APM

POSEIDON HOUSE • CASTLE PARK • CAMBRIDGE CB3 0RD UNITED KINGDOM
+44 1223 515010 • Fax +44 1223 359779 • Email: apm@ansa.co.uk • URL: http://www.ansa.co.uk

ANSA Phase III

FlexiNet Open ORB Framework

Richard Hayton

Abstract

This document describes the architectural approach taken in the design of the FlexiNet ORB framework. This provides a transparent and highly modular remote procedure call transport, which will be utilised by the FlexiNet project to expose binding decisions to application policy.

2047.01.00	**Approved** Technical Report	06 October 1997

Distribution:
Supersedes:
Superseded by:

This paper makes reference to a number of trademarks. The ownership of all such trademarks by their respective organisations is hereby acknowledged.

TABLE OF CONTENTS

1 INTRODUCTION

1.1 Overview

The FlexiNet ORB framework forms the lower layers of the FlexiNet architecture. FlexiNet is an architecture for the dynamic association of infrastructure components to support application deployment in mobile code environments. The focus of the FlexiNet ORB framework is an open binding model with the emphasis on the reuse of components, heterogeneous mechanism and the ability to enforce policy constraints. The framework consists of an architectural design, and a set of component classes that are sufficient to implement remote method invocation. Future work will add to this set of classes, to provide additional functionality, and to exploit the flexibility provided by this framework.

1.2 Current Functionality

The framework can provide communications functionality similar to that provided by RMI[1] or other packages. FlexiNet provides four major advantages over other approaches

- Transparency

  FlexiNet allows remote access to any *public* interface on any accessible object. That is any interface that has been exported, either explicitly, or implicitly by referencing it during another remote call. This approach can be used to turn existing objects into services without the need to modify (or access) the source code. It also simplifies the task of writing distributed applications as there is little additional syntax over standard Java.

- Interface based

  In FlexiNet *interfaces* are the points of access, not objects. It is possible for an object to implement many interfaces, and to implement access to these interfaces via different communications mechanisms. This is closer to the ODP model than other RPC systems, leading to more natural implementation of distributed applications.

- Security

FlexiNet provides dynamic stub creation *on the client*. Unlike other approaches this does not require the downloading of stub code from server to client. This removes a potential security risk. The current incarnation of the FlexiNet ORB framework does not provide security functions such as proof of authenticity or secrecy, although these may be added as optional component layers in future.

- Flexibility

The FlexiNet ORB framework is designed as a set of components to be dynamically assembled by other parts of the FlexiNet framework. Components are highly independent, considerably simplifying the task of tailoring the framework to suit particular application classes.

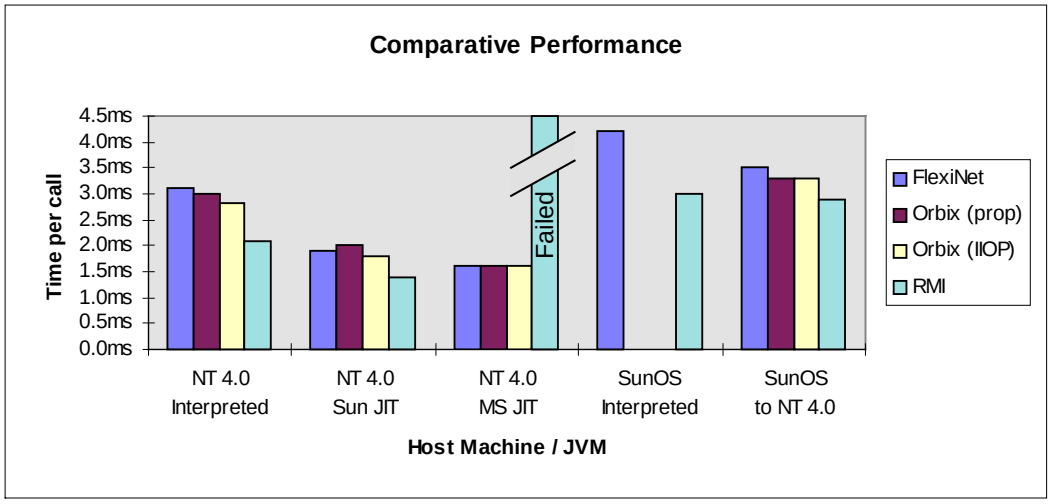
1.3 Performance

The framework has been tested by performing repeated remote invocations between a single client and a single service. The method chosen was the echoing of a null string, as this demonstrates the minimal RPC overhead, and is used as a standard benchmark for distributed object systems. It should be noted that benchmarks were not taken under ideal conditions. RMI[1] and OrbixWeb[2] measurements are given for comparison.

Host Machine / JVM	**FlexiNet**	**OrbixWeb**		**RMI**
	REX	Prop.	IIOP	Prop.
NT 4.0 Pentium Pro 200MHz Ethernet Java 1.1.3 interpreted	3.1ms	3.0ms	2.8ms	2.1ms
NT 4.0 Pentium Pro 200MHz Ethernet Java 1.1.3 Sun JIT	1.9ms	2.0ms	1.8ms	1.4ms
NT 4.0 Pentium Pro 200MHz Ethernet Java 1.1.3 Microsoft JIT	1.6ms	1.6ms	1.6ms	Failed
SunOS 5.5 Ultra 1 167MHz Same Machine Java 1.1.3 interpreted	4.2ms	not tested	not tested	3.0ms
SunOS 5.5 to NT 4.0 Ethernet Java 1.1.3 interpreted	3.5ms	3.3ms	3.3ms	2.9ms

The FlexiNet ORB framework can support multiple protocols. The protocol illustrated is based on the REX protocol over UDP. This has several advantages over TCP based protocols (such as IIOP), in particular it can support larger numbers of clients, and allows more control over resource allocation. However, the disadvantage is that reliable message passing must

be implemented above the socket layer (i.e. in Java) rather than in the operating system kernel (in C,C++ or assembler). This explains the difference in performance between REX and the other protocols. There is no reason why IIOP should not be supported by FlexiNet, and a FlexiNet implementation would be expected to give similar performance to RMI.



1.4 Future Work

This framework document describes the key concepts for the assembly of communications components. The ORB framework distribution consists of sufficient components to perform basic RPC operations. Future work will concentrate on the provision of specialist components (for example security components) and on the specification and automation of assembly of these components to build a distributed application.

2 ARCHITECTURE

2.1 Introduction

The framework consists of a number of components making up a protocol stack, together with resource management classes and a generalised naming model. In addition a temporary API has been created to allow test programs to be written and to demonstrate how the framework may be used in place of other RPC transports such as RMI. This API will not form part of FlexiNet proper.

2.2 Protocol Stack

The protocol stack consists of a number of layers that transform a call from a simple invocation on the client, through a generic invocation, a marshalled invocation to a "conversation" of messages passed between client and server. On the server the call is received as a number of messages and passes up through the layers to ultimately result in a call on a particular interface on a particular object.

An important design issue was ensuring that the layers remained highly autonomous, so that individual layers could be replaced, or additional layers inserted. For example, it is possible to replace the naming scheme used within FlexiNet, in order to inter-work with other systems, or to replace the protocol used for message passing.

There are a number of different styles of communication that take place in the protocol stack. Figure 1 illustrates the basic protocol stack and indicates where additional functionality may be placed for a particular protocol.

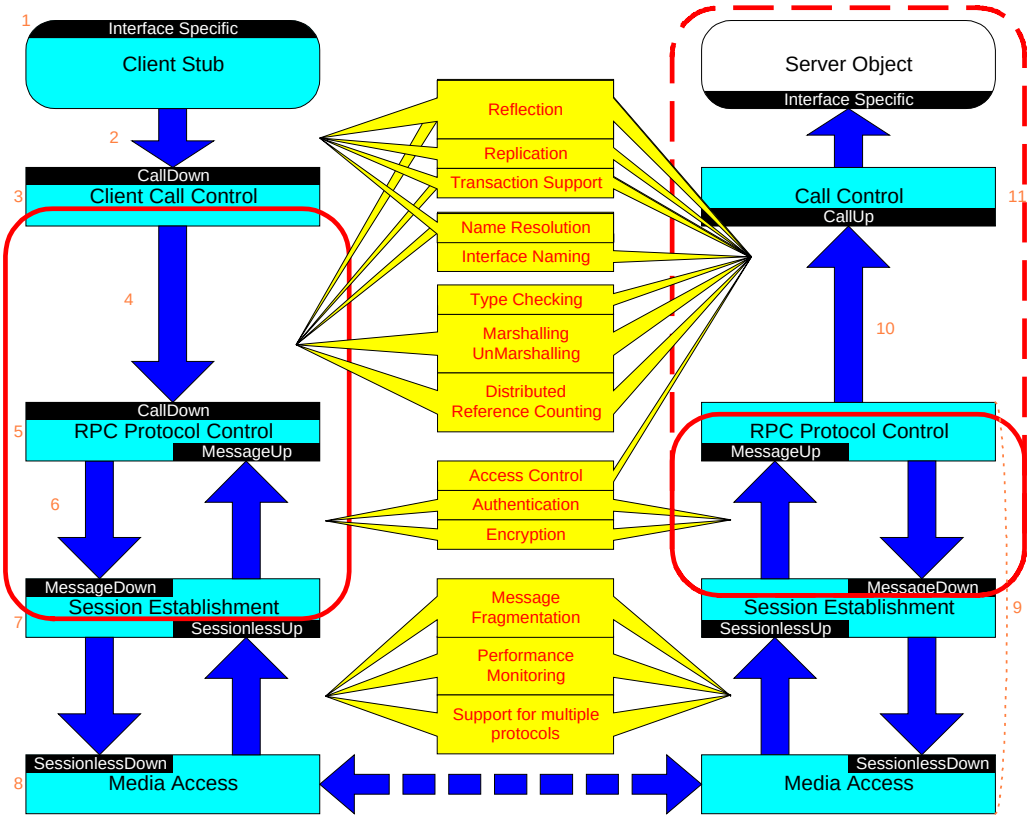


Figure 1

2.3 A Walk-through from Client to Server

Using Figure 1, this section describes the functions provided by the different protocol layers, and follows 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[3]. 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. Normally the call will eventually pass to the “Client Call Layer”, however for a local, reflective call, the call may instead by diverted directly to the (Server) Call Control layer, and for replicated objects, the call may be multiplexed to a number of calls on the same or different protocol stacks.

3. The “Client Call Layer” will choose an appropriate session on which to perform the call. The session is a grouping of a number of invocations that

share the same service endpoint, and a session abstraction is common in RPC protocols. Other layers may utilise the session structure to cache information relating to a particular endpoint (for example encryption keys). In Figure 1, the red boxes indicate the layers that may make use of the session in this way.

4. The next few layers are responsible for functions such as marshalling the data so that when the call reaches the RPC Protocol Layer the call is in the form of a request message that is to be paired with a reply.

5. Below the RPC layer, communication is via message passing and the RPC layer is responsible for pairing requests with replies, and managing acknowledgement or other protocol messages. It is also responsible for dealing with timeouts, and deciding when a session should be abandoned.

6. Further layers perform processing that is message, rather than call, specific. For example encryption or compression.

7. The outgoing message ultimately reaches the Session Establishment Layer. For outgoing client messages, this layer simple encodes the session id in the message, to allow the session to be re-established on the server.

   The Session Establishment Layer marks the lower boundary of the region of mutual exclusion. Only one call or message per session will be in progress within the red session zone. This simplifies the coding of session related functions, and reduces the number of race conditions (such as duplicate messages or simultaneous messages and timeouts).

8. The final client layer is a media access layer. This performs reads and writes on a socket. Once the request has been written the client thread will typically return to the RPC layer where it will block until it receives a matching reply.

9. On the server we will examine message flow from the wire to the object and back. A received message passes through similar Media Access and session establishment layers to reach the RPC Protocol Layer. This will typically spawn a new thread to manage the request, allowing the initial thread to service further incoming requests. However, for some services, the call may proceed inline. This will depend on the RPC Protocol Layer being used, and the management of the thread pool (see later).

   The RPC Protocol Layer marks the upper region of mutual exclusion for sessions on the server. Only one thread per session will be active within the red session zone. There is a second, dashed, zone in which the session remains available. However the session is *not* locked in the dashed zone. This is because the session may be required for protocol exchanges between the peer RPC layers whilst the call is still in progress. As the session cannot be timed out whilst the call is in progress, the session may be safely used, for example to store the identity of the client.

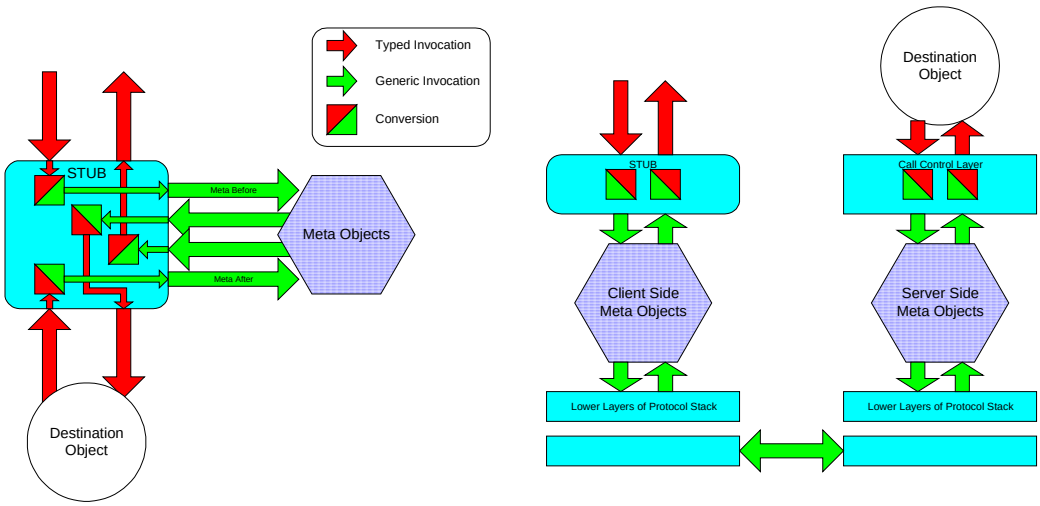
10. Layers above the RPC layer will unmarshall the call, and translate from the supplied address to the destination object. Additional functions such

as method matching (for type conformance) and access control may also take place. There may also be server side reflective layers.

11. The final layer to be called is a Call Control Layer. This will translate the generic call into a call on the server object using Java Core Reflection. The result (normal or exception) is processed in a similar fashion and is returned to the client thread.

2.4 Reflection

Reflection is a technique whereby objects are split into a number of parts; the core object, which performs the function of the object, and a number of *meta-objects* which are responsible for non-functional aspects. Meta-objects might be responsible for ensuring transactional properties, managing replication, or performing access control. Reflective techniques allow a separation of concerns. For example it is possible to change the transactional meta-object without recourse to re-implementing the core object. More information about reflection can be found in [4]. Method reflection is implemented by intercepting all method calls to an object, and passing the arguments to a `meta_before` method for acceptance testing and possible modification, and then performing a `meta_after` call to perform meta transformations of the reply. This is illustrated in Figure 2(a). FlexiNet enables reflection and distribution to go hand in hand, and provides a framework for reflective execution in the upper layers of the protocol stack. This framework supports both client side and server side reflection, in addition to reflection for local objects. This is illustrated by Figure 2(b).



(a) **Figure 2** **(b)**

This approach has the benefit that only one stub is required to perform the function of both a reflective and remote execution stub. This reduces the overhead of converting the call from a typed invocation to a generic invocation, and allows reflective techniques to be used to implement high level ORB semantics, such as replication or transactions.

2.5 Naming

The FlexiNet framework takes an open view of interface naming. There is support for multiple name spaces and multiple protocols. The basic structure is illustrated in Figure 3. There is a single global namer that implements the interface `Namer`, and which maps from interfaces to names, and from names to interfaces. If an attempt is made to name a previously unnamed interface (for example by passing it as a parameter) then the global namer will choose an appropriate `generator` to perform the translation. In the initial framework, there is only one generator, and this choice is therefore trivial. Equally, when a name needs to be resolved for the first time, the global namer will choose an appropriate `resolver` to perform the translation. The resolver will resolve the name to a local object. Typically this will be a stub object instantiated with the name of the remote interface and a reference to the appropriate protocol stack.

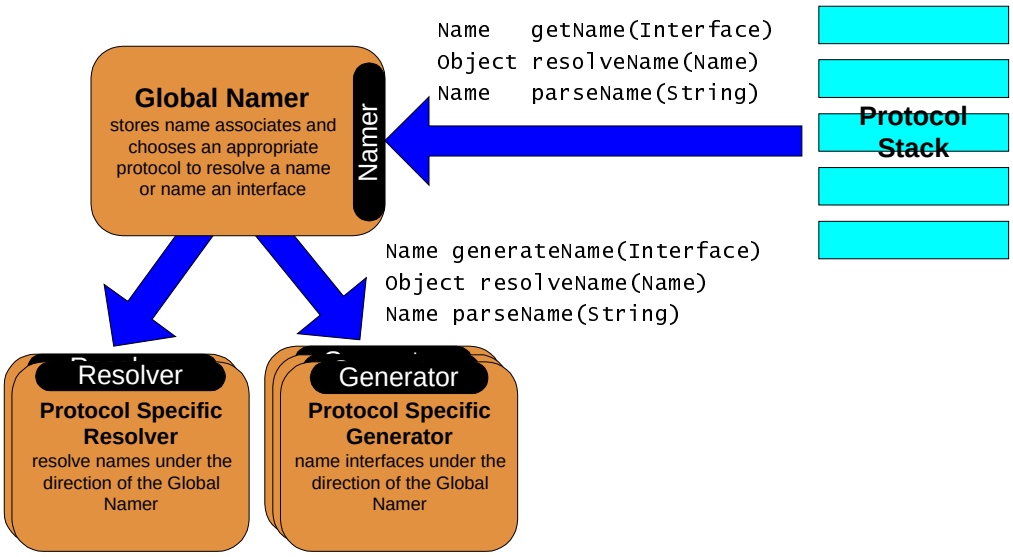


Figure 3

2.5.1 Names and Addresses

The type hierarchy for names is illustrated in Figure 4. A *name* is the most general representation of a reference to a remote object. Names may require trading to resolve them, and may represent a set of remote objects, for example a set of replicas, or a set of aliases for a remote interface. An *address* is a representation of a particular remote interface. Each address consists of a low level endpoint (for example a socket) that may be used to choose an appropriate session, and a further part that can be resolved by the remote protocol stack to identify a particular destination interface.

Layers in the protocol stack have limited knowledge about naming, and names are translated as they pass up and down the stack. For example, when an outgoing call reaches the Client Call Layer, the remote name must

be an instance of class `Address`, so that the layer can choose an appropriate session. However, the Client Call Layer does not understand the format or semantics of particular address classes, or of the endpoints associated with them. This enables the same layer to be used in protocol stacks implementing different RPC protocols, or utilising different transport mechanisms.

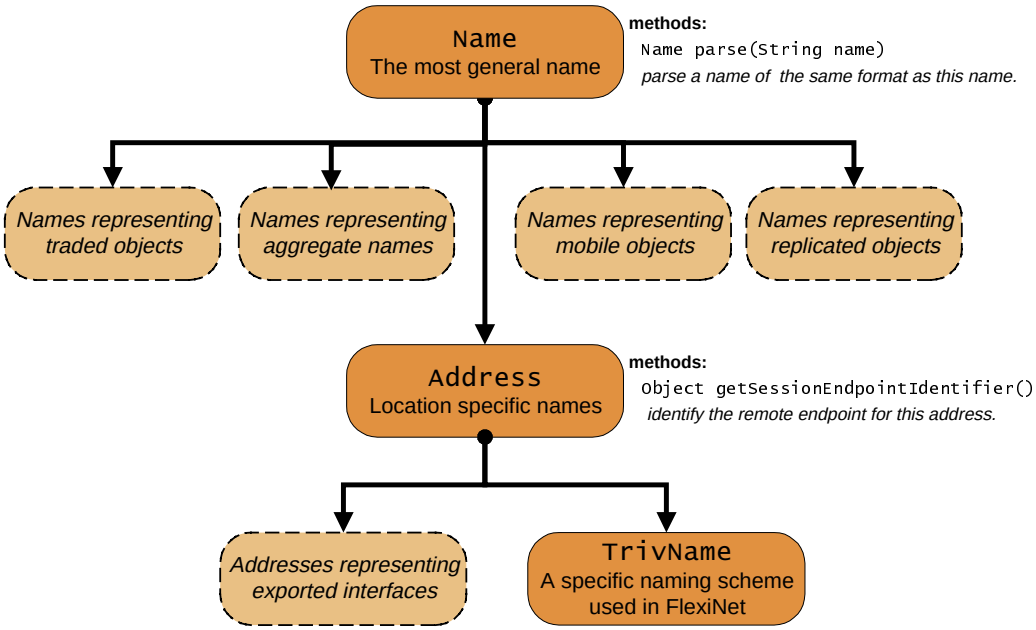


Figure 4

2.6 Layer Binding

In order to form a protocol stack, each layer must be aware of the layers with which it must communicate (typically those immediately above and below it). The layer may also require other information such as the buffer segment to use, or the pool for thread resources (see later). This information is not coded into the definition of each layer, but is provided by an initialisation method. An object called a *binder* is responsible for building a protocol stack of a particular kind, and initialising each of the layers. Currently only one binding object exists, which generates a protocol stack for REX based communication (REX is a reliable RPC protocol based on UDP). Current work is investigating other binding objects, and the automate creation of binders which fulfil particular requirements. The Binder effectively defines the communications protocol as it defines the transformations that interface references and data undergo in the processes of an invocation. In addition, as a binder specifies the transformation that an interface reference undergoes, it acts as both a `Generator` and a `Resolver`. Binders may be compatible in the sense that names generated by one namer may be resolvable by another.

2.7 Resource Management

The FlexiNet framework manages a number of resources, including threads, sessions and memory in the form of input and output buffers. There are several reasons why these resources need to be managed.

- Quality of Service Provision

  We may wish to reserve resources in advance to ensure that an agreed quality of service can be provided. Equally, we may wish to limit the resource used on behalf of a particular client, or set of clients, to ensure that a service does not become overloaded.

- Abstract Resource Allocation

  An important goal of the FlexiNet framework is to ensure that the different modules used in a protocol stack remain independent from each other, to allow code reuse and dynamic reconfiguration. It is often the case that one layer of the stack will need to create a resource that will ultimately be used by another layer. It is important that creation of resources is managed to ensure that the resource is of the appropriate sub-class. For example, the session establishment layer may need to create sessions. If the RPC Protocol Layer is a REX layer, then these must be REX sessions; whereas for a IIOP Protocol Layer, a different class of sessions is required.

- Sharing between components

  Use of mobile code may result in several independent application components residing in the same virtual machine, and hence sharing the same resources. Although managed resources cannot stop one application component from starving the other of memory, threads or other limited resources; the use of managed resources can reduce the unintentional interaction between well-behaved components.

Each FlexiNet resource class implements the interface `Resource`, and is managed by one or more `Pool` classes. The resource interface provides a `free()` method for indicating that a resource is no longer required, and the Pool interface provides a number of `get(…)` methods for obtaining resources. The `get` methods take appropriate arguments for the particular resource (for example buffers may be of a particular size). Each pool is controlled by a `PoolManager`, which is responsible for managing resource reservation and reuse. The reason for this separation is to de-couple the initialisation of a particular class of resource from the *policy* for reusing resources. There may be many pools managing resources of a particular class, each using a different instance of a pool manager. `PoolManager`s understand only generic resources, and so may be used to manage pools of any type of resource.

For example, there may be three input buffer pools each of which manages a separate pool of resources. Two of these may use `RecycleManager`s, which manage the reuse of resources, each from a different sized pool. The third input buffer pool may use an `AllocManager`, which creates a fresh resource instance whenever needed, from a conceptually infinite pool.

This approach de-couples generic resource management (reuse, reservation etc.) from type specific resource management (creating instances, initialising resources etc.) and allows abstract resource allocation. Figure 5 gives an example of the classes used in the allocation of REX Sessions.

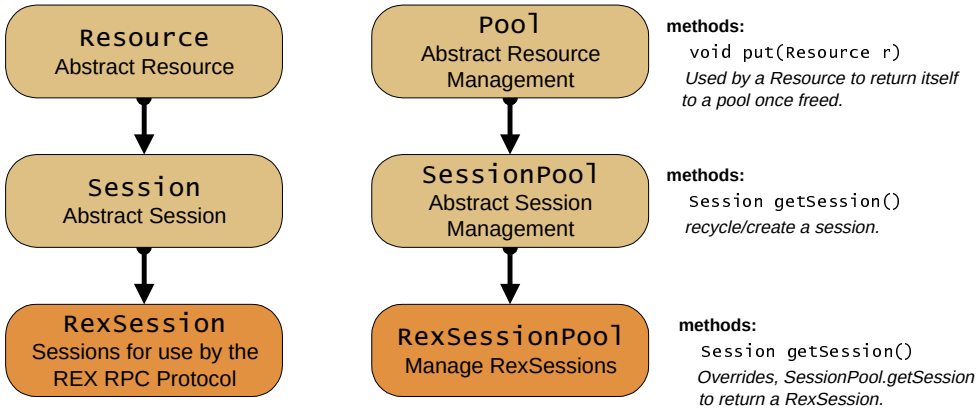


Figure 5

2.7.1 Resource management under load

When a service is heavily loaded, resources may be scarce. Different resource pool managers may implement different strategies for managing resources, for example by placing an upper limit on the number of active sessions in order to reduce thrashing between calls in progress. In addition, when resources are requested, there are two forms of `get()` call. The first, `getResourceB()` is blocking. If no suitable resource is currently available, the thread will block until such time that there is. The second, `getResourceNB()` is non-blocking; if a suitable resource is unavailable, then null is returned, and the caller must act appropriately (for example by dropping the request).

2.7.2 Buffers

There are two distinct buffer types in the FlexiNet framework, `InputBuffer` and `OutputBuffer`. This is due to the distinction between input and output streams used by the core `java.io` classes. Other that this, input and output buffers are almost identical. When designing buffers, the intention was to make it as easy as possible for different layers in the protocol stack to insert and remove data from the buffer, without the need to understand the format of data used by other layers. This is essential if protocol stacks are to be built dynamically. An additional goal was to keep the amount of data copying to a minimum, to ensure that the system is as efficient as possible.

The buffer system is based on a multi-segment abstraction of a single byte array. Each layer is aware of the segment(s) that it utilises, and may use the buffer as an input or output stream to read from, or writing to, these segments. Segments are generally fixed size, although there is the ability to extend buffers or remove empty space. The current REX based protocol

implemented in the FlexiNet framework uses three fixed sized segments, and one variable sized segment. This is illustrated in Figure 6.

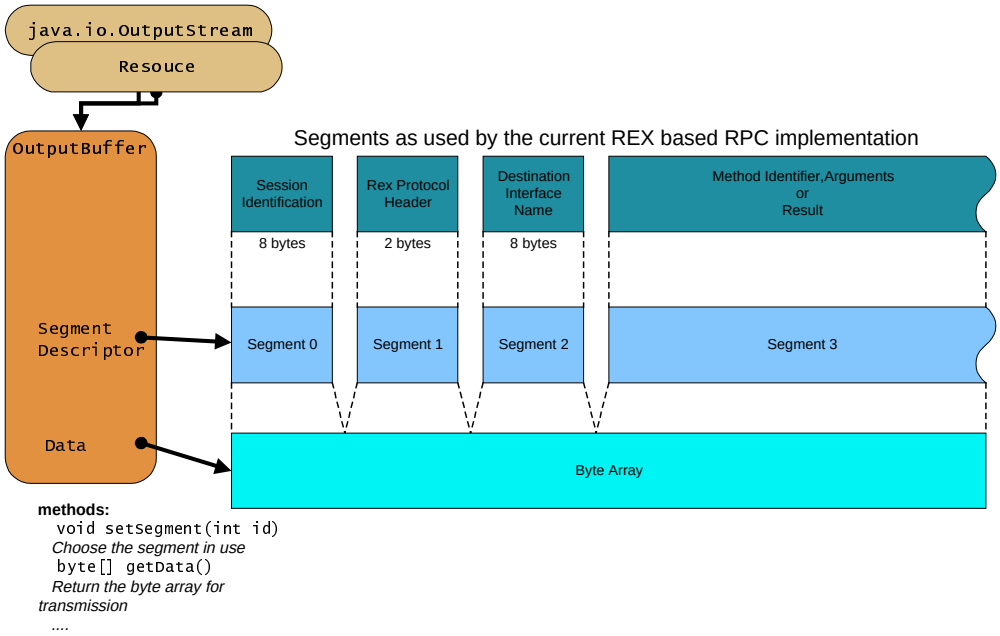


Figure 6

2.7.3 Threads

Client side thread multiplexing is under the control of the RPC Protocol layer, as this must be aware of the relationship between synchronous calls and asynchronous messages. Below the RPC layer, there may be many simultaneous messages flowing up or down the protocol stack (although only one per session). Above the RPC layer there may be many simultaneous calls.

On the server, the Media Access Layer is responsible for receiving messages and transmitting them up the protocol stack. When a message reaches the RPC layer, it may be converted to a call that proceeds up the protocol stack. The upper layers of the stack (and in particular the object implementing the service) may take a long time to process the request and during this period, it is important that messages continue to be received. There are two options; either the call proceeds using the original thread and a second thread takes its place to listen for further messages, or the original thread passes the call state to a new thread and returns to the Media Access Layer itself. Figure 7 illustrates the general case.

Clearly, the Media Access Layer and the RPC Layer must co-ordinate to ensure that threads are created as required, so that message are not missed. In the current REX implementation, the Media Access Layer is responsible for thread creation. It maintains a pool of threads, one of which acts as the “active listener”. When a message is received by this thread, it proceeds up the protocol stack, and another thread from the pool takes its place. This scheme has the advantages that no thread switch is required on the server during the processing of a call (leading to rapid response) and that the size of

the thread pool may be simply specified by a tuple of two values; the maximum number of idle thread waiting to receive messages, and the maximum number of threads involved in call processing. Threads are dynamically created and destroyed as required. Figure 8 illustrates this protocol (timeouts are omitted for clarity). Note that the layers other than the RPC and Media Access Layers are unaffected by these decisions.

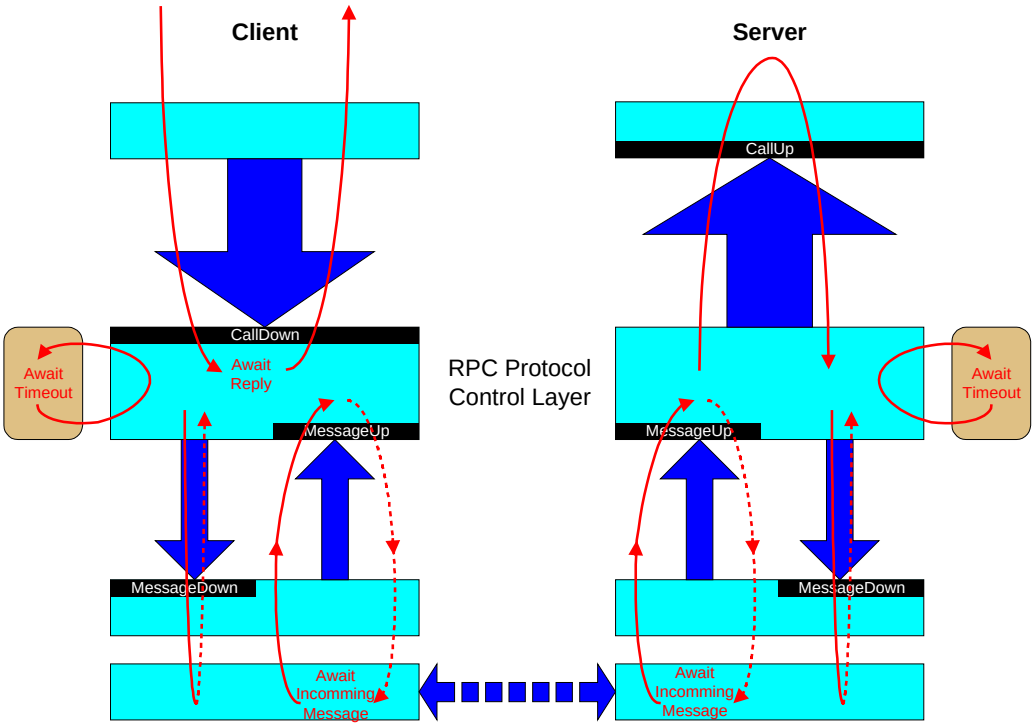


Figure 7

2.7.4 Timers

A utility resource is provided in the form of a timer thread. This maintains a priority queue of `Queuable` objects ordered by timeout, and calls each back when the timeout occurs. Items may be added and removed from the queue at will. This resource is utilised by the RPC layer to manage failures and control retransmission. The locking strategy for the timer queue is that objects must be locked to be added or removed from the queue, and that the callback occurs with the lock held. This ensures that there are no race conditions with queue management. For example if a lock is held on an object, and the timeout occurs, the callback will not occur until the lock 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. This limitation is not considered important at present, as the only items currently queued are sessions, which are only locked for short periods.

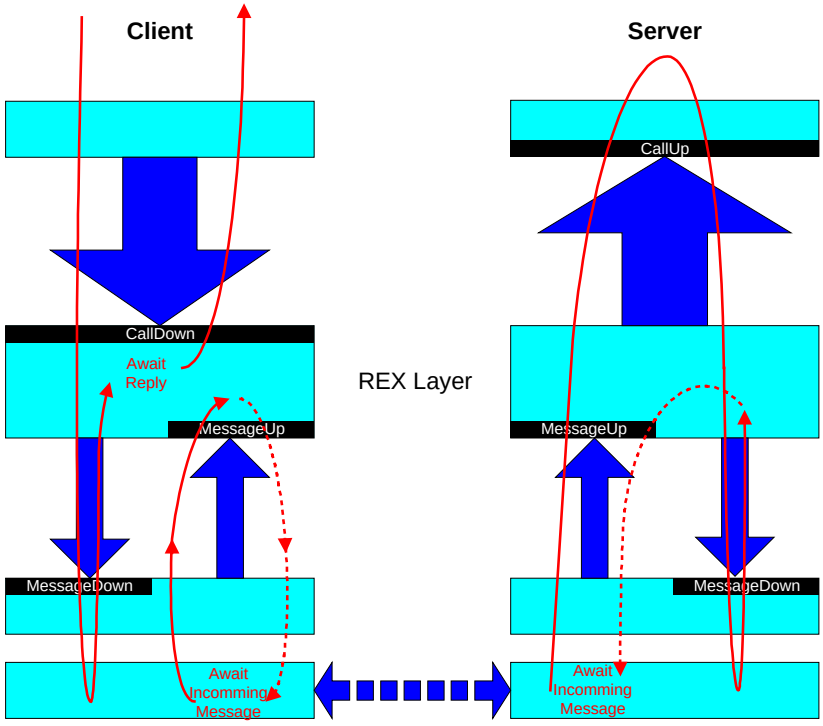


Figure 8

2.8 Stub Creation

In the FlexiNet framework, like other RPC systems, is it necessary to have stubs to represent remote objects. FlexiNet avoids the need for similar stubs (or skeletons) on the server side of a connection by making use of Java core reflection.

In FlexiNet the purpose of a stub is to convert a typed method call into a generic method call. This may undergo reflective processing before being passed to a protocol layer to be processed for remote execution. As the function of stubs is well defined, and they are not responsible for object naming or data marshalling, it is possible to have a single stub class per interface class that may be used by all RPC protocols. This simplifies design and aids reuse.

In traditional systems, stubs are generated by a special compiler that builds the stubs from a definition of the interface to be represented (a Java interface class, or statements in a special interface definition language). There is

typically an additional compilation phase that adds to the complexity of distributed programming. In addition, there are security concerns to be considered. The client application must trust the repository that stores the stubs, as the stubs are objects that will be executed in the client domain. In general the stubs are stored in a repository linked to the server code. This places a trust requirement between client and server that may not be warranted.

In the FlexiNet framework, the stubs are automatically generated as required *by the client*. This reduces the trust requirements to agreement on the interface used for interaction (which is a declarative statement, not an executable object). In addition this gives completely transparent stub creation, easing distributed programming.

The stub generator works by constructing a stub class in Java byte code that implements the interface exported by the server object. This byte code is then loaded like any other class. The client does not require access to any special resources (such as file access) as the 'load' operation uses an in memory image as the source location.

2.9 Serialisation

In order to transfer data between the client and server, it must first be converted into serialized form that may be passed across the wire. In object oriented systems, the data to be transferred typically consists of data objects, each of which may be a subclass of the corresponding class defined in the server interface. It is therefore necessary to pass information pertaining to the type of the data, in addition to the data itself. The process of converting a graph of structured objects into a serial stream of bytes for transmission is often referred to as serialisation. Within the data to be passed, there may be references to objects implementing other services, which should be passed *by reference* rather than by simply serialising the state of the objects. Any RPC system that supports both *by value* and *by reference* semantics must apply some criteria to determine how a particular object should be transferred.

The criteria used by Sun's RMI protocol is to pass objects that extend a particular super-class by reference, and to pass all other objects by value. This approach is clearly not transparent, as objects must be designed for one mode of operation, and is unhelpful when there are a range of possibilities -- for example a number of transmission protocols. The approach taken in FlexiNet is to examine the class of the *reference* rather than the class of the object. If the reference is of an interface class (i.e. it refers to an interface on the object) then this reference is passed by reference. If a reference is of an object class then the object referred to is passed by value. These semantics are natural, as they allow re-implementation behind an interface; simple to use and most importantly, transparent. No special action has to be taken in the design of a service object. An additional advantage is that one server object may implement a variety of different interfaces, and a different approach may be taken to realising remote references to each one. For

example an object may have both a reliable management interface, and a high performance stream interface.

Unfortunately, Sun's serialisation classes are not suitable for serialising classes where distinctions about interface need to be made, so a new set of serialisation classes was designed. The new serialisation scheme does not require classes to be serialised to implement any special interfaces, but there is currently a limitation that the classes must have a no-arg constructor, and only public fields are serialized. Sun's scheme also suffers from the first restriction, but uses native methods to overcome the second. This is an approach we are reluctant to take with FlexiNet. An alternative solution has been architected.

Some classes cannot be serialized in the normal way, because the fields have special meaning. Sun's scheme allows such classes to provide specialist serialisation code, by implementing methods in the `Serializable` or `Externalizable` interface. FlexiNet serialization is compatible with the Externalizable approach, and objects written with Sun's serializer in mind may be used with the FlexiNet serializer. Unfortunately the FlexiNet serializer cannot make use of specialist serialization methods of the original `Serializable` form. This is due to Sun's decision to tie the signature of these methods directly to Sun's current implementation of its serializer, rather than to an abstract specification of all possible serializers.

2.10 Widening and the 'Tagged' Class

FlexiNet performs remote invocation based on exported interfaces. One object may implement many interfaces, and a client stub produced to access the object is specific to a particular interface class. The class of an interface to be exported may be determined by the way in which it is called.

For example

```
public interface Foo
{
  public int method1(Bar b);
}
```

If `Bar` is an interface class, then an invocation of `method1` on a remote object implementing `Foo`, should pass a reference to an object implementing interface `Bar`. This will have a stub created for it that also implements `Bar`. If the original object implemented other interfaces as well as Bar, then this will not have any effect on the generation of the stub. In particular the stub cannot be cast to be of another class *even if this was possible with the original object*. This limitation is due to security requirements, in general a client should not be able to convert a name for one interface into a name to another interface.

Although this limitation is imposed for a good reason, it causes problems when it is necessary to pass an interface *generically*, i.e. by passing it as if it

were of type `Object`. In the local case we may pass an object as any interface that it implements, and then cast the object back to some other interface.

For example

```
public class Baz
{
  public int method1(Object o)
  {
    Bar b = (Bar) o;
    ...
  }
}
```

This is not possible if the call is made remotely.

The `Tagged` class is a solution to this problem. The `Tagged` class is used to represent a generic reference to an interface, together with the class of this interface. When instances of class `Tagged` are serialized or unserialized, a special technique is adopted so that it will always be possible to cast the reference to be of the named class.

For example

```
Bar b;
Baz z;
. . .
z.method2(new Tagged(b,"Bar"));

public class Baz
{
  public void method2(Tagged a)
  {
    Bar b = (Bar) a.iface;
    ….
  }
}
```

This technique is used in the Trader described in the following section. A shorthand constructor may be used for tagged,

```
new Tagged(obj)
```

produces a reference to the object `obj` which is tagged with the *first* interface implemented by `obj`. This is defined as the first interface name to appear in the '`implements`' clause of `obj`'s class definition, or if `obj` implements no interfaces directly, then this is the first interface implemented by its direct superclass, or nearest ancestor that implements an interface.

3 TEMPORARY API

3.1 Introduction

In order for programs to use make use of the FlexiNet framework directly, a temporary API has been designed to manage trading of interfaces. This API is used by the test and demonstration programs, but does not form part of the overall FlexiNet architecture.

3.2 Trivial Naming API

The static class `UK.co.ansa.flexinet.FNetTest` provides basic facilities for naming interfaces, and resolving these names.

```
Name FNetTest.name(Object o)
```

will name all the interfaces to an object `o` and return a name which can be resolved by a remote machine. The name can be converted to a string and printed to the screen, or written to a file.

An example server:

```
Class Server
{
  public static void main(String[] args)
  {
    A a = new AImp();
      // A is an interface
      // AImp is an class that implements this interface
    Object n = FNetTest.name(a);
    System.out.println("Service name =  " + n );

    // Don't let the server exit!
    Thread.currentThread().suspend();
  }
}
```

The method

```
Object FNetTest.resolve(String ifaceclass,String name)
```

will resolve the name represented by the string `name.` `The` object returned will implements the class `ifaceclass` by performing a remote method invocation on the object named by `name`.

An example client to use the above server is

```
Class Client
{
  public static void main(String[] args)
  {
    String name = args[0];
    A a = (A) FNetTest.resolve("A",name);
    int result = a.add(2,3);
    System.out.println("2 + 3 = " + result);
  }
}
```

The full example may be found in `FlexiNet/TestCode/TestCall`

3.3 Trivial Trading

A trivial trader is provided to allow clients and services to locate each other. The trader stores a mapping between names and references to exported interfaces. Services may add to this dictionary, and clients may request the interface to a service by supplying its name. This is not intended as a fully featured trader, merely as a simple utility to aid experimentation.

Only one interface is stored per name, and the trader does not perform any checking to ensure that the object that provided the interface is still in existence. If a (name, reference) pair is added to the trader and the name is already in use, the old (name, reference) pair is discarded.

The system property "`flexinet.trader`" is used to establish a reference to the trader. This may be set by defining it on the command line when the Java interpreter is run. The trader itself must be running in order for clients and services using it to work. Typically there is only one trader, and it need not be on the same machine as the client and/or server. If more than one trader is running, then each will manage a separate name space; i.e. names exported to one, cannot be imported via another.

3.3.1 Starting the Trader

The trader may be started at a random address using the command

```
java UK.co.ansa.flexinet.trader.TrivTrader
```

The address of the trader will be printed to `System.out`

To start the trader at a particular address, use the command

```
java -Dflexinet.trader=<address >
     UK.co.ansa.flexinet.trader.TrivTrader
```

where <*address*> is the address given by a previous incarnation of the trader, when run *on the same machine*. Note. The trader may be unable to use this address, for example if the protocol being used requires access to a socket that is in use. If the trader cannot use the address given, it will fail.

3.3.2 Exporting an interface to the Trader

A service may export an interface on any object to the trader by calling one the static methods

```
void FNetTrader.put(String name,Object obj,
                        String ifaceclass)
```

or

```
void FNetTrader.put(String name,Object obj)
```

where `name` is the name the interface is to be called in the trader (for example "MyService", `obj` is the object implementing the interface to be exported and `ifaceclass` is the class of the interface to be exported. If the interface class is omitted, then the *first* interface implemented by `obj` will be use (as described in section 2.10). Any string may be used as the name for an interface; however if two or more services use the same name, then the last service to export an interface to the trader will displace the other interfaces.

`UK.co.ansa.flexinet.trader.FNetTrader`, is simply a static wrapper class for the `Trader` interface that deals with the initial binding to the trader and hides the use of the `Tagged` class. A client or server may of course choose not to use this wrapper class.

Clients and services wishing to use the trader must have the address of the trader set in the system properties. For Sun's Java interpreter this may be done by using the `-D` flag. For example

```
java -Dflexinet.trader=<address> MyServiceClass
```

3.3.3 Importing an interface from the Trader

A client may import an interface from the trader by calling the static method

```
Object FNetTrader.get(String name)
```

Where `name` is the name the interface to be imported was given in the corresponding `put` call. The object returned should then be cast to be of the appropriate interface class.

4 EXAMPLE APPLICATIONS AND APPLETS

4.1 Introduction

With the FlexiNet code are a number of example applications to demonstrate how to use the ORB framework, and to allow some simple performance measurements to be taken. Each example it in its own directory and typically consists of two files `Client.java` and `Server.java`. With each example is a `ReadMe.txt` file explaining how to use it.

4.2 FlexiNet and Applets

There is nothing inherent in the design of FlexiNet that prevents it from being used from within applets, as well as applications. However due to the security restrictions placed upon applets by applet viewers/browsers, there are a number of points to note.

1. FlexiNet requires JDK 1.1. FlexiNet use in applets has only been tested using Sun's `appletviewer` (1.1.3).

2. The FlexiNet libraries must be present on the browser's `CLASSPATH`. This is because locally loaded code has is able to access the network, whereas code loaded remotely may not (in general).

3. During execution, FlexiNet will detect that it is running under a browser, and will take special action to circumvent the normal security restrictions that would be placed on a remotely loaded applet. There is a performance hit of two thread switches per packet sent associated with this. This is equivalent to approximately 0.5ms per remote invocation if either of the server or client is an applet, and 1ms if both the client and server are applets.

4. *IMPORTANT* By placing FlexiNet on the browser's `CLASSPATH`, the facility to send data to any host on the Internet is made available to any applet. Effectively, FlexiNet changes the browser's security policy to allow network access *for all applets*. FlexiNet also allows applets to determine the IP address of the host on which they are running.

   The draft specification of JDK 1.2 includes changes to the way in which

applet security is managed, so that the current applet specific mechanisms would not be required.

5. When objects, or references to interfaces are passed between FlexiNet clients and servers, the classes of the objects and interfaces must be available to the FlexiNet code. By default, FlexiNet can only access classes on the browser's `CLASSPATH`. To allow access from other class repositories (e.g. The applet's source directory) use the static method

```
FNetTest.addCodeBase(ClassLoader loader)
```

or

```
FNetTest.addCodeBase(Object o)
                    // Allow classes from same location as O
```

Alternatively the applet can be made to inherit from `UK.co.ansa.flexinet.FlexiNetApplet`, rather than `java.applet.Applet`

4.3 Example Programs

4.3.1 TestCall

This is the simplest test, and performs simple RPCs between a client application and a server application. No traders or other network services are involved.

4.3.2 AppletTest

This is a version of TestCall where the client and server are both applets. The client from TestCall can be used with the server from AppletTest and visa-versa.

4.3.3 TestRefPassing

This is an extension of TestCall where the client passes the server a reference to an interface to an object on the client, and then invokes a method on that interface.

4.3.4 AppletTestRefPassing

This is the applet version of TestRefPassing. The client from TestRefPassing can be used with the server from AppletTestRefPassing and visa-versa.

4.3.5 ChineseWhispers

This is a more elaborate reference passing example. The client invokes a method on the server, and passes a reference to a local interface. The server

invokes a method on this interface and then returns the result to the client. This demonstrates applications acting as clients and servers simultaneously (in the RPC sense) and the nesting of calls.

4.3.6 TestTrader

This example demonstrates how a network service called a trader can be used to allow a client to lookup the network address of a server. The server *exports* an interface to the trader, giving it a textual name. The client then lookups the name in the trader and *imports it*. Both the client and server must be given the network address of the trader, so that they may communicate with it. This is done by setting a Java property.

Once the client has located the server, it performs a simple method invocation.

4.3.7 SoakTest

This is a simple soak test to demonstrate the performance of long running servers with large numbers of clients. It also simulates network failures by occasionally pausing for up to a minute. This tests the retransmission mechanisms present in the RPC protocol. The server maintains a list of active clients, and for each validates each call to ensure that no messages have been duplicated or lost. Any number of clients may be started or destroyed at will.

Both the client and the server actively call the Java garbage collector, so that the amount of memory consumed by the processes may be monitored.

Soak test clients make use of the Trader to locate the server.

4.3.8 TestPerformance

This test performs repeated simple invocations between client and server. It was used to produce the performance figures given in section 1.3

4.3.9 RMIPerformance

This example has identical functionality to TestPerformance, but uses Sun's RMI rather than the FlexiNet ORB. As well as being used for comparative performance, this code illustrates the additional 'code pollution' required for RMI.

5 SUMMARY

This document has outlined the basic ORB framework designed as a test bench for the FlexiNet Architecture. The key features of this framework are transparent remote method invocation, and the ability to construct protocol stacks dynamically. The FlexiNet project is currently investigating how a policy annotated description of application component interaction can be used to automatically choose where to place application components, and how to bind them together.

6 REFERENCES

[1] RMI, Java's native remote method invocation, from Sun.

http://java.sun.com/marketing/collateral/javarmi.html

[2] OrbixWeb for Java, a CORBA compliant ORB from Iona

http://www.iona.com/Products/Orbix/OrbixWeb/index.html

[3] Java Core Reflection, a new facility in Sun's JDK 1.1

http://java.sun.com/products/jdk/1.1/docs/guide/reflection/index.html

[4] Reflective Java, meta-level programming from APM.

http://www.ansa.co.uk/Research/ReflectiveJava.htm