FlexiNet

A flexible component oriented middleware system

Richard Hayton, Andrew Herbert. APM Ltd.

Introduction

Generally, research middleware platforms have provided application programmers with facilities for just one form of communication, for example remote procedure call, or transactions or replication. Industry middleware platforms are been asked to provided a kit bag of "plug and play" capabilities, for example transactions, replication, authentication, privacy, auditing and many other features. In CORBA, for example, a single set of nested choices is effectively being made. 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.

This suggests a component-oriented view of middleware is more appropriate. This would allow an application developer to choose appropriate components that embodied appropriate trade-offs. Designing middleware in a component oriented fashion is a difficult problem. Middleware tends to make use of automatically generated code (for example stubs) that are by their very nature hard to change. Traditionally a stub will be responsible for converting an invocation into an untyped byte array representation. This may be manipulated efficiently, but as language level typing is discarded, or separately encoded, the code required to plug-in additional capabilities is often esoteric. For all of 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 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. This allows programmers to tailor the platform for a particular application domain or deployment scenario.

We have used FlexiNet to create a "Mobile Object Workbench", which can provide location transparent access to interfaces on objects which themselves may move from host to host. The next phase of the project is to provide security services, so that objects may authenticate communications, and ensure that they themselves have not been tampered with whilst in transit. Other ongoing work includes an IIOP protocol stack and SSL integration.

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

Architecting for Components

FlexiNet has a straightforward computational model. Communication is *selectively transparent*. We endeavour to make remote communication appear identical to local communication, but also allow a program to control the trade-offs made when binding to the remote destination. We do this to maximise the independence of application code from ' systems' code. We use the ODP notion of interfaces as the access points for objects, and provide transparent interface proxies, so that objects may be accessed remotely. 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.

The FlexiNet engineering model has three central concepts. Interfaces are represented by **proxies**. Proxies enforce the typing of the remote interface, and perform remote access by utilising **binders**. Each binder is an object capable of creating a *generic* binding to a local or remote object, and different binders embody different application requirements or engineering strategies. Binders may make use of other binders in a recursive way. This keeps individual binders small, and allows application domain-specific binders to be easily created. To manage the flexibility, FlexiNet supports the notion of **multiple name spaces** for interfaces, and names are both strongly typed and structured. Names may be constructed out of other names, or arbitrary data, making the management of aggregate and indirected names straightforward.

Components and configuration were the central motivations of the design of FlexiNet. Indeed, had a sufficiently configurable system have been available, then FlexiNet itself would never have been built. As an example some of the components involved with the implementation of an interface proxy are illustrated in Figure 1. Two significant features are that the standard stub implementation allows a wide variety of binding protocols to be used, and secondly that the implementation of each binder consists of a composition of communication layers, with a high degree of reuse of layers between different binders. Different binders might use different standard protocols (e.g. IIOP, ANSAware REX), different marshalling techniques, different policies for buffer reservation or thread usage and have other performance related features. Binders may also provide different communication abstractions, for example a binder may manage communication to a replicated object, or perform encryption or auditing. In the design of the Mobile Object Workbench, a binder was created to manage communication with objects that are able to move between hosts, by tracking them using a distributed location service. This is an example of a specialist binder created by re-plumbing existing layer components, and adding a small amount of custom functionality.

This approach to software engineering keeps the infrastructure small and, together with Java's dynamic loading of classes, ensures that an application will only load those components which it actually needs.

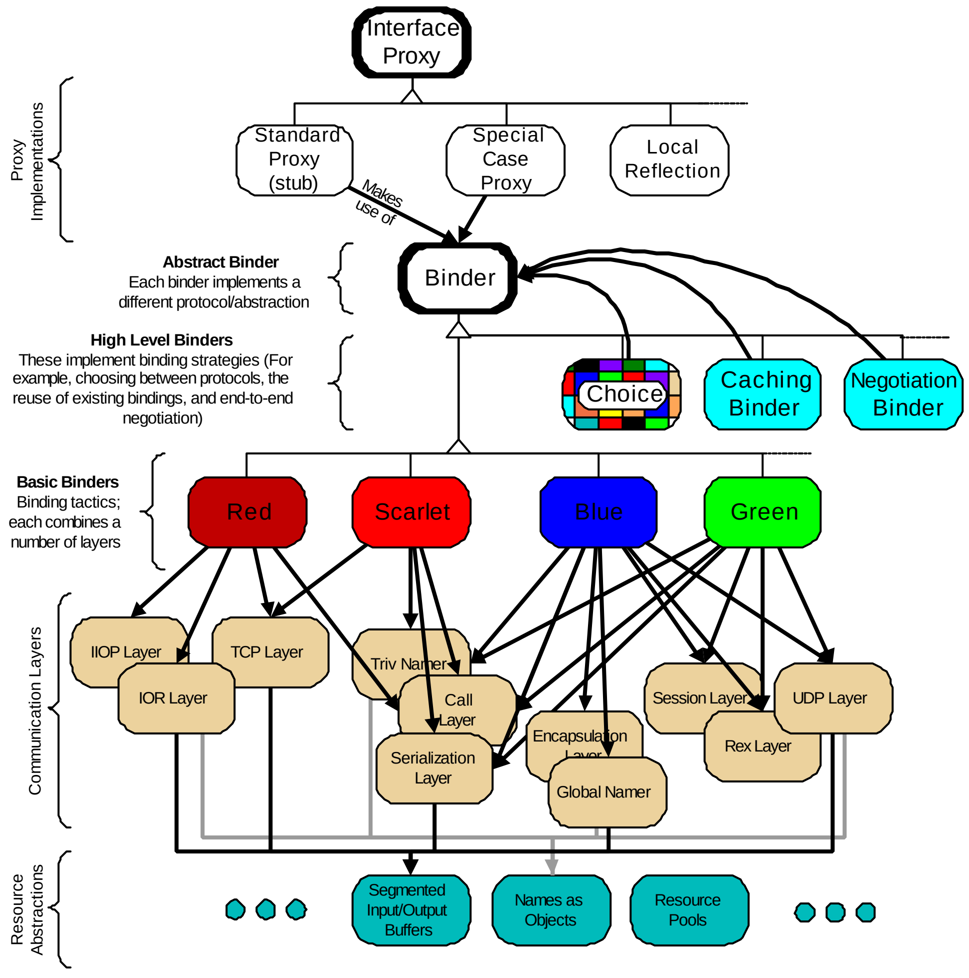

Figure 1

Generic Communications

This is a reflexive technique central to the design of FlexiNet. Rather than using stubs to convert an invocation directly into a byte array representation, instead we leverage Java's runtime typing support to represent the invocation in a generic (but fully typed) form. The layers of the FlexiNet communication stack may then be viewed as reflexive meta-objects that manipulate the invocation before it is ultimately invoked on the destination object using Java core reflection.

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. In fact the role of the stub is reduced to a conversion from a type specific to type independent invocation. All marshalling/serialization can be performed by using the run-time type introspection facilities, and is not dependant on the application or auto-generated code.

Figure 2 illustrates how a communication stack can be considered as a number of meta-objects that perform reflective transformations on an invocation. Third part meta-objects can be fully general and are fully type-safe. For example a replication meta-object might extract replica names from an interface reference by consulting an external database. It might then perform invocations on each replica in turn. As this processing is performed in terms of generic invocations, the is no need for each of these calls to pass through stubs and so the code can be both straightforward and efficient.

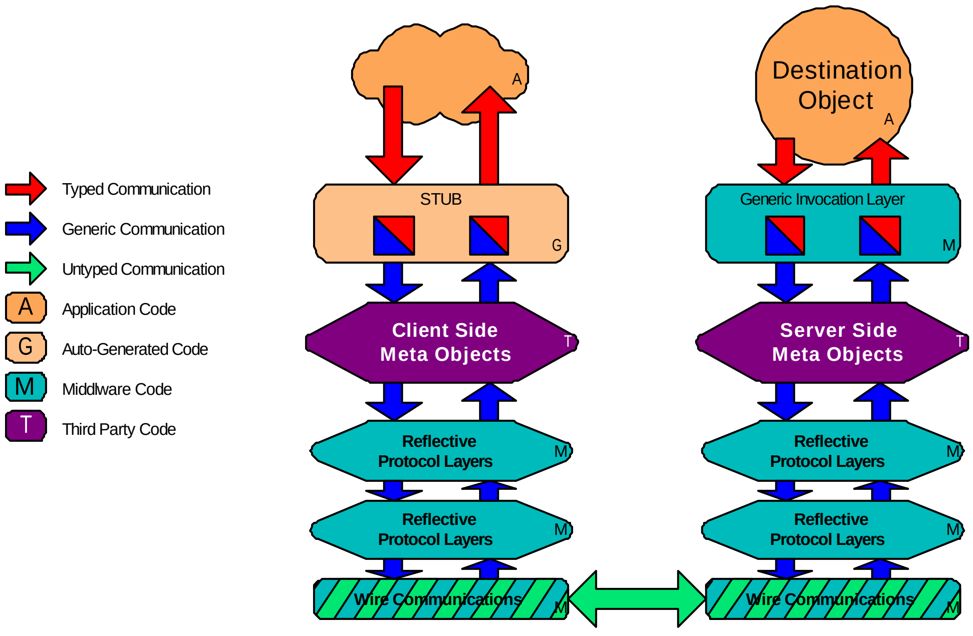

Figure 2

Naming Interface References

FlexiNet allows great flexibility, both in the design of a communications stack and the wire format of messages. In order to take advantage of this, the format use for interface references must be equally flexible. Traditionally, interface references are represented by ‘bundles of bits’, and extracting fields from these bundles must be done with care, and without the aid of language level typing. This makes it difficult, if not impossible, to extend a system by adding new types of names, as this leads to changes in the name format and has repercussions throughout the code.

In FlexiNet we took the approach that interface references are themselves objects. As objects they may be converted to and from a serial byte representation in the same way as other objects are converted. This removes the need for a specific wire format for names, and allows us to use the full generality of object typing to subclass and otherwise manipulate interface references. This is only possible because Java provides the run time type information required for serialization. As an example the decomposition of the name representing an interface on a mobile object is shown in Figure 3. This is necessarily complex, but it considerably easier to manipulate that its ‘bundle of bits’ representation. As the components for storing and manipulating interface references all use class

`Name` and `MobileName` is a subclass of this, mobile names may be treated like any other in the majority of code. This again increases the reuse of components.

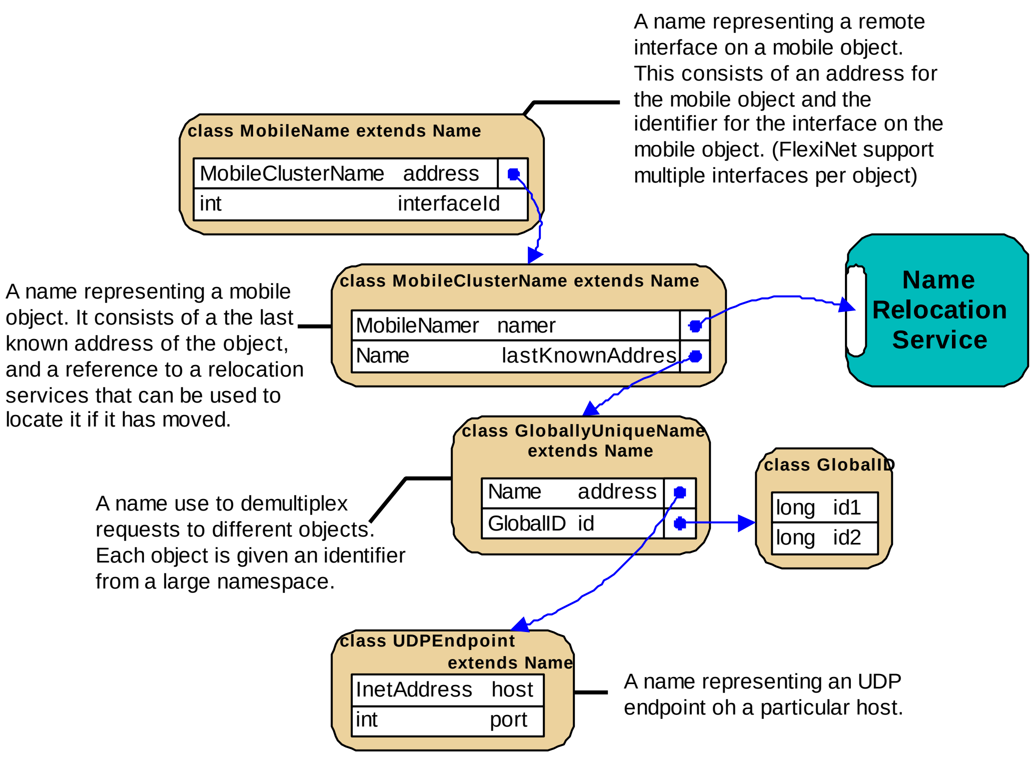
A name representing a remote interface on a mobile object. This consists of an address for the mobile object and the identifier for the interface on the mobile object. (FlexiNet support multiple interfaces per object)
class MobileName extends Name
MobileClusterName address
int interfaceId
class MobileClusterName extends Name
MobileNamer namer
Name lastKnownAddres
Name Relocation Service
A name representing a mobile object. It consists of a the last known address of the object, and a reference to a relocation services that can be used to locate it if it has moved.
class GloballyUniqueName extends Name
Name address
GlobalID id
class GlobalID
long id1
long id2
A name use to demultiplex requests to different objects. Each object is given an identifier from a large namespace.
class UDPEndpoint extends Name
InetAddress host
int port
A name representing an UDP endpoint oh a particular host.

Figure 3

Performance

FlexiNet is a component based framework, and the protocols and abstractions which 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. 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 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.

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 reflexive 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 as been built on top of FlexiNet as part of an European ESPRIT project related to the investigation of Agent based programming. This design and construction of the mobile object workbench has served both as a demonstration of the flexibility of FlexiNet, and as a powerful example of the way in which distributed computing abstractions can be extended to embrace mobility. Ongoing work investigating issues of security and evolution of autonomous, distributed systems.