FlexiNet - A flexible component oriented middleware system

Richard Hayton, Andrew Herbert, Douglas Donaldson.
APM Ltd, Poseidon House, Castle Park, Cambridge, CB3 0RD, United Kingdom
Richard.Hayton@ansa.co.uk

Abstract
The FlexiNet Platform is a Java middleware platform that features 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. In this paper we give an overview of the FlexiNet architecture, highlighting how its approach differs from other middleware architectures, and illustrate the benefits that result from the new approach. We believe the FlexiNet gives a clear example of the advantages of language level introspection and reflective techniques.

1 INTRODUCTION

Generally, research middleware platforms have provided application programmers with facilities for just one form of interaction, for example remote procedure call, or transactions or replication. Increasingly, industry middleware platforms are been asked to provide a kit bag of "plug and play" capabilities, for example transactions, replication, authentication, privacy, auditing and many other features. Inherent in the design of middleware is the need to make appropriate trade-offs. Current systems remove choice and impose 'one size for all'. Unsurprisingly, this does not fit all needs.

In resolving this difficulty, platform developers are torn between the competing goals of transparency and application control. Transparency dictates that the specific interaction model should be hidden behind a generic invocation interface (e.g., method invocation) whereas application control requires that a full set of interaction primitives (e.g., bind, commit, abort) should be exposed.

The problem is further compounded by the fact that different applications require different combinations of middleware features and therefore a compositional approach is required. In CORBA [1], for example, a single set of nested choices is effectively being made through the definition of Object Services. Each Object Service provides an interface for application control and overall the CORBA Services framework offers the option of transparency for ordered subsets of the available capabilities. Whilst the CORBA framework is comprehensive, it is very fragile in the sense that there are complex relationships between the various services and the basic Object Request Broker itself, leading to bloat in implementations and developers being forced to manage more capabilities than they necessarily need in any particular situation.

This suggests a more component-oriented view of middleware is required. Component-oriented middleware would allow an application developer to choose individual components that embody appropriate trade-offs, and may even substitute alternative components, for example to accommodate different security mechanisms or transaction protocols. However, unlike designing applications components such as customer accounts and inventories, which are separable parts, designing middleware components is a difficult problem. In particular, middleware tends to make use of automatically generated code (for example stubs) that is by its very nature hard to change. Traditionally a stub converts an invocation into an untyped byte array representation to pass to a communications layer. Having discarded language level typing and introspection facilities in this way, it is hard to provide developer-written components with the information they need, and complex conventions have to be followed to tie application level events to middleware events. For example Orbix's filters and transformers [2] provide a means to modify how communication events are handled, but relating filter events (i.e. pre-stub events) to transformer events (i.e., post-stub events) is a major challenge. Behaviour at the filter level is modelled by CORBA type codes [1] and the dynamic checking of these against the programming language type system has to be managed by the developer. For all these reasons, existing middleware systems tend to be monolithic, or only partly configurable.

The FlexiNet Platform is a Java middleware system built as part of a larger project [3] 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. This gives us the benefit of remaining within the language type system and saves us from having to conduct the book-keeping that would otherwise been needed to remember relationships between components.

In this paper we give an overview of the FlexiNet architecture, highlighting how its approach differs from other middleware architectures, and illustrating the benefits that result from the new approach. We believe the FlexiNet is a clear example of the advantages of language level introspection and reflective techniques.

2 RELATED WORK

There are many other systems purporting flexible binding, particular with respect to performance and resource tradeoffs. Traditionally this work has been driven either from a quality of service perspective [4] or from an aim to simplify protocol implementation by building complex protocols out of simpler micro-protocol engines. FlexiNet differs from these systems in that it provides flexibility as a higher level – in addition to controlling protocol level choices, higher level transparencies such as replication, security, persistence and mobility can be managed. A companion paper [5] describes how mobility, in particular, has been tackled. In providing these capabilities as components we have to tie together different pieces in a consistent and structured manner, and therefore cannot work at a generic buffer and channel level like these other systems.

The key advantage that FlexiNet has over other schemes is the Java language itself. This provides a great deal of support, in particular for introspection (runtime examination and control of types) and reflection (generic invocation of methods). This allows FlexiNet to provide a middleware platform that extends the core language features, and is internally strongly typed – rather than having to separately manage the type system as is found with CORBA’s typecodes.

3 SELECTIVE TRANSPARENCY

FlexiNet provides a straightforward computational model defined entirely in terms of objects, interfaces and method invocation. Remote method invocation is given semantics as close to possible as local method invocation. This uniformity (access transparency) helps keep application code separated from ‘systems’ code, making it easier to move applications from one environment to another. To allow control over middleware decisions, the application programmer may control the selection of mechanisms, either at runtime via an explicit binding interface, or at design time by controlling the mixture of protocols and transparency components used, and the size and nature of resource pools assigned to those components. Other work within the FlexiNet project is concerned with third party control of these decisions, so that an application may be linked with appropriate protocols at deployment or bind time.

We use the ODP [6] notion of interfaces as the access points for objects, and provide transparent interface proxies, so that objects may be accessed remotely, rather than following the less clean semantics of Java Remote method Invocation (RMI) [7]. In FlexiNet, when parameters are passed to a method, we pass references to interfaces by reference, and pass object values by copying. This follows the 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 importance of determining the transfer semantics (value or reference) within the context of a particular call can be seen when mobile objects are considered. Using the tagged object approach, each object is classified as either a server or a data object. It is therefore not possible to have the concept of a mobile service object – since it must be passed between hosts as data and this contradicts its definition as a server.

3.1 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. Our stubs are very lightweight compared to other systems, and in particular the stub is not responsible for the wire representation of the invocation, and embodies no implicit or explicit policy about how the call will be handled. As stubs are so simple, we may leverage Java’s run-time linking support and generate stub classes on demand within the client process,

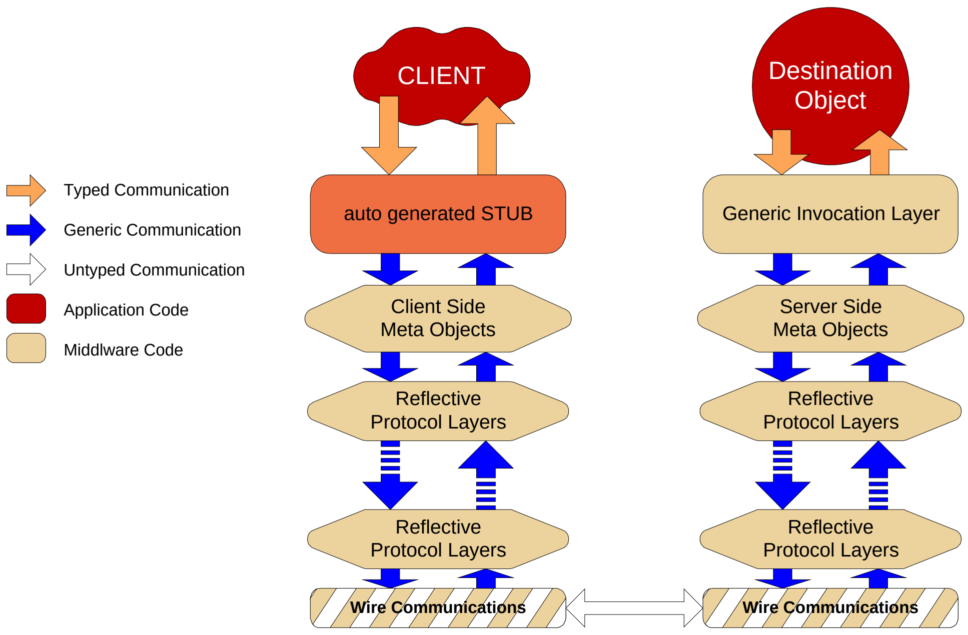

Figure 1 A Reflective Protocol Stack

by examining the interface definition dynamically. The same stubs classes may be used by all protocols that treat service objects in a uniform way.

The FlexiNet protocol stacks are correspondingly more complex than those of a 'heavyweight stub' system, since we make them responsible for all aspects of call processing. This includes high level aspects, such as management of replication, in addition to the serialization of the invocation parameters and driving the execution of a remote procedure call protocol (In our implementation we support IIOP [1], ANSAware Rex [8] and a native TCP based protocol).

The layers of a FlexiNet communication stack may be viewed as reflective meta-objects which manipulate the invocation before it is ultimately invoked on the destination object using Java Core Reflection [9] (this removes the need for a server-side stub or skeleton). This approach has a number of advantages. Middleware (or application) components may examine or modify the parameters to the invocation using the full Java language typing support. Debugging is made straightforward as information is kept 'in clear' and splitting the middleware 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. Third party meta-objects can be fully general 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).

As the call proceeds down the stack, different layers manipulate the invocation so that by the bottom of the stack, the original abstract name has been resolved to an appropriate endpoint or connection identifier, and sufficient information has been serialized 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.

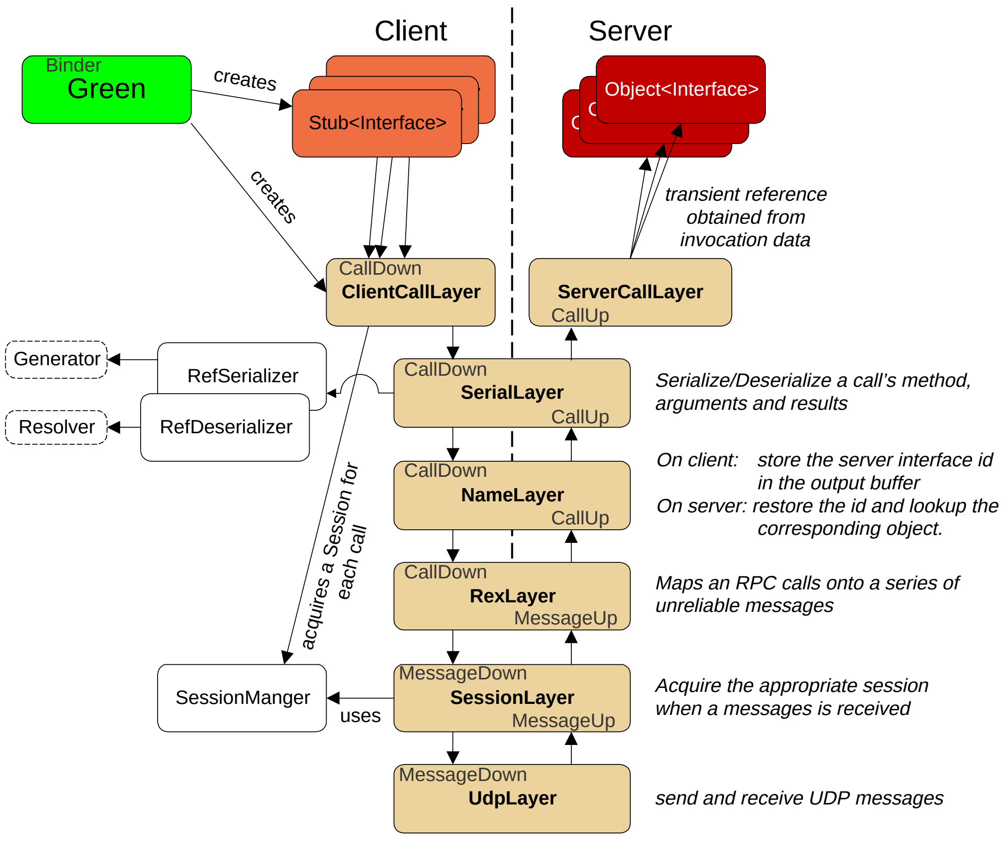
Client
Server
Binder
Green
creates
Stub<Interface>
Object<Interface>
creates
transient reference obtained from invocation data
CallDown
ClientCallLayer
ServerCallLayer
CallUp
Generator
RefSerializer
Resolver
RefDeserializer
CallDown
SerialLayer
CallUp
Serialize/Deserialize a call's method, arguments 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

The notion of a *session* is introduced to abstract this information, so that layers may retain client-related information over the duration of a number of calls. Sessions also provide an in-call mutex, and may be used to ensure that per-client resources are cleanly allocated and freed across a number of essentially independent components. In other work, our colleagues have had experience adding security features to a number of RPC protocols. They have 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.

3.2 An example protocol

Figure 2, gives an overview of the 'Green' protocol stack in our 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 may be integrated into a single stack. 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 to multiplex the call over a number of replicas, or perform other client-side reflection. 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 will acquire 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 (per replica) reflective layers to further manipulate the call. After this manipulation, the call passes to the Serial Layer, which serializes (marshals) the method and parameters into an output buffer.
5. The next layer, Name Layer, extracts the de-multiplexing information from the name of the interface being referenced, 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 to encrypt or compress the outgoing message, before it reaches the session layer. This stores information to allow the session to be re-established on the server. The session layer marks a region of mutual exclusion. 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.
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 uses lazy acknowledgements). 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-serialized 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 serialization layer, the result is serialized, 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. Eventually, the result is returned to the client using standard Java means.

3.3 Binders

In the discussion so far, we have describe how a binding to a remote interface is represented, and how an invocation may be processed. In one or more of the protocol layers, it may be necessary to serialize references to local or remote interfaces, and during deserialization, proxies must be constructed to represent these exported references. This is the mechanism by which all bindings (other than the first) 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 objects that are capable of both the generation and resolution of names. A typical binder will both generate names and convert names generated by other (compatible) binders into the (stub, stack) pair previously described. A 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, each with potentially 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. As binders are responsible for constructing protocol stacks, each binder in turn contains a reference to the generator and resolver to be used by newly created stacks.

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, which generates bindings using a protocol based on Rex over UDP, and Red, which generates bindings

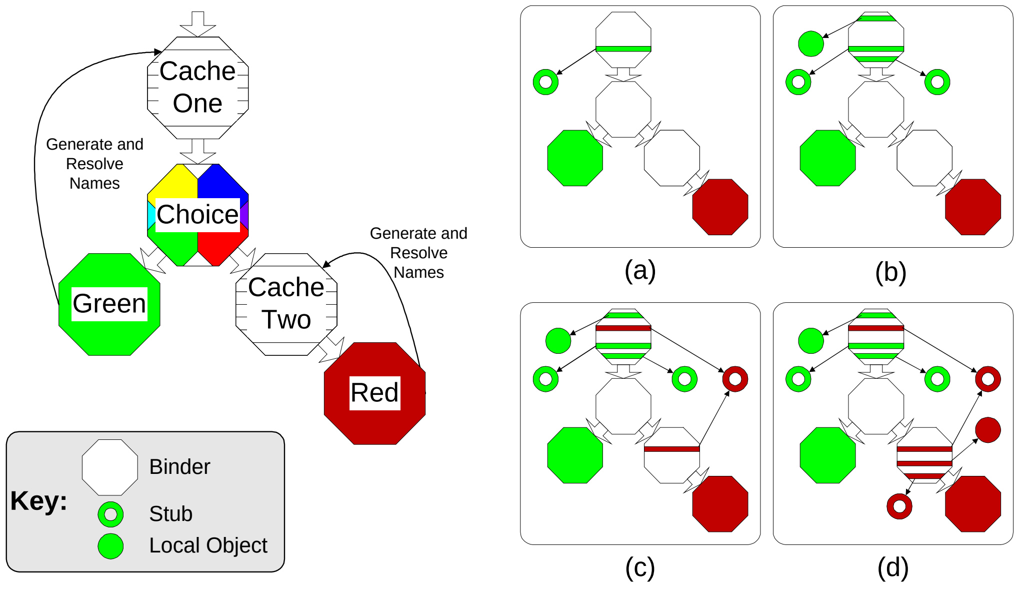

Figure 3 A hierarchy of binders

using IIOP over TCP. There are three additional binders, two Caches which store tables of previously resolved bindings and one 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 our example, imagine that the choice binder has been initialised to always use Green when generating names unless explicit QoS parameters are specified indicating that Red should be used. When resolving names, Green or Red will be used as is appropriate.

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 client 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. We have constructed a ‘negotiation binder’ 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 are to be used dependant on the location of the client and server (for example we could choose a 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, we could dynamically augment the binder graph 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.

4 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

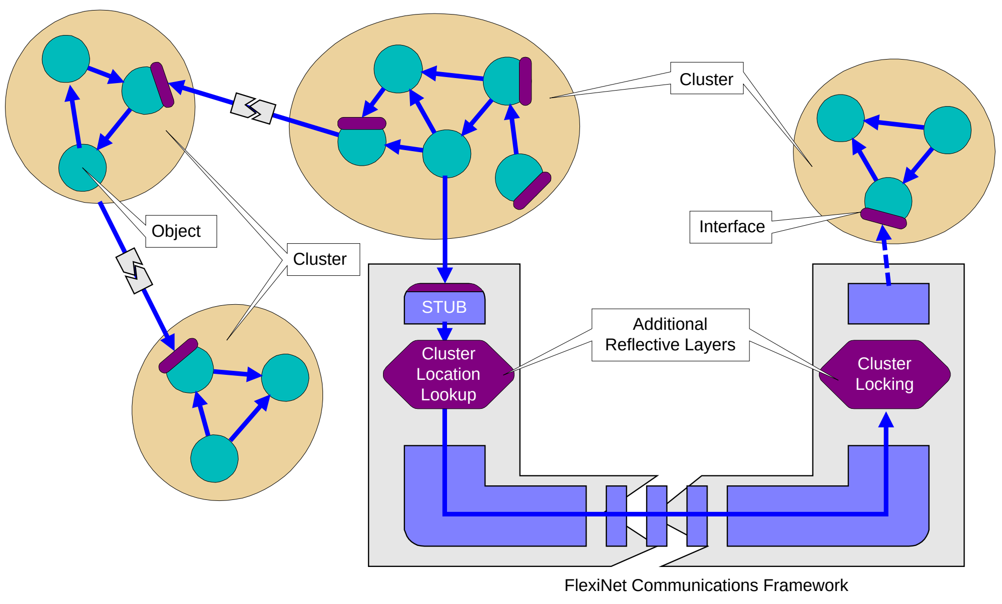

Figure 4 Encapsulation using FlexiNet

be intercepted and managed. The RM-ODP standard introduces the notion of a *cluster* as a unit of encapsulation. We have implemented this abstraction as part of a Mobile Object Workbench constructed using FlexiNet [5]. This is illustrated in Figure 4. Using this encapsulation construct FlexiNet can support mobile objects (requiring Migration and Relocation transparencies), persistent objects (requiring Persistence and Failure transparency) and current work is adding Transactional transparencies using the same clustering mechanisms.

Once this work is complete, we will have distributed platform that can support all of the high-level distribution transparencies identified in the RM-ODP standard, as a natural extension of non-distributed Java programming. Furthermore, the RM-ODP cluster concept can be considered as a special kind of Java Bean [10], allowing transactions, persistence and mobility to be expressed in terms familiar to Java developers, and allowing integration with visual development tools.

5 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 [11]. The modularity is an advantage here, as different pool management policies may be 'slotted in' in order to trade off performance against resource usage. We benchmarked an early version of FlexiNet with UDP based communication against Sun's RMI and IONA's OrbixWeb[12] and found that over a range of computers, Java interpreters and JIT compilers, FlexiNet performs as efficiently as either of these offerings.

Recently, we have expanded FlexiNet's repertoire of protocols, and anticipate a significant performance increase. Unlike RMI, which relies on native methods in order to function, or OrbixWeb, which utilises additional stub compilation tools, 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.

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. 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.

6 SUMMARY

FlexiNet was designed to provide a flexible test-bench on which to perform code deployment and binding related experiments. As such, the emphasis was on modularity and flexible configuration. FlexiNet has make considerable use of language level introspection and has embraced reflective techniques. Not only does the resulting system have the desired modularity properties, but it also performs as efficiently as other Java middleware offerings. A mobile object workbench has been built on top of FlexiNet as part of a European ESPRIT project related to the investigation of Agent based programming [13,14]. This provides Relocation and Migration transparencies, in addition to the ODP call transparencies, such as Location and Access transparency, that 'vanilla' FlexiNet provides. Ongoing work is adding Transaction and Persistence transparencies, as these require similar mechanisms to those required for Mobility. 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.

REFERENCES

1. Object Management Group (1997) CORBA/IIOP 2.1 Specification http://www.omg.org/corba/corbiiop.htm
2. Iona Technologies, Orbix Programmers Guide (Chapter 16,22) (1997) http://www.iona.com/
3. APM (1996) FlexiNet - Automating application deployment and Evolution http://www.ansa.co.uk/Research/FlexiNet.htm
4. Blair G.S., Stefani, J.B. Open Distributed Processing and Multimedia. Published by Perseus Pr; ISBN: 0201177943 (1997)
5. Hayton R.J, Bursell M.H., Donaldson D., Herbert A.J. Mobile Java Objects, Middleware '98
6. International Standards Organisation: Open Distributed Processing - Reference Model. Sep. 1995 http://www.iso.ch:8000/RM-ODP/
7. Sun Microsystems (1996) Java Remote Method Invocation (RMI)" Specification http://www.sun.com/products/jdk/1.1/docs/guide/rmi/
8. O'Connell, J. Edwards, N. and Cole, R. A review of four distribution infrastructures. Distributed Systems Engineering 1 (1994) 202-211
9. Sun Microsystems: Java Core Reflection http://java.sun.com/products/jdk/1.1/docs/guide/reflection/index.html
10. Sun Microsystems: Java Beans http://java.sun.com/beans/
11. Herbert et.al. DIMMA – A Multi-Media ORB, Middleware'98
12. Iona Technologies: OrbixWeb http://www-usa.iona.com/products/internet/orbixweb/
13. FAST (1997) FollowMe project overview http://hyperwav.fast.de/generalprojectinformation
14. Bursell, M.H., Hayton R.J., Donaldson, D. Herbert A.J. A Mobile Object Workbench. Mobile Agents '98.