as follows:

- 'Black box' storable methods (`copyInto`, `lookupCopy`, etc.), for copying objects by value into and out of the directory. These methods are analogous to file operations.
- 'White box' storable methods (`newStorable`, `lookupStorable`, etc.), for creating empty storable clusters. A 'white box' storable is *transparently* persistent. That is to say that the creator is passed a reference to (an interface on) an object within the storable, and may the access it as it were a local Java object. This reference may be passed to other clients, or even stored within other storables. The fact that the object is both remote and persistent is make transparent.
- Directory methods (`newDirectory`, `lookupDirectory`, `getParent`, etc.), for creating and managing a hierarchy of directories.

`Storable`

This is the management interface of a storable, and includes operations such as `copy` and `destroy`.

Figure 86 shows the relations between these classes. See the extensive JavaDoc in the code for a full description of these interfaces.

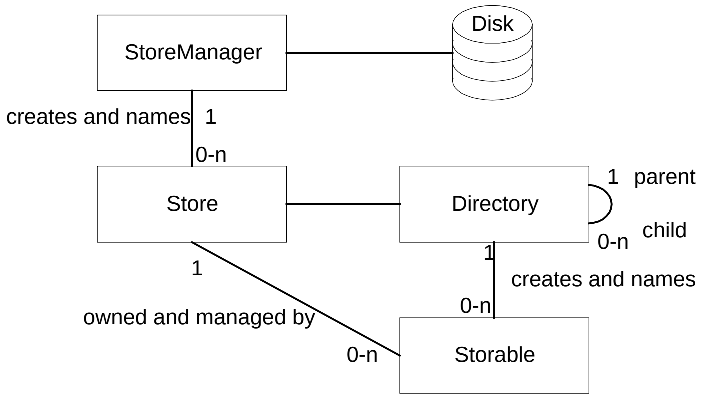


Figure 86 Class Diagram of Stores, Directories and Storables

37.7 Design

The design of the Information Space is in terms of clusters and capsules.

There are three levels of grouping. Each `StoreFactory` manages a number of `Stores`. Each `Store` manages a number of `Storables`, and each `Storable` contains a number of separate `Objects`. Working in reverse, the objects that constitute one storable must be kept separate from those in other storables, both for security reasons, and in order to identify a boundary for persistence. Storables therefore correspond to the FlexiNet (and ODP) notion of a cluster. Stores are consequently factories for storables, i.e. capsules in FlexiNet/ODP terms. The `StoreFactory`, as a factory for stores is an ODP Nucleus. In FlexiNet terms, this is just another capsule.

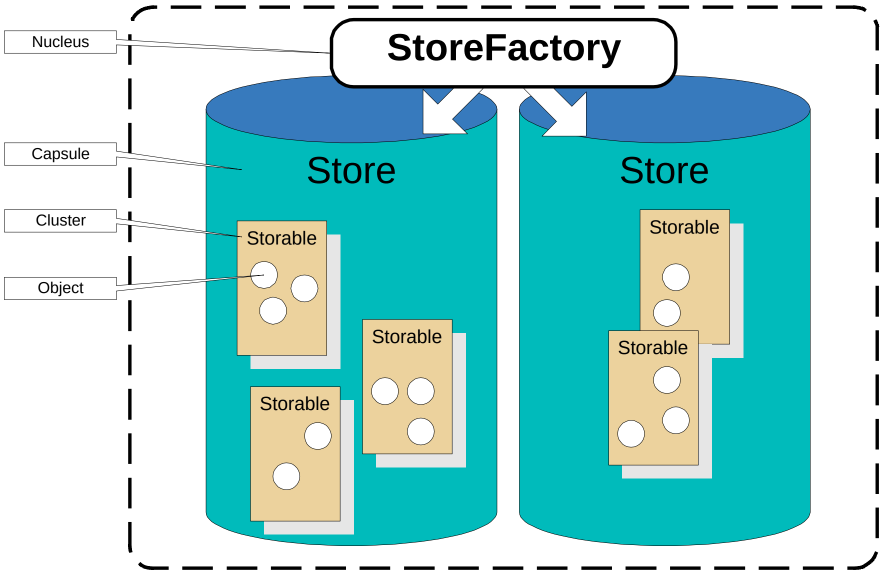


Figure 87 Stores and Storables

Each cluster is encapsulated so that it may be treated as a unit. In practical terms, this means that no objects are shared between clusters, and the cluster abstraction therefore defines a boundary that may be used when serialising a storable to disk.

At a computational level, clusters and capsules communicate with each other, and with remote clients via location transparent communication. In engineering terms, this is achieved by each Cluster and Capsule having a separate 'Cluster Comms' communications system managing its personal set of imported and exported references. It would be inefficient for each Cluster Comms to be implemented entirely separately, so to allow sharing, all the Cluster Comms within a capsule are multiplexed over a Capsule Comms, which manages low level communication, such as access to the network, and multiplexing. In an insecure implementation, different capsules may also share the same Capsule Comms, however in a secure implementation, where each store is 'owned' by a different client, it is more robust, and straightforward for each capsule to have a separate Capsule Comms. Capsules are therefore entirely separate from each other (and may even reside in different processes or on different physical machines). In addition to the Cluster Comms and Capsule Comms, the other related engineering component is the Cluster Manager. This is a management object with each cluster, which provides two management interfaces, a public interface for initialising a cluster and a private interface for use by the capsule. In the case of a storable, these interfaces are `Storable` and `StorableManager` respectively. Figure 88 gives a more detailed view of the internal relationship between a store and its Storables.

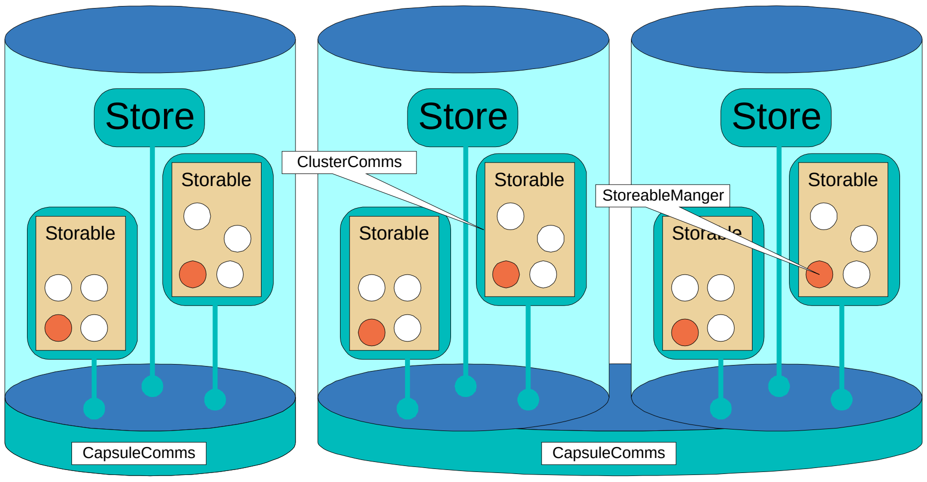


Figure 88 Implementation of the Store and Storables

37.7.1 Underlying Storage

The underlying storage is supplied by implementations of `DataDirectory`. The `DataDirectory` interface abstracts the provision of storage for byte arrays. A description of this is given in section 37.8.6.

37.8 Components

In this section, the primary components in the Information Space are described in turn.

37.8.1 StoreFactoryImpl

A `StoreFactoryImpl` creates and manages `StoreImpls`. The information needed to recreate the stores after a crash is held in a `MetaStoreFactory` object, saved as a black box storable in the store factory's directory. This directory is independent from the stores' directory hierarchies. The `MetaStoreFactory` object is saved under the name ***`storename`***`.msf`. The information needed for efficient lookup of stores created is held in a Hashtable of `StoreFactoryEntries`, hashed by store name.

The principle subtlety in the implementation of `StoreFactoryImpl` is the need to create new stores in their own capsule, and to preventing the sharing of any Java objects between the store factory capsule and the store capsules.

The implementation of `newStore` checks for the existence of a ***`storename`*** directory and ***`storename`***`.msf` object. If they exist, the store is recreated based on the previously saved parameters. (This is used for restart after failure.) If the store really doesn't exist already, then an appropriate root directory is created and `initStore` is called to initialise a new capsule for the store. `InitStore` 'wraps' the proto-capsule to isolate it from the rest of the system, and to turn it into a true capsule, thereby preventing the accidental sharing of objects between capsules. The wrap operation returns an interface to the capsule of type StoreManager. Once wrapped, `StoreManager.init` may be called to initialise the capsule.

Note that a similar wrapping process in undergone to wrap the `StoreFactoryImpl` itself, however this is slightly simplified as the `StoreFactoryImpl` is the only capsule (at that time) so does not need to be encapsulated from anything else.

37.8.2 StoreImpl

`StoreImpl` is the store capsule implementation. It performs several roles, which are separately defined in the interfaces it implements.

- As a `CapsuleManager`, its behaviour is inherited from `CapsuleManagerImp`.
- As a `StoreManager` it allows the store factory to initialise and destroy it.
- As a `Store`, external Clients can gain access to its directory hierarchy.
- As a `PartNameHandler`, it restores Storables that have not yet been loaded from disc.
- As a `PartNameHandler`, it restores Directories that have not yet been loaded from disc.
- As an `XStore`, it offers back door services to its Directories.

Analogously to a `StoreFactoryImpl`, it keeps information about the `Storables` and `Directories` within it. This is held in a `MetaStore` object, saved as a black box storable in the Store's `meta` directory. This is a directory just for this purpose, and is independent from the store's directory hierarchy. The `MetaStoreFactory` object is saved under the name *`storename`*`.mst`. (The `StoreFactoryImpl` actually initialises the `StoreImpl` with a `meta` directory that shares the same file system directory as itself, so the Store Factory's *`storename`*`.msf` meta information and the Store's *`storename`*`.mst` meta information appear in the same file system directory.)

The `MetaStore` object holds two mappings and their inverses, one from absolute directory name to directory interface Id, and one from absolute storable names to `ClusterAddress`. (See the interface definition of directory for a definition of absolute names in a directory hierarchy). These mappings allow `Storables` and `Directories` to be recreated at their old addresses after a crash, whether they are looked up by name in a directory, or referenced by a location and persistent transparent object reference.

The `MetaStore` object is saved whenever it changes, that is whenever a 'white box' storable or directory is created or destroyed.

As with `StoreFactoryImpl`, the principle subtlety in the implementation of `Store` is the need to create new `Storables` in their own clusters. The `StoreImpl` uses its Capsule Comms to ‘wrap’ each Storable and is returned an encapsulated `StorableManager` interface. This interface is used for most communications with the Storable. However, the `StoreImpl` needs to use the unwrapped `StorableManager` interface to bypass encapsulation when a Storable is being destroyed. Similarly, the `StorableManager` uses an unwrapped `XDirectory` interface to its directory for efficient access to the storage.

37.8.3 DirectoryImpl

`DirectoryImpl` relies on a `DataDirectory` object for creating sub-directories and storing storables. It also keeps a hashtable of its black box storables, white box storables and sub-directory members, indexed by name.

It implements `copyInto` by serialising the object onto a buffer, and then using the `DataDirectory` to store the buffer’s byte array. Note that to obtain a deserialiser, the directory needs to be given a reference to its Cluster Comms. This means that the store should not be initialised to construct its root directory until after it has been wrapped with Capsule and Cluster communications.

It delegates implementation of `newStorable`, `restoreStorable` and `removeStorable` to its Store, through the Store’s `XStore` interface. `StoreImpl` implements `newStorable` by creating and wrapping a new `StorableManagerImpl`.

After creating, restoring or removing a sub-directory, `DirectoryImpl` informs the store through the store’s `XStore` interface. This allows the `StoreImpl` to keep its meta information up to date.

`DirectoryImpl` prepends a byte to each `Storable` in its `DataDirectory`, signifying whether the Storable is a black box or a white box object. This means that the directory does not need to keep an extra meta information object in its `DataDirectory` to signify this. The directory can efficiently read the first byte of each Storable after a crash and work out which are black box and which are white box objects.

Locking of directory members is fairly complicated. Locking is needed in case two clients try and change a directory simultaneously, or try to access a white box storable simultaneously. The locking needs to be at a fine granularity so that independent entries in a directory can be accessed simultaneously.

A directory creates a `Lock` for each of its members. The lock for a white box storable is shared with the storable’s StorableManager. This allows the StorableManager to lock the storable from when method activity starts until after the changed Storable has been stored. The storable cannot be deleted in this interval. The lock for a sub-directory is shared with that directory. This allows a parent to lock its child, remove all its members and then remove it.

When a store is destroyed, or a directory is recursively removed, the lock acquisition recurses down the directory hierarchy. The lock for a deleted object can be killed (moved to a dead state). Any waiting activity then fails to acquire the lock, leading to an exception propagating back to the client.

37.8.4 StorableManagerImpl

`StorableManagerImpl` performs several roles, which are separately defined in the interfaces it implements:

- As a `ClusterManager`, its behaviour is inherited from `ClusterManagerImp` and augmented to implement persistence transparency.
- As a `StorableManager` it allows stores to initialise and destroy it.
- As a `Storable` (an extension of `Cluster`), external clients can create objects and copy them to new directories.

The persistence transparency is implemented in `endCall`, called by the Cluster Comms after a call to the storable has finished. `EndCall` waits for thread activity in the cluster to finish, and then stores the cluster state. This is obtained by a call to `ClusterManagerImp.getContents`, and consists of the root object itself (possibly null), the table of exported interfaces, and the distinguished interface on the root object returned after the object was initialised. This is precisely the information needed to restore the storable after a crash. The `StoreImpl` keeps the storable's cluster address in its MetaStore object, so the table of exported interfaces only needs to map the interfaces to their Ids. The work of restoring all the interfaces at their old identifiers is done by `ClusterManagerImp.setContents`.

The `ClusterState` is serialised into a buffer using a 'by value' serialiser obtained from the Cluster Comms.

`StartCall` differs from `ClusterManagerImp's startCall` because it locks the Storable, serialising calls to the cluster. `EndCall` then waits for thread activity to finish, meaning that the Storable is in a stable state to be stored.

The buffer used for the cluster's state is segmented, with a one-byte segment for the `DirectoryImpl's` flag, and the rest for the Storable. The `StorableManagerImpl` passes the buffer to the directory using its `XDirectory` interface, and the directory fills in the flag byte and saves the buffer's byte array to its `DataDirectory`.

37.8.5 Implementation of Directory and StoreManager

For flexible implementation of Directory and Store, a layer of abstraction needs to be introduced between them and the storage area. `DataDirectory` forms such a layer (Figure 89)

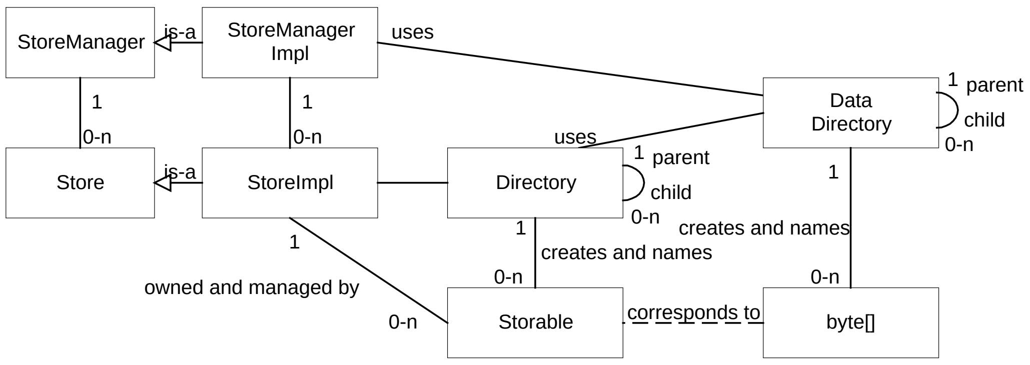


Figure 89 Class Diagram for DataDirectory

37.8.6 Implementation of DataDirectory

The current implementation of `DataDirectory` is `FSDirectory` (File System Directory), which uses a file system for its implementation (Figure 90). An alternative implementation might be use a database.

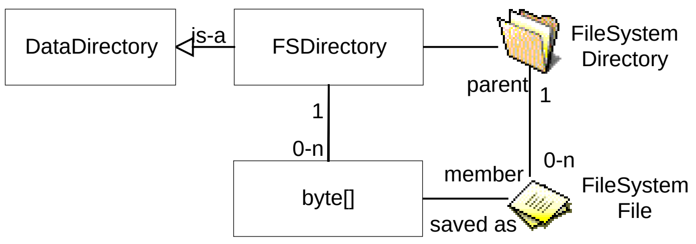


Figure 90 Class Diagram for FSDirectory

37.8.7 Implementation of FSDirectory

`FSDirectory` needs to make named sub-directories, and save byte arrays as named files. Two issues are atomic write of data, and file naming.

A naive implementation of 'write' might leave an inconsistent state of a crash occurred part way through overwriting a previous storable representation with a new one: the copyIn operation of DataDirectory must be atomic.

This is achieved by writing the data to a new file name, then renaming the new to the actual file name. The rename must be done in two steps (first renaming the old to an old file), because not all file systems support atomic rename over an existing file (NTFS does not). The state diagram for copyIn is shown in Figure 91.

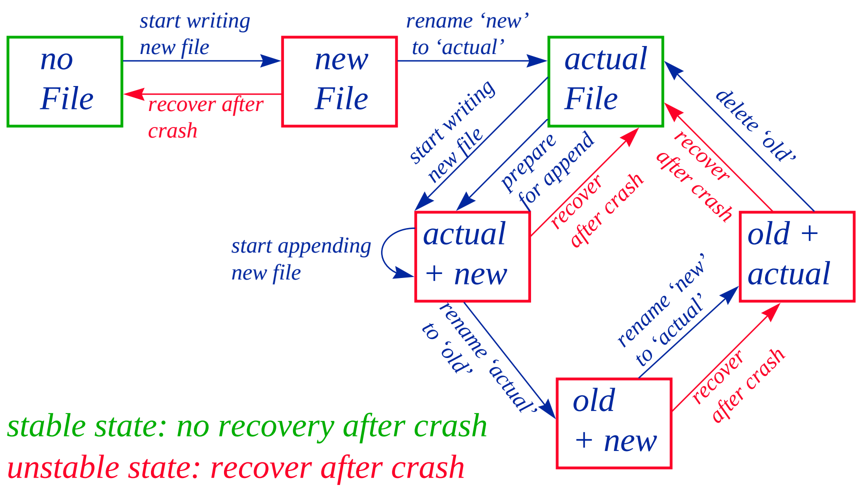


Figure 91 State Diagram for Atomic Write

The names supplied `FSDirectory` to it are arbitrary Java strings. They may not be valid file names. A `StringMapper` is used to map Java strings to file names. The current implementation simple converts each Java character to a two-byte hexadecimal code.

38 TRANSACTIONS

38.1 The Runtime Execution Model

Figure 92 illustrates the runtime model of the transaction framework.

- An enterprise bean instance is an object whose class was provided by the enterprise bean developer.
- An EJB object is an object whose class was generated at deployment time by the EnterpriseBeanBox. The EJB object class implements the enterprise bean's remote interface. A client never references an enterprise bean instance directly; instead, a client always references an EJB object whose implementation is provided by the container.
- An EJB home object provides the life cycle operations (create, remove, find) for its EJB objects. The class for the EJB home object was generated by EnterpriseBeanBox at deployment time. The home object implements the enterprise bean's home interface that was defined by the EJB provider.

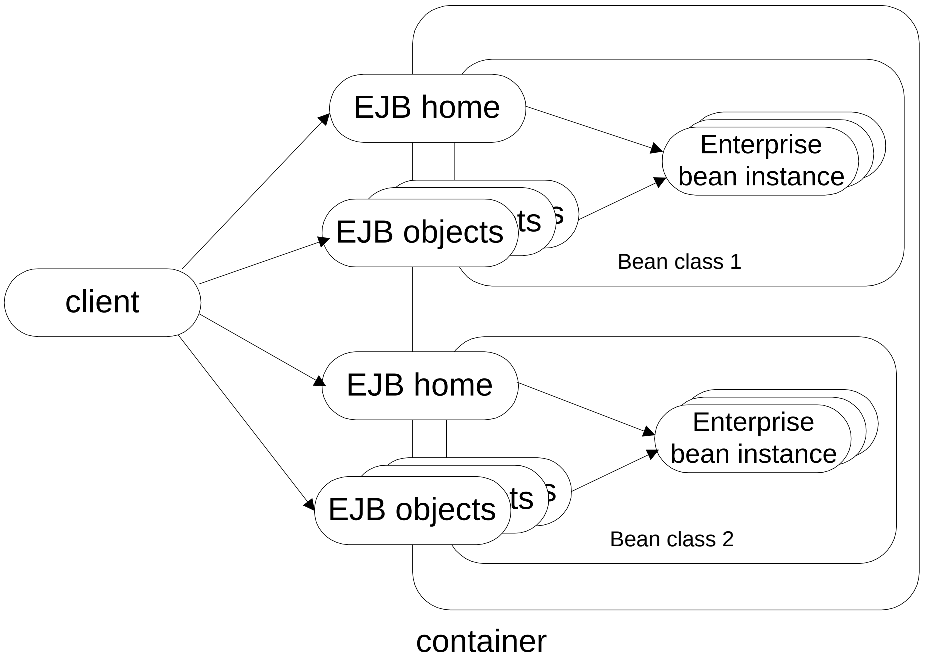


Figure 92 The EJB Container Model

38.2 Transaction Model

Because the transaction architecture provides implicit transactions, an enterprise bean developer or client programmer is not exposed to the complexity of distributed transactions. The burden of managing transactions is shifted to the container and the underlying transaction system. The container implements the declarative transaction scopes. It also, together with the underlying transaction system, implements necessary low-level transaction protocols, such as the two-phase commit protocol and transaction context propagation.

38.3 Transaction Context Management

The scope of a transaction is defined by a transaction context that is shared by the participating objects. A transaction context records all the information related to the transaction, namely the name and identifier of the transaction, the participating objects, and the status of the transaction. Each client may only exist in one context at a time, but EJB objects may be involved in many. (This is an extension to the standard EJB model). The transaction service is available in every context. (Figure 93).

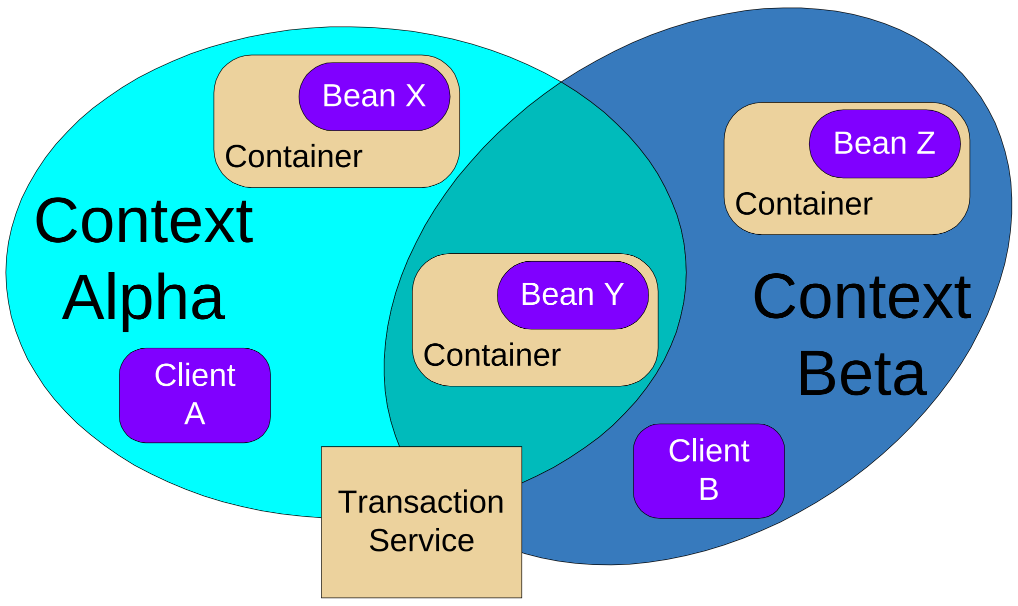


Figure 93 Transactional Contexts

When a container processes a client request on an enterprise bean, it must be told the sender's transactional context. The container may need to access and update the transaction context, for example to check the status of the transaction and/or to register a resource to the transaction. In the local case, a transaction context is associated with each thread. Therefore, when invoking an operation remotely the related transaction context must be read from the client thread, and propagated to the peer thread on the server (Figure 94).

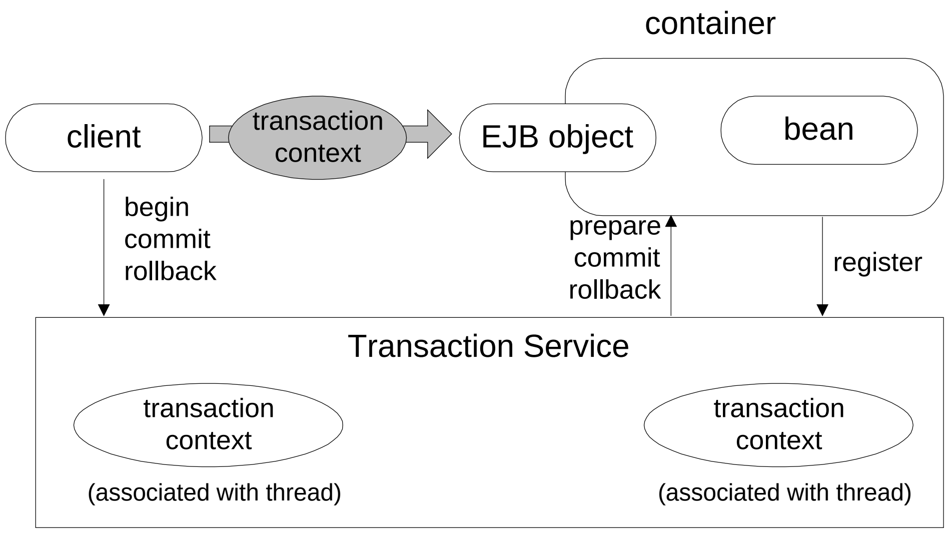


Figure 94 Contexts via Threads

To provide transparency, it would be ideal if the transaction context could be propagated implicitly. For example, we could make the underlying remote method invocation mechanism pass a transaction context to the server side automatically when it deals with a remote method invocation. This requires support from the underlying remote object platform. Because of the flexibility of FlexiNet, we can customise the basic FlexiNet protocol to provide this capability.

38.4 Implementation of Transaction Context Propagation

In our implementation, when a transaction is created by a client thread we create a transaction context object and associate it with that thread. We may use smart proxies to pass copy this information from client to server thread. As the transaction platform is generic, we must use *generic proxies* to achieve this.

We arrange that the client uses a `TransactionalProxy` whenever accessing a transactional object. When a method is invoked on the client stub, this is passed to the `TransactionalProxy`, which reads the transaction context related to the calling thread and pushes this into the stack of 'additional arguments'. This is illustrated in Figure 95.

```
public class  TransactionProxy extends SimpleGenericProxy
{
  // for TransactionProxyGenerator
  public TransactionProxy(Name n)
  { super(n); }

  // for serialization
  public TransactionProxy()
  {}

  public void invoke(Invocation i) throws BadCallException
  {
    // determine the current transaction context
    Coordinator coordinator =
             Thread2Transaction.get(Thread.currentThread());

    // save this on the stack of extra aguments
    i.push(Coordinator.class, coordinator);

    // continue the invocation
    super.invoke(i);
  }
}
```

Figure 95 The Transactional Proxy

On the server side, we arrange that a `TransactionalSkeleton` is invoked in place of the real target object. This reads the transaction context from the stack, and assigns this to the current thread. The invocation is then invoked on the object ‘for real’. This is illustrated in Figure 96.

The use of proxies and skeletons in this case is extremely straightforward, and exactly follows the procedure outlined in section 17.4.

```
public class  TransactionSkeleton implements GenericCall
{
  private Object target;

  public TransactionSkeleton(Object target)
  {
    this.target = target;
  }

  public void invoke(Invocation i) throws BadCallException
  {
    // initially, assume no context
    Thread2Transaction.put(Thread.currentThread(), null);

    if(!i.stackEmpty()) // in a transactional context
    {
     try
     {
      // read context from stack of extra arguments
      Coordinator coordinator =
                  (Coordinator) i.pop(Coordinator.class);

      Thread2Transaction.put(Thread.currentThread(),
                             coordinator);
      }
      catch (Exception e)
      { // ignore exceptions (context left as null)
      }
    }
    // invoke the method
    i.invoke(target);
  }
}
```

Figure 96 The Transactional Skeleton

38.5 Summary

The transactional architecture was developed in parallel with, and on top of, the core FlexiNet architecture. It takes advantage of the same concepts of reflection and component technologies to provide a flexible, scalable and adaptive system.

Like most component models, it enables users to develop portable, customisable components, and assemble them into applications. It enables rapid application development and deployment using standard components and off-the-shelf tools. Moreover, unique to our architecture, by supporting reflection it allows a server component container to be easily customised, for example, in order to cater for new application demands, or to adapt to a new environment. We also allow application developers to provide application-specific information, declaratively and separately from application code. This allows the transactional container to improve system performance.

PART SEVEN: SERVICES

39 CLASS REPOSITORIES

39.1 Background

When the components of a distributed application communicate with each other using remote method invocation, then they may pass objects *by value*. The object-oriented paradigm allows for object subclassing, and so a service expecting an object of class `Foo`, may in fact, be passed an object of type `FooSub`. This facility is essential when building the *generic* services. For example, a `Place` is a service with a `receive` method that takes as an argument an object of a (sub class) of `MobileCluster`.

Whenever an object is created, the JVM must first load the necessary code to implement the object's class. This class must either be available on the local class path, or be accessible by some other means. Without special mechanisms, a service is only able to load classes available locally. This severely restricts the interoperability between applications and services, and adds an additional degree of complexity to application deployment and upgrade.

In newer versions of Sun's RMI, there is some provision for remote class loading in order to tackle this issue. However, there is one significant limitation. If, by chance, two client of a service both make use of the same class name to refer to different classes, then the class loading system is unable to distinguish between them, and unexpected behaviour will result. Although apparently an obscure and insignificant limitation. This is likely to arise in two different circumstances.

- If programmers are undisciplined or unlucky in the choice of class names (for example, if they build applications in the default package) then they are likely to choose the same names for different classes. Names such as `App1`, `Test` and `HelloWorld` are obvious candidates.
- If programmers rely on particular versions of a class that has been upgraded or modified. By chance, a service might load an early version of a class, as requested by one client, and cannot then load a later version of a class, as requested by a second client. In practice, the service will not even be able to identify which version of a class it requires: there must be at most one version of each class available to it.

The FlexiNet class repository has been designed to manage classes so that (remote) services may load classes as required. In particular, it does not suffer from the limitations of the RMI scheme. Its key features are

- Classes are loaded from a 'nearby' repository one at a time, reducing the overhead of loading entire jars from distant servers.
- Application programs do not need to act as class repositories themselves, so there is low per-process overhead. This is particularly significant for weakly connected machines (e.g. portables).
- Repositories are federated in a scalable manner. Each repository only serves local clients (e.g. same sub-net); remote clients are served via traditional web-server needs.
- Repositories act as caches for remote jars. This reduces wide-area traffic, and provides a single point for management of foreign classes (for example signature checking).
- Applications can load and use any classes that fit within the typing and semantics of the application. The application is statically typed, and there can be no runtime type errors due to network class loading – even if class names are poorly chosen.
- There is zero impact on application code. Applications need not be aware of the class loading system, and need not be written differently because of it.
- Classes on the local class path will be used in preference to network classes, if available. This reduces the load on the class repository, and taken to an extreme, allows applications to run without a class repository being available.

39.2 Class Loading Overview

In this section we describe a number of case studies, and use these to illustrate the key requirements of a networked class repository.

39.2.1 Base Case – No Repository

When distributed components communicate via remote method invocation, then may use pass objects of any class available on both machines. Figure 97 shows two machines, each with a single Java process, and a number of Jars of class files available via the class path. The left-hand process is using classes from the black and white jars. The right hand one is using classes from the white and grey jars. The processes may communicate using remote method invocation as expected. However, if during this communication, one machine tries to send the other a class from a jar that is not available on the remote machine; then the communications will fail.

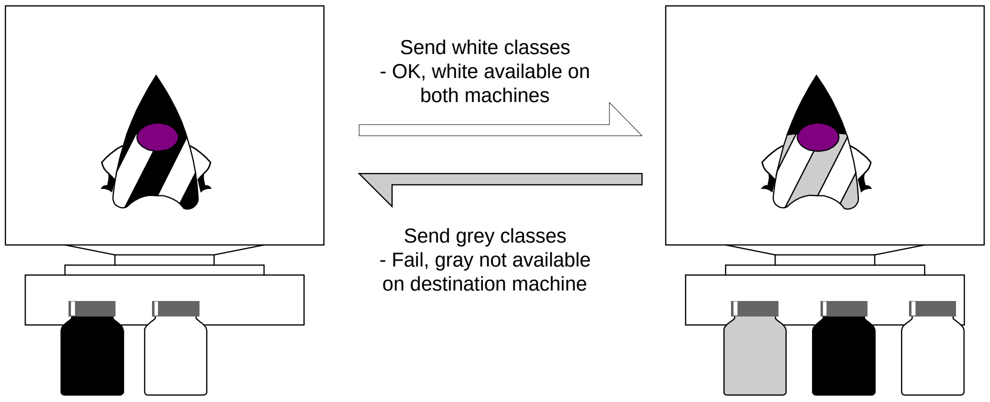

Send white classes
- OK, white available on both machines
Send grey classes
- Fail, gray not available on destination machine

Figure 97 Communication Without a Repository

39.2.2 Communication Backed by a Class Repository

Figure 98 shows the same two hosts communicating as before. However in this example, the classes are stored in Jars on a remote repository that is available to both machines. The two processes may communicate freely, and classes will be retrieved from the repository as and when needed.

An alternative solution would be to place all jars on all machines. This approach does not scale in general, but might be appropriate in special circumstances, for example if the machines had a shared filing system. A specialist class loading system would still be required in order to overcome issues relating to Jar versions.

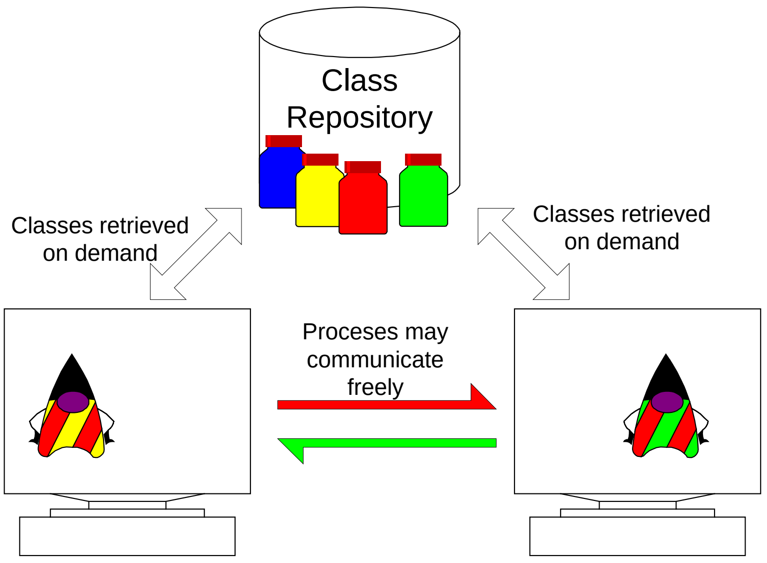

Class Repository
Classes retrieved on demand
Classes retrieved on demand
Proceses may communicate freely

Figure 98 Communication Backed by a Repository

39.2.3 Wide Area Communication

If there are many sites or organisations that wish to communicate, it is unrealistic for them to all share the same class repository. A single, global, repository would be a single point of failure, and would be prone to overload. In addition, for management reasons it is likely that each organisation would prefer to maintain its own class repository. For example, a repository might contain software that was proprietary or licensed.

Although they might not share class repositories, processes in different sites may still need to communicate and share classes. This is an analogous problem to the communication between hosts without a class repository. This is illustrated in Figure 99.

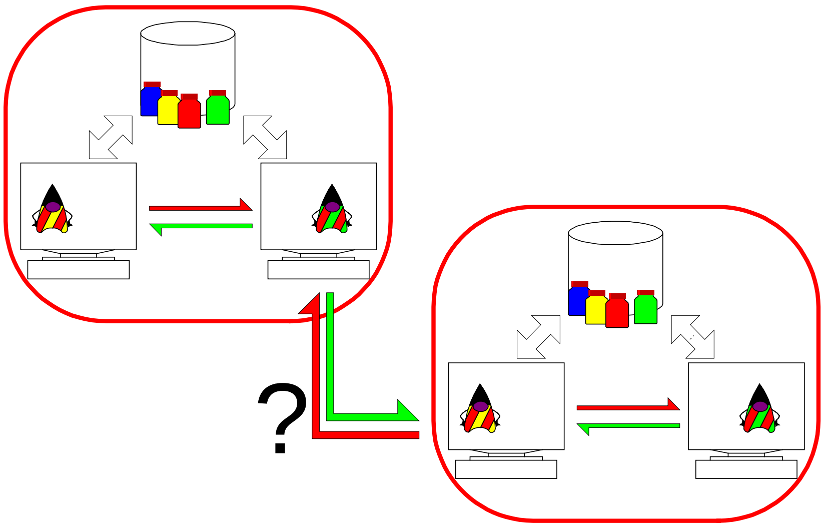


Figure 99 Communication Between Sites

39.2.4 Federation Using Web Servers

Web servers provide the ideal mechanism for publishing classes for use in other sites. They are a mature technology, and are already used for providing a gateway to information stored a sites protected by firewalls. In addition, the auditing and access control mechanisms they provide are exactly what is required when publishing Java classes. Figure 100 illustrates how web servers are used by the class repository architecture.

If a class is requested from a repository, and that repository does not contain a copy of the class; then the repository contacts the class's web server. To support this, every class identifier sent in a remote method invocation contains details about the web server containing the original class. The class repository may then download *the entire jar* containing the requested class. It then validates the jar and stores it for later use. Certificates are checked and site policies relating to the use of imported classes will be consulted. (Certificate checks are not currently implemented).

Downloading an entire jar is the appropriate granularity to reduce the load on the web server, and to maximise throughout over the Internet. Once the jar has been downloaded it may be treated like any local jar in the repository, and classes may be served from it. A class repository may later delete the jar to recover disc space, as it can always be retrieved from the web server later. In this sense, the class repository is acting as a web cache for Java Jars.

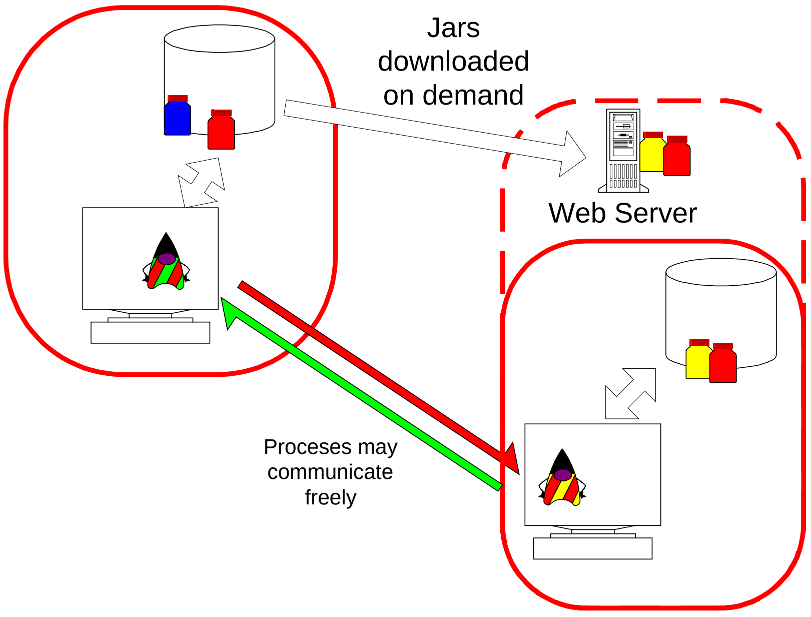


Figure 100 Federation Using Web Servers

39.3 Architecture

The class repository architecture has two main components. The *repository* itself is shared between a number of client processes. Each client process contains *class manager* which co-ordinates its use of the repository. Java class files are collected into *bundles* which are stored in the repository. Each class manager maintains a *proxy bundle* for each bundle in use. Additional proxies are created as and when needed.

During normal operation, classes are loaded from the appropriate bundles by implicit calls made by the JVM on behalf of the client application. When the client communicates via FlexiNet, then a special *class serialiser* is used to serialise information about each required class. In the destination machine, a *class deserializer* can convert this information into real classes, by explicitly calling the class manager. This in turn will contact its local repository as required. If a class is requested from a repository that it does not possess, then it contacts the appropriate web server, and downloads the Jar containing the class using `http`. The key components and interfaces are illustrated in Figure 101.

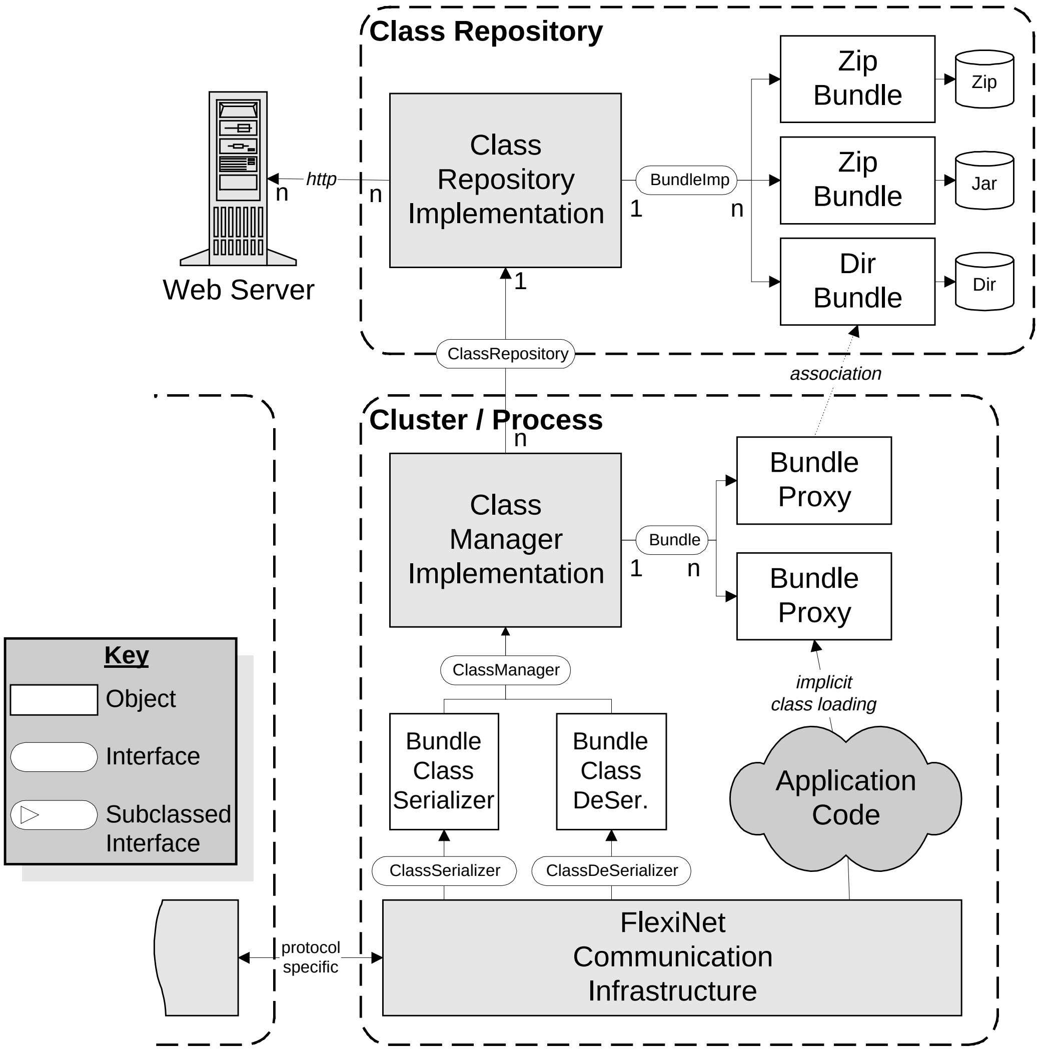


Figure 101 Class Repository Architecture

39.3.1 Bundles

Jars stored in the class repository are abstracted by *Bundles*. This allows other collections to be stored (for example directory based collections) and gives additional design freedom. Each bundle has an identifier, which is globally unique. For `jar` and `zip` files, this identifier is a secure hash of the bundle's contents. This ensures that the identifier is unique, prevents reuse of names due to human error, and is used to aid the secure download of bundles over insecure communications.

Each bundle also contains a list of other bundles that it imports. Bundle imports are analogous to Java import statements. Each bundle lists all of the bundles that contain classes required by the classes within it – in effect the set of 'library' bundles that are needed. The intention is that reusable code is placed in separate bundles from application specific code, so that a class repository may cache small bundles that contain frequently used classes. This approach is similar to JDK 1.2 'Jar extensions', and the format of the import statements within the Jar manifest is compatible with it. The format is extended to include the identity (secure hash) of imported bundles in addition to their URLs. This allows secure retrieval of imported classes.

It is envisaged that a later implementation might allow more complex import statements that allow a runtime choice to be made. For example an import statement that requested "AWT version 2.0 or above" would allow a class repository more choice when servicing a request. There is more research needed on the format and impact of such import statements.

A bundle may have other meta-data associated with it. Currently this, together with import statements, is stored in the manifest file (`META-INF/MANIFEST.MF`) in a format compatible with `.jar` manifests. The current implementation of the class repository ignores all manifest sections other than those specifically designed for it.

39.3.2 Class Managers

Each FlexiNet process (or cluster) has a local class manager object associated with it. This provides a context for class loading. It is used by the serialisation and deserialisation code to read and write class identifiers, and to obtain classes from the class repository. The class manager is not itself a class loader, instead it managers a number of bundle proxies, each of which is a class loader. Each bundle proxy loads classes related to a particular bundle, and uses other bundle proxies to load classes imported from other bundles.

Although one class loader per bundle may seem excessive, it is necessary, as two bundles may both contain different classes of the same name. In cluster based applications, each cluster has its own class manager, to allow different clusters to obtain different instances of classes – this is required for strong encapsulation, in order to prevent sharing of static data. In addition, to reduce the overhead of different class instances, a single shared class manager is used to service requests for bundles that *may* be safely shared.

These must be explicitly marked as sharable (within the manifest), as sharing is a potential security risk unless the classes are 'cluster aware'.

39.3.3 Local Classes

The class repository scheme does not (and cannot) completely replace the system class loader. For example the code required to access the class repository must itself be loaded somehow. Typically, the JDK and FlexiNet will both be loaded from the system class path. As the class path will vary from machine to machine, it would be inadvisable to treat local classes as a special case, and instead each locally loaded class should be made available in a bundle on the class repository. Bundles may be marked as 'shadowable' meaning that (some of) their contents may be available locally. For shadowable bundles, the local class path is searched first, before the class repository is contacted. Shadowed bundles must be sharable (as all clusters can access the system class loader). A secondary limitation is that it must be possible to determine which bundle a locally loaded class really belongs to. This involves (at worst) a search of all shadowable bundles in the class repository. Shadowable bundles must therefore contain classes with unambiguous class names – or it would not be possible to work out which bundle contained the appropriate class.

An example snapshot of a JVM is shown in Figure 102. Here there are two clusters, A and B each of which has its own Class Manager. The clusters are both sharing classes managed by the shared class manager. In the example, cluster A has access to classes A,B,C from bundle `JDK`, classes D,E,F from bundle `FlexiNet` classes G and H from bundle `Lib` and classes I and J from bundle `AppA`. Cluster B has access to A-F and K.

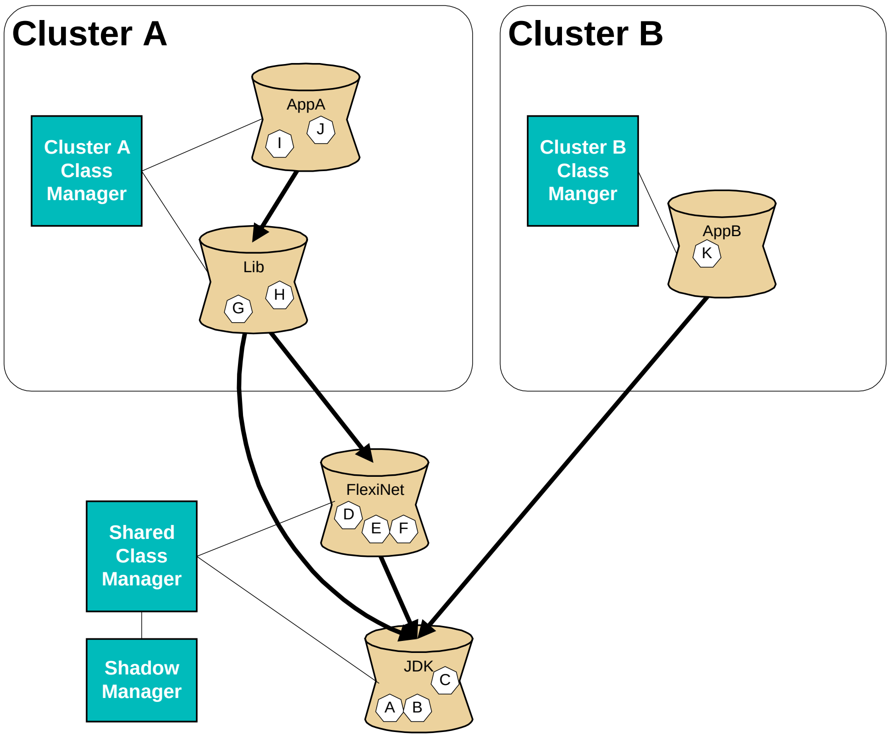


Figure 102 An Example use of Bundles Within a JVM

If bundle `AppA` or `Lib` also contained a class called K, then this would be loaded on demand, and would not affect the execution of Cluster B. Similarly, if Cluster A was passed a reference to a *different* class called 'H' then this would necessarily live in a different bundle. A's cluster manager would create a bundle proxy, and the class would be loaded correctly. The fact that A would be using two different classes with the same name seems counter-intuitive, but is perfectly correct.

Each bundle in the example contains a reference to other bundles that it imports. For example, the definition of class G might contain references to classes E and B.

39.4 Engineering Details

39.4.1 Class Serialisation

The following information must be passed to uniquely identify a class

- The identity of the bundle containing the class. This is a secure has of the bundle's contents (MD5).
- An identifier for the class within the bundle. The fully qualified class name is used for this. In an early implementation, classes within a bundle were sorted, and an integer identifier was used. This approach was rejected because of the additional complexity and overhead caused by mapping shadowed classes.

In order to ensure that the bundle can be *located* by the destination host's class repository it is also necessary to pass location information. For this, the URL of the bundle is passed. It up to (human) management to ensure that the bundle remains available at this URL during the its lifetime.

To minimise the amount of information written when serialising a particular object graph we note the following possibilities for optimisation.

- A graph will typically consist of many objects of classes in the same package
- A large graph will typically consists of many objects of classes in different packages, but within the same bundle
- A large graph will typically consist of many objects of a given class.

To take advantage of these observations, the class serialiser uses three dictionaries, one for package names, one for bundles and one for classes. The dictionaries are initially empty, and the following algorithm is used.

- Lookup the class in the class dictionary. If it is present, write the index and exit, if not add to the end of the dictionary and write the class name (unqualified).

- Lookup the bundle in the bundle dictionary. If it is present, write the index, if not add to the end of the dictionary and write the bundle identity and URL.
- Lookup the package name in the dictionary. If it is present, write the index, if not add to the end of the dictionary and write the package name.

The Deserializer created an analogous set of dictionaries, and uses these to decompress class information as it deserialises it.

39.4.2 FlexiNet without Class Repositories

FlexiNet can (and by default is) used without a class repository. During remote method invocation, the bundle each class belong to is set to unknown. All classes are loaded from the local class path, and ambiguous class naming is not allowed (i.e. the class path should not contain two classes of the same name). It is envisaged that this arrangement will be used for application *development*, and class repositories will only be used at *deployment* time. It is possible for two processes to communicate if one uses a class repository and the other does not, but in general, this is to be avoided.

39.4.3 Class.forName

The use of class repositories is, for the most part, transparent to the application programmer. There is however, an issue with unrestricted use of `Class.forName`. This will attempt to load a class from one of two locations: the class loader used to load the *calling* class, and the system class loader. In applications that make use of many class loaders (for example if many bundles are used) then it might be necessary to load named classes from other locations. This issue will occur in any system making use of multiple class loaders, and is not FlexiNet specific. The following code fragment is an enhanced version of `Class.forName` that will allow classes to be loaded from the same location as any existing class or object. Note that in JDK 1.2, an equivalent method is provided in class `Class`.

```
static Class Class_forName(Object context,String name)
                            throws ClassNotFoundException
{
  Class c;
  if(context instanceof Class)
    c = (Class) context;
  else
    c = context.getClass();

  ClassLoader l = c.getClassLoader();

  if(l==null)
    return class.forName(name);
  else
    return l.loadClass(name);
}
```

39.4.4 ClassLoader Resources

In JDK 1.1, it is possible for an application to store resources such as images or samples together with class files. These may then be accessed via

```
   URL ClassLoader.getSystemResource(…)
```
and
```
   URL ClassLoader.getResource(…)
```

The class repository implements this interface by acting as an `http` server for these resources (as resources are loaded via URLs). This implementation choice is less efficient than a FlexiNet based protocol, however it was unavoidable as `getSystemResource` and `getResource` return URLs. It should be noted that as `http` is used to communicate with the (local) class repository when loading resources, this may be less secure that the protocol used to load classes.

39.5 Bundle Integrity

The scheme for identifying bundles been designed to ensure that the wrong bundle cannot be accidentally used. This covers misnamed bundles, corrupted bundles, or those that have been maliciously modified (either on the original web-server, or during communications). The integrity scheme relies on the use of the bundle's secure hash code as an identifier for it. It is illustrated in Figure 103.

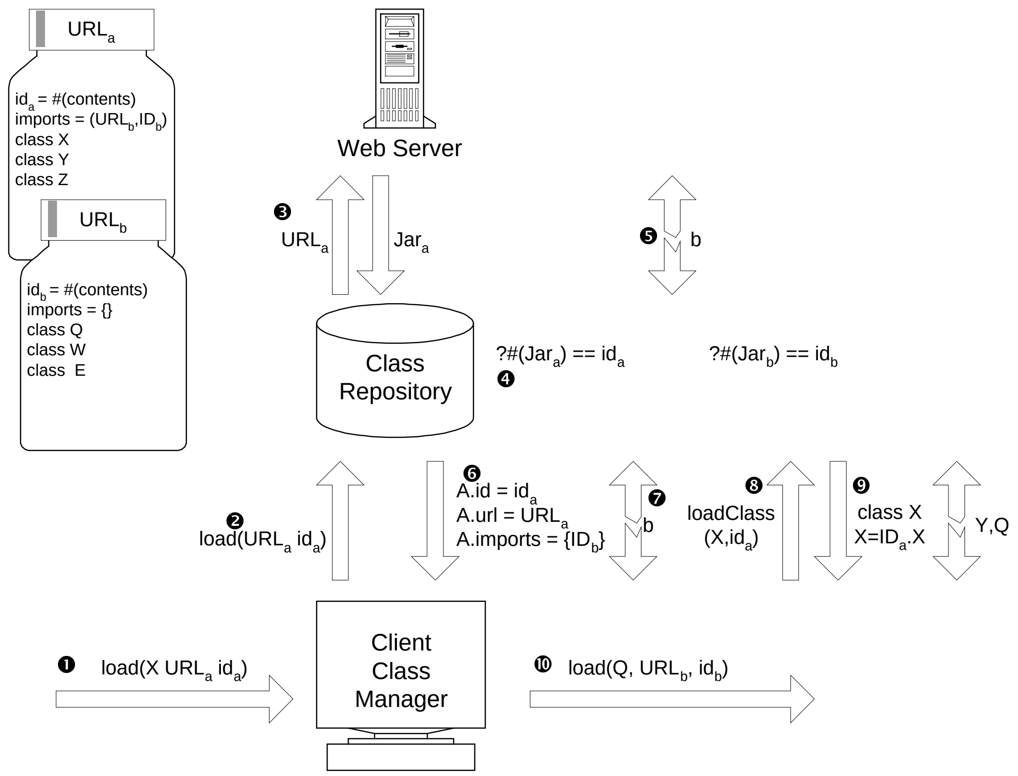
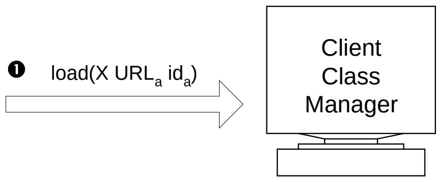


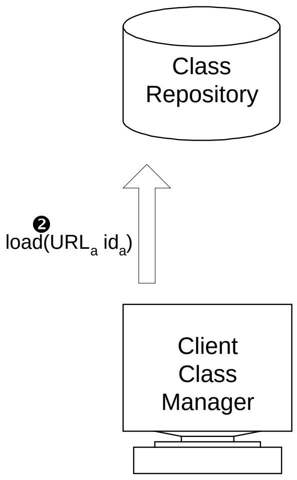
Figure 103 Secure Class Loading

Under normal use, the loading of classes proceeds as illustrated in Figure 103. We will walk through this in stages.

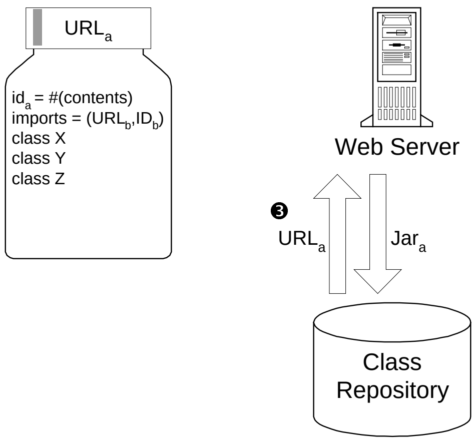
1) First, a client class manager is requested to load a class. Three parameters are supplied, the name of the class (`X`), the URL of the bundle containing the class (`URL`$_a$) and the unique identifier for the bundle (`id`$_a$)



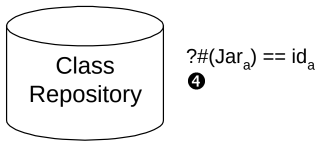
2) As the class is not already loaded, the Class Manager first attempts to load a record representing the bundle from the client's local class repository. We will assume that all communication with the class repository is authenticated. Usually the class repository and client are within a firewall, removing the need for further mechanisms, but in general, an arbitrary technique can be used to insure the authenticity of information transferred across this link. (Note. This is a standard Flexi-Net remote method invocation, so may be secured by using a secure FlexiNet protocol).



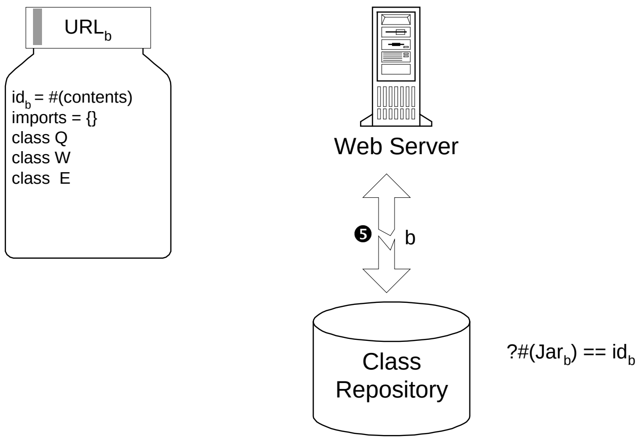
3) We will assume the class repository does not already have the class (if it does, skip to stage 6). The class repository downloads the bundle from the specified URL using (insecure) `http`.



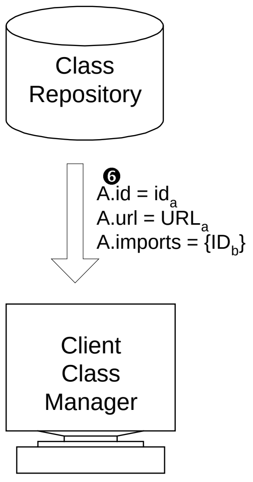
4) Once downloaded, a secure hash of the bundle is created, and checked against the identifier supplied by the client. This ensures that the bundle was not tampered with during download.



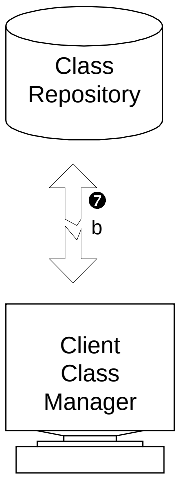
5) The class repository examines the bundle and downloads and verifies the transitive closure of all imported bundles in a similar manner. (Note. The repository may have already downloaded some of these when servicing other requests).



6) The class repository supplies the original client with a meta-data record for the bundle. This includes the identity of all imported bundles, but not their URLs (this will be obtained later).

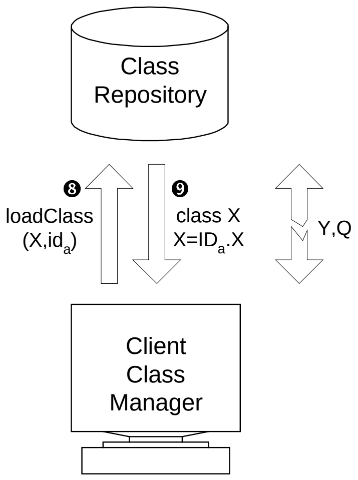


7) The client obtains meta-data records for the transitive closure of the imported bundles (again, some may have already been loaded for other reasons). Note that the client may load these using only their identifiers – the URLs are not required as the Class Repository has already loaded the bundles.

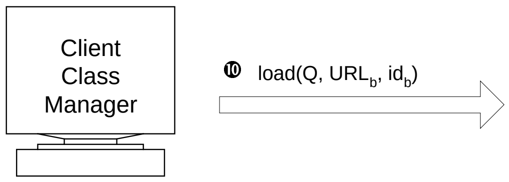


8) The client is now able to load classes, one at a time as required, from the class repository.

9) For each class loaded, the repository informs the client of the actual bundle it came from (it may be a bundle imported from the requested bundle). This ensures that the client can manage classes and share them when appropriate. It also ensure that the client has the `(BundleID, URL, ClassName)` tuple for each loaded class.



10) If the client requests a subsequent client to load a class, it can supply all the necessary information.



39.6 Discussion

It is not possible for an aggressor to replace a real class or bundle with an alternative, without first compromising the client or class repository. It is not sufficient to subvert the web server or the web server to repository communication.

We assume that the original request to the client to load a class is authentic and the client decided to agree to the request. In practice, the client may ask for additional meta-information from the repository, such as certificates relating to the bundle's authorship. In this case, it is the repository, not the client, which performs cryptographic checks on supplied certificates, but the client interprets these certificates, and has the option not to load a class. This removes the burden of cryptographic processing from the client, which is important if the client has limited resources. It also allows organisation wide policy to be enforced at the Class Repository – for example the repository may disallow the use of certain classes.

In general, the class repository is shared between many clients, and caching in the client and repository will reduce the amount of communication.

39.7 Special Cases

In some special circumstances, the class repository may be used to securely load classes, even if the secure identity of the containing bundle is not known. This is particularly useful when bootstrapping from a command line, or when bridging to less secure technology.

39.7.1 Special Case 1

The system remains secure if the initial 'loadClass' request to the client does not contain a bundle ID if the following assumptions hold

- The Class Repository has previously loaded and validated the Jar at the named URL, and the contents of the URL have not changed

or

- The Class Repository can securely load the Jar at the named URL without the need to validate it.

These circumstances are likely to arise if the client is requested to load a 'local' application which exists on an intra-net web server, or that has been permanently installed on the class repository (rather than just cached). It is a useful special case, as it allows simple command line use of the class repository (e.g. `Java CRHarness File://MyProg.jar main`).

The class repository can be configured to allow this special case, but to disallow use of a URL without ID when this would be insecure.

39.7.2 Special Case 2

If a standard Jar is loaded using this system, then it will contain the URLs of imported Jar's but not their identities. This is because the JDK1.2 'Jar extensions' makes no provision for securing imported jars. We employ a special tool to 'seal' a jar by adding security information about imported jars to the jar's Meta directory. This is backward compatible with standard jars.

If the system is used with such 'insecure' jars, then it will remain secure if the following assumptions hold

- Jars loaded in this way have no imports that have not already been loaded into the class repository, and the jars at the import URLs have not changed since they were loaded into the repository

or

- The class repository can securely load the Jar at the named URL without the need to validate it.

The class repository can be configured to reject insecure jars for which these assumptions do not hold.

Note. The Class Repository implementation is complete except that it does not read bundle identities from Manifest import statements. In addition, the seal tool has not yet been written.

40 TRIVIAL TRADER

40.1 Introduction

The Trivial Trader is a name server used by FlexiNet applications and services to locate each other. Services may *publish* an interface by entering a record into the trader giving a textual name for the service, and a reference to the exported interface implementing the service. Clients may then query the trader and obtain references to services.

It is important to note that the trader is *not* an integral part of FlexiNet. Services may publish interfaces by other means, such as by passing them to a third party service, or even by outputting a stringified form of a FlexiNet name to the screen (although the latter case is to be avoided). The trader is, however, a commonly used service. In particular, the static class `FlexiNet` may be used to obtain a reference to the trader. This is obtained by parsing a Java system property, which is supplied as a command line argument or by other means.

40.2 Naïve Design

The current trader implementation is extremely trivial. It maintains a simple dictionary of (service name, interface) pairs. It is not persistent, not does it perform access control. In particular any client may add, delete or overwrite entries within the trader. For these reasons, and the fact that complex queries are not supported, the trader is called the *trivial* trader. For any significant use of FlexiNet, this would have to be replaced with a more robust and comprehensive service.

There are two subtleties with the trader design.

- The trader is able to restart at a previous (well known) address. It does this by explicitly initialising the `FNetTest` test bench with a stringified FlexiNet name.
- The trader actually stores byte arrays rather than interfaces. This is to overcome various security and scaling issues described in the following section. This subtlety is hidden by the use of a smart proxy to access the trader.

40.3 Scaling Issues

The Trader is a general purpose service, and is expected to be used to store a large number of interface from a variety of different services. If implemented naïvely this would lead to the trade having to load and resolver a large number of different interface classes. This would have the following unwanted side effects.

- The trader would have to generate stubs for a large number of interfaces, despite the fact that it would never actually use these stubs.
- The trader would have to load a large number of interfaces classes, and a large number of (object) classes imported by these interface classes. This would lead to bloat, and security issues if the classes were not trusted.
- The trader would have to load and resolve a number of names that represented Smart Proxies (if a stored interface exploited Smart Proxies)
- The trader would have to support a large number of protocols, one for each protocol used by a name stored in the trader, even if communication with the trader itself was always made using the same protocol.

These issues can (and are) all avoided by using a Smart Proxy to access the trader. This serialises and deserialises interface references into a byte array. The trader itself therefore only has to store untyped data. As the clients of the trader who wish to use an interface already have to load and resolve the interface and related stubs and protocols, this introduces no significant overhead.

40.4 Trader Proxy

The trader proxy is a smart proxy that performs serialisation and deserialisation. To do this, it must obtain a reference to 'Binder Top', so that names can be generated/resolved as required. This is obtained from the *binding context* stored in the `SmartChoice` binder, during resolution. An overview of the proxy is given in Figure 104. Note that clients of the trader do not need to be aware that they are using a proxy.

```
class  TraderProxy extends SmartProxy implements FNetTrader
{
  public Trader trader;
  ...

  // called during resolution of the SmartProxy
  protected void init(BinderDB ctxt)
  {
    binderTop = ctxt.getBinderTop();
    ...
  }

  public void put(String name,Object obj,Class cls)
  {
    // Create a byte array & serializer
    ByteArrayOutputStream ba = new ByteArrayOutputStream();
    Serializer s = getSerializer(new DataOutputStream(ba));
    // write the interface to the byte array
    s.writeObject(FlexiNet.tag(obj,cls),Iface.class);
    // put the byte array into the trader
    trader.put(name, ba.toByteArray());
  }

  public Object get(String name)
  {
    // get the byte array from the trader
    byte[] data = trader.get(name);
    // create a deserializer
    ByteArrayInputStream s = new ByteArrayInputStream(data);
    DeSerializer d =getDeSerializer(new DataInputStream(s));
    // read from the byte array
    return (Proxy) d.readObject(Iface.class);
  }
}
```

Figure 104 A Simplified Trader Proxy

40.5 Discussion

The trader is both trivial and complex. As a trader, it is extremely simple, and does not have the feature set that might be expected of a full trader. However, as a distributed service, it has been designed to scale. The use of a proxy allows this scaling without recourse to 'special techniques'. The trader is a standard FlexiNet program, and as such, can easily be replaced by an application specific trader.

PART EIGHT: API AND EXAMPLES

41 BASIC API

41.1 Introduction

The computational API for FlexiNet is extremely small. This is as should be, as the main purpose of the FlexiNet framework is to provide *transparency*. Once a programmer has a reference to a remote service, they may use it, more or less, as if it were a local reference. The FlexiNet computational API is therefore only concerned with the small number of aspects of the reference that cannot be made completely transparent to the programmer.

For a particular FlexiNet component, protocol or binder, there may be additional APIs to control configurable aspects of that component. For example, defining security policy for secure bindings; creating and managing clusters in a cluster based system or setting binder specific QoS. These aspects have been discussed in the relevant chapters.

In addition to the 'programmer API', the binders and protocols available to a particular process may be configured using the 'Testbench' interface. This is described in chapter 42.

41.2 FlexiNet Initialisation

The trader (currently the trivial trader) is used as a bootstrapping mechanism for FlexiNet applications. A program may trivially obtain a reference to the trader, and may then use this reference to discover references to other services, or to publish its own service. This is the *only* FlexiNet-specific call in the majority of application code.

A reference to a trader may be obtained using the following call

```
FNetTrader trader = FlexiNet.getTrader();
```

Note. In a cluster environment, static methods should not be used from within clusters, and this call should only be made from the initial (nucleus) capsule. This may then pass the reference into other clusters if required. If a cluster security manager it used, this should be enforced.

The interface to the trader is given in Figure 105.

```
public interface FNetTrader
{
  // Add an interface to the trader.
  // name - A textual name for the interface
  // obj  - The object which implements the interface
  // cls  - The class of the interface
  public void put(String name,Object obj,Class cls);

  // Add the first interface implemented by an object
  // to the trader. (first as defined by Java spec.)
  public void put(String name,Object obj);

  // Lookup an interface name and return the interface
  public Object get(String name);

  // Remove an interface from the trader.
  public void delete(String name);
}
```

Figure 105 The Trader Interface

41.3 Configuring a Binding

For most applications, bindings are created implicitly and used 'as is'. The binders themselves and the binding graph are pre-configured to use the appropriate protocols and QoS settings, and application code is not polluted with this information. This pre-configuration may be done using the Testbench interface.

It is sometimes necessary for a service to *explicitly* specify the protocol or configuration to be used for a particular binding on a per-client or per-object basis. This is required for protocol-sensitive services if the correct protocol cannot be trivially determined by a binder (for example by examining the service object).

Equally, a client may post-configure a binding by changing the QoS settings relating to it. However, a client is more limited, as they may not (normally) affect the server-side of the binding. Typical client configurations might be to change the authentication information implicitly passed in a secure binding or to change the buffering policy on the client side of a binding. If the name resolved to generate the binding consisted of a set of choices, (for example for replica servers, or different protocols) then the client might change which choice was used. Such 'tinkering' is highly protocol dependant and should be used with care.

41.3.1 Use of Proxies for Server Side Binding Configuration

A server may create a 'tagged proxy' to a local interface, indicating desired QoS properties. By passing a reference to this proxy, rather than the local interface, then the server can arrange that a particular binder, or binding configuration is used, when a client binds to the endpoint. This is analogous to the way in which a server passes a smart proxy to specify application-level binding configuration (see section 17.3). It provides the same functionality as explicit binding interfaces found in other middleware offerings. An example is given in Figure 106.

```
Foo f = new FooImpl();

// create a tagged reference to f that indicates a
// specific protocol should be used when a client
// binds to f.

FlexiProps qos = new FlexiProps();
qos.setProperty(“protocol”,”Rainbow”);

Foo rainbowF  = (Foo) FlexiNet.tag(f,Foo.class,qos)

// put the reference in the trader

FlexiNet.getTrader().put(“MyService”,rainbowF);
```

Figure 106 An Example Use of Explicit Binding Configuration

Note. The format of the QoS properties, and their meaning is entirely binder dependent. At present, no standard format has been adopted for QoS specification.

41.3.2 Use of QoS Control Interface for Client-side Binding Configuration

A client that has a proxy for a (remote) interface may use the `QoSControl` interface (if supported) to configure the binding that the proxy represents. The degree of configuration will depend on the particular binder/protocol used by the proxy. The `QoSControl` interface has two operations; modifying the QoS on an existing proxy; and creating a new proxy with different QoS. This interface is shown in Figure 107.

```
public interface QoSControl
{
  // set the QoS of the object
  // return false on failure
  public boolean setQos(FlexiProps qos);

  // create a clone of the object with differnet
  // QoS. Return null on failure
  public Object qosClone(FlexiProps qos);
}
```

Figure 107 The QoSControl Interface

An example use of the `QoSControl` interface is shown in Figure 108. This is part of the cluster implementation, and creates a new client-side binding to the same server endpoint (interface), but that uses a different serialisation policy when serialising parameters to method invocations.

```
public Object getByValueReference(Object originalRef,
                                  Class ifaceClass)
{
  // check QoSControl is supported
  if(!originalRef instanceof QoSControl)
     return null;

  // create desired QoS properties
  FlexiProps qos = new FlexiProps();
  qos.setProperty("serialize.interfacesAsObjects",
                  new Boolean(true));

  // create a qos-clone. (Will return null if the
  // QoS cannot be met/is not understood).
  return ((QoSControl) originalRef).qosClone(qos);
}
```

Figure 108 An Example Use of the QosControl Interface

41.4 Passing Interface References

Normally, a reference to an interface may be passed using FlexiNet simply by referring to it by its interface class (as described in section 9.4). However, for security reasons, a client receiving an interface passed in this way will be unable to *widen* the interface – i.e. cast it to a subclass. This is because a malicious client might use this mechanism to gain access to other interfaces on the same service object. If this client behaviour *is* to be allowed, then the service must explicitly indicate this, by tagging the returned interface with its intended class. The client may then freely cast the returned interface to this class, or any of its superclasses. This concept was described in detail in section 26.11. It is particularly relevant when a client or service wishes to pass a reference generically (i.e. as an instance of class `Object`). To distinguish between generically passed objects and generically passed interfaces, the class `Iface` is introduced. This may be used in the definition of a class/interface to indicate that particular field or parameter must be of interface type (and must be tagged with its interface type). The `Iface` class is provided purely to allow additional type-safety. Its use is entirely optional.

An interface may be tagged to allow widening in the same was as it is tagged for QoS control. An example is shown in Figure 109. The `Iface` class is shown in Figure 110. Although it is not actually a superclass of all interfaces, it may be used as if it were. An example is shown in Figure 111.

```
Foo f = new FooImpl();

// tag to allow widening
Foo tf = (Foo) FlexiNet.tag(f,Foo.class)

remoteHash.put(“myfoo”,tf);

…
// same or different client

Object o = remoteHash.get(“myfoo”);

Foo f = (Foo) o; // no special action on widening
```

Figure 109 Tagging to Allow Widening

```
public interface Iface
{
  // return the concrete class of this interface
  public Class getIfaceClass();
}
```

Figure 110 The Iface Class

```
// a trader-like interface
public interface NamingService
{
  public void register(String name,Iface service);

  public Iface lookup(String name);
}


// A client of this interface
Foo f = new FooImpl();
namer.register(“myfoo”,FlexiNet.tag(f,Foo.class));


// a second client of the interface
Foo f = (Foo) namer.lookup(“myfoo”);
```

Figure 111 An Interface Making use of ‘Iface’

41.5 Local Reflection

As FlexiNet is heavily based on reflection, an API is provided to allow a programmer to use FlexiNet style reflection on local objects, even if FlexiNet is not used for remote invocation. Details are given in section 17.5.

41.6 The Full FlexiNet programmer API.

A summary of the API described in this chapter is shown in Figure 112.

```
public class FlexiNet
{
  // obtain a reference to the trader
  public static FNetTrader getTrader();

  // tag an interface to allow widening by a client
  public static Iface tag(Object obj,Class cls);

  // tag an interface to allow widening, and specify QoS
  // constraings on bindings created using this tag.
  public static Iface tag(Object obj,Class cls,
                          FlexiProps qos);

  // reflect a local object
  public static Iface reflect(GenericCall metaObject,
                              Class interfaceClass)

  // reflect a local object, and store QoS information
  // for use by the metaObject
  public static Iface reflect(GenericCall metaObject,
                              Class interfaceClass,
                              FlexiProps qos)

  // unexport an interface that was previously exported
  // Subsiqnently remote use of this interface will fail
  public static void unexport(Object obj,Class cls);
}
```

Figure 112 The FlexiNet Computational API

41.7 Running FlexiNet applications

FlexiNet applications are Java applications that make use of FlexiNet. Various parts of the FlexiNet infrastructure will use Java properties (environment variables) to determine configuration information. In general, a user must only ensure that the property `flexinet.trader` is defined, and contains a stringified reference to the correct trivial trader. This is most easily set on the command line. For example:

```
java –Dflexinet.trader=rrp:(192.5.254.101:1234)(0)
     MyApplication arg1 arg 2 arg3
```

The trader itself will read this property and attempt to run at this address. If the property is unset, the trader will choose a random address, and output this to the screen. Typically, a user will run the trader once at a random address, and then use this address in future. Note that the address is host specific, so a trader cannot be started on a different machine, at the old address. Example applications using FlexiNet are described in chapter 43.

42 THE FLEXINET TESTBENCH

42.1 Introduction

The testbench architecture has been designed to allow the binding graph used by a program to be changed. An application may explicitly request that a particular configuration is used, and this may be overridden by a command line argument.

A FlexiNet testbench is a static class that defines and configures a binder graph. The static class `FNetTest` is used to choose and initialise a particular testbench. This is then used by the application, via calls to `FNetTest` or `FlexiNet.getTrader()`.

In addition, an application may dynamically augment the binder graph using `FNetTest`. This adds a new generator to the binder represented by `BinderTop` using a call to `addGenerator`. `FNetTest` also provides other test and debugging related access to the binder graph. The application programmer interface to `FNetTest` is shown in Figure 113.

42.2 Format of a Testbench

A testbench is a class containing a method with the following signature:

```
public static Binder init()
```

It may also contain a second init method which takes a single bootstrap string as an argument.

When called, the `init` method should create a binder graph and return a reference to the top of the graph. The `init` method will only be called once.

Typically, a testbench is created to test use of each new binder. This testbench creates a binder graph consisting of (only) a generator and resolver for the new protocol, and sufficient other binders to allow the protocol to be tested. This usually consists of a `Cache` and `SmartChoice` binder. Other testbenches test combinations of binders, or binders configured with particular QoS parameters.

```
public class FNetTest
{
  // initialize FlexiNet, optionally specifying a test
  // harness to be used if there is no command line
  // argument; and a bootstrap string for that harness.
  public static boolean init(Class defaultBinderFactory)
  public static boolean init(Class defaultBinderFactory,
                             String bootstrap)
  public static boolean init(String bootstrap)
  public static void init()

  // aliases for methods in class FlexiNet
  public static void dropNames(Object obj,Class cls)
  public static FNetTrader getTrader()

  // generate a stringified name for an interface
  // used for debugging, or if there is no trader available
   public static String name(Object obj, Class iface)

  // resolve a stringified name. Used for debugging,
  // or if there is no trader available
  public static Object resolve(String name,Class iface)

  // return a reference to BinderTop, for debugging
  public static Binder getBinderTop()

  // add a generator to binderTop
  public static boolean addGenerator(Generator g)

  // Add knowledge of a new class loader.
  // This is only required if the TestHarness specified by
  // a command-line argument must be loaded from by a
  // different class loader that FNetTest itself.
  public static void addCodeBase(ClassLoader loader)
  public static void addCodeBase(Object obj)
}
```

Figure 113 FNetTest API

The testbench abstraction is designed to aid debugging of new protocols. It is envisaged that an alternative abstraction will be created to support deployed FlexiNet applications.

42.3 Existing Testbenches

The following testbenches are included in the FlexiNet distribution. These fall into two categories; those that support a single protocol, and those that support a number of protocols

Single Protocol Testbenches

`MagentaTest`
: This is the default testbench. It provides a binder graph that supports the RRP protocol ("rrp"), by using the `Magenta` binder. This is configured to use the Class Repository (if available) for serialising references to classes.

`GreenTest`
: This testbench supports the REX protocol ("rex") using the `Green` binder. It uses the default setting for green.

`YellowTest`
: This supports the REX protocol over TCP ("tcprex") using the `Yellow` binder with default settings.

`CrimsonTest`
: This supports RRP with SSL ("rrpSSL") using the `Crimson` binder. `Crimson` also supports RRP without SSL ("rrp").

`BurgundyTest`
: This supports RRP ("rrp") using the Burgundy binder. `Burgundy` is a variant of `Magenta` that uses blueprints for configuration. `Bugundy` uses the class repository by default, so `BurgundyTest` contains no additional binder configuration.

`NegotiatorTest`
: This supports the negotiation protocol ("neg") using the `Negotiation` binder.

`IIOPonlyTest`
: This supports IIOP ("iiop") using the `IIOPBinder`. It uses the 'basic' IDLMapper.

`IIOPobjByValueTest`
: This supports IIOP ("iiop-v") using the `IIOPBinder`. It uses the object-by-value IDL mapper.

Multi-Protocol Testbenches

`MagentaGreenTest`
: This supports RRP without class repository support ("rrp-lite") for the generation and resolution of names, and will resolve interfaces exported using REX ("rex"). RRPlite is supported

through use of the `Magenta` binder. REX is supported via the `Green` resolver (a resolve only variant of `Green`).

`BlackTest`
: This supports the RMP group messaging protocol ("rmp"). As this protocols is only useful for group-based communication, RRP ("rrp") is also supported for normal operation. RMP must be specified explicitly as a QoS parameter.

`IIOPandGreenTest`
: This supports IIOP using the basic IDLMapper ("iiop") via the `IIOPBinder`, and will resolve "rex" interfaces using the `Green` resolver.

`GreenAndIIOPTest`
: This testbench supports a "rex" binder and resolves "iiop" interfaces. It uses `Green` and `IIOPBinder`. For interfaces resolved using IIOP, interface references passed across the interface have IIOP names generated for them. A graph containing two Caches is used, as explained in section 14.3.1.

42.4 An example Testbench

The `GreenAndIIOPTest` testbench is shown in Figure 114. The binder graph it generates is shown in Figure 115.

```
public class GreenAndIIOPTest
{
  public static Binder init()
  {
    // create components
    Cache topCache = new Cache();
    Cache iiopCache = new Cache();
    SmartChoice choice = new SmartChoice(topCache);
    Green green = new Green();
    BinderIIOP iiop = BinderIIOP.create(topCache,topCache,
                            iiopCache,0,
                            IDLmapperBasic.mapper,null);
    // build a graph
    topCache.init( green, choice );
    iiopCache.init( iiop );
    green.init( topCache );
    choice.addResolver( green );
    choice.addResolver( iiopCache );
    green.addGenerator( iiopCache);
    // return the root of the graph
    return topCache;
  }
}
```

Figure 114 An Example Testbench

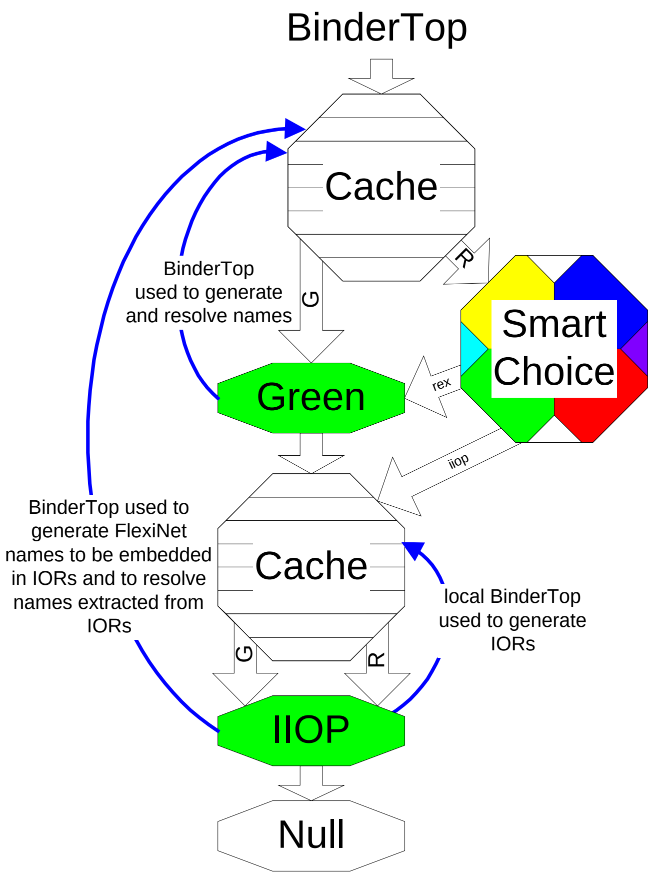

BinderTop
Cache
R
Smart Choice
BinderTop used to generate and resolve names
G
Green
rex
iiop
BinderTop used to generate FlexiNet names to be embedded in IORs and to resolve names extracted from IORs
Cache
local BinderTop used to generate IORs
G
R
IIOP
Null

Figure 115 The Binder Graph Constructed by the Example Testbench

42.5 Specifying a Testbench

An application can be made to use a none-default testbench in one of two ways.

- The application can contain an explicit call to `FNetTest.init`. This will be used instead of the default testbench, but is overridden by the second mechanism.
- The Java property “`flexinet.binderinit`” can be set to the name of a testbench class. This is most easily set by a command line augment, for example:

```
java -Dflexinet.binderinit=UK.co.ansa.flexinet.
 protocols.rex.binders.green.GreenTest MyApp arg1 arg2
```

42.6 Cluster Testbenches

Clusters use cluster binder graphs rather than ‘standard’ binder graphs. The testbench function is subsumed by a `CapsuleComms`. The default `CapsuleComms` class used is `JustMangenta` which supports the RRP protocol with class repository support ("rrp") and is equivalent to the non-cluster `MagentaTest` testbench. This is done using the `BlueClusterBinder` and `MagentaCapsuleBinder` components.

The only other cluster testbench in the current distribution is `BlueGreen`. This supports a variant of REX for cluster naming ("rex-c") using the `BlueClusterBinder` and `BlueCapsuleBinder` and standard REX ("rex") for resolving names. This is done using a `GreenClusterResolver` and a `GreenCapsuleResolver`.

`CapsuleComms` classes are more complex than the corresponding testbenches, as they must support other functions, such as the creation of clusters. However it is not difficult to create additional `CapsuleComms` implementations by using the existing implementations as templates.

The `CapsuleComms` uses by an application may be chosen by calling `Nucleus.init` or setting the Java property `flexinet.capsulecommsinit`. The format and semantics of these calls is the same as the standard testbench initialisation.

43 EXAMPLE PROGRAMS

43.1 Introduction

A number of test and example programs are provided with the FlexiNet distribution. These serve the dual roles of illustration of existing components and test suite for new components. Virtually all of the test programs can be configured to use a different binder or protocol by supplying a test bench parameter on the command line (see section 42.5).

Detailed instructions on how to run each test make be found in a ReadMe.txt file within the appropriate directory.

The examples are arranged into a number of sub-directories:

`simple`	This contains simple examples that illustrate/test limited features of FlexiNet. They may be used for testing features such as simple invocation, message size, reference passing etc.
`trader`	These examples all make use of the FlexiNet interface to obtain a reference to the Trader, which must be running. This is then used to pass a reference from server to client. These examples are trivial applications, and form a basis upon which larger applications can be built.
`proxy`	This sub-directory contains a number of smart proxy and generic proxy examples.
`binders`	This sub-directory contains a number of examples designed to illustrated the features of particular binders, protocols or binding graphs. It includes the worked example from chapter 44.
`applet`	This contains a small number of applet examples. FlexiNet can be used with applets *in theory*. In practice security restrictions and bugs can make this difficult.
`clusters`	This contains a number of examples using clusters. These include mobile object examples, and persistent object examples.
`transactions`	This contains the transaction example from chapter 12.

43.2 Trivial Remote Invocation Examples

These examples are all in the simple subdirectory. They use the `FNetTest` interface to output a server's address to the screen, so that it can be passed as a parameter to the client. This reduces the amount of infrastructure that must be working in order to use these examples. This makes them ideal for testing new protocols or binders.

`TestCall` This is the simplest FlexiNet test. A server process is started which creates a single service interface. The client process is then started and binds to this interface and performs some simple invocations. Only primitive types are passed as parameters.

`TestRefPassing`
This tests simple reference passing. Two processes are used, a client and a service. The service publishes an interface, and the client uses this to obtain an interface to a second service within the same server. It then performs a simple invocation on this. This is a useful test of the serialisation of interface references.

`ChineseWhispers`
This test is a more complex version of TestRefPassing. It tests nested callbacks from server to client. The client creates an object A and then passes a reference to it in a call to the server. During this call, the server invokes an operation on A. This is a simple concurrency test, and illustrated processes acting as both client and server.

`BulkTest` This test may be used to check support for large invocations. A simple service provides a string append function. The client calls this repeatedly with ever increasing string sizes until the maximum Java string length is reached.

43.3 Examples Using the Trader

These are simple application programs. They are written in the style of 'standard' applications, and make use of the trader for publishing/discovery of services. These examples are a good starting point for a developer who wishes to *use* FlexiNet, rather than develop components for it.

`TestTrader`
This is a simple test of the trader. A service is started, which enters the name of an interface into the trader. The client is then started. This contacts the trader and obtains the interface to the published service. It then performs a simple invocation on this.

`BankExample`
This is the example illustrated in Chapter 9. It is a simple bank

service and client. The client adds and withdraws money from the account and on one occasion receives an overdrawn exception.

`TestPerformance`
This is a simple performance test for FlexiNet. The client performs a large number of invocations on the server, in bursts of 1000. It then displays the average time the invocations took. It is interesting to use this example with a number of clients and a single server. This may be used to determine how well a particular protocol or resource policy scales.

`RMIPerformance`
This is an equivalent program to TestPerformace, but written using Sun's RMI, rather that FlexiNet. It may be used to compare performance, and coding style.

`SoakTest` This is the 'big daddy' of protocol testers. It allows a large number of different clients to contact the same server. The server maintains state for each client, and checks that they behave as expected. Both the clients and server may pause between/during the processing of a call. This will test the timeouts and keep-alives of a protocol (for example RRP will close an idle connection). In addition to this, both client and server actively call the garbage collector, so that memory usage can be monitored to aid protocol debugging.

43.4 Proxy Examples

These examples illustrate the different forms of generic and smart proxies available in FlexiNet. The first three of these are described in Chapter 16.

`SmartProxy`
This is an example use of a Smart Proxy to a read-only service. The proxy performs caching, to reduce the number of remote invocations.

`SimpleGenericProxy`
This is the same example as SmartProxy, but makes us of a generic proxy via the `SimpleGenericProxy` interface. It illustrates the difference in coding style. For this example, smart proxies are more straightforward.

`GenericProxy`
The same example again, using a generic proxy using the GenericProxy interface. This is the most flexible proxy interface available, but is overkill for this example.

`MultiProxy`
A simple service offering four different instances of a service, each using a different proxy.

43.5 Applet Examples

Note. FlexiNet can only work in applets if the applet is able to access the network. In these examples, a special class is used to allow this access. Even so, the examples will only work under the following circumstances

- The applet must be run in a full JDK1.1 environment with no (restrictive) `SecurityManager`.
- FlexiNet itself must be loaded locally

These examples are very prone to minor changes in FlexiNet or the Applet environment, as the applet environments are still relatively buggy with respect to 'advanced' JDK1.1 features like introspection and inner classes. In principle, FlexiNet can run in an applet environment without these restrictions, but more work is required to identify and overcome the idiosyncrasies of the current applet viewers and browsers.

`AppletTest`
A version of `TestCall` that uses an applet for client and server.

`AppletTestRefPassing`
A version of `TestRefPassing` that uses an applet for client and server.

43.6 Binder Examples

These examples illustrate particular binder functionality. The majority of the other examples can be run using any binder (Magenta is used by default).

43.6.1 SSL Examples

The SSL binder (Crimson) can be used with any of the other non-cluster examples. A text file in the `TestCode/binders/SSL` directory explains how this is done.

43.6.2 IIOP Examples

The IIOP binder may used with any of the test code, providing it is using the objects-by-value IDLMapper. However, specific test code is provided to illustrate standard CORBA examples.

`Grid` This is a standard CORBA example. It creates a grid of values on a server, and a client then manipuates these.

`ObjByValue`
This is a demonstration of the use of the CORBA objects-by-value

RFP. A number of objects are passed by reference and by value between a client and server.

`Specials`	This tests the handling of various CORBA 'specials' such as in-out parameters an enumerations.

43.6.3 Lamport Clocks

The construction of an example binder is illustrated in chapter 44. The corresponding source code is in the `TestCode/binders/LamportClocks` directory.

43.6.4 Class Repository

This is a simple example use of the class repository. It explains how to create bundles for use in the repository, and how to use it. It contains a simple example program that will fail if used without the repository.

43.7 Cluster Examples

There is one 'vanilla cluster' example, and a number of examples for mobile and storable clusters.

43.7.1 Vanilla Clusters

`ClusterTest`

This example consists of a single application program. When run, this creates a local capsule, and a cluster within that capsule. The capsule then communicates with the cluster to perform a simple invocation. This tests the basic clustering mechanism. With tracing enabled, the non-trivial behaviour can be seen.

43.7.2 Mobile Object Examples

`TweetiePie`

This is a simple non-graphical mobile object example. A 'place server' process creates a Place that mobile objects can move to. The 'client' process then creates a second place and a mobile object (a bird) which 'flies' between the two places. In addition, the client process periodically invokes operations on the bird, to illustrate mobile naming in action.

`Tama` = (Tamagotchi)

This is a complex, graphical demonstration. In essence, a factory is created for 'Tamagotchi' mobile objects. These have a standard

AWT interface, and may also be requested to move between places on a pre-defined itinerary. Internally, the Tamagotchi code performs many consistency checks, making this both a demonstration and a piece of test code.

`EventTest`
: This is a demonstration of the ‘event’ subsystem developed for use with mobile objects.

`MagicPlace`
: This is a demonstration of how Places, or other Capsules, can be subclassed to provide specialist functionality. The place created conforms to the standard Place interface, but performs additional auditing of mobile object activities.

43.7.3 Persistent Object Examples

`Black`	This is a simple example of ‘black box’ use of the Information Space. Objects are stored by copying them into an information space, and then retrieved by a second client by copying them out.
`White`	This demonstrates ‘white box’ use of the Information Space. A persistent account object is created by one client. A second client updates this. Neither client need be aware that the object is actually persistent.
`WhitePages`	Another ‘white box’ test.
`Ibrowser`	This provides a simple graphical interface that may be used to browse an Information Space.

43.8 Transactional Examples

This contains a single example `BankTest` that was described in chapter 12. As this illustrates both the assembly and execution of a bean-based application, it is correspondingly more complex than the other examples.

44 LAMPORT CLOCKS – AN EXAMPLE BINDER

44.1 Introduction

As an worked example of binder engineering, a simple binder is designed and illustrated in this chapter. This binder, `LamportBinder`, is based on the Magenta binder, but adds an additional layer used to pass sequencing information between client and server, and to produce debugging information.

The Lamport Binder uses the concept of Lamport Clocks [LAMPORT78]. The basic scheme is as follows:

- Each process maintains a *logical* clock. The clock is logical in that it starts at zero and 'ticks' only when some even happens.
- Each message sent contains the sender's clock value.
- On receipt of a message, the local clock is advanced (if necessary) to ensure that the receive timestamp is larger than the send timestamp.
- Whenever an 'interesting event' occurs, the local clock is read, and incremented. This ensures that no two local events occur at the same time.

The information from the timestamps is used to print debugging information of the form

```
  client invoke @4
  server received invocation @5
  server replied @7
  client received reply @8
```

The code described in this chapter is available in the "`LamportClock`" test code directory.

44.2 Approach

For simplicity of example, we construct our binder from scratch. In practice, this binder could be a subclass of `Burgundy`. If this approach were taken, only the method `setRequirements` would be required in the subclass, the rest being inherited from Burgundy.

Our binder is going to support a variant on the “rrp” protocol. At the low level, this will be precisely “rrp”, however as we are adding an additional argument to every call and return, our protocol is strictly incompatible with rrp. We therefore choose a different protocol name, “rrp-lamport”.

The Magenta and Burgundy binders are complicated by two facts:

- They can be bootstrapped at a previous address, cluster address or port
- They support the parsing and stringification of names that include cluster or mobile namer information.

For simplicity of example, we will avoid both of these issues. Although our binder will be able to handle mobile names and cluster-based addresses for remote services, it will be unable to use mobile names for local interfaces, nor parse or stringify these names. In addition we avoid the bootstrap code as it is complex and covers many cases. Our binder will always use a random port. If the alternative approach of subclassing `Burgundy` were taken, then these limitations would be avoided.

44.3 Components

There are three components in our new binder environment:

- The `TimerLayer` which performs the actual work – maintaining Lamport Clocks, and printing debugging information.
- The binder (`LamportBinder`) which creates a binding stack containing this layer.
- A testbench to allow use of the LamportBinder (`LamportTest`)

These will be described in turn.

44.4 Timing Layer

This layer is summarised in Figure 116. It has two static methods (both called `getLamportTime`) which maintain the local clock. The other methods fall into two categories; initialisation methods and operation methods.

44.4.1 Lamport Clock Methods

These methods implement the Lamport clock abstraction. They are show in Figure 117.

```
public class  TimingLayer
      implements CallUp, CallDown, UniformStartup
{
  // static data & methods for Lamport Clock
  protected static int lamportClock=0;
  public synchronized static int getLamportTime()
  public synchronized static int getLamportTime(
                                   int remoteTime)
  // per-instance data
  protected CallDown down;
  protected CallUp up;

  // initialization methods
  public static void setRequirements(Blueprint props)
  public static Object createUninitialised()
  public int initialise(Blueprint props)

  // operation methods
  public void calldown(Invocation data)
  public void callup(Invocation data)
}
```

Figure 116 Overview of TimerLayer

```
public synchronized static int getLamportTime()
{
  lamportClock++;
  return lamportClock;
}

public synchronized static int getLamportTime(int remoteTime)
{
  lamportClock = Math.max(lamportClock,remoteTime);
  return getLamportTime();
}
```

Figure 117 Lamport Clock Methods

44.4.2 Initialisation Methods

The `TimerLayer` is controlled using blueprints. It has three initialisation methods that together implement the `UniformStartup` interface. These are illustrated in Figure 118.

`setRequirements`
    This defines Blueprint properties indicating constraints on the layers above and below this in the stack.

`createUnitialised`
    This simple creates a new instance of the layer

`initialise`
    This reads property values from the supplied blueprint, and sets the per-instance fields within the layer.

```
public static void setRequirements(Blueprint props)
                   throws ConstraintContentionException
{
  props.constrain("up",CallUp.class);
  props.constrain("down",CallDown.class);
}

public static Object createUninitialised()
{
  return new TimingLayer();
}

public int initialise(Blueprint props)
{
  up=(CallUp)props.get("up");
  down=(CallDown)props.get("down");
  return Blueprint.COMPLETE;
}
```

Figure 118 Initialisation Methods for TimerLayer

44.4.3 Operation Methods

These are the methods called during normal operation of the binder. `Calldown` is called on the client in order to process an invocation. This leads to a nested call of `Callup` on the server. Both methods make use of the 'additional argument stack' stored in the invocation. When sending a message, the sender *pushes* the current time onto the stack. When receiving a message, the receiver *pops* the sending time off the stack. Because of this used of the stack, this layer must exist above the serialisation layer in the protocol stack. If this were not possible, a variant on the layer could be devised that used its own buffer segment for storing the Lamport Time. The Fragmentation layer in the Green protocol is an example of such a layer. The `CallUp` and `CallDown` methods are shown in Figure 119. For brevity, exception handling is omitted.

```
public void calldown(Invocation data)
                     throws BadCallException
{
  // start of invocation
  int localTime = getLamportTime();
  System.out.println("INVOKE   @ " + localTime +
                     " " + data);
  data.push(Integer.class,new Integer(localTime));
  down.calldown(data);
  // reply recieved
  Integer i = (Integer) data.pop(Integer.class);
  int remoteTime = i.intValue();
  localTime = getLamportTime(remoteTime);
  System.out.println("RETURNED @"+ localTime +
                     " (sent @" + remoteTime+")");
}

public void callup(Invocation data)
                   throws BadCallException
{
  // invocation received
  Integer i = (Integer) data.pop(Integer.class);
  int remoteTime = i.intValue();
  int localTime  = getLamportTime(remoteTime);
  System.out.println("RECIEVED @" + localTime +
                     " (sent @" + remoteTime+")");
  System.out.println("INVOCATION = " + data);
  up.callup(data);
  // invocation has been processed, send reply
  localTime = getLamportTime();
  System.out.println("RETURN   @" + localTime);
  data.push(Integer.class,new Integer(localTime));
}
```

Figure 119 Operation Methods for TimingLayer

44.5 A Binder Using TimingLayer (LamportBinder)

The binder used with `TimingLayer` is a simplified version of Burgundy. As explained in section 44.2, this simplification of to aid explanation. A binder constructed using blueprints, such as this, must support all of the following methods:

Methods that create an instance of the binder using Blueprints:

```
static void   setRequirements(Blueprint b)
static Object createUninitialised()
int initialise(Blueprint props)
```

Standard binder methods – generate and resolve names:

```
Name generateName (Object obj,Class cls,FlexiProps qos)
boolean grantName (Object obj,Class cls,Name name,
                  FlexiProps qos)
```

```
Object resolveName(Name name, Class cls, FlexiProps qos)
```

Utility methods that are part of the Binder interface:

```
boolean resolvesProtocol(String p)
boolean addGenerator    (Generator g)
void    dropNames       (Object obj, Class cls)
String  stringifyName   (Name name) throws BadName
Name    parseName       (String name) throws BadName
```

These sets of methods will be considered in turn.

44.5.1 Binder Construction using Blueprints

Of the three `UniformStartup` methods, `setRequirements` is by far the most complex. This must define all of the layers that make up the protocol stack, and give 'suggestions' on resources that they may use and how the layers are related to each other. The statements are 'suggestions' because it is possible for the creator of the binder to override any of these suggestions, with any other suggestion that meets the constraints specified by that component. Where the binder itself wishes to constrain a value it uses the blueprint `constrain` method.

The initialisation methods are shown in Figure 120 and Figure 121. Figure 120 shows `createUnitialised initialise` and the first part of `setRequirements`. This part is responsible for identifying sub-parts of the binder (layers and resources). The second part is shown in Figure 121. This shows dependencies between the components. The differences from Burgundy are minor and relate to the introduction of the additional layer. They are highlighted in the figures.

Two sets of properties are worthy of further comment:

- The buffer set up makes use of a set of constants; `RRSEGMENT`, `CLUSTERNAMESEGMENT`, `NAMESEGMENT` and `DATASEGMENT`. These are defined to the values `0`,`1`,`2`,`3` in the static initialisation of the class, and represent the buffer segments used by the various layers in the stack. Using segmented buffers helps keep layers independent.
- The default action of the serial layer is overwritten by setting a number of properties to change the class of the serialiser used. This is a clone of the code in `Burgundy`. This is to force the serial layer to use the class repository, if available. Similar customisation can be made to any component, including the binder itself.

```
public static Object createUninitialised()
{
  return new LamportBinder();
}

public int initialise(Blueprint props)
{
  clientTop = (CallDown)      props.get("locatelayer");
  nameLayer= (TrivNameLayer)  props.get("namelayer");
  binderTop = (Binder)        props.get("binderTop");
  RRPLayer rpcLayer=(RRPLayer)props.get("rpclayer");
  baseAddress =  rpcLayer.getAddr();

  return Blueprint.COMPLETE;
}

public static void setRequirements(Blueprint b)
                   throws ConstraintContentionException
{
 // The context in which this may be used
 b.constrain ("binderTop",Binder.class);
 // set up buffers
 b.constrain("buffer.outputfactory",OutputBufferFactory.class);
 b.link("..template."+RRSEGMENT    ,"rpclayer.segmentsize");
 b.link("..template."+CLUSTERNAMESEGMENT
                             ,"clusternamelayer.segmentsize");
 b.link("..template."+NAMESEGMENT  ,"namelayer.segmentsize");
 b.link("..template."+DATASEGMENT  ,"buffer.datasize");
 b.constrain("buffer.inputfactory" ,InputBufferFactory.class);
 b.link("..template"               ,".outputfactory.template");
 // Suggest suitable buffer abstraction and maximum size
 b.suggest("buffer.datasize"  ,1024*128);
 b.suggest(".inputfactory"    ,BasicInputBufferFactory.class);
 b.suggest(".outputfactory"   ,BasicOutputBufferFactory.class);
 // Suggest a threading policy
 b.suggest("tcp.maxthreads"      ,6);
 b.suggest("tcp.maxwaiters"      ,6);
 // set up session manager
 b.require("sessionmanager"        ,SessionManagerImp.class);
 b.suggest("sessionmanager.factory",RRPSessionFactory.class);
 b.link   ("sessionmanager.factory.layer"  ,"rpclayer");
 b.link   ("sessionmanager.factory.manager","sessionmanager");
 // Layer of the shared protocol stack
 b.require("locatelayer"        ,LocateLayer.class);
 b.require("clientcalllayer"    ,ClientCallLayer.class);
 b.require("servercalllayer"    ,CallLayer.class);
 b.require("lamportlayer"       ,TimingLayer.class);
 b.require("serial"             ,SerialLayer.class);
 b.require("namelayer"          ,TrivNameLayer.class);
 b.require("clusternamelayer"   ,SingleClusterMuxLayer.class);
 b.require("rpclayer"           ,RRPLayer.class);
```

Figure 120 Initialisation Methods (part 1)

```
 // Links down the stack (client)
 b.link   ("locatelayer.down"     ,"clientcalllayer");
 b.link   ("clientcalllayer.down" ,"lamportlayer");
 b.link   ("lamportlayer.down"    ,"serial");
 b.link   ("serial.down"          ,"namelayer");
 b.link   ("namelayer.down"       ,"clusternamelayer");
 b.link   ("clusternamelayer.down","rpclayer");
 // Links up the stack (server)
 b.link   ("rpclayer.up"          ,"clusternamelayer");
 b.link   ("clusternamelayer.up"  ,"namelayer");
 b.link   ("namelayer.up"         ,"serial");
 b.link   ("serial.up"            ,"lamportlayer");
 b.link   ("lamportlayer.up"      ,"servercalllayer");
 // set segments for use by the various layers
 b.require("serial.datasegment"                 ,DATASEGMENT);
 b.require("namelayer.namesegment"              ,NAMESEGMENT);
 b.require("clusternamelayer.clusternamesegment",
                                          CLUSTERNAMESEGMENT);
 b.require("rpclayer.datasegment"               ,DATASEGMENT);
 // layers using the session manger
 b.link   ("clientcalllayer.sessionmanager"     ,"sessionmanager");
 b.link   ("rpclayer.sessionmanager"            ,"sessionmanager");
 // layers using buffers
 b.link   ("serial.outputbufferfactory"   ,"buffer.outputfactory");
 b.link   ("namelayer.outputbufferfactory","buffer.outputfactory");
 b.link   ("clusernamelayer.outputbufferfactory"
                                          ,"buffer.outputfactory");
 b.link   ("rpclayer.outputbufferfactory" ,"buffer.outputfactory");
 b.link   ("rpclayer.inputbufferfactory"  ,"buffer.inputfactory");
 // Serial layer extra setup
 b.suggest("serial.serializerFactory",RefSerializerFactory.class);
 b.link   ("..generator"             ,"binderTop");
 b.suggest("..classSerializerFactory"
                             ,ClassBundleSerializerFactory.class);
 b.suggest("serial.deserializerFactory"
                                    ,RefDeSerializerFactory.class);
 b.link   ("..resolver"            ,"binderTop");
 b.suggest("..classDeSerializerFactory"
                           ,ClassBundleDeSerializerFactory.class);
 // RPC Layer extra setup
 b.suggest("rpclayer.port" ,"0");
 b.link   (".maxthreads"   ,"tcp.maxthreads");
 b.link   (".maxwaiters"   ,"tcp.maxwaiters");
}
```

Figure 121 Initialisation Methods (Part 2)

44.5.2 Standard Binder Functions

The three standard binder functions, `generateName`, `grantName` and `resolveName` are shown in Figure 122. The first two make use of the name layer to manage multiplexing of a number of interfaces over a single shared base address. The third function trivially creates a new stub and links it to the top of the stack. No additional resolution of the name is made until a method on the stub is invoked.

```
public Name generateName (Object obj, Class cls,
                          FlexiProps qos)
{
  if (qos != null) // cannot handle QoS, hand off
    return nextGenerator.generateName(obj,cls,qos);

  int id =  nameLayer.generatePartName(obj,cls);
  return  new TrivName(protocol,baseAddress,id);
}

public boolean grantName (Object obj, Class cls,
                          Name name,FlexiProps qos)
{
  if((qos==null) && (name instanceof TrivName))
  {
    TrivName tname =  (TrivName) name;
    if ((baseAddress.equals(tname.ref)) &&
        (nameLayer.grantName(obj, cls, tname.id)))
         return true;
  }
  // fall through if we can’t grant this name
  return nextGenerator.grantName(obj,cls,name,qos);
}

public Object resolveName (Name name, Class cls,
                           FlexiProps qos) throws BadName
{
  if (qos != null) // cannot handle QoS
    return  null;
  // return a new stub, linked to the top of the stack
  return  Stub.stub(cls, clientTop, name);
}
```

Figure 122 Standard Binder Functions

44.5.3 Additional Binder Methods

The rest of the binder methods are straightforward and are illustrated in Figure 123. Equivalent (or identical) methods are found in the majority of binders. The purpose of these methods is described in sections 14.6 and 14.7.

```
String protocol="rrp-lamport";
public boolean resolvesProtocol(String p)
{
 return (protocol.equals(p)); // only resolve one protocol
}

// we are one of a list of generators, initially
// termiate this list with a null generator.
Generator nextGenerator = new NullGenerator();
public boolean addGenerator(Generator g)
{
  // add a generator to the front of the list
  g.addGenerator(nextGenerator);
  nextGenerator = g;
  return true;
}

public void dropNames (Object obj, Class cls)
{
   nameLayer.dropNames(obj, cls);    // names we generated
   nextGenerator.dropNames(obj,cls); // recurse down list
}

public String stringifyName (Name name) throws BadName
{
  // ignore other generator's names
  if(!name.getProtocol().equals(protocol))
    return nextGenerator.stringifyName(name);

  if (!name instanceof TrivName)
    throw  new BadName(); // sanity check

  // the stringified name is of the form protocol:(addr)(id)
  TrivName mname =  (TrivName)name;
  return  protocol+":("+RRPLayer.stringifyName(mname.ref)+
           ")("+TrivNameLayer.stringifyPartName(mname.id)+
           ")";
}

public Name parseName (String name) throws BadName
{
  if(!protocol.equals(NameUtil.getProtocol(name)))
    throw new BadName(); // sanity check

  String addressPart =  NameUtil.getPartName(name, 0);
  String idPart      =  NameUtil.getPartName(name, 1);

  Address baseAddress =  RRPLayer.parseName(addressPart);
  int id =  TrivNameLayer.parsePartName(idPart);

  return new TrivName(protocol, baseAddress, id);
}
```

Figure 123 Additional Binder Methods

44.6 A Testbench for Lamport Binder

Figure 124 shows a complete testbench for the Lamport Binder. This sets the binder graph to a simple graph containing three components. A Cache binder (as BinderTop), a SmartChoice resolver (to allow the use of proxies and dynamically loaded protocols), and a LamportBinder. This is the only generator, and is also a resolver used for the “rrp-lamport” protocol.

Note that as LamportBinder does not understand bootstrap strings, this testbench can only be used for applications that can run at an arbitrary address. In particular, it may be used for the trader, but the trader will run at a different address each time it is started. This limitation is overcome if LamportBinder is built as a subclass of Burgundy (as described in section 44.2).

```
public class LamportTest
{
  public static Binder init()
  {
    try
    {
      Cache binderTop = new Cache();

      // set up blueprints
      Blueprint bp = new Blueprint(LamportBinder.class,
                 new ClassConstraints(LamportBinder.class));

      bp.set("binderTop",binderTop);

      // build LamportTest using blueprints
      bp.construct();
      Binder lamport = (Binder) bp.get(null);

      // set up rest of binder graph
      SmartChoice choice = new SmartChoice(binderTop);
      binderTop.init( lamport,choice );
      choice.addResolver(lamport);

      return binderTop;
    }
   catch (Exception e)
    {
      return  null;
    }
  }
}
```

Figure 124 A Testbench for LamportBinder

44.7 A Simple Example

In this section, we demonstrate how to use the `LamportBinder` in a trivial example, `TestCall` (section 43.2).

1. Ensure that the `LamportBinder`, FlexiNet and the `TestCall` directory are on the class path.

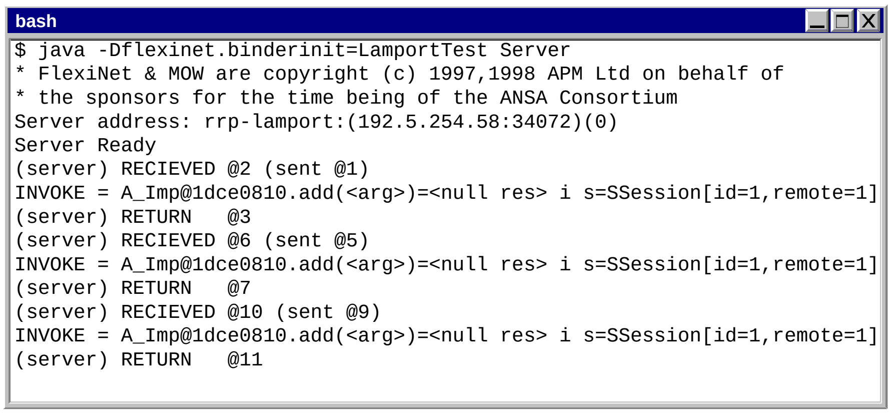
2. Run the server:

   ```
   java -Dflexinet.binderinit=LamportTest Server
   ```

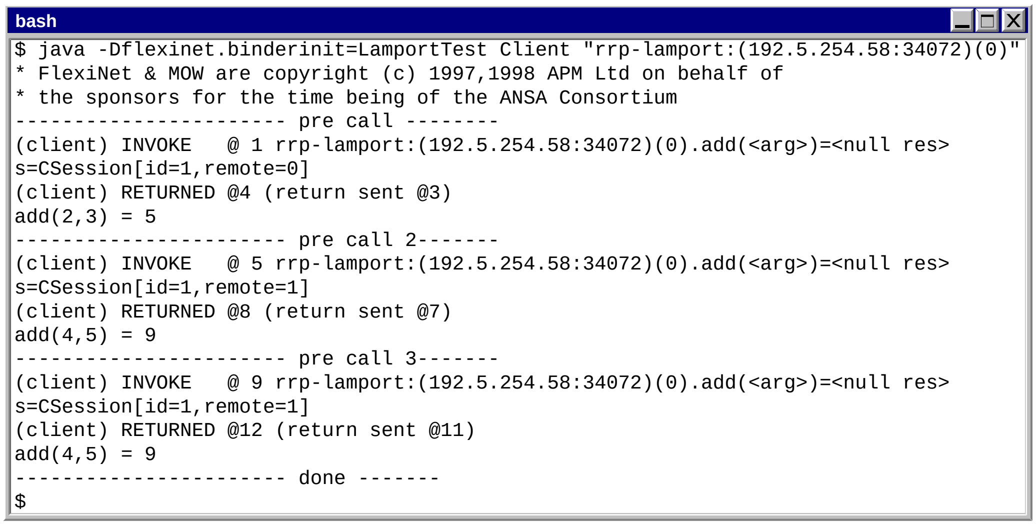
3. Run the client

   ```
   java -Dflexinet.binderinit=LamportTest Client server_addr
   ```

The trace of an example run is shown in Figure 125.



```
$ java -Dflexinet.binderinit=LamportTest Server
* FlexiNet & MOW are copyright (c) 1997,1998 APM Ltd on behalf of
* the sponsors for the time being of the ANSA Consortium
Server address: rrp-lamport:(192.5.254.58:34072)(0)
Server Ready
(server) RECIEVED @2 (sent @1)
INVOKE = A_Imp@1dce0810.add(<arg>)=<null res> i s=SSession[id=1,remote=1]
(server) RETURN   @3
(server) RECIEVED @6 (sent @5)
INVOKE = A_Imp@1dce0810.add(<arg>)=<null res> i s=SSession[id=1,remote=1]
(server) RETURN   @7
(server) RECIEVED @10 (sent @9)
INVOKE = A_Imp@1dce0810.add(<arg>)=<null res> i s=SSession[id=1,remote=1]
(server) RETURN   @11
```



```
$ java -Dflexinet.binderinit=LamportTest Client "rrp-lamport:(192.5.254.58:34072)(0)"
* FlexiNet & MOW are copyright (c) 1997,1998 APM Ltd on behalf of
* the sponsors for the time being of the ANSA Consortium
----------------------- pre call --------
(client) INVOKE   @ 1 rrp-lamport:(192.5.254.58:34072)(0).add(<arg>)=<null res>
s=CSession[id=1,remote=0]
(client) RETURNED @4 (return sent @3)
add(2,3) = 5
----------------------- pre call 2-------
(client) INVOKE   @ 5 rrp-lamport:(192.5.254.58:34072)(0).add(<arg>)=<null res>
s=CSession[id=1,remote=1]
(client) RETURNED @8 (return sent @7)
add(4,5) = 9
----------------------- pre call 3-------
(client) INVOKE   @ 9 rrp-lamport:(192.5.254.58:34072)(0).add(<arg>)=<null res>
s=CSession[id=1,remote=1]
(client) RETURNED @12 (return sent @11)
add(4,5) = 9
----------------------- done -------
$
```

Figure 125 An Example Use of LamportTest

PART NINE: ADVANCED TOPICS

45 INTRODUCTION

In the following two chapters, we outline the design of two advanced FlexiNet sub-systems. These are paper designs, and full implementations of the systems have not been constructed. They are the fruits of two advanced research activities under the ANSA/FlexiNet umbrella.

46 MOBILE AGENT SECURITY

46.1 Introduction

We have identified six basic areas of security concern when mobile objects are used in an open or insecure environment.

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.

Of these six areas three have already been tackled, namely host integrity (via strong encapsulation and cluster security managers); access control (via FlexiNet reflection and/or SSL certificates) and secure communication (via SSL). The remaining areas require that a Cluster may carry secret information (to support confidentiality) and that this information cannot be modified or separated from the cluster (to support integrity).

The approach to these problems is to devise "secure objects". A secure object is an object that encapsulates state that may only be accessed or updated at particular hosts, according to a pre-configured security policy stored in the object. The secure object is protected cryptographically, so that a malicious host cannot ‘break’ the object to gain access, and so that illicit modifications made by a malicious host can be detected at any later host. A mobile cluster may contain one or more secure objects, and use these to carry anything of value. To prevent the secure object from being disassociated with the cluster, the infrastructure allows the secure object to validate its containing cluster prior to allowing access. This may include a check on the cluster’s code (classes) and/or a check on the state of any (unprotected) fields within the cluster. A malicious host may fake this validation, but this will give it an advantage over other attacks.

46.2 General Overview

A “secure object” is a cluster of objects within another cluster such as a mobile cluster. The secure object 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 126).

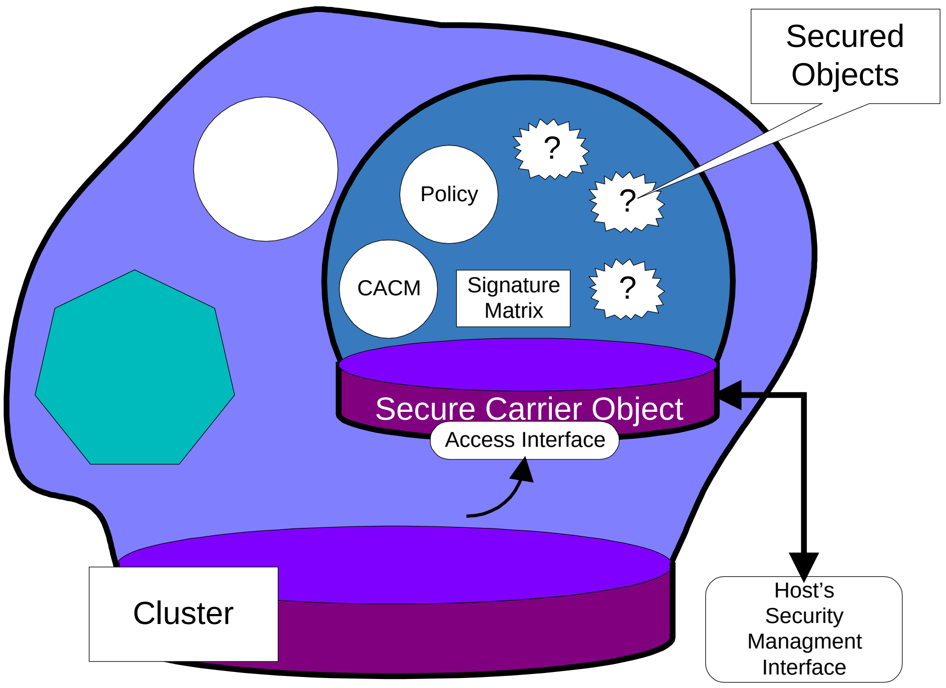


Figure 126 A Secure Inner Cluster

The “secure cluster” is an “inner cluster” that is treated as an object by the top level 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 127.

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

Figure 127 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 128.

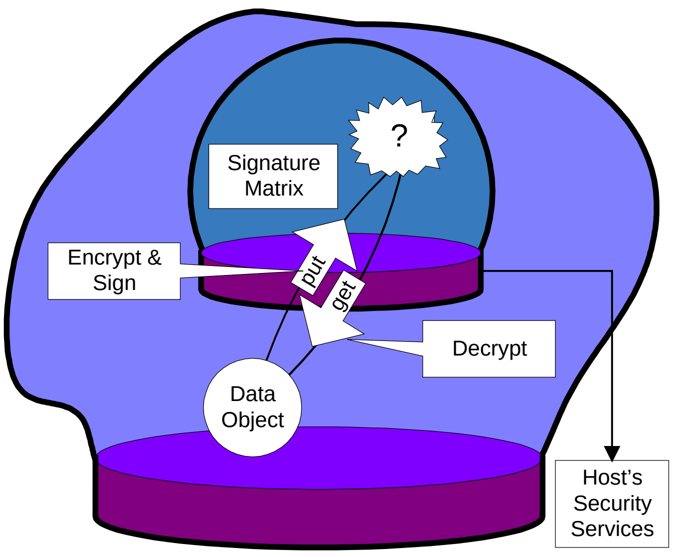


Figure 128 Operation of a Secure Carrier Object

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 Class Repository. The serialised object will actually be a secure reference to a class stored in the repository.

Figure 129 shows component interactions.

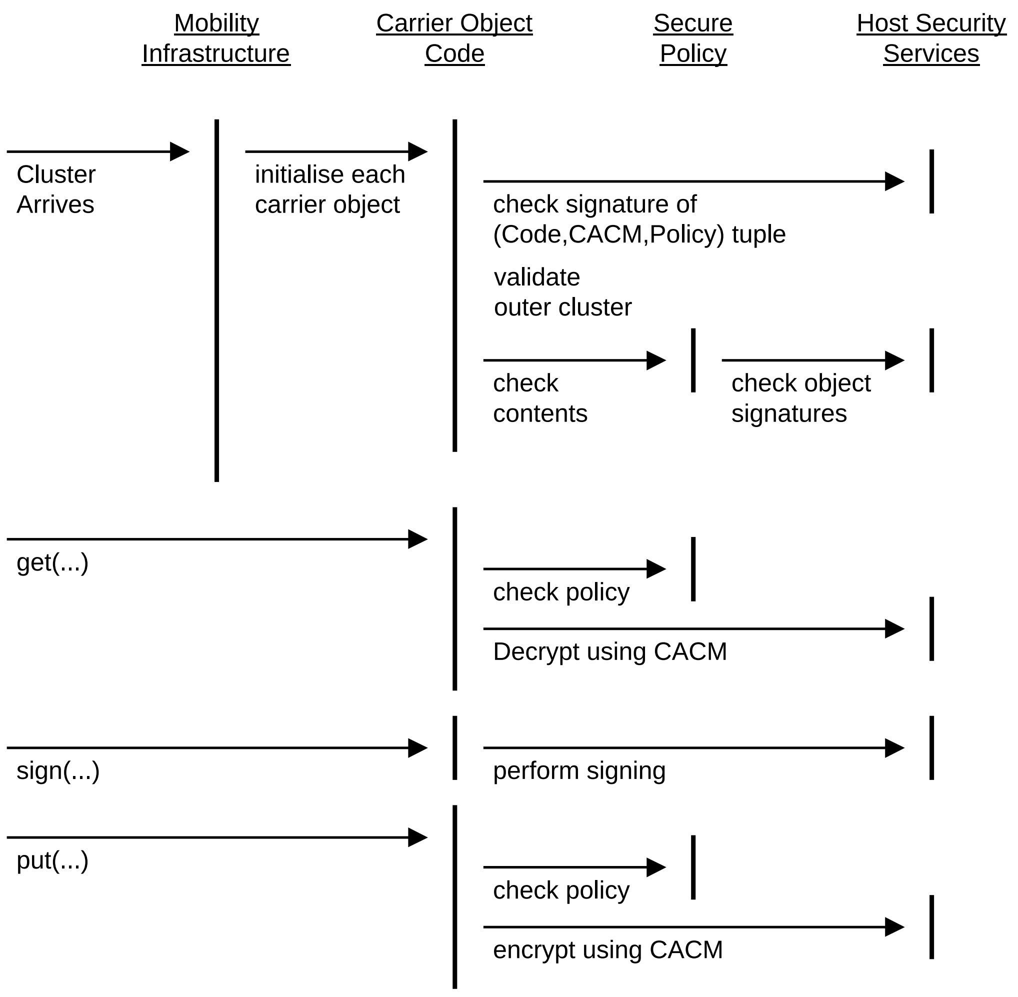


Figure 129 Interaction of Security Components

The method (MobileCluster)`.getPlace` returns an interface to host specific services. This interface is extended to include the method `getSecurityInterface` that returns the hosts security interface for mobile objects. This interface provides for the encrypting, decrypting, signature checking and signing of objects using the cryptographic access control matrix and itinerary definitions passed as parameters to the functions.

47 AN ADVANCED RELOCATION SERVICE

47.1 Introduction

The relocation service described in section 35.3 suffers from three serious defects.

- It is possible for an accidental name clash to prevent correct operation
- Name clashes can be created by a malicious client and used to attack the system
- An aggressor may alter information in the relocation service, as there is no authentication or integrity checks.

In this chapter, we described a more robust relocation architecture. This is based on the original design, but is extended to deal with accidental and maliciously generated clashes.

The approach taken is to allow clashes to occur, and to use a secondary mechanism to deal with them. This mechanism is sufficiently cheap that intentionally creating clashing names ceases to be an effective attack. The new architecture has also been designed to allow the integrity of a naming record to be assured, so that only the ‘owner’ may modify or delete it.

There are also optional extensions to support periodic refresh of references to allow some naming records to be recycled. These extensions allow a trade-off to be made between storage space in the name servers, and additional background computation. Experience is required to determine if this trade-off is worthwhile.

47.2 Basic Approach

Each named entity (FlexiNet Cluster) is identified by a tuple

(identifier, current address, reference to relocation service)

The *current address* is used to communicate with the entity. If the entity moves, then an attempt to communicate with it will fail. The callee then contacts the *relocation service* (Relocator) identified in the tuple and asks for the latest address of the entity. It then updates its stored tuple, and uses this new address. As each naming tuple contains an explicit reference to a

Relocator, there may be many different Relocators. Typically, sufficient instances will exist to spread the load of supporting a large population of names. Which Relocator to use for a particular entity is determined when that entity is first named (although the Relocator itself is not contacted until later). It is possible to change the service associated with a particular entity, and this is important over long time scales – if entities and their clients migrate away from an instance of a Relocator.

Individual Relocators may themselves be replicated in a number of ways for performance and/or availability. The naming scheme has been designed so that this is orthogonal to the design of the naming architecture. However, if a concession is that a Relocator previously associated with an entity plays an ever-decreasing role in the resolution of its names, and need only store read-only state associated with it. With the 'lease' optimisation, it may discard all knowledge of an entity after a relatively short time.

As the population of names is divided between a number of Relocators, the failure of one Relocator will only affect a small proportion of the total population of names. As Relocators are only contacted when an entity moves, or when a client attempts to communicate with an entity that has moved since its last contact; the affect of a transient Relocator failure is limited.

47.3 Naming Records

The role of the Relocator is to store a mapping from entity identifiers to their current addresses. Such a record is called an *Address* record. If the Relocator associated with an entity is changed, then some clients may still have naming tuples containing a reference to the original Relocator. That Relocator must therefore store a *Forward* record indicating that it is no longer the Relocator for the entity. A Relocator will store at most one Address or Forward record per identifier.

Problems arise if two names clash. That is if two entities are accidentally or maliciously given the same identifier. If these two entities exist on different hosts, and used different Relocators, then the clash will not be detected, and will not cause any problems. However, if the entities move to the same host or to the same Relocator then the basic scheme will fail.

47.4 Types of Clashes

A naming clash occurs when two entities have the same name. It may be detected in a number of different circumstances.

- A host creates an entity, and chooses a Relocator for it. When the entity moves, and the Relocator is updated, it is discovered that the Relocator already contains a record for a different entity with the same identifier.
- An entity moves to a host that already contains an entity with the same identifier, which uses the same or a different Relocator.

- The Relocator used by an entity is changed, and the new Relocator already has an entry for an entity with the same identifier.

These may occur in combination, for example an entity may move host and Relocator simultaneous and clash on the second two cases, or may move for the first time, and clash on the first two cases.

47.5 Approach to Clash Recovery

The approach taken to handling clashes is to allow them to occur, and then assign a new identity to the moving entity if it is found to clash with another entity. As clients will already hold references to the entity with its previous identity, these must be continued to be resolved. This is possible as all (identifier, address, Relocator) tuples held by clients uniquely identify a particular entity, *even if other entities have the same identifier*. We ensure this by construction.

Relocators may contain two additional record types to deal with clashes. An identifier within the Relocator may therefore map to one of four different records

- An *address* record. This is the normal case. The address record contains the current address of the entity.
- A *forward* record. This record contains the identity of the Relocator that is now responsible for managing the entity (This Relocator may itself forward the request).
- A *remap* record. This record contains the previous address of an entity that has been assigned a new identity, and the new identity. This is used to handle clashes detected in the Relocator itself.
- A *universal remap* record. This record contains the new identity of an entity that has been renamed. This is used to handle clashes detected on a host an entity moves to.

A particular identifier will in general map to a set of *remap* records and zero or one other record, which will be an address, forward or universal remap record.

47.6 Clash Resolution Algorithm

Identifiers are structured and have two fields. (R, N). R is a large random number. The larger the random number, the less likely accidental clashes are. However, as clashes can be dealt with, it need not be overly large. It is expected that 48 bits will suffice for a global system, although analysis has yet to be undertaken. The second field, N, is a counter, used to enumerate entities created by a host. This removes the possibility of a host accidentally creating two entities with the same identifier, and allows a host to detect a possible clash between a ‘foreign’ name and one that it may created in future.

116 bits is sufficient for N. If a host expects to create more than 2^{16} names, it can simply choose a number of different values for R.

A host therefore will never create identifiers that clash with other identifiers that it generates. This is important as clash resolution takes place at the Relocator, and the Relocator is only involved when (if) an entity first moves.

There are a number of different operations on the Relocator. We will consider these in turn.

47.6.1 An Entity Moves for the First Time

This is the point at which the identifier is first presented to the chosen Relocator. The source site first contacts the destination site and initiates the move. The destination host validates that the entity identifier does not clash with any entity currently at the host, or that may be created there in future.

If there is no clash at the destination host, the source host enters the mapping (identifier,source address) → destination address into the Relocator. If the Relocator does not have a record for that identifier, it stores an address record, mapping the identifier to the destination address. The source address information is discarded. This is the normal case. It is expected to occur 99%+ of the time.

If there *is* a clash at the destination host, or the Relocator already contains an address, forward or universal remap record for the identifier, then a fresh identifier is generated which does not to clash with the destination host or the Relocator.

> *Note*. If the Relocator contains one or more remap records for the identifier, then it is not necessary to create a new identifier, unless the Relocator has a remap record containing the destination address. The Relocator cannot contain a remap record containing the source address, as a different entity with the same identifier could never have existed on the source host.

If a new identifier is required then a remap record is created on the Relocator containing the following tuple:

(source address, new identifier)

This will allow any clients who later contact the Relocator searching for the identifier to be directed towards the new identifier. The clients will always supply the correct source address, as the entity has never moved before. In addition to the remap record, a mapping from the new identifier to an address record is created.

47.6.2 An Entity Moves Again, but Stays With The Same Relocator

During a subsequent move, a clash on the Relocator cannot occur. If there is no clash on the destination host, then the current address record on the Relocator is overwritten with a new address record.

If there is a clash at the destination host, then a fresh identifier is created. The previous address record on the Relocator is overwritten with a universal remap record that directs all clients searching for this entity to look under a new identifier.

Note A universal remap record does not contain the source address, as clients may supply any of the entities previous addresses as the source address. Storing all possible values would be impracticable, but is unnecessary, as the universal record will suffice. The algorithm can never require that a Relocator stores two universal records for one identifier, as universal records are only ever created to replace 'address' records, and there is only ever one of these.

47.6.3 An Entity Changes Which Relocator It Is Associated With

In practice, this is only likely to occur as a side effect of moving, however conceptually it is a separate operation. A clash will require that the entity is assigned a new identifier, and this may be impracticable for engineering reasons – however it if a change of Relocator is associated with a move, then this is straightforward.

If the new Relocator contains no record for the identifier, then an address record is created in the new Relocator, and the old Relocator overwrites its address record with a forward record. This is the normal case.

If the new Relocator contains a record for the identifier then a fresh identifier is created. The original Relocator replaces its address record with a universal remap record, and then stores a mapping from the new identifier to a forward record. The new Relocator stores an address record.

47.6.4 A Client Resolves An Address

A client will only ever contact a Relocator if the current address it has for an entity is proved to be out of date. This may be checked by querying the entity at the specified address. If there is no entity at the address, then the entity has moved. If there is an entity, then both the identifier and Relocator for the entity must be checked.

Note. Intuitively, only the identifier need be checked, however consider the following attack:

1. An entity e_1 with identifier i moves to a host h. This is handled by Relocator N_1.
2. A client C, obtains a name for this entity (i,h,N_1).
3. The entity moves on. Later a second entity e_2 with the same identifier moves to h. This is handled by Relocator N_2, so there is no clash.

4. C uses the address h. The entity at this address has identifier i, but is e_2. However, the Relocator for e_2 is N_2. A check of the Relocator will spot the discrepancy.

If the entity has moved from the address known by the client, the Relocator is contacted. The client supplies the address where the entity used to be (the previous address) and the identifier. The Relocator looks up the records for the identifier and performs the following checks:

- If it has any remap records, then it checks to see if one applies to the specified address. If it does, then it rechecks using the new identifier specified in the record.
- If the Relocator has a universal remap record, then it rechecks using the new identifier specified in the record.
- If the Relocator has a forward record, then it contacts the specified Relocator, and makes a nested request to it.
- If the Relocator has an address record, then it returns the client a new (identifier, address, Relocator) tuple, as any of these may have changed from the client's current values. The client may then use the specified address. This will succeed, unless the entity has moved whilst the lookup was taking place. It is up to the client whether to try again or not.
- If the Relocator has no record, then the entity no longer exists.

47.7 Security Attacks

There are various attacks on the system. All of these can be countered, or reduced to resource attacks that are no worse than trivial resource attacks. To reduce corruption attacks in general, address records contain the identity of the hosts that the address maps to. This is the only authority that may cause this record to be changed. Other records (forward, remap, and universal remap) need never be changed by a third party, so no authority information is stored.

Attacks may be made one three classes of malicious agent

47.7.1 Attacks by a Malicious Host

A malicious host can destroy or corrupt any entities moved to it. This is to be expected and is unavoidable. A host may also claim to have entities it does not (to a client). This can only be used to attack use of entities that once existed on the host. This is (arguably) no worse than a corruption attack on the entity when it was at the host. For other entities, the client will only contact hosts that it is directed to contact via the Relocator.

A malicious host may enter identifier to address mappings into a Relocator that intentionally clash with existing mappings, or mappings that may be correctly added in future. This will lead to the creation of additional remap records within the Relocator, but will only lead to (at worst) two remap records per attack, in addition to an address record for the entered mapping. This is not significantly worse than a straightforward 'bombardment' resource attack, where a host simply adds millions of entries to a Relocator. It will lead to an additional cost during lookup of the attacked identifier, of (at worst) two records. This is again not significant.

These resource attacks are mitigated by the use of many Relocators. A particular Relocator is likely to only support specific trusted hosts (for example those within an administrative domain). This reduces the likelihood of attacks. As the algorithm allows the creation of additional mutually distrustful Relocators, it is reasonable to force an untrusted party to create their own Relocator, rather than have them use a critical resource.

47.7.2 Attacks by a Malicious Relocator

A malicious Relocator is free to misdirect any of its clients, and corrupt or discard any information stored in it. This is to be expected, and for this reason, hosts should only use Relocators that they trust. Beyond this, Relocators are not privileged, and in particular cannot affect the use of other Relocators.

When the Relocator associated with an entity is changed, then either Relocator can prevent the correct resolution of that entity. The Relocators cannot otherwise adversely affect each other (other than by claiming a clash, and forcing the creation of a universal remap record). Here, and in other cases where parties must agree on a fresh identifier a simple iterative algorithm is used, and if agreement cannot be found after a statistically significant number of iterations, then the operation is abandoned. The probability of, say, ten randomly chosen identifiers all clashing is extremely low.

47.7.3 Attacks by a Malicious Client

A malicious client has little power. It may make 'nuisance' calls to a host or Relocator, but cannot change information stored in the Relocator. Relocators may therefore freely allow client access – providing the simple bombardment attack can be countered. There is no need to authenticate clients.

47.8 Optimisations

There are three classes of optimisation to the standard algorithm.

Client Tracking
: When a client is informed of the current identifier, address or Relocator for an entity, it may store these, to reduce the cost of subsequent lookups. This optimisation is assumed in normal operation.

Relocator Tracking

A Relocator may examine records it contains. It may search for chains of records and shorten these, to speed subsequent lookups. These may be local chains of Remap and Universal Remap records, distributed chains of forward records, or combinations of the two. When shortening chains, all entries are made to point to the final entry (or latest entry in the current Relocator), however no entries may be deleted as a client may still hold tuples containing information they match.

A Relocator may examine records periodically, or as a side effect of client lookups.

Leasing

A final optimisation allows a Relocator to ultimately discard forwarding and remap references, at the cost of periodic communication between *all* clients and Relocators. A Relocator 'leases' names to clients, and a client is responsible for contacting the Relocator for a new lease, before the old one expires. As a side effect of doing this, the client is supplied with up to date information about the name. The Relocator may then discard remap and forward records that have no outstanding leases.

This optimisation is significant as it allows a Relocator to completely forget a name it no longer manages, and recover all state associated with it. However, it requires co-operation from all clients. This precludes the use of clients who are disconnected for periods longer than the lease period. It is debatable whether this complexity is justified, especially considering that forward and remap references never need updating, and so may be archived by a Relocator, reducing the gain from deleting them.

47.9 Destroying Entities

There are several issues surrounding the destruction of an entity. Two alternatives may be taken:

- The Relocator(s) remove all knowledge of the entity
- The Relocator stores a 'tombstone record' indicating that the entity has been destroyed.

The former approach is preferable, however it removes the possibility of detecting clashes with destroyed entities. A client may have a name for an entity e_1, and then be unable to determine that is has been destroyed, and a new entity e_2 is accidentally/maliciously using the same identifier and Relocator. This may (or may not) be considered an important issue.

The second approach solves this problem, but at the cost of records that the Relocator can never destroy.

The ‘solution’ to this dilemma is to use time-limited identifiers. Each identifier is extended to contain its creation date. This is then given a lifetime. If during this lifetime, the entity is destroyed, then the Relocator keeps a tombstone for it until after the lifetime expires. If the entity survives until after its lifetime, then on the next change of Relocator, its identifier is treated as having clashed. This is because the Relocator may have previously had a tombstone for that identifier, which has been discarded. A sixteen-bit creation time would allow a 180 years of daily timestamps. Given that the Relocator is rarely changed this will lead to few additional remap records, and no tombstone records that last more than a day.

For entities that use an authentication mechanism, this is not required, as a client cannot accidentally use the wrong entity, and the reuse of identifiers will not lead to problems.

47.10 Discussion

The algorithm presented is relatively complex, but straightforward to implement, and efficient in use. More importantly, it does not require any global co-ordination between the parties (clients, hosts and Relocators), and has no reliance on ‘truly unique’ identifiers. This is a significant strength over other systems that require co-ordination, or that assume hosts or domains have unique identifiers or closely synchronised clocks.

The fact that the system copes with, rather than prevents, identifier clash allows it to be more secure, and to be more resilient to accidental clashes. This allows the use of small identifiers, where the likelihood of clash is perhaps 1%, rather than large identifiers that are required in systems where clash leads to failure and must be avoided at all cost.

The algorithm can also be used in a second form. Rather than store the reference to the Relocator explicitly in the name tuple, it may be inferred from the address field, with one Relocator used for all entities stored in a set of hosts (for example a sub-net). This implies that the Relocator is changed whenever an entity moves from one set of hosts to another, but is a reasonable optimisation if this is a rare occurrence, and the size of the name is considered more important.

REFERENCES

[BLAIR97]	Blair G.S., Stefani, J.B. Open Distributed Processing and Multimedia. Published by Perseus Pr; ISBN: 0201177943 (1997)
[BURSELL98]	Bursell, M.H., Hayton R.J., Donaldson, D. Herbert A.J. A Mobile Object Workbench. Mobile Agents '98.
[CM84]	J.M.Chang and N.F.Maxemchuk. Reliable Broadcast Protocols. *ACM Transactions on Computer Systems*, 2(3):251-273, August 1984.
[FAST97]	FAST (1997) FollowMe project overview *http://hyperwav.fast.de/generalprojectinformation*
[HAYTON98]	Hayton R.J, Bursell M.H., Donaldson D., Herbert A.J. Mobile Java Objects, Middleware '98
[HERBERT98]	Herbert et.al. DIMMA – A Multi-Media ORB, Middleware'98
[IAIK]	iSaSiLk. IAIK SSL for Java *http://jcewww.iaik.tu-graz.ac.at/index.htm*
[IONA]	Iona Technologies: OrbixWeb *http://www-usa.iona.com/products/internet/orbixweb/*
[IONA97]	Iona Technologies, Orbix Programmers Guide (Chapter 16,22) (1997) *http://www.iona.com/*
[ISO95]	International Standards Organisation: Open Distributed Processing - Reference Model. Sep. 1995 *http://www.iso.ch:8000/RM-ODP/*
[KICZALES91]	G. Kiczales, J. des Rivieres, and D. G. Bobrow: The Art of the Metaobject Proto- col. MIT Press, 1991.
[LAMPORT78]	L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System," Comm. of the ACM 21,7 (July 1978)
[MAES87]	P. Maes: Concepts and experiments in computational reflection. In OOPSLA '87 Proceedings, pages 147--155, October 1987. [OMG98a] Objects by Value, Joint Revised Submission. Object Management Group, Inc. ORBOS/98-01-01.

[MICROSOFT] Microsoft Transaction Server White Paper *http://www.microsoft.com/transaction/learn/mtswp.htm*

[O'CONNEL94] O'Connell, J. Edwards, N. and Cole, R. A review of four distribution infrastructures. Distributed Systems Engineering 1 (1994) 202-211

[OMG94] Object Transaction Service. OMG document 94.8.4, August 1994.

[OMG97a] Object Management Group (1997) CORBA/IIOP 2.1 Specification *http://www.omg.org/corba/corbiiop.htm*

[OMG97b] IDL/Java Language Mapping, Joint Revised Submission. Object Management Group, Inc. ORBOS/97-03-01.

[OMG98b] Java to IDL Mapping, Joint Submission. Object Management Group, Inc. ORBOS/98-01-07.

[OMG98c] The Common Object Request Broker: Architecture and Specification, revision 2.2. Object Mangement Group, Inc. ORBOS/98-02-33.

[ORL98] Sai-Lai Lo and S.Pope, The Implementation of a High Performance ORB over Multiple Network Transports. Middleware 98.

[RMP] The Reliable Multicast Protocol *http://research.ivv.nasa.gov/RMP/*

[SUNa] Sun Microsystems: Java Core Reflection *http://java.sun.com/products/jdk/1.1/docs/guide/reflection/index.html*

[SUBb] Sun Microsystems: JavaBeans *http://java.sun.com/beans/*

[SUNc] Sun Microsystems: Enteprise JavaBeans *http://www.javasoft.com/products/ejb/docs.html*

[SUNd] The Beans Development Kit *http://java.sun.com/beans/software/bdk_download.html*

[SUN96] Sun Microsystems (1996) Java Remote Method Invocation (RMI) Specification *http://www.sun.com/products/jdk/1.1/docs/guide/rmi/*

[SYBASE] Jaguar: Java Components and Transactions. Byte, February 1998

[WEIHL85] W. E. Weihl and B. Liskov: Implementation of resilient, atomic data types. ACM Transactions on Programming Languages and Systems, 7(2):244--269, April 1985.

APPENDIX I – PACKAGE HIERARCHY

1.1 UK.co.ansa

Top level package prefix for ANSA related work. Classes in this package and its sub-packages are Copyright © Citrix Systems (Cambridge) Ltd, on behalf of the sponsors for the time being of the ANSA project.

1.2 UK.co.ansa.flexinet

FlexiNet packages and classes, as described the majority of this document.

1.2.1 UK.co.ansa.flexinet.core

Central concepts and abstractions.

- `UK.co.ansa.flexinet.core.naming`
  Naming concepts and abstract implementations.
- `UK.co.ansa.flexinet.core.call`
  Classes defining Invocations and the CallUp/CallDown abstraction.
- `UK.co.ansa.flexinet.core.resource`
  Definitions of resources and pools.

1.2.2 UK.co.ansa.flexinet.basecomms

Standard implementation components used by the majority of binders and protocol stacks.

- `UK.co.ansa.flexinet.basecomms.serialize`
  Serialisation abstraction.
  - `UK.co.ansa.flexinet.basecomms.serialize.engine`
    Implementation of the basic serialisation engine.
  - `UK.co.ansa.flexinet.basecomms.serialize.ref`
  - `UK.co.ansa.flexinet.basecomms.serialize.stub`
  - `UK.co.ansa.flexinet.basecomms.serialize.sun`
    Specialisations of the basic serialiser, that provide additional functionality.

  - `UK.co.ansa.flexinet.basecomms.serialize.classname`
    Class serialiser that uses a class's name to identify it.
  - `UK.co.ansa.flexinet.basecomms.serialize.layers`
    Layers of a protocol stack that relate to serialisation.

- `UK.co.ansa.flexinet.basecomms.layers`
  Commonly used layers of protocol stacks.
  - `UK.co.ansa.flexinet.basecomms.layers.echo`
  - `UK.co.ansa.flexinet.basecomms.layers.trivname`
  - `UK.co.ansa.flexinet.basecomms.layers.gname`
  - `UK.co.ansa.flexinet.basecomms.layers.tcp`
  - `UK.co.ansa.flexinet.basecomms.layers.udp`
  - `UK.co.ansa.flexinet.basecomms.layers.call`
  - `UK.co.ansa.flexinet.basecomms.layers.session`

- `UK.co.ansa.flexinet.basecomms.binders`
  Common protocol independent binders.
  - `UK.co.ansa.flexinet.basecomms.binders.cache`
  - `UK.co.ansa.flexinet.basecomms.binders.choice`
  - `UK.co.ansa.flexinet.basecomms.binders.generic`

- `UK.co.ansa.flexinet.basecomms.socket`
  Socket implementations.

- `UK.co.ansa.flexinet.basecomms.stub`
  Stub generation
  - `UK.co.ansa.flexinet.basecomms.stub.generator`
  - `UK.co.ansa.flexinet.basecomms.stub.bytecode`

- `UK.co.ansa.flexinet.basecomms.buffer`
  Segmented buffer abstraction.
  - `UK.co.ansa.flexinet.basecomms.buffer.basic`
    A concrete implementation used by most protocols.

1.2.3 UK.co.ansa.flexinet.protocols

A top level package for protocol and binder specifications. Each sub package defines a particular protocol, binder or protocol family.

- `UK.co.ansa.flexinet.protocols.rex`
  The REX protocol family.
  - `UK.co.ansa.flexinet.protocols.rex.layers`
  - `UK.co.ansa.flexinet.protocols.rex.layers.rex`
    Protocol stack layers required by REX binders.
  - `UK.co.ansa.flexinet.protocols.rex.binders`
  - `UK.co.ansa.flexinet.protocols.rex.binders.green`
  - `UK.co.ansa.flexinet.protocols.rex.binders.yellow`
    REX-based binders.

- `UK.co.ansa.flexinet.protocols.requestreply`
  The RRP protocol family.

  - `UK.co.ansa.flexinet.protocols.requestreply.layers`
    Protocol stack layers required by RRP binders.
  - `UK.co.ansa.flexinet.protocols.requestreply.binders`
    RRP-based binders.

- `UK.co.ansa.flexinet.protocols.giop`
  The GIOP protocol family.
  - `UK.co.ansa.flexinet.protocols.giop.layers`
  - `UK.co.ansa.flexinet.protocols.giop.serialize`
  - `UK.co.ansa.flexinet.protocols.giop.buffer`
  - `UK.co.ansa.flexinet.protocols.giop.corba`
  - `UK.co.ansa.flexinet.protocols.giop.util`
    Resource implementations required by GIOP binders
  - `UK.co.ansa.flexinet.protocols.giop.binders`
  - `UK.co.ansa.flexinet.protocols.giop.binders.giop`
  - `UK.co.ansa.flexinet.protocols.giop.binders.iiop`
    GIOP-based binders.

- `UK.co.ansa.flexinet.protocols.group`
  Group-based (RMP) protocol.
  - `UK.co.ansa.flexinet.protocols.group.call`
  - `UK.co.ansa.flexinet.protocols.group.layers`
  - `UK.co.ansa.flexinet.protocols.group.binders`
  - `UK.co.ansa.flexinet.protocols.group.test`
  - `UK.co.ansa.flexinet.protocols.group.java2util`

- `UK.co.ansa.flexinet.protocols.negotiator`
  Negotiation-based binder.
  - `UK.co.ansa.flexinet.protocols.negotiator.binders`
  - `UK.co.ansa.flexinet.protocols.negotiator.layers`

1.2.4 UK.co.ansa.flexinet.cluster

Implementation of the ‘cluster’ abstraction. The top-level package contains the abstract interfaces, and the sub-packages provide implementations.

- `UK.co.ansa.flexinet.cluster.binders`
  Cluster and Capsule binder implementations.
- `UK.co.ansa.flexinet.cluster.naming`
  Cluster related naming classes.
- `UK.co.ansa.flexinet.cluster.layers`
  Protocol stack layers relating to clusters.
- `UK.co.ansa.flexinet.cluster.comms`
  Other communications components.
- `UK.co.ansa.flexinet.cluster.manager`
  Implementations of cluster managers.
- `UK.co.ansa.flexinet.cluster.awt`
  AWT-wrapper package.

- `UK.co.ansa.flexinet.cluster.events`
  Cluster-friendly event package.

1.2.5 UK.co.ansa.flexinet.util

Utility packages and classes. These are components of FlexiNet that do not fit into the rest of the package structure. Util has the following sub-packages.

- `UK.co.ansa.flexinet.util.properties`
  The FlexiProps scheme – mainly used for specifying QoS constraints.
- `UK.co.ansa.flexinet.util.queue`
  Priority queue and timer queue implementations.
- `UK.co.ansa.flexinet.util.pool`
  Various pool manager implementations, for different policies.
- `UK.co.ansa.flexinet.util.applet`
  Code to aid the use of FlexiNet in applets.
- `UK.co.ansa.flexinet.util.debug`
  Replacements for standard components that perform additional tracing or validation.
- `UK.co.ansa.flexinet.util.thread`
  Thread resource abstraction, for thread pools.
- `UK.co.ansa.flexinet.util.cache`
  Various cache implementations (e.g. leaky bucket).
- `UK.co.ansa.flexinet.util.blueprint`
  Blueprint system used for binder specification and construction.
- `UK.co.ansa.flexinet.util.locale`
  Localisation system, including example code.

1.2.6 UK.co.ansa.flexinet.services

Standard FlexiNet services and utility programs.

- `UK.co.ansa.flexinet.services.trivtrader`
  The Trivial trader.
- `UK.co.ansa.flexinet.services.sslcert`
  A utility program for generating SSL certificates.
- `UK.co.ansa.flexinet.services.classrepository`
  The class repository service. The top-level package contains definitions for the basic components. There are three sub-packages.
  - `...flexinet.services.classrepository.repository`
    The repository service itself.
  - `...flexinet.services.classrepository.serialize`
    A class serialiser and deserialiser that pass references to classes stored in the repository.

- `...flexinet.services.classrepository.client`
  Client side class manager; a local interface to the repository.

1.3 UK.co.ansa.mobility

Packages relating to the ‘mobile object’ abstraction. This work was conducted under the ‘Mobile Object Workbench’ package of the FollowMe project.

The top-level package contains the definitions for the main components. These are specialisations of the cluster abstraction.

- `UK.co.ansa.mobility.secure`
  Mobile object security. In particular, per-cluster security managers.
- `UK.co.ansa.mobility.namer`
  An implementation of the ‘mobile namer’ relocation service for (mobile) clusters.

1.4 UK.co.ansa.ispace

Packages relating to the ‘information space’ abstraction. This provides persistent objects accessible transparently, or via a directory interface. This work was conducted under the ‘Information Space’ package of the FollowMe project.

The top-level package contains the definitions for the main components. These are specialisations of the cluster abstraction.

- `UK.co.ansa.ispace.test`
  Test code.
- `UK.co.ansa.ispace.util`
- `UK.co.ansa.ispace.util.file`
  Utility code.
- `UK.co.ansa.ispace.service`
  Definition of the ‘StoreFactoryImpl’ service.
- `UK.co.ansa.ispace.manager`
  Cluster managers for storables, and other management code.

1.5 UK.co.ansa.beanbox

This is a visual assembly tool for EJBs. It is based on the BDK beanbox, but modified to support Enterprise Java Beans.

1.6 UK.co.ansa.transaction

The FlexiNet transactional system for Enterprise Java Beans.

- `UK.co.ansa.transaction.concurrency`
  Two phase locking concurrency control implementation.
- `UK.co.ansa.transaction.container`
  EJB container for FlexiNet transactions.
- `UK.co.ansa.transaction.javaxImpl`
- `UK.co.ansa.transaction.javaxImpl.ejb`
- `UK.co.ansa.transaction.javaxImpl.jts`
  FlexiNet implementation of standard EJB interfaces.
- `UK.co.ansa.transaction.reflect`
- `UK.co.ansa.transaction.reflect.transactional`
- `UK.co.ansa.transaction.reflection`
  Distributed transaction implementation.
- `UK.co.ansa.transaction.rmi`
  proxies to support the passing of transactional context.
- `UK.co.ansa.transaction.util`
  Utility code.

1.7 javax

An implementation of classes defined in the “Java Extensions” specification. These are required for the FlexiNet transaction system.

1.8 sunw

A subset of Sun’s BDK distribution. Required for the FlexiNet transactional beanbox, and beans it creates.

1.9 org.omg

Top level package for classes defined by the Object Management Group (OMG). These classes and packages are defined in the OMG standard, and include classes relating to the ‘Objects by Value’ RFP. A small number of classes have been modified for use in FlexiNet.

- `org.omg.CONV_FRAME`
- `org.omg.CORBA`
- `org.omg.IIOP`

- `org.omg.IOP`
- `org.omg.GIOP`
  CORBA related classes required for IIOP binder
- `org.omg.CosTransactions`
  CORBA Object Services transaction classes required for the FlexiNet transaction system.

1.11 Licensing and Libraries

The vast majority of the code in the FlexiNet distribution is a clean room implementation and Copyright © Citrix Systems (Cambridge) on behalf of the sponsors for the time being of the ANSA Consortium. FlexiNet is an ANSA deliverable, and is covered by the standard ANSA Consortium licensing terms. Code based on, or requiring access to, none ANSA code is listed below.

- Java 1.1
  FlexiNet requires access to many of the standard Java 1.1 classes. These are not supplied as part of the FlexiNet distribution. FlexiNet will also compile against Java 1.2, however there has been limited testing in this environment, and in particular, the IIOP binder will not function, as it requires a different version of the standard OMG classes.
- IAIK SSL
  Some parts of FlexiNet require the IAIK libraries in order to function [IAIK]. These are not supplied as part of the FlexiNet distribution, but must be licensed from IAIK. If the IAIK libraries are not available, then the following classes will fail to compile, and SSL based protocols and utilities will be unavailable.
  - `UK.co.ansa.flexinet.basecomms.socket.ConfigurableSocketFactory`
  - `UK.co.ansa.flexinet.services.sslcert.CertificateGUI`
  - `UK.co.ansa.flexinet.services.sslcert.TempKeyGUI`
- Java Bean Development Kit (BDK)
  The Enterprise Bean Box is based on Sun's BDK [SUNd]. The modified version is supplied as part of the distribution (in package `UK.co.ansa.beanbox`). The BDK licence is unclear about the legality of distributing this outside of the consortium, and consortium members as requested to consult the BDK licence before doing so. In addition to the classes in `UK.co.ansa.beanbox`, the classes in package `sunw.beanbox` are copied from the BDK distribution.

  These classes are required for the construction of EJBs used in the FlexiNet transactional framework.
- OMG Classes (package `org.omg`)
  These classes are either taken from publicly available distributions of OMG classes, or implemented from scratch based on OMG

specifications. There are not believed to be any licensing issues related to their use. They are used in the IIOP binder, and in the transactional framework.

- Javax Classes (package `javax`)
  These are a clean room implementation based on published specifications. They are used by the transactional framework.