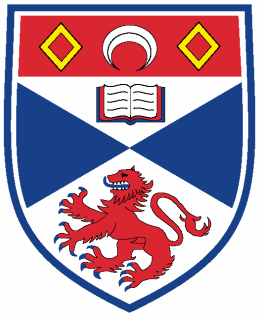
A Flexible, Policy-Aware Middleware System

Scott M. Walker

PhD Thesis

September 2005



School of Computer Science
University of St Andrews
St Andrews
Fife
Scotland
KY16 9SX

Abstract

Middleware augments operating systems and network infrastructure to assist in the creation of distributed applications in a heterogeneous environment. Current middleware systems exhibit some or all of the following five main problems:

1. Decisions must be made early in the design process.
2. Applications are inflexible to dynamic changes in their distribution.
3. Application development is complex and error-prone.
4. Existing systems force an unnatural encoding of application-level semantics.
5. Approaches to the specification of distribution policy are limited.

This thesis defines a taxonomy of existing middleware systems and describes their limitations. The requirements that must be met by a third generation middleware system are defined and implemented by a system called the RAFDA Run-Time (RRT). The RRT allows control over the extent to which inter-address-space communication is exposed to programmers, aiding the creation, maintenance and evolution of distributed applications.

The RRT permits the introduction of distribution into applications quickly and with minimal programmer effort, allowing for quick application prototyping. Programmers can conceal or expose the distributed nature of applications as required. The RRT allows instances of arbitrary application classes to be exposed to remote access as Web Services, provides control over the parameter-passing semantics applied to remote method calls and permits the creation of flexible distribution policies. The design of the RRT is described and evaluated qualitatively in the context of a case study based around the implementation of a peer-to-peer overlay network. A prototype implementation of the RRT is examined and evaluated quantitatively.

Programmers determine the trade off between flexibility and simplicity offered by the RRT on a per-application basis, by concealing or exposing inter-address-space communication. The RRT is a middleware system that adapts to the needs of applications, rather than forcing distributed applications to adapt to the needs of the middleware system.

Acknowledgements

I'd like to thank the following people because this thesis would not have been possible without them:

- Professor Al Dearle and Dr Graham Kirby, my supervisors, for their help, patience and support. They have provided a great introduction to research.

- Stuart and Alvaro for the useful feedback and for some good fun.

- Professor Ron Morrison for the opportunity to do this work.

- Conor and Dave for their proof-reading.

- Graeme, Iain, Kath, Andy and Davie for just being around at the right time.

- Mervyn and Margery Walker, my family, for their unerring belief in me.

- Finally, Jen Walker, my wife, who has been a great support for the last four years. This thesis is for her.

This work was funded by EPSRC grant GR/R51872/01: *"Reflective Application Framework for Distributed Architectures"* (RAFDA).

Declarations

I, Scott Mervyn Walker, hereby certify that this thesis, which is approximately 55,000 words in length, has been written by me, that it is the record of work carried out by me and that it has not been submitted in any previous application for a higher degree.

Date ….....… Signature of Candidate ….....……

I was admitted as a research student in September 2001 and as a candidate for the degree of Doctor of Philosophy in September 2002; the higher study for which this is a record was carried out in the University of St Andrews between 2001 and 2005.

Date ….....… Signature of Candidate ….....……

I hereby certify that the candidate has fulfilled the conditions of the Resolution and Regulations appropriate for the degree of Doctor of Philosophy in the University of St Andrews and that the candidate is qualified to submit this thesis in application for that degree.

Date ….....… Signature of Supervisor ….....……

In submitting this thesis to the University of St Andrews I understand that I am giving permission for it to be made available for use in accordance with the regulations of the University Library for the time being in force, subject to any copyright vested in the work not being affected thereby. I also understand that the title and abstract will be published, and that a copy of the work may be made and supplied to any bona fide library or research worker.

Date ….....… Signature of Candidate ….....……

Contents

Chapter 1

Introduction

This chapter introduces middleware and briefly describes the approaches to distributed application development that are available to programmers. The need for a new middleware system is justified and the outline of the thesis described.

1.1 Introduction

Lamport defines *distributed systems* as collections of distinct, spatially separate processes that communicate by exchanging messages [1]. Distributed systems consist of multiple physical machines connected via a network, and exhibit the following properties:

- Processes are autonomous. There are no central authorities that control all processes.
- Processes do not share memory. Each has direct access only to its own *address-space*.
- Processes execute concurrently.
- Systems exhibit multiple, independent points of failure. Partial failure can occur, for example, if machines or their interconnecting networks fail.
- Systems can be constructed from heterogeneous machines with different physical architectures running different operating systems.

An application that executes across a distributed system is known as a *distributed application.* Distributed applications employ the resources of multiple machines allowing programmers to create applications that are more scalable and resilient than their non-distributed equivalents. Scaling may be achieved through the introduction of additional machines, and resilience to failure by employing redundant machines then replicating application code and data across them.

1.2 Creating Distributed Applications

Inter-address-space communication occurs between processes in distributed applications. Programmers can perform all inter-address-space communication directly or employ *middleware systems* to simplify the software engineering process. This thesis defines middleware as *software that augments the operating system and network infrastructure to make the creation of distributed applications in a heterogeneous environment easier*. There are multiple approaches to middleware, which differ in the extent to which inter-address-space communication is exposed to programmers.

1.2.1 Approaches to Creating Distributed Applications

Figure 1.1 lists the possible approaches to distributed application creation, shown in a spectrum that ranges from those that expose inter-address-space communication

completely to the programmers, to those that conceal all inter-address-space communication. All but the left-most approach meets the definition of middleware provided previously. The software engineering process adopted when using each of these approaches is examined below.

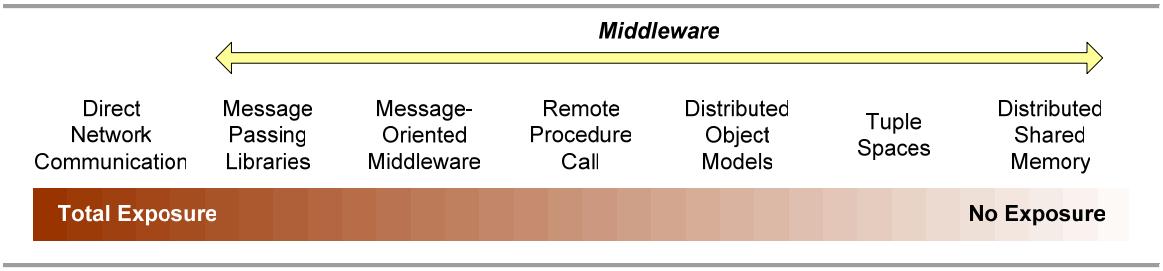


Figure 1.1: Approaches to constructing distributed applications.

1.2.1.1 Direct Network Communication

Operating systems provide abstractions over network communication (e.g. sockets) which allow programmers to pass data across the network in datagrams (e.g. UDP) or through streams (e.g. TCP). When performing inter-address-space communication directly, programmers are responsible for the end-to-end encoding of data. Programmers write all the code to manage inter-address-space communication and perform all message construction and transmission. Programmers define protocols and message formats on a per-application basis. The encoding scheme applied to data passed across the network must be defined. This encoding scheme must handle differences in architecture, operating system and implementation language between processes. Programmers open network connections between address-spaces and directly pass data across these channels. Performing all inter-address-space communication directly is difficult, error-prone and results in applications that are expensive to change.

1.2.1.2 Message Passing Libraries

Message passing libraries such as the *Message Passing Interface (MPI)* standard [2] or *Parallel Virtual Machine (PVM)* [3] abstract over the inter-address-space communication by formalizing the message passing mechanisms. These libraries allow application-level processes to exchange structured messages, which may contain application data. Data that is passed across the network in messages is encoded automatically to mask any differences in architecture, operating system or implementation languages that exist between processes. Message passing libraries provide a send/receive model through which programmers can pass messages, but

message structure must be defined on a per-application basis. Therefore, inter-address-space communication is simplified but not concealed from programmers.

1.2.1.3 Message-Oriented Middleware

Message-Oriented Middleware (MOM) systems such as the *Java Message Service (JMS)* [4] extend the message passing model provided by message passing libraries. Messages are transmitted via message queues in the system infrastructure from which they can be retrieved by recipients asynchronously, allowing for location transparency between message senders and receivers. MOM systems also provide a publish/subscribe model that allows many-to-many relationships between senders and receivers. Recipients can subscribe to particular topics in order to receive all messages published with those topics. Additional features such as persistency of message queues, guaranteed delivery or support for transactions may be provided. MOM systems expose inter-address-space communication to programmers in the same way as message passing libraries but tend to provide a richer feature set to application developers.

1.2.1.4 Remote Procedure Call

Remote Procedure Call (RPC) [5, 6] systems such as *Sun RPC* [7] provide a request-response model similar to local procedure calls. When an application performs a remote procedure call, the RPC system constructs a request message that identifies the procedure to call and contains serialized representations of any arguments. Calls are performed on static code so the environment for each call is constructed at call-time from the passed arguments and any static data referenced from the code. After the call is complete, a response message containing serialized representations of the return values is sent back to the caller.

1.2.1.5 Distributed Object Models

Distributed Object Models (DOMs) such as *CORBA* [8], *Java RMI* [9] and *Microsoft .NET remoting* [10] provide *Remote Method Invocation (RMI)*, the object-oriented equivalent to RPC. Remote method calls are performed on identifiable closures of code and data. Consequently, the environment of each call is partially formed before call-time, thereby differentiating DOMs from RPC systems. Application objects can hold remote references to objects that exist in different address-spaces. Each remotely

accessible object is associated with an identifier. A programmer can obtain a reference to a remote object from the middleware infrastructure based on its identifier.

1.2.1.6 Tuple Spaces

Tuple Space systems such as *Linda* [11] and *JavaSpaces* [12] conceptually provide a shared memory space in the distributed system in which name/value pairs called tuples are stored. This shared space is called a tuple space and is accessible to all processes in the system. Tuples written by one process are visible to all. Processes communicate and synchronize their behaviour by reading and writing tuples. If a process tries to read a tuple that is not yet present in the tuple space, it is blocked until that tuple is written by another process. The distributed nature of applications is hidden from programmers through the abstraction of shared tuple spaces. No explicit inter-address-space communication is performed.

1.2.1.7 Distributed Shared Memory

Using *Distributed Shared Memory (DSM)* systems [13] such as Java/DSM [14] and Orca [15], all processes in the distributed system appear to have access to shared memory though no physical memory is shared. DSM systems ensure that any updates made to the shared memory by one process are visible to all other processes. Using DSM systems, the distributed nature of an application can be completely hidden from the programmer.

DSM systems can provide shared memory at the byte/page level (e.g. Java/DSM) or at the object level (e.g. Orca). DSM systems that share at the byte or page level emulate physical shared memory, which leads to problems in heterogeneous distributed systems because different machines represent data internally in different ways. DSM at this level requires low level support from hardware or the operating system and so cannot be introduced easily into a system that does not already support it.

DSM systems that share memory at the object level allow processes to access shared objects using well-defined operations in a location transparent manner. Object-based DSM systems achieve shared memory through the caching and replication of shared objects, combined with coherency protocols. It is this location transparency and the approach to implementation adopted by object-based DSM systems that differentiate them from DOMs, which also permit access to shared objects.

1.2.1.8 Conclusion

Multiple approaches to middleware and the creation of distributed applications have been introduced. A more thorough discussion of these different kinds of middleware can be found in Coulouris, Dollimore and Kindberg [16].

This thesis tests the following hypothesis: A middleware system that provides control over the extent to which inter-address-space communication is exposed to programmers aids the creation, maintenance and evolution of distributed applications.

Such a middleware system needs to conceal inter-address-space communication from programmers yet allow control where required. The DOM model has been adopted in this thesis as it matches the language model of currently popular object-oriented languages such as Java [17], C# [18] or C++ [19] and provides a compromise between complete exposure and complete concealment of inter-address-space communication. Message-based systems do not provide sufficient abstraction over the network while DSM systems do not permit the fine-grained control required. A DOM that permits arbitrary objects to be exposed to remote access, and that supports caching and replication of code and data, achieves the abstraction over the network offered by DSM systems. Unlike DSM systems, DOMs allow programmers explicit control over the behaviour of remote references and remote method calls.

1.3 Limitations of Existing Systems

Middleware systems aim to make it easy to create distributed applications; however, while existing middleware systems solve some problems, they introduce others:

1. Programmers are forced to make decisions early in the design process about which types of application component may participate in inter-address-space communication. Distribution boundaries are decided statically and cannot be altered without changes to application source code. Thus, application distributions are difficult to change.
2. Distribution-related code permeates application logic meaning that applications created using existing middleware systems are inflexible to dynamic changes in their distribution. An application cannot adapt to changes to the underlying distributed system or to the flow of control within the application itself, for example, by dynamically collocating objects that interact frequently.

3. The creation of code to handle inter-address-space communication is complex and additional points of potential failure are introduced into the software engineering process.
4. It is difficult to understand and maintain distributed applications because middleware systems may force an unnatural encoding of application-level semantics. Application classes may be forced to extend special base classes, implement particular interfaces or handle distribution-related errors explicitly. Flexibility in distribution boundaries is limited and the re-use of non-distributed classes in distributed contexts, and vice versa, is hampered.
5. It is difficult to control the way in which objects are distributed among available address-spaces. Using existing systems, programmers have limited control over the policies deciding object placement, which leads to policies that are inflexible and non-adaptive.

1.4 The RAFDA Run-Time

This thesis describes the design of the RAFDA Run-Time (RRT), a 'third generation' middleware system that aids the creation, maintenance and evolution of distributed applications, thereby tackling the problems inherent in previous systems.

The RRT simplifies the software engineering process by:

- Permitting the introduction of distribution into applications quickly and with minimal programmer effort, allowing for quick application prototyping.
- Allowing programmers to conceal or expose the distributed nature of applications as appropriate.

The RRT allows the creation of flexible applications by:

- Providing fine-grained control over all aspects of middleware behaviour. The same underlying middleware system can be used to create prototypes and complete distributed applications. A prototype application can be evolved by exposing and controlling more aspects of its distribution.
- Permitting control over the parameter-passing semantics applied to remote method calls. Flexibility in application semantics is increased and is decoupled from application distribution, thereby promoting code reuse.
- Permitting the creation of object placement policies, constructed from individual policy components. This placement policy framework allows the

specification of rules that control how application objects are instantiated and migrated dynamically.

This thesis shows that the RRT directly benefits programmers by allowing application logic, parameter-passing semantics and application distribution to become orthogonal considerations, thereby aiding application design, creation, maintenance and evolution. Re-use of existing code in distributed contexts is promoted and distributed applications are capable of adapting to changes in their requirements or in the distributed system. These benefits are illustrated in the context of a case study that provides a qualitative evaluation of the RRT and a prototype RRT implementation that provides a quantitative evaluation. The RRT is designed primarily for applications distributed at "LAN-scale", though is capable of supporting "internet-scale" applications provided such applications do not attempt to preserve non-distributed application semantics.

1.5 Thesis Contribution

This thesis makes a four-fold contribution. Firstly, a taxonomy of existing middleware systems is created. From this, their limitations are evaluated and the requirements of a middleware system that permits the creation of flexible distributed applications are specified.

Secondly, the design and implementation of a third generation middleware system that meets these requirements is described. This system permits arbitrary application objects to be exposed to remote access or migrated between address-spaces without modifications to their underlying source code. It also provides a mechanism to allow the parameter-passing mechanisms applied to objects that cross address-space boundaries to be decided dynamically.

Thirdly, a framework for the specification of parameter-passing policy, known as *transmission policy*, is created. This framework allows programmers to dictate in a dynamic manner how objects participating in inter-address-space communication are passed across the network. Policies can be defined on a per-address-space basis and can be associated with classes, methods or individual parameters.

Finally, a framework for the specification of object placement policy and migration policy, known collectively as *distribution policy*, is created. Programmers can define policies of arbitrary complexity that can be reused and recombined to create complete distribution policies on a per-address-space basis.

The work described in this thesis has been developed as part of the *Reflective Application Framework for Distributed Architectures (RAFDA)* project [20]. The objectives of this project are to investigate flexible distributed object-oriented application architectures, the key components of which are:

- Transformation tools capable of transforming an application into an isomorphic distributed version in which the distribution boundaries are not fixed [21].
- A novel middleware system that tackles the limitations of current systems with respect to flexibility by allowing the exposure of arbitrary application objects to remote access.
- The creation of policy frameworks that separate parameter-passing semantics and object placement policy from functional application requirements.

1.6 Thesis Structure

Chapter 2 describes the middleware system model adopted in this thesis without reference to specific systems in order to provide a conceptual framework against which existing systems can be evaluated and compared. Chapter 2 also introduces the case study used throughout this thesis. Chapter 3 examines the limitations of existing systems in more detail and evaluates related work. Chapter 4 defines the requirements of a third generation middleware system. Chapter 5 describes the design of the RRT and evaluates it qualitatively. Chapter 6 describes the implementation of an RRT prototype and provides quantitative evaluation of this implementation. Chapter 7 concludes this thesis by summarizing the contribution of the research carried out and stating plans for future work.

1.7 Summary

Current middleware systems aim to simplify the creation of distributed applications but exhibit problems. This thesis shows that a middleware system that can separate application logic from distribution allows the creation of applications with flexible distribution architectures and has direct benefits for programmers. The design of the RRT, a middleware system that provides this flexibility, is described. The RRT conceals the complexity of distribution where appropriate, allowing distribution to be introduced into applications quickly. This reduces the software engineering effort

required to create distributed applications, leading to quick application prototyping. However, the RRT also permits programmers to expose all aspects of application distribution if required, allowing the creation of applications that can exploit their distributed nature and are flexible with respect to change. The RRT has advantages over traditional middleware approaches as it adapts its behaviour to suit the requirements of a given distributed application, rather than forcing the programmer to adapt the application to the requirements of the middleware system.

Chapter 2

Middleware System Concepts

This chapter describes the middleware system model in a system-independent manner. The case study that is adopted throughout this thesis is also introduced.

2.1 Introduction to Chapter 2

The middleware system model adopted in this thesis is a Distributed Object Model (DOM) since DOMs are able to conceal inter-address-space communication from programmers yet allow control where required. This chapter describes the DOM middleware system model without reference to any particular technologies. The list of features described here is neither exhaustive nor implemented fully by all DOMs, but all systems implement at least a subset of these features. The DOM middleware system model provides a general framework against which the features of existing systems may be judged and compared.

Terminology varies from system to system and the terminology defined here is reused throughout the thesis to provide a consistent universe of discourse. A glossary summarizing this terminology is provided in Appendix A.

This chapter also introduces a case study that is used throughout the thesis to exemplify the limitations of current systems and to illustrate the design and implementation of the RRT. This case study consists of JChord, an implementation of the Chord [22] peer-to-peer overlay network, and the Data Store service, a generic distributed object store that builds on the JChord platform.

2.2 Middleware System Model

The objects in a distributed application are partitioned among the address-spaces in the distributed system. Application objects can hold both intra-address-space references and inter-address-space references to other objects. Intra-address-space references are known as *local references*, while inter-address-space references are known as *remote references*. From the perspective of a particular object, those objects in the same address-space are *local objects*, while those in remote address-spaces are *remote objects*.

A method call performed on a local object is known as a *local method call* and a call made on a remote object is known as a *remote method call*. In a method call, the *caller* is the object that performs the invocation and the *target* is the object on which the invocation is performed. In a remote method call, the caller is said to exist in the *client-side* address-space while the target is said to exist in the *server-side* address-space. It is important to note that the terms *local* and *remote* are relative in these contexts.

Figure 2.1 shows a distributed application in which there are four address-spaces (represented by solid squares) that exist on three machines (represented by dotted rectangles). There are a number of application objects (represented by circles), which hold both intra-address-space references (solid arrows) and inter-address-space references (dashed arrows).

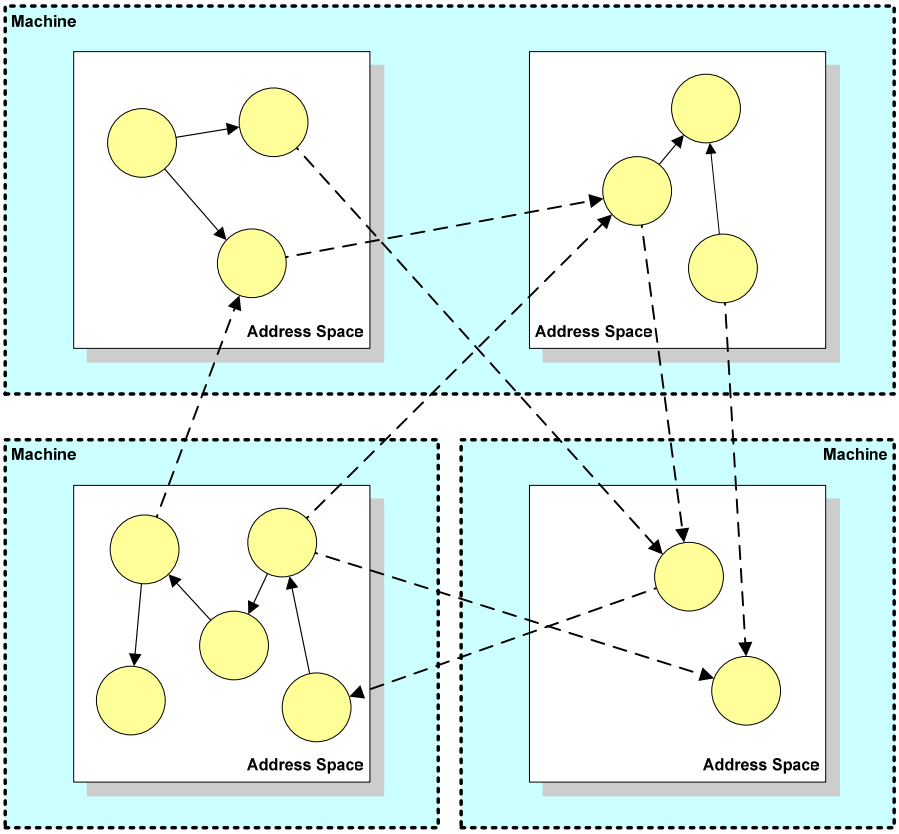


Figure 2.1: A distributed application showing both intra- and inter-address-space references.

Every middleware system provides an *infrastructure*, which is its point-of-presence in each address-space in the distributed system. The infrastructure is responsible for handling communication between address-spaces and is commonly implemented as a run-time system or a set of libraries.

In practice, direct remote references cannot be implemented without support from the operating system or virtual machine in which the application executes. Middleware systems typically implement remote references by associating an object identifier with each remote accessible object. This object identifier differentiates a given object from all other remotely accessible objects in the same address-space. An object can be uniquely identified in the distributed system by its object identifier and the identity of the address-space in which the object exists.

Each remote reference is a combination of an object identifier and an address-space identifier. An application object that holds a remote reference really holds a local reference to an object that contains this information. *Proxy objects* [23] can be used to make remote references type-compatible with the corresponding remote objects. Proxy objects are local handles to remote objects that implement the same methods as the corresponding remote objects.

Any methods that the application invokes on the proxy object are propagated across the network to the associated remote object. The client-side application therefore uses the proxy object as though it were the remote object, thereby introducing location transparency into the application. The abstraction of proxy objects is not logically necessary but many middleware systems provide it to allow local calls and remote calls to be performed in the same manner.

When a remote method call is performed, the call is propagated through the client-side infrastructure, across the network to the server-side infrastructure based on the address-space identifier, and onto the referenced object based on the object identifier. The part of the server-side infrastructure that performs the local method call on the exposed object is known as the *skeleton*.

Figure 2.2 shows the same application as Figure 2.1, with the middleware infrastructure revealed. The middleware infrastructure in each address-space associates an identifier with each local object that is remotely accessible (labelled A, B, etc.). Each address-space is identified by a number (1, 2, etc.). The remote references are shown as infrastructure objects that identify remote objects using a combination of address-space identifier and object identifier (labelled 2A, 4B, etc.).

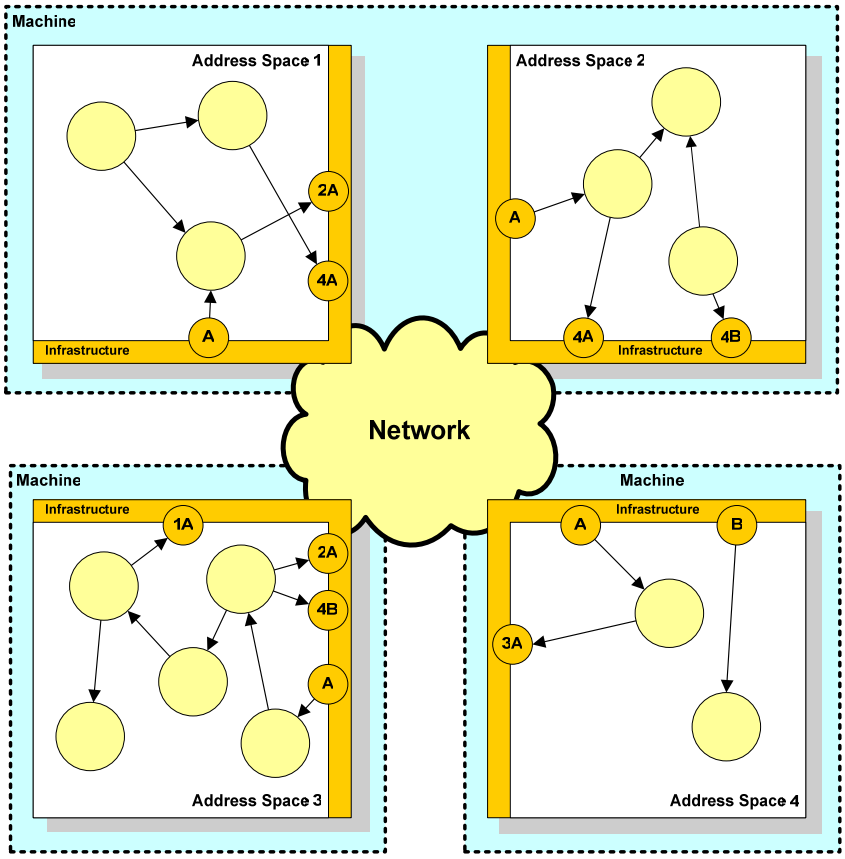


Figure 2.2: A distributed application in which the middleware infrastructure is shown.

Programmers must register objects with the middleware infrastructure in order to expose them to remote access. On registration, an identifier for the object is created and a mapping from this identifier to the exposed object is defined. Registration prepares the infrastructure to handle incoming calls to the object.

In some middleware systems, programmers are required to introduce support for distribution into application classes by extending special middleware infrastructure base classes or implementing special interfaces. These semantic restrictions vary from system to system and are investigated in more detail in the next chapter.

2.2.1 Remote Method Calls

Figure 2.3 shows the flow of control when object *A* performs a remote method call on the *sayHello()* method of object *B*.

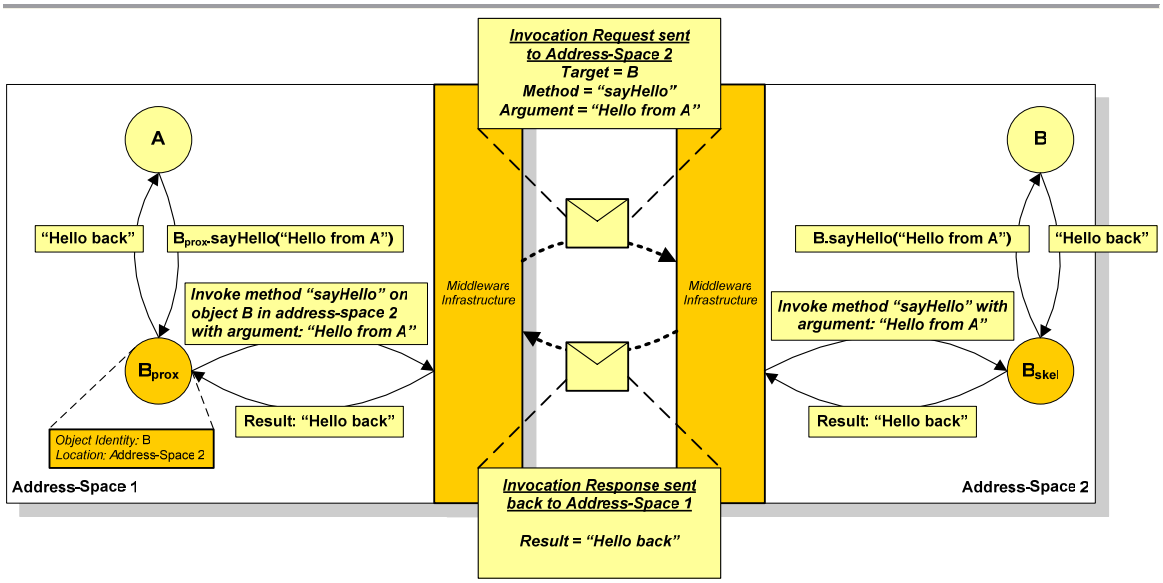


Figure 2.3: Flow of control in a remote method call.

Conceptually, object *A* holds a remote reference to object *B*. In reality, object *A* holds a local reference to a proxy object called B_{prox}. Object B_{prox} is type-compatible with object *B* and appears to object *A* as though it really is object *B*. The proxy object stores an address-space identifier and object identifier, which in combination identify object *B* uniquely within the distributed system. When object *A* attempts to call *sayHello()* on object *B*, it calls the *sayHello()* method on B_{prox}. This proxy object accesses the local middleware infrastructure and instructs it to perform a remote call on the *sayHello()* method of object *B* in address-space 2 with the supplied argument.

The client-side middleware infrastructure constructs and transmits an *invocation request* to the server-side middleware infrastructure in the same address-space as object *B*. The invocation request contains the target object information, method name and associated argument. The server-side middleware infrastructure uses the target object information to identify the skeleton associated with object *B*, B_{skel} in this case. Once the call has been performed, the results are returned via the same path in reverse, in an invocation response that travels back across the network.

2.2.2 Marshalling

Method calls performed on proxy objects are converted into invocation requests in a process known as *marshalling*. The server-side conversion of an incoming invocation request back into a method call is known as *un-marshalling*. When marshalling arguments and return values, the middleware must determine which parameter-passing

mechanism will be applied to each of the objects that cross address-space boundaries. Most DOMs provide pass-by-value and pass-by-reference mechanisms which behave as follows. Consider the distributed application shown in Figure 2.4, in which object *A* holds a remote reference to object *B* and a local reference to object *C*.

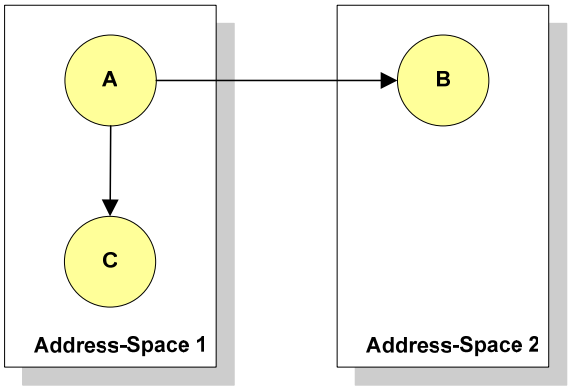


Figure 2.4: An application before a remote method call is performed on object *B*.

If object *A* calls a method on object *B* and passes object *C* as an argument, then *C* can be marshalled in one of several ways:

- If passed *by-value,* object *C* will be copied to address-space 2 as shown in Figure 2.5. Any method calls that object *B* performs on the argument will be executed on the copy in address-space 2. The middleware system must encode any object that is passed by-value into a stream of bytes that represents its internal state, for transmission across the network. This process is known as *serialization*. The decoding of a byte stream back into application objects is known as *deserialization*.

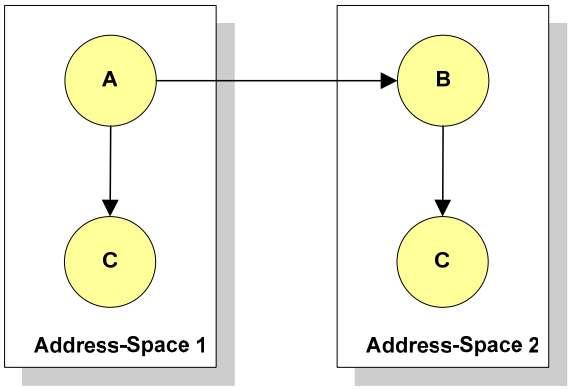


Figure 2.5: Object *C* has been passed by-value.

- If passed *by-reference,* a remote reference to object *C* will be passed to address-space 2 as shown in Figure 2.6. Any method calls that object *B*

performs on the argument will result in remote method calls to object *C* in address-space 1. The middleware system must serialize references to objects that are passed by-reference, rather than serializing the objects themselves.

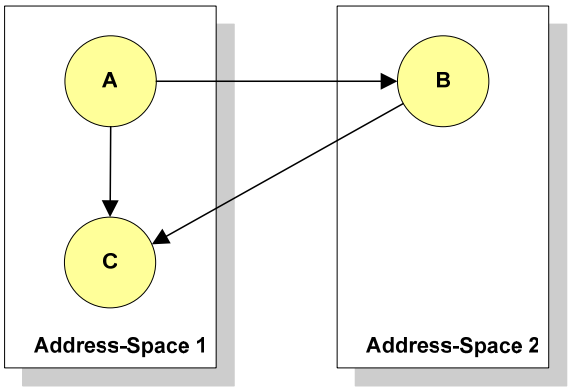


Figure 2.6: Object *C* has been passed by-reference.

- Some systems also offer pass-by-migrate semantics. If passed *by-migrate,* object *C* will be moved to address-space 2 as shown in Figure 2.7. Any method calls that object *B* performs on the argument will execute locally on object *C*, which is now in address-space 2.

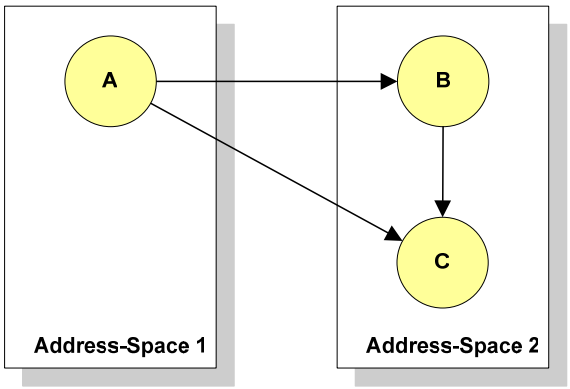


Figure 2.7: Object *C* has been passed by-migrate.

2.2.3 Smart Proxies

The cost of performing remote method calls is many orders of magnitude greater than local method calls so it is often desirable to minimize the number of remote method calls that are performed by distributed applications. *Smart proxies* [24] are proxy objects that cache some of the target objects' state thus allowing access without the expense of remote method calls. Smart proxy functionality overlaps with that provided by Distributed Shared Memory systems. Smart proxies allow multiple copies of cached state

to exist, analogous to object-based DSM systems. However, DOMs using smart proxies do not typically provide integrated coherency mechanisms, while DSM systems do.

2.2.4 Remote Object Instantiation

The creation of objects in remote address-spaces is known as *remote object instantiation*. Instead of performing object instantiation directly in the local address-space, an application can instruct the middleware to perform instantiation in a remote address-space on its behalf. The application specifies the class of object it wishes to create and possibly supplies some initialization arguments to its local middleware infrastructure. This information is propagated across the network, the object instantiated and a remote reference returned to the application.

2.2.5 Object Migration

Object *migration* is the movement of application objects between address-spaces in the distributed system without loss of referential integrity. Migration is useful as it allows applications to adapt their distribution dynamically to handle changing requirements. It is implemented by copying the migrating object to the new address-space, then updating all references to the old copy to refer to the new copy. The middleware must ensure that the application remains consistent while these operations are performed. References may be duplicated, passed between address-spaces, on the execution stack or inaccessibly in-flight between address-spaces, making them difficult to update. When object *A* moves from address-space 1 to address-space 2, there are four steps that must be carried out:

- A copy of object *A* is created in address-space 2.
- Local references that exist to the copy of object *A* in address-space 1 must be changed into remote references to the migrated copy of object *A* in address-space 2.
- Remote references from application objects in address-space 2 to the copy of object *A* in address-space 1 must be changed into local references to the migrated copy of object *A* in address-space 2.
- Remote references from application objects in any other address-spaces must be updated to reference object *A* in address-space 2, rather than address-space 1. This can be achieved lazily using a *tombstone,* which is an object that remains in address-space 1 to record that object *A* has migrated. Remote

references are updated using the information in the tombstone only when they attempt to access object *A* in address-space 1.

These steps need not be carried out eagerly at migration time, but references must be updated before any operations are performed on them.

2.2.6 Code Distribution

The code distribution problem is that of ensuring that the necessary application code is available in each address-space. Some middleware systems assume that all code is available in all address-spaces and defer responsibility for code distribution to programmers. In languages that support the dynamic loading of classes, such as Java, a network-accessible code repository available to all address-spaces in the distributed system can be created. If the infrastructure requires particular code, it can be obtained from the repository and dynamically loaded. Code repositories implemented in a logically centralized manner are a single point of failure and may become heavily loaded so it may be desirable that they are implemented in a scalable, distributed fashion.

2.3 JChord Case Study

The case study employed in this thesis consist of two parts: a Java implementation of the Chord [22] peer-to-peer protocol called JChord and a Data Store service that makes use of this peer-to-peer network. This case study is used to exemplify the limitations of current middleware systems and to describe the design and implementation of the RRT.

2.3.1 JChord

The following quotation from Norcross, Dearle, Kirby, and Walker [25] describes JChord:

> *JChord is our implementation of the Chord [22] peer-to-peer look-up protocol. This implementation provides a peer-to-peer overlay that supports Key-Based-Routing (KBR) [26] for addressing nodes in the underlying network. Under a KBR scheme every entity addressable by an application has an associated* M*-bit key value (where M is a system constant), and every key value maps to a unique live node in the overlay network. Up-calls from the routing layer inform the application layers of changes to the key space, thus allowing an application to be aware of changes to the set of keys that map to the local node.*

> *Chord is a ring-based protocol, which at the simplest level requires each node to maintain only a pointer to its immediate successor in the ring. Each node also has a unique key and the ring is arranged in key order modulo* 2^M*. The Chord protocol supports a single lookup operation, which takes a key value and returns the network address of the Chord node to which the key value maps. A look-up on key* K *will yield the address of the node* N *whose key* K_N *is the first of the ring members to succeed* K *in the key space. In this way the Chord protocol provides a distributed hash function that maps from keys to overlay nodes. Each node maintains a list of nodes that follow it in the ring, known as its successor list. A successor list of size* L *allows the ring to survive the failure of up to* L-1 *adjacent nodes. This provides resiliency of the ring and the look-up protocol, though further measures are required to ensure integrity of the data structures hosted by ring nodes.*

The following example illustrates how an archetypal middleware system could support remote calls in JChord. All code fragments are shown in Java [17]. Figure 2.8 shows the classes in the example application and indicates whether these classes are created by programmers, generated using middleware tools or provided as part of the middleware infrastructure.

To create a class supporting remote access, the programmer initially creates an interface (*IJChordNode*) that defines the operations that will be provided by this class. An implementation of this interface is also created by the programmer (*JChordNode*). The main application (*Application*) accesses this implementation through the interface only. A proxy class called *JChordNodeProxy* that implements the *IJChordNode* interface is generated using middleware system tools. It makes use of the middleware infrastructure classes *Middleware* and *RemoteReference* to perform remote method calls and identify objects in the distributed system.

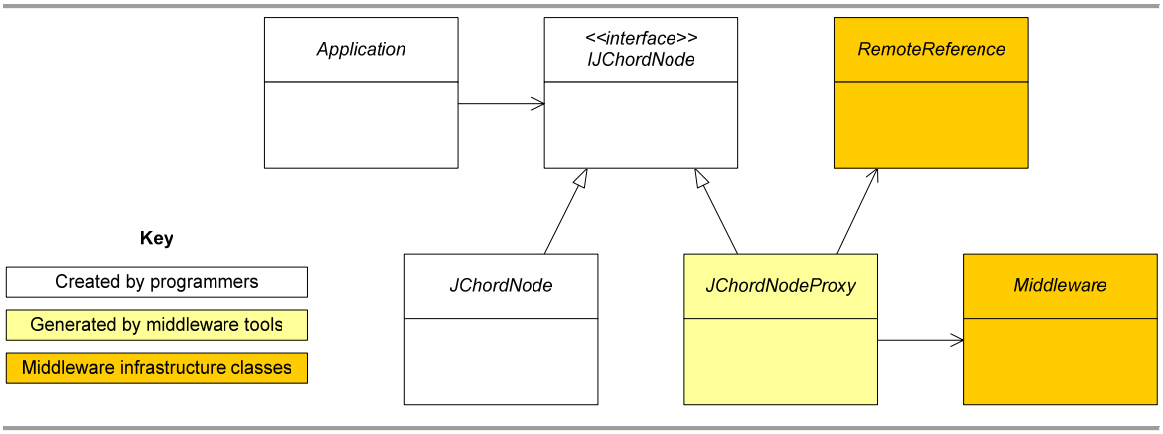


Figure 2.8: The classes involved in the remote method call example.

The programmer defines the common interface, *IJChordNode*, and corresponding implementation, class *JChordNode*, as shown in Figure 2.9.

```
public interface IJChordNode {
      IJChordNode lookup(Key key);
}

public class JChordNode implements IJChordNode {
      public IJChordNode lookup(Key key) {
            /* Find JChordNode associated with the supplied key */
            return …;
      }
}
```

Figure 2.9: The *IJChordNode* interface and *JChordNode* class.

Figure 2.10 shows the remote reference implementation class, called *RemoteReference*, which uses an *InetSocketAddress* object (containing an IP address and port pair) to identify the target address-space and an object counter to identify the remote object within its address-space.

```
public class RemoteReference {
      public InetSocketAddress isa = null;
      public int objectID = 0;
}
```

Figure 2.10: The remote reference class.

Figure 2.11 shows the signature of the *Middleware* class, which provides a method called *callRemoteMethod()* through which remote method invocations can be performed. This method takes a remote reference to the target object, the name of the

method to call, and any associated arguments packaged into an array. This method performs all marshalling, including serialization of arguments, and transmits the invocation request to the remote address-space identified by the remote reference. The method call is then performed and the return value passed back across the network.

```
public class Middleware {
      public static Object callRemoteMethod(
            RemoteReference remoteRef,
            String methodName,
            Object[] arguments) {…}
}
```

Figure 2.11: The middleware infrastructure class.

The proxy class associated with class *JChordNode*, called *JChordNodeProxy*, can be implemented as shown in Figure 2.12. Typically this class would be generated using middleware tools. This proxy class implements the *IJChordNode* interface and so is type-compatible with the associated application class. Instances of the proxy class hold remote references to the instances of *JChordNode* with which they are associated, in the *remoteRef* field. This field is initialized by the middleware system when the proxy object is created. The *lookup()* method forwards calls to a remote instance of *JChordNode* using the remote invocation method in the *Middleware* class, based on this remote reference.

```
public class JChordNodeProxy implements IJChordNode {
      /* Reference to remote instance of JChordNode */
      private RemoteReference remoteRef = …;

      public IJChordNode lookup(Key key) {
            Object[] args = new Object[] { key };
            return (IJChordNode) Middleware.callRemoteMethod(
                  remoteRef,
                  "lookup",
                  args);
      }
}
```

Figure 2.12: The *JChordNodeProxy* proxy class.

The *main()* method of the *Application* class shown in Figure 2.13 contains a reference typed as *IJChordNode*. This allows the application to reference either an instance of *JChordNode* or an instance of *JChordNodeProxy* associated with a remote instance of *JChordNode*. The use of proxy objects conceals the inter-address-space communication from the application, allowing remote method calls to be made in the same manner as local method calls.

```
public class Application {
      public static void main(String[] args) {
            /* Object 'node' may be local or remote */
            IJChordNode node = …;
            /* Create a key */
            Key key = new Key(12345);
            /* Method call may be local or remote */
            IJChordNode result = node.lookup(key);
      }
}
```

Figure 2.13: The *Application* class, which can perform remote method calls.

2.3.2 Data Store

A generic key-based distributed Data Store service has been constructed using JChord. The Data Store service is implemented as a series of Data Store objects distributed across the available machines. When an application object is inserted into the store, the object is associated with a key. A mapping between key and object is recorded by one of the Data Store objects. Each Data Store object is collocated with a JChord node.

The Data Store service decides in which Data Store object to store a given application object based on the application object's key. Application object *A* with key *K* is stored in the same address-space as the JChord node which owns key *K* (that is, the address-space returned when a JChord lookup of key *K* is performed). The JChord ring does not store application objects. Rather it acts as a distributed hashing mechanism by which the application objects stored by the Data Store service can be partitioned among the available Data Store objects.

Applications store objects in the Data Store service using a local point-of-presence (POP), which exists in all address-spaces. The Data Store POP allows applications to store and retrieve objects, and implements the interface shown in Figure 2.14. Each Data

Store object can hold local references to copies of stored application objects or remote references to stored application objects in remote address-spaces. When storing an application object in the Data Store service, the programmer decides whether the Data Store should hold a remote reference to the stored object or hold a duplicate copy of it, using the *storeByReference* argument of the *store()* method.

```
public interface IDataStorePOP {
      Key store(Object objectToStore, boolean storeByReference);
      Object retrieve(Key key);
}
```

Figure 2.14: The *IDataStorePOP* interface.

Each Data Store object presents the *IDataStoreInternal* interface shown in Figure 2.15 to the Data Store point-of-presence. This interface allows the Data Store POP to add and remove mappings between a particular key and object in a particular Data Store object.

```
public interface IDataStoreInternal {
      void put(Key key, Object objectToStore);
      Object get(Key key);
}
```

Figure 2.15: The *IDataStoreInternal* interface.

The *store()* method provided by the *IDataStorePOP* interface generates a key for each stored application object. It determines which remote Data Store object should store the application object by performing a JChord lookup to find an address-space and obtaining a reference to the Data Store object in that address-space using the underlying middleware system. The stored object must then be passed by-reference or by-value to the *put()* method provided by the chosen Data Store object. The parameter-passing semantics of this method must be decided dynamically based on programmer input, irrespective of the class of object stored.

Figure 2.16 shows a distributed system in which the Data Store service is deployed. Each Data Store object (labelled *DS1 – DS5*) is collocated with a JChord node (labelled *JC1 – JC5*). Each JChord node holds a remote reference to the next JChord node in the ring. The Data Store POP (labelled *DS POP*) can remotely access individual Data Store objects. Note that the Data Store POP need not be collocated with JChord nodes or

Data Store objects. The diagram shows the result of storing application objects *A1* and *A2* (in the top left address-space) in the Data Store object *DS1* by-reference and by-value respectively. *DS1* holds a remote reference to *A1* and a copy of *A2*.

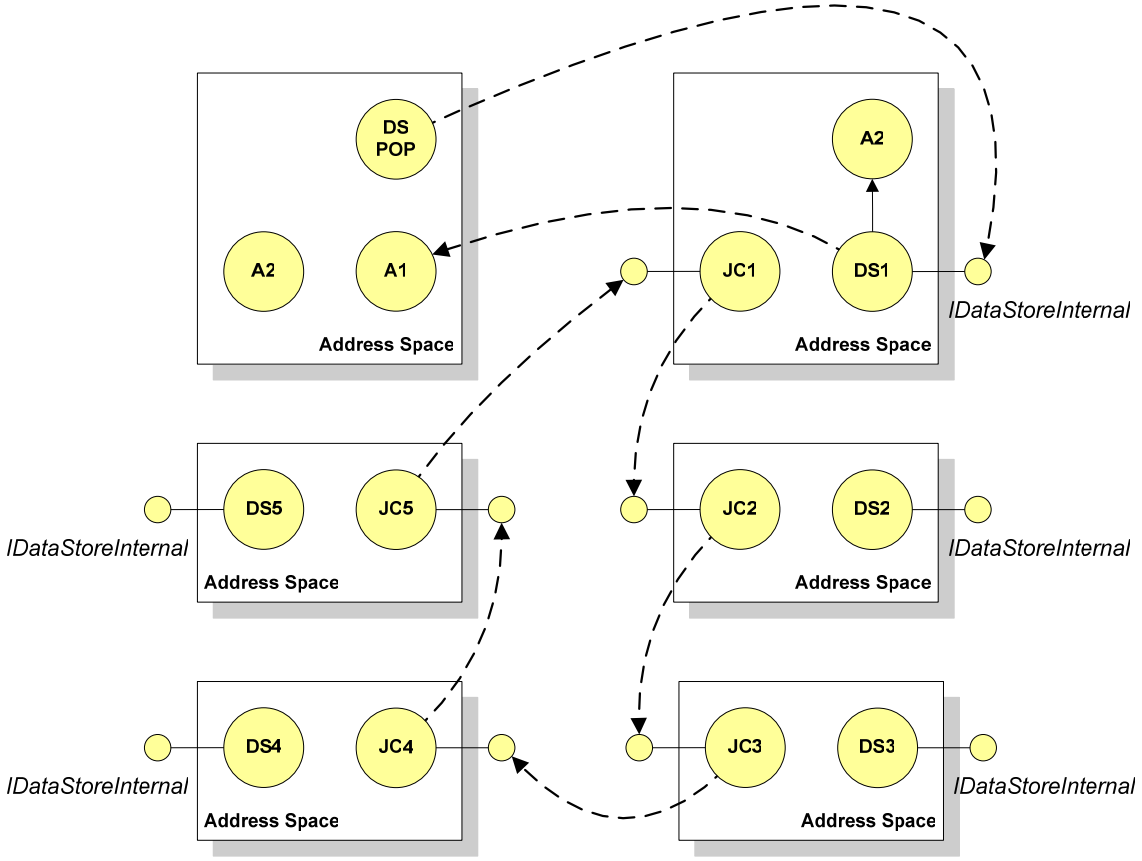


Figure 2.16: A JChord ring showing the remotely accessible Data Store objects.

If the application executing in the top left address-space shuts down, it is desirable that object *A1* be moved to another live address-space in the distributed system. However, it may be remotely referenced by clients that accessed it through the Data Store service so, to ensure referential integrity, object *A1* must be migrated to another address-space rather than simply copied.

2.3.3 Implementing the JChord Case Study

JChord has been implemented as part of the *Secure Location-Independent Autonomic Storage Architectures (ASA)* project [27] using the RRT prototype developed as part of the research described in this thesis. This case study is used to illustrate the novel features of the RRT.

JChord and the Data Store have a number of properties that would make them difficult to implement using traditional middleware systems and so are particularly suitable as a case study, namely:

- JChord was initially developed as a local application used to simulate peer-to-peer networks on a single machine. The application was tested and stable so it was desirable that distribution be introduced with minimal changes to application logic, in order to reduce the likelihood of new errors.
- JChord is used as a tool in a research environment to investigate the properties of peer-to-peer systems and so must adapt to changing requirements, such as the introduction of new routing algorithms, with minimal programmer effort.
- Each JChord node presents a multiplicity of interfaces to clients. It provides lookup functionality to applications, low-level ring maintenance operations to other nodes and control over ring configuration to managers.
- References to remote JChord nodes must cache some of the state of the remote nodes locally for efficiency and for use during failure, to identify the failed nodes.
- It must be possible to create nodes on remote machines to automate ring deployment.
- The Data Store point-of-presence needs to alter parameter-passing semantics dynamically when accessing the individual Data Store objects.
- It must be possible to migrate objects that are remotely referenced by Data Store objects from one address-space to another. This allows the Data Store service to adapt to changes in the distributed system without loss of referential integrity.
- Distribution policies to control the deployment of JChord ring nodes and the migration of stored objects must be defined.

2.4 Summary

This chapter has described the middleware system model adopted in this thesis in a system-independent manner. The concepts described here do not comprise an exhaustive list of all features that a middleware system must provide, nor are all possible middleware features described. This chapter provides a framework describing common

functionality against which existing systems can be evaluated. The JChord case study is introduced and its requirements summarized.

Chapter 3

Related Work

This chapter investigates and evaluates 'first generation' industry-standard and 'second generation' research-based middleware systems. The limitations of existing systems are described in the context of this related work.

3.1 Introduction to Chapter 3

This chapter categorizes middleware systems into 'first generation' systems, which are industry-standard systems in wide use, and 'second generation' systems, which are research-based systems that are not in extensive use outside of academia. The chapter concludes by re-examining the limitations of existing systems, which were introduced in Chapter 1, in the context of the middleware systems described here.

3.2 First Generation RPC Systems

RPC systems allow clients to call remotely accessible procedures, rather than the methods of specific objects. RPC systems provide two mechanisms to permit callers to execute code on remote machines, namely:

- A mechanism to identify remote procedures.
- A mechanism to encode arguments and return values that are passed across address-space boundaries.

The main limitation of RPC systems is a lack of support for remote references, with the consequence that arguments cannot be passed by-reference during remote calls. RPC systems allow applications to provide service-oriented functionality to remote clients. All data required to perform any operations must be passed as arguments during calls.

The most widely used traditional RPC system is *Sun RPC* [7] though there are others, such as *DCE-RPC* [28] (from which Microsoft's COM RPC mechanism was developed) and *ISO-RPC* [29]. The differences between these different approaches are discussed in Barkley [30]. There are also modern RPC systems such as *XML-RPC [31]*.

3.2.1 Sun RPC

SunRPC [7] allows clients to execute code on remote machines. A programmer defines a numbered list of procedure declarations and creates a server application that implements these procedures. The server registers itself with the RPC infrastructure to make the procedures available. Clients perform remote calls via proxies which access the remote procedures based on the identities of the server applications, procedure numbers and the names of the machines on which the servers run.

Programmers initially define application structure in terms of programs, versions and procedures using *RPC Language (RPCL)*. A program declaration describes a single remotely accessible server application and contains one or more version declarations. Version declarations define the particular procedures provided by each version of the server and provide multiple views over the server.

A code generation tool called *RPCGen* is used to generate server implementations containing method stubs, to which programmers add application logic, and any associated proxies. When run, the server applications register themselves with the RPC infrastructure's connection listener, known as the *port mapper*. The port mapper listens for invocation requests from clients.

To perform a remote method call, a client performs a local method call on a proxy, which marshals the arguments. The call is passed into the RPC infrastructure which creates an invocation request containing a program identifier, a version number and a procedure identifier along with the marshalled arguments. The request is passed across the network to the server-side port mapper. The port mapper executes the described procedure on behalf of the client.

All arguments and return values are passed across address-space boundaries by-value. A data representation scheme called the *External Data Representation (XDR)* is used to define how primitive values and data structures are encoded. Encoded values do not contain any type information and so it is not possible to determine which values an arbitrary block of encoded data represents unless the types are known in advance.

SunRPC is representative of the functionality provided by all RPC systems. It provides a service-oriented model that allows clients to pass some arguments to a remote server, which performs some computation and returns a result. SunRPC, like all RPC systems, is not a suitable middleware system to use when creating distributed applications that depend on pass-by-reference semantics.

3.2.2 XML-RPC

XML-RPC [31] represents remote calls and serialized values in XML. Remote procedures are invoked by performing an HTTP POST request on an XML-RPC compliant web server. The POST data contains XML describing the:

- Method name to call.
- Method call arguments.

The XML-RPC specification is simplistic and much of the behaviour of an XML-RPC system is implementation-specific, preventing interoperability between different implementations. The structure of valid XML-RPC calls is described but there is no indication of how method names are mapped to application methods. The specification states that servers interpret the method name in any way they deem appropriate. XML-RPC provides no service description mechanism from which clients can determine the set of operations provided by a server.

Only pass by-value semantics are supported for remote calls and though complex types can be passed across the network, XML-RPC provides no type mappings to indicate how to associate programming language types with serialized objects. Further, the current specification does not indicate how cycles within the closures of serialized objects can be handled.

3.3 First Generation Distributed Object Models

Distributed Object Models (DOMs) represent a more object-oriented approach to distributed application development. They provide Remote Method Invocation (RMI) functionality that allows clients to call methods on identifiable objects in remote address-spaces. Applications can hold references to objects in remote address-spaces and both pass-by-reference and pass-by-value semantics are available for remote method calls.

This section describes the first generation DOMs, which are CORBA [8], Java RMI [9], Microsoft COM technologies [32] and Microsoft .NET remoting [10]. Each of these middleware systems has unique properties but all require programmers to follow similar steps in order to create remotely accessible objects:

1. Programmers must define the interfaces between distribution boundaries statically.
2. Programmers must decide statically which classes will implement these interfaces and thus support remote access. These classes are known as *remote classes* and must meet certain semantic requirements.
3. Programmers instantiate the remote classes and register them with the middleware infrastructure. Objects are associated with names that allow clients to obtain remote reference to them.

Only instances of classes that have been designed to support distribution can be exposed to remote access. Further, distribution-related decisions must be made at class

granularity in the first step but at object granularity in the third step, making these middleware systems difficult to use. The decisions that programmers make statically concerning support for remote access place constraints on the ways in which applications can be distributed.

All of the first generation DOMs described here exhibit similar limitations. The creation of distributed applications is both complex and error-prone. Programmers must decide which classes and interfaces will support remote access early in the design phase. They do not abstract over the distributed nature of the application meaning that the semantics of an application are tightly bound to its distribution. The introduction of distribution into an existing local application is difficult. Extensive changes to source code and possibly the application semantics are required. These DOMs do not adapt easily, rendering them inflexible to the requirements of applications that evolve over time.

3.3.1 Common Object Request Broker Architecture (CORBA)

The *Common Object Request Broker Architecture (CORBA)* [8, 33-35] is a complex DOM specification [33] that allows object interaction across address-spaces in an operating system and language independent manner. Multiple implementations of CORBA exist, such as MICO [36] and Orbix [37, 38].

The local infrastructure in a CORBA implementation is known as the *Object Request Broker (ORB)*. There is a single ORB per address-space, which is responsible for constructing all outgoing invocation requests to remote objects and handling all incoming invocation requests to local objects.

3.3.1.1 Creating a Distributed Application

The first stage in the design and creation of distributed applications using CORBA is the definition of interfaces for the remotely accessible classes. CORBA interfaces are defined independently of implementation language using the *Interface Definition Language (IDL)* [8]. All complex types that can be passed as arguments or return values are also defined in the IDL. IDL is object-oriented and is used only to specify the structure of the interfaces and types.

CORBA defines mappings between IDL and many programming languages, including Java, C, C++, ADA, Lisp and others. In order to implement the interfaces and complex types defined in IDL, programmers use language-specific IDL compilers to generate partial implementations of the interfaces and their associated proxy classes in a

programming language chosen by the programmers. These partial implementations contain method stubs without execution logic, which is subsequently provided by programmers. Support for remote access permeates classes making it difficult to change classes that are not accessible remotely into ones that are and vice versa.

Remotely accessible objects are associated with names in a CORBA *name service* when exposed to remote access. The name service is available to both servers and clients through the ORB. Clients in remote address-spaces can obtain references to the exposed objects by name. Implementations of remote classes are known as *servants*. IDL compilers typical generate one partial servant implementation per type defined in IDL though a single servant may implement several CORBA types.

Servants provide functionality to both the ORB and application programmers. It is common to inherit some of this functionality from a special base class. However, when the implementation language does not support multiple inheritance, servants cannot extend arbitrary other application classes, thereby affecting application semantics.

The *tie* approach solves this problem by allowing programmers to implement the servant functionality and application logic in separate classes. Using this approach, the servant extends the special base class but holds a reference to an instance of the remote class that implements the application logic. The remote class is free to extend any arbitrary base class as shown in Figure 3.1.

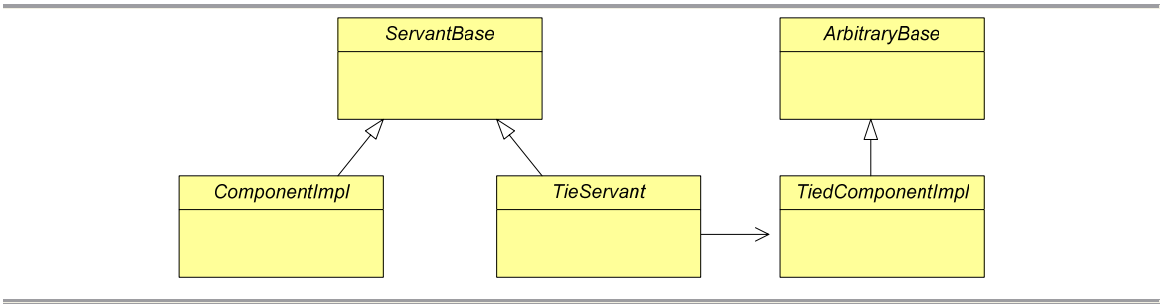


Figure 3.1: CORBA servant classes.

3.3.1.2 Dynamic Invocation

Programmers typically need static knowledge of the objects that clients will access at client compile-time. CORBA solves this problem by providing the *Dynamic Invocation Interface*, which allows clients to access remote objects for which static type information was unavailable at client compilation. Programmers must explicitly construct remote method call requests by defining the names of the operations and providing

arguments. If the arguments are of unknown types, the client must also describe the structure of these types to the infrastructure in order than they can be marshalled.

To allow programmers to construct servants that can *tie* to objects for which static type information was not available at servant compilation time, the *Dynamic Skeleton Interface (DSI)* is provided. DSI servants extract the name of the operation to call and any associated arguments from an incoming remote method call request, and perform the requisite call. CORBA does not serialize objects in a self-describing manner and so programmers must write code to extract type information from the request and deserialize the arguments. The *tie approach* offers increased flexibility at the cost of increased complexity. Programmers lose the abstraction over the inter-address-space communication afforded when using proxies and objects created from IDL. Since serialized data is not self-describing it is possible to construct applications in which methods are invoked with inappropriate arguments, particularly when using dynamic invocation [33]. This leads to unexpected application semantics and may cause run-time problems in strongly typed languages.

3.3.1.3 Implementation

To ensure interoperability among different CORBA implementations, there is a standard representation of remote references called the *Interoperable Object Reference (IOR)* that must be supported by all ORBs. ORBs must also be capable of performing inter-address-space communication using the *Internet Inter-ORB Protocol (IIOP)*. This protocol specifies the range, size and encoding for basic types and an encoding for complex types called the *Common Data Representation*.

CORBA objects, data structures and primitive values may be passed as arguments or return values when remote methods are called. Each parameter of the methods defined in IDL is marked as *in*, *out* or *inout* indicating whether that parameter should be passed by-value, by-result or by-value-result [39] respectively. When arguments are passed by-result or passed by-value-result, this indicates that the server will modify the arguments and copy the modified values back to the client. Pass-by-result semantics indicate that the server is not interested in the client-side value of the argument before the call and is simply using the argument as a way to return values to the caller. In languages that have no support for pass-by-result, such as Java, programmers must wrap, unwrap and update *out* and *inout* parameters to ensure the by-result semantics are preserved.

CORBA objects are passed by-reference by default and the *in*, *out* and *inout* semantics are applied to the associated IORs, not the objects themselves. Types may alternatively be defined in IDL as pass-by-value types and in such cases these semantics are adopted instead of the default. Only CORBA objects may be passed by-reference, leading to limitations with respect to shared data. Application flexibility and code reuse are hindered because passing semantics are defined statically.

CORBA is a powerful but complex middleware tool that exposes much of the complexity of distribution to programmers, particularly those using the dynamic invocation mechanisms. Programmers must define distribution boundaries statically, leading to inflexibility in application distribution.

3.3.2 Java Remote Method Invocation (Java RMI)

Java Remote Method Invocation (Java RMI) [9] is a DOM included as part of Java 2 Standard Edition (J2SE) [40]. It permits the creation of classes whose instances can be accessed remotely from other Java Virtual Machines (JVMs) [41]. Remotely accessible classes must implement interfaces that meet a number of requirements:

- The interfaces must extend a special marker interface (*java.rmi.Remote*) either directly or indirectly.
- Each method must throw a special remote exception class (*java.rmi.RemoteException*).

Programmers must ensure that the methods inherited from class *java.lang.Object*, the root of the class hierarchy in Java, are modified to support distribution. This is achieved either by extending a special Java RMI base class or by overriding these methods in any classes that support remote access. These requirements erode the abstraction over the network provided by the middleware system because support for distribution pervades classes.

3.3.2.1 Creating a Distributed Application

In the most recent version of Java (J2SE 5.0 [40]), the Java RMI infrastructure can make use of the reflection technology integrated into Java to provide generic skeleton and proxy implementations. Programmers can instead employ a Java RMI compiler to generate ancillary distribution-related code such as skeletons and proxies (the latter known in Java RMI as stubs) for all remotely accessible classes to avoid the run-time cost of reflection.

The process of exposing an object to remote access is known as *export.* On export, objects are registered with a name service called the *rmiregistry* that runs in a separate process on a known machine and port. Clients can obtain references to exported objects by contacting the *rmiregistry* and specifying the names with which the objects were registered initially. The *rmiregistry* returns remote references to the exported objects.

Java RMI determines the parameter-passing semantics to apply when remote methods are called based on whether the argument objects have been exported to remote access. Exported objects are always passed by-reference and all other objects are passed by-value. Even instances of classes supporting remote access are passed by-value if they have not yet been exported. Consequently, the parameter-passing semantics are tightly bound to the distribution of the application and can be unpredictable.

Java RMI implements distributed garbage collection using a reference-counting scheme in which each JVM keeps track of its own reference count for each remote object. This distributed garbage collection scheme cannot detect distributed cycles of garbage [42] and so an additional lease-based scheme is employed to ensure that exported objects in cycles will eventually be collected. Distributed garbage collection in Java RMI is complete but not safe - the integrity of remote references cannot be ensured.

3.3.2.2 Semantic Limitations

Java RMI places a number of semantic limitations on classes that support remote access. If a remote class extends a super-class that is not remote, it must override all of the inherited methods to meet the semantic restrictions placed on remotely accessible methods with respect to network error related exceptions. Only methods written with concern for distribution can be inherited directly. Programmers may be forced to re-implement inherited methods as only arguments of interface types can be passed by-reference. The advantages of inheritance in terms of code reuse can be lost.

Classes supporting remote access and their associated proxy classes are type-compatible in terms of interface only. Instances of the proxy class cannot be cast into the associated remote class, which can have consequences for the client applications. For example, the observer/observable design pattern [43] is implemented in Java using the *Observable* class and the *Observer* interface. Instances of the classes that extend *Observable* allow observers to register an interest in them. These observers implement the *Observer* interface, which includes a call-back method that is invoked by *Observable* objects to indicate that events have occurred.

Each observer can be registered with multiple *Observable* objects and so the *Observable* objects pass references to themselves to these call-back methods, to allow the observers to identify where the events originated. Since the call-back methods take arguments typed as the *Observable* class, proxies to instances of classes that extend *Observable* cannot be passed in their place. Proxies to instances of *Observable* classes are not themselves instances of the *Observable* class. Observers and *Observable* objects cannot therefore be separated into different address-spaces.

One solution appears to be the conversion of the *Observable* class into an interface. This is a viable solution to the problem but, because the newly created interface would need to have explicit support for remote access, it would be forced to extend the *Remote* interface and meet the semantics limitations described above. Support for distribution would permeate library code, whether the classes were used in distributed applications or not.

Java RMI places major semantic limitations on classes that support remote access and the parameter-passing semantics are tightly bound to the application distribution. It provides no support for the dynamic reconfiguration of application distribution.

3.3.3 Distributed COM

Distributed COM (DCOM) introduces distribution into the *Component Object Model (COM)* [32, 34, 44]. COM is a Microsoft developed component technology, primarily for use with the Windows operating system. Although reduced feature implementations exist on other operating systems (MacOSX and several flavours of Unix), this tight association with Windows restricts interoperability across operating systems. COM and DCOM have been subsumed into a single entity called COM+ with the release of Windows 2000.

COM is a binary standard that permits programmers to create *components*, which are coarse-grained reusable units of software that can be combined to create complete applications. COM components implement multiple interfaces that are defined in *Microsoft Interface Description Language (MIDL)*, a language that provides mechanisms similar to CORBA IDL. An MIDL compiler is used to produce partial implementations in one of several programming languages including C, C++ and Visual Basic. The COM specification indicates how the compiled code in the component is structured and provides binary level interoperability between components written using different implementation languages.

Each interface and implementation class is identified using a *Globally Unique IDentifier (GUID)*. A component is created using a factory by specifying the GUID of the required component and the context in which the component should be created. Programmers can instruct factories to instantiate components on remote machines by either identifying the machine explicitly or deferring to the *Service Control Manager (SCM)*.

The SCM makes decisions based on component information that has been explicitly stored in the Windows registry by programmers. This information may include details about the machines that are present in the distributed system and the component factories provided by these machines. In this way, object placement policies are defined in terms of one-to-one mappings between component identifiers and machines.

Clients can access (particular interfaces of) remotely accessible components using proxies in the conventional manner. Though a DOM, DCOM is built on top of DCE-RPC [28], an RPC technology, and supports pass-by-reference semantics by extending the data representation to include interface references. Like CORBA, DCOM determines parameter-passing semantics based on the statically defined MIDL.

DCOM does not support garbage collection. Once instantiated, component lifetime must be managed manually by programmers using a built-in reference-counting scheme. Programmers must increase and decrease the reference count associated with each component as they create and destroy references to it.

Programmers using DCOM must support remote access explicitly in each component. Though DCOM enforces the use of factories when instantiating components thereby making the location of components transparent with respect to the client, it is inflexible in terms of changing distribution boundaries. DCOM does not support dynamic alterations to application distribution and does not permit migration. Modifications to existing components require considerable programmer effort which, combined with the lack of support for automatic memory management, makes the creation of distributed applications difficult and error-prone.

3.3.4 Microsoft .NET framework

The *.NET framework* [45] is a component technology that includes operating system extensions to provide a run-time infrastructure for applications, DOM functionality known as *.NET remoting* [10] and technology for the provision of Web

Services [46]. The .NET framework supersedes COM though the two technologies are interoperable for legacy reasons.

Currently, the only fully featured implementation of the .NET framework runs on the Windows operating system and so cross-platform interoperability is limited. Two different feature-limited versions of the framework for other operating systems including MacOSX and GNU/Linux exist, namely Microsoft's own *Shared Source Common Language Infrastructure (SSCLI)* [47] and the *Mono* project [48].

3.3.4.1 Microsoft Intermediate Language (MSIL)

.NET components are known as *assemblies* and can be written in any .NET enabled language, of which the main ones are C#, Managed C++ and Visual Basic. All .NET-enabled languages are compiled into a single intermediate language called *Microsoft Intermediate Language (MSIL)*, a singly-inherited, object-oriented language with automatic memory management. The run-time system, called the *Common Language Runtime (CLR)* compiles the MSIL into platform-specific binary code at run-time.

The use of this common intermediate language allows cross-language interoperability. Classes written in one .NET enabled language may extend classes written in another or may throw exceptions across language boundaries. Assemblies written in different languages can be tightly integrated but the cost of this interoperability is the loss of each language's unique properties. Managed C++ illustrates this problem clearly; it is not source compatible with C++ and, in effect, is C# with C++ syntax. In general, there are no reasons to choose one .NET implementation language over another beyond programmer preference.

3.3.4.2 Creating a Distributed Application

Programmers need not define separate interfaces for the classes that support remote access though all remotely accessible classes must extend a special base class called *MarshalByRefObject*. There are two conceptually different approaches to making instances of classes available: object-based and class-based. The first approach adheres to the typical DOM model in which programmers instantiate objects then make them accessible remotely by registering them with the infrastructure. The second approach adheres to a Web Service style model in which programmers register classes with the infrastructure rather than objects. This indicates to the run-time that any instances of the

specified classes can be used to handle incoming method calls. The .NET infrastructure then creates instances of these classes on each call or on first access.

The first step using either approach is to create and register a communications channel for the local application domain. Application domains are analogous to address-spaces and multiple channels can be registered with a single application domain. Channels bind to particular ports and can use either a proprietary binary transport protocol or SOAP. To make objects remotely accessible, the programmer registers them with channels using names that identify them uniquely among the other remotely accessible objects in the application domain. All remotely accessible objects registered with the .NET run-time in that application domain will be accessible via any of that domain's channels.

Clients using the .NET framework obtain proxy objects that reference objects in remote address-spaces using the registered names and invoke methods in the usual DOM fashion. The .NET infrastructure provides special treatment for proxy objects allowing a client to cast a proxy object into the same class as its associated remote object.

The .NET framework implements a lease-based distributed garbage collection scheme and garbage collects objects with expired leases. This approach is complete but not safe and can result in the collection of live objects.

The .NET remoting infrastructure places semantic restrictions on the inheritance hierarchies of classes supporting remote access and tightly binds parameter-passing semantics to the distribution of the application. No support for migration or the instantiation of objects in remote address-spaces is provided, meaning that applications are inflexible to dynamically changing requirements.

3.3.5 Component Models

Component models such as *Enterprise Java Beans (EJB)* [49] or the *CORBA Component Model (CCM)* [50] are specifications defining how programmers can create software components that execute within server-side application containers. The application containers manage non-functional considerations such as transactions, persistence, load-balancing or security, allowing programmers to abstract over these concerns.

Component models use first generation middleware systems to provide their underlying inter-address-space connectivity, for example, EJB builds on Java RMI and

CCM is part of the CORBA specification. Consequently, these systems are not examined any further.

3.4 First Generation Service-Oriented Architectures

Service-Oriented Architectures (SOAs) provide one-way-call or call-return semantics and so superficially appear similar to RPC systems or DOMs. However, there are a series of differences between SOAs and more traditional systems [51]. DOMs provide an abstraction over operations on remote objects by implicitly passing messages between address-spaces to perform remote calls, as the result of local calls on proxy objects. SOAs are often more explicit about this message passing, bringing it up into the application level. Indeed, some authors define services as entities that operate over messages [52]. SOAs operate at the granularity of the service, rather than the object, resulting in differences in the way in which application requirements are modelled. Services are intended to provide abstractions at the level of business entities.

3.4.1 Web Services Architecture

The *Web Services Architecture* [46] is a W3C specification for *Web Services*, which provides a standardized mechanism to allow interoperability between applications across programming languages and operating systems (promoted via the WS-I Basic Profile [53]). Web Services allow web servers to expose methods to remote access by clients using an XML-based protocol called *SOAP* [54-56][1]. SOAP implementations are available on many platforms and for many languages, for example, *Apache Axis* [57] and *Microsoft .NET Web Services* [45].

Each Web Service is associated with a particular URL. HTTP requests posted to that URL correspond to calls on that service. The body of each request is a SOAP message containing the name of the method to invoke and any arguments in serialized form. Web Services do not support remote references and can employ only pass-by-value semantics for remote method calls. Consequently, references are typically exchanged between web services in an ad-hoc fashion, for example, using invoice numbers rather than remote programming language references to invoice objects.

[1] Prior to version 1.2, SOAP stood for *Simple Object Access Protocol*, though it was also known as the *Service Oriented Architecture Protocol*. The latest specification does not spell out the acronym.

Methods invoked on Web Services are performed on underlying *service objects*. The majority of Web Service technologies do not allow programmers to associate particular service objects with Web Services. They operate at class granularity, allowing programmers to associate implementation classes with Web Services. The Web Services infrastructure instantiates the specified class to handle incoming calls on a per-call basis, a per-session basis or on first access.

The *Web Services Description Language (WSDL)* [58] is used to describe the methods provided by particular Web Services. WSDL defines the available methods in an abstract manner, in terms of the requests that clients can make and the responses they can expect to receive. WSDL then defines services in terms of:

- Abstract method definitions.
- URLs for the services.
- The transport protocols that must be employed to access the services.

Programmers can access Web Service functionality from client applications by generating proxies based on the WSDL. Using Web Services, all types that may be passed as arguments are described and associated with name-spaces in the WSDL describing the service. Programmers define application-specific mappings between namespaces and concrete programming language classes.

3.4.2 JBoss Remoting

The *JBoss Enterprise Middleware System*, known as JBoss or JEMS [59], is a Java-based application server for developing enterprise and web applications. JBoss *AOP Remoting* uses aspect-oriented programming techniques to instrument instances of existing classes for remote access. AOP Remoting allows the exposure of application objects to remote access as services using SOAP or Java RMI (described later in Section 3.3.2).

AOP Remoting places some semantic restrictions on the classes of object that can be exposed. All classes must provide default constructors and all method arguments and return values must be *Serializable*. AOP Remoting adopts a service-oriented model in which methods of the underlying objects are provided to remote clients, if the objects meet the above semantic requirements. Pass-by-value semantics are always employed.

AOP Remoting simplifies the process of service design, implementation and exposure of objects to remote access, provided they meet some minor semantic

restrictions, but fixes the parameter-passing semantics. No dynamic control over object placement, via remote instantiation or migration, is provided.

3.5 Second Generation Middleware Systems

There exist several second generation middleware systems that build on the first generation systems described so far. These second generation systems aim to tackle the limitations of first generation systems either by performing code transformations that help programmers to introduce distribution or by providing libraries that programmers can employ directly.

3.5.1 Emerald

Emerald [60, 61] is an object-based language and associated run-time infrastructure with integrated support for distribution and object mobility. While Emerald pre-dates all of the first generation systems described previously, it provides functionality such as migration and dynamic control over application distribution that is found only in more recent second generation systems.

Emerald does not adopt the usual object-oriented paradigm in which programmers specify classes that are instantiated to create objects. Instead each object is declared and constructed in a single operation that defines its state and the operations that it supports. Emerald provides a number of primitive instructions that allow control over the placement of objects:

- *Locate X*, returns the identity of the address-space in which object *X* exists.
- *Move X to Y*, tells the Emerald infrastructure that object *X* should be migrated to the address-space in which object *Y* exists.
- *Fix X at Y*, is similar to a *move* operation. It migrates object *X* to the address-space in which object *Y* exists but will not subsequently allow object *X* to move away from that address-space. It is not permitted to *move* or *fix* an object that is already fixed.
- *Unfix X*, removes the fixed status from *X*, allowing it to be migrated or fixed again. Once fixed, an object cannot be migrated or fixed until it is unfixed.
- *Refix X at Y*, atomically performs an *unfix* of object *X* then a *fix* of object *X* at the address-space of object *Y*.

When object migration occurs, the Emerald infrastructure updates all references internal to the object, such that references previously held by the migratory object to local objects are converted into remote references, and vice versa as required. Remote references held by other objects to migratory objects are updated lazily, through a scheme whereby each address-space holds tombstones for the objects that migrate away from that address-space. The first time a client attempts to access an object at an address-space from which it has migrated, the call is forwarded automatically to the correct address-space. For example, if object *A* migrates from address-space 1 to address-space 2, each time a client holding a reference to *A* attempts to access it at address-space 1, the call is forwarded to address-space 2. The invocation response sent back to the client indicates that the object has moved to the new address-space. The client can then update its remote reference.

Emerald provides several different parameter-passing mechanisms. Arguments to remote method calls are passed by-reference by default though programmers can choose to mark arguments statically as *by-move* or *by-visit*. Pass-by-move is identical to pass-by-migrate, as described in Chapter 2. Pass-by-visit is similar to pass-by-migrate but the arguments are migrated back to their original locations once the remote method calls are complete.

Emerald is an impractical choice for creating real-world applications because it is a relatively unknown and unsupported language, with little library support in comparison to language such as Java or C++. However, it provides true transparency between local and remote method calls and allows programmer control over application distribution through its provision of object migration.

3.5.2 JavaParty

JavaParty [62] extends the Java language with the addition of the keyword *remote*, which is permissible only in class signatures and indicates that the class must support remote access. Classes are compiled with the JavaParty compiler, which generates pure Java source code that uses Java RMI to implement remote accessibility.

3.5.2.1 Introducing Support for Remote Access

The *remote* keyword acts as a marker indicating to the JavaParty compiler which classes must be transformed into remotely accessible versions. The JavaParty compiler makes all members (that is, methods and fields) of each *remote* application class public

and generates get/set accessor methods for all fields. Each *remote* application class is then transformed into five distinct classes. The non-static and static members of the original class are separated into two Java RMI-enabled implementation classes. One implementation class contains only the non-static members of the application class and the other contains only the static members transformed into isomorphic non-static versions. Java RMI compliant interfaces are then extracted from both implementation classes. Finally, a wrapper class with the same name as the original class is generated.

Figure 3.2 shows an application class *X* and the five classes that are generated from it. Class *XImpl* contains Java RMI-compliant versions of all the non-static methods that were in *X* and class *XImplStatic* contains Java RMI-compliant non-static versions of all the static methods that were in *X*. Interfaces *IX* and *IXStatic* are extracted from the implementation classes. The wrapper class *X* is structurally equivalent to the original *X* but all methods have been converted into wrapper methods.

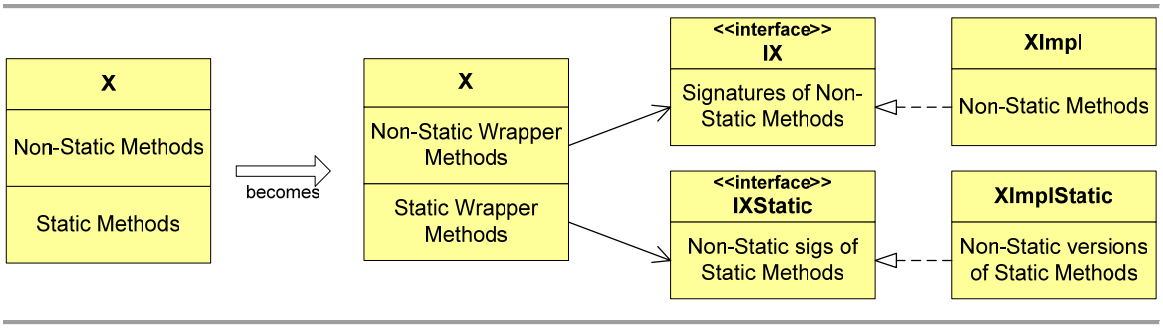


Figure 3.2: The JavaParty remote class transformations.

Each instance of the wrapper class references an instance of each implementation class via its Java RMI-compliant interface. These implementation instances may be in the same address-space, in which case the associated wrapper object references them directly, or may be in remote address-spaces, in which case the wrapper object references the associated Java RMI proxy objects.

The wrapper implements the same methods and has the same name as the original class. When called, each method in the wrapper calls its counterpart in one of the implementation objects. The JavaParty wrapper ignores any Java RMI exceptions that occur, except for those that are caused by unchecked exceptions[2], which are thrown back to the application.

[2] Java has two kinds of exception, checked and unchecked. Programmers must explicitly handle checked exceptions. Unchecked exceptions may be handled or ignored, in which case the JVM terminates.

3.5.2.2 Additional Functionality

JavaParty introduces new functionality that is not provided by Java RMI. It allows programmers to instantiate objects in any remote address-spaces in which the JavaParty infrastructure is executing. Object placement policies may be associated with classes, meaning schemes such as round-robin placement can be implemented. However, by applying policy at the granularity of class, the same placement policies must be applied to all instances of a class, limiting the flexibility of these policies.

Objects may be migrated from one address-space to another, provided no methods are currently executing on those objects. Instances of the wrapper classes keep track of the threads that are executing on the wrapped objects and ensure that this rule is not broken. To migrate an object, the two implementation class instances and their closures are serialized using Java RMI then passed across the network by-value. The wrapper remains in the original address-space and is updated to reference the migrated implementation objects remotely using Java RMI.

JavaParty implements migration without concern for referential integrity. The entire closures of the implementation instances are serialized. Programmers must ensure that references to objects within the migrated objects' closures remain consistent. All remotely accessible objects in JavaParty directly or indirectly extend a special base class that implements the migration functionality. This approach cannot support the migration of non-transformable classes, such as system classes or those with native members.

3.5.2.3 JavaParty Semantics

JavaParty allows instances of non-transformed classes to hold references to instances of the transformed class wrapper and treat them as though they were instances of the original, untransformed class. The underlying transformations are based on Java RMI and consequently JavaParty exhibits Java RMI remote call semantics. If instances of non-remote classes are passed as arguments to remote method calls, they are passed by-value. Passing semantics are decided based on whether arguments support remote access, but as JavaParty is designed to hide this information, these semantics can be difficult for programmers to predict.

JavaParty is a tool to simplify creation of Java RMI-based applications. It allows programmers to define Java RMI-compliant classes without needing to write the Java RMI code manually. Using Java RMI directly, programmers create remote interfaces and

associated implementations. They ensure that all references to instances of the implementation classes are interface references. JavaParty reduces the number of steps required to create Java RMI compliant classes. Programmers identify the remote classes and the compiler alters method signatures and extracts interfaces automatically.

JavaParty applications cannot introduce support for remote access into classes dynamically. Programmers must know statically which classes will need to support distribution. The parameter-passing semantics are the same as Java RMI and are decided based on the remote accessibility of the arguments. Though JavaParty hides the distribution-related code from programmers, it does not attempt to preserve non-distributed calling semantics. Further, it transforms code at the source level and so cannot modify classes for which source is unavailable, such as system classes or native classes.

3.5.3 J-Orchestra

J-Orchestra [63-66] transforms non-distributed Java applications into isomorphic distributed versions. J-Orchestra introduces distribution into existing applications while retaining local method calling semantics using byte-code transformations and Java RMI. It is not a tool set for the creation of general distributed applications. Applications must have a single entry point and must push objects across the address-spaces in the distributed system, rendering it unsuitable as a platform on which to build distributed applications with multiple entry points.

Programmers use the J-Orchestra tool to describe how a given application will be distributed among the available address-spaces. To transform an application, J-Orchestra analyses it statically and determines all places in the code where object construction is performed. Programmers are shown a list of these constructor calls and a list of all address-spaces in the distributed system. Programmers use a graphical tool to associate constructor calls with address-spaces to create a *distribution plan*. Constructor calls are replaced with calls into factories that will instantiate the requisite objects in remote address-spaces according to the distribution plan.

J-Orchestra modifies applications statically and creates multiple transformed classes for each application class in almost the same manner as JavaParty. However, J-Orchestra transforms all the classes it can instead of transforming only particular *remote* classes.

J-Orchestra supports the migration of instances of transformed classes though non-transformable classes cannot migrate from the address-spaces in which they were

instantiated. Using the J-Orchestra tool, programmers can statically associate a pass-by-move [61] policy with classes that support migration. If instances of these classes are passed as arguments to remote method calls, they will migrate to the target address-space.

J-Orchestra is designed to analyse a non-distributed application statically in order to create a distributed version of that application. The application begins execution in a single address-space and creates objects in remote address-spaces in the distributed system. The overhead of indirection exists between all application objects, whether or not the objects exist in the same address-space, thereby increasing the cost of application execution.

J-Orchestra is unsuitable for the creation of distributed applications in general. It cannot support multiple entry points into the application to allow asynchronous deployment of the application across the distributed system. Programmers cannot alter application semantics to take advantage of their application-specific knowledge to replicate or cache objects. J-Orchestra preserves local application semantics strictly. Control over object placement policy is provided, through distribution plans and pass-by-move semantics when remote methods are called. This control is limited as distribution polices may only be applied at class granularity and are limited in their expressiveness.

3.5.4 Do!

Do! [67, 68] is a Java RMI-based system that aims to reduce the complexity of distributed application creation by generating the Java RMI code automatically but does not hide the distributed nature of the applications. The system deliberately exposes distribution and provides tools to enable programmers to create parallel distributed applications.

As with Java RMI and JavaParty, programmers determine at design-time which classes are to be accessible remotely. Classes that must support remote access are modified to implement a marker interface called *Accessible*. The Do! framework generates Java RMI code by reflectively analysing all *Accessible* application classes in order to create:

- Java RMI-compliant interfaces that capture the functionality of each *Accessible* class.
- Implementations of these interfaces that are functionally identical to the original classes but have modified method signatures which throw Java RMI exceptions.

- Wrapper classes with the same names as the original classes.

These transformations are similar to those carried out by both JavaParty and J-Orchestra though Do! provides no support for static members. Consequently, Do! is limited in comparison to these other systems as programmers cannot access static members remotely.

Programmers instantiate objects in remote address-spaces explicitly using factory methods provided by the Do! infrastructure, which registers the objects with the *rmiregistry* automatically. Do! provides no bootstrapping mechanism beyond that which is supplied by the *rmiregistry*. Each application starts running in a single JVM and pushes objects out to remote machines during execution. The Do! framework adopts conventional Java RMI passing semantics.

Do! is a toolkit to allow programmers to create parallel applications in which units of work can be pushed out to machines in the distributed system. In comparison to JavaParty, it provides little support for the creation of general distributed applications.

3.5.5 Pangaea

Pangaea [69, 70] is a Java-based system that introduces distribution into non-distributed applications according to programmer supplied constraints using static code analysis. Existing non-distributed applications are instrumented to introduce distribution, using JavaParty as the underlying distribution mechanism.

Pangaea performs static code analysis on an existing application to create an application graph that approximates the set of objects that will exist when the application executes. Pangaea identifies constructor calls in the non-distributed application and estimates how many times each constructor will be called at run-time using static byte-code analysis. Though not decidable in general, Pangaea can identify the number of times certain constructors are called, for example, those inside loops that execute a statically defined number of times. Pangaea cannot analyse classes that contain native code, which includes some system classes.

Each node represents a particular constructor call in the application and represents the zero or more Java objects that will be instantiated by that constructor call at run-time. The distribution of the application is constrained by associating each graph node with a single address-space or some set of address-spaces. Constructor calls are replaced with factory calls which instantiate objects in the appropriate address-spaces. Programmers do

not need to assign every graph node to an address-space since Pangaea will instantiate unassigned objects in the same address-spaces as the objects that created them.

Pangaea makes use of the underlying migration support in JavaParty. It is therefore not concerned with the implementation of an object migration mechanism. Instead, it provides a policy mechanism that can determine when and to where object migration should take place [71]. The same application graph that is used to define the application distribution is used by programmers to determine which objects support migration. Pangaea evaluates migration policies by polling *migration strategy* objects on a synchronous basis, after a certain number of calls, or on an asynchronous basis, after a certain period has elapsed. The migration strategies indicate whether the objects should be migrated based on one of three strategies:

- Never advocate a migration.
- Choose an address-space at random.
- Move the object to the address-space from which it has received most remote calls.

The Pangaea infrastructure associates a *watcher* object with each of the objects that supports migration. The watcher objects perform these polling operations on behalf of their associated objects.

Pangaea allows support for distribution and migration in an application by building on an existing middleware system. It has advantages over the underlying middleware system because it employs static code analysis to help programmers visualize the approximate run-time structure of applications. Like J-Orchestra, the object placement policies are based on constructor calls and cannot be extended to take advantage of run-time information about the context in which instantiations are performed to guide the policy decisions. Similarly, migration policies cannot evolve at run-time or respond to application events. In order to modify placement policies or alter the set of objects that supports migration, the programmer must stop and re-analyse applications.

3.5.6 XRMI

XRMI [72] is designed to allow the dynamic reconfiguration of Java RMI-based distributed applications. XRMI does not attempt to hide distribution from programmers but rather introduces a layer of indirection in order that applications can be reconfigured dynamically. Reconfiguration is the replacement of one Java RMI object with another in a

manner transparent with respect to the object's reference holders. XRMI can convert references to local objects into references to remote objects and vice versa.

A programmer creates distributed applications using standard Java RMI in the traditional manner. Consequently, XRMI exhibits all the limitations of Java RMI in terms of flexibility to change, complexity and restricted application semantics. XRMI uses a custom compiler to generate replacement Java RMI proxies, known as *virtual stub*s, which can be used to control access to the remotely accessible objects.

Virtual stubs are wrappers that can reference either local application objects or Java RMI proxies to remote objects. XRMI keeps references from remotely accessible objects back to the virtual stubs that reference them. When replacing one object with another, XRMI eagerly updates all the virtual stubs that reference the old object to reference the new one. XRMI provides a locking mechanism to ensure that no threads are executing on a wrapped object before substitution occurs. The implementation of this mechanism, as it is described in Chen [72], is not thread-safe and so may lead to loss of referential integrity in multi-threaded applications.

XRMI aims to provide dynamic reconfiguration of Java RMI applications to allow flexibility in application distribution. However, it provides a sub-set of the functionality found in JavaParty or Do! with no advantages over either system.

3.5.7 Coign

Coign [73-75] automatically redistributes client/server distributed COM applications based on run-time profiling in order to minimize the inter-address-space communication cost incurred. Coign instruments an application in order to establish which components interact at run-time [76]. The application is run several times under typical conditions in order that usage data can be collected. This data allows the Coign system to determine how much inter-address-space communication occurred during these typical runs and to assign a cost to it based on the number of remote calls and the amount of serialized data passed. Coign transforms the application so that its distribution can be decided dynamically. At run-time, the Coign infrastructure decides the application distribution based on the usage data, in order to minimize the inter-address-space communication cost.

3.5.7.1 Application Profiling

During the profiling stage, Coign tracks contextual information about each component instantiation, such as the class of component created or the name of the method that performed the instantiation. The number of inter-component method calls performed by each component to both local and remote components is recorded. Coign also tracks the amount of data that was serialized for these calls, or would have been serialized had the components been in separate address-spaces.

Using this profiling information, Coign creates a graph representing the inter-component communication that occurred in the application. A graph-cutting algorithm is applied to determine how the applications could be distributed to reduce the inter-address-space communication cost. One of the main limitations of Coign is that it can only re-distribute applications that are already divided into two partitions. Coign assumes that some components are fixed in each partition and uses the graph-cutting techniques to determine how the remaining components are divided between these two partitions. It cannot transform non-distributed applications or applications that are distributed into three or more parts.

A cut graph indicates the optimal distribution of an application with respect to inter-address-space communication cost, given a particular set of usage data. This distribution is optimal if the application behaviour does not change and so it is important that the profiled runs of the application are representative of typical application usage. During future application runs, Coign distributes the application dynamically based on the optimal distribution. In order to make use of information gained during profiling, Coign classifies components and assumes that any components existing in the future that are classified as the same can be placed in the same partition.

3.5.7.2 Distributing Applications

Coign transforms each application so that its distribution can be decided dynamically. When an application is run, Coign classifies each new component before creation then finds the component from the profiled run that most closely matches it in classification. The optimal distribution is examined to determine in which address-space the matching component from the profiled runs was placed. Coign then instantiates the new component in that address-space. Coign places the new component in the address-

space in which the profiled component *should* have been placed, not necessarily the one in which it *was* placed.

Coign does not support component migration and so it must ensure that components are instantiated in the correct address-space, since they cannot be moved after creation. Components must be classified before they are instantiated and a variety of different component classification schemes are employed [73]. These schemes classify components based on combinations of the Coign classification, the class of the component performing the instantiation and the specific method in which the instantiation is performed.

Coign provides completely automated re-distribution of applications based on typical usage data gained by profiling the original untransformed applications. Coign relies on the enforced use of factories in COM to allow application instrumentation. Further, it can distribute only at the granularity of COM components, which have been explicitly created with support for remote accessibility.

3.5.8 JavaSymphony

JavaSymphony [77-79] provides Java libraries to help programmers create distributed applications with particular emphasis on parallel tasks. JavaSymphony does not attempt to hide the distributed nature of applications from programmers. Instead, it allows programmers to specify the resource requirements for applications in abstract terms and employs system profiling to match the resources that are available at run-time to those specified.

The JavaSymphony run-time executes on all machines in the distributed system and performs profiling of metrics such as CPU load or available memory. When creating a distributed application, the programmer defines the architecture in which the application must run in abstract terms using constraints based on the profiled metrics. For example, programmers may specify that the application runs on machines with at least 128MB of free memory and a processor with no more than 50% load. When the application is run, JavaSymphony distributes it over the sub-set of the available machines that best reflects the abstract architecture defined by the programmer.

JavaSymphony allows programmers to make instances of any class accessible remotely using *JSObjects*. JSObjects adopt the *active object* pattern [80] in which each object has its own unique thread. All method calls on an object are queued and executed serially by the same thread. JavaSymphony extends this model by permitting concurrency

and providing thread pooling to reduce the overhead inherent in a strict one-to-one correspondence between threads and objects.

Programmers can introduce support for remote access to arbitrary application classes, providing the classes are instantiated using JavaSymphony factory methods. When using one of these factory methods, programmers specify an application class and a series of constraints. The JavaSymphony infrastructure creates an instance of the specified class on a machine that meets these constraints then wraps it in a JSObject. A remote reference to the JSObject, not the application object, is returned to the client application.

JavaSymphony does not offer the usual abstraction of proxies and so programmers must explicitly call a remote invocation method provided by JSObjects, supplying method names and arguments. This invocation method propagates the remote method call across the network. JSObjects cannot be passed as arguments to remote methods and so all method arguments are passed by-value. JavaSymphony does not provide DOM functionality that can be used to preserve local by-reference calling semantics, making it is unsuitable for the introduction of distribution into existing applications.

JavaSymphony implements both persistence and migration but without concern for referential integrity. When an object is serialized for persistent storage or for migration, the object's closure is serialized. Programmers must ensure that references to any objects within the closures are updated appropriately.

JavaSymphony is primarily aimed at tasks for which local application semantics need not be preserved. It does not hide the fact that the application is distributed. Instead it attempts to make inter-address-space communication explicit, though with minimal programmer effort.

3.5.9 ProActive

ProActive [81] is a Java library that provides tools for the creation of distributed applications using Java RMI or the Java Message Service (JMS) [4] as the underlying transport protocol. Remotely accessible objects in ProActive adopt the *active object* pattern. Each remotely accessible application object is wrapped and accessed only via this wrapper. Method calls are queued by the wrapper and there is a thread per active object that executes the calls serially. Concurrent execution of methods on active objects is not permitted.

Remotely accessible objects are known as *active objects* while non-remotely-accessible objects are known as *passive objects*. Objects may only be activated if they are instances of non-final classes that support serialization. The programmer must ensure manually that no two active objects reference a single shared passive object and that all passive objects support serialization. If these rules are not adhered to, threading and migration semantics become inconsistent. Remote method calls may only be performed on active objects and, since passive objects may not be shared, only active objects may be passed as arguments. ProActive is aimed at applications in which active objects are coarse-grained application components rather than individual programming language objects.

Proxy classes in ProActive are sub-classes of the original application classes with which they are associated (hence the requirement that classes are non-final) and so a client can treat a proxy object as if it were an instance of the same class as the corresponding remote object. In this way, ProActive avoids one of the main limitations of Java RMI, which is that interface references must be used when referring to objects that support remote access.

ProActive permits migration of active objects but relies on Java serialization, so all active objects and the objects in their closures must support serialization. When an active object migrates to a new address-space, its wrapper remains in the old one and acts as a tombstone. When a client attempts to access an object that has migrated using a remote reference that is out-of-date, the remote reference is updated lazily using the information in the tombstone.

By strictly adopting the policy that passive objects may not be shared ProActive avoids the difficult problems inherent in preserving referential integrity when performing object migration. It further ensures that thread synchronization problems do not result as a consequence of the changes made to each application's threading model. The active model allows no concurrent access to any objects, active or passive and so avoids the need for standard thread synchronization.

This approach avoids problems with referential integrity and thread synchronization but is inflexible as a result. Programmers must ensure that they adhere to the rules and so the semantics of existing applications must be modified to fit the active object model. Further, programmers must rely on the thread synchronization provided by the active object model. Standard thread synchronization techniques cannot be used in a ProActive application without risking unpredictable application behaviour. ProActive has

significant overheads because the number of threads required for each application is potentially unbounded. There is further overhead in terms of the ancillary objects required to implement the active object model, such as the wrappers and associated method calls queues.

3.5.10 FarGo

FarGo [82-84] is a Java RMI-based DOM that allows programmers to create classes with explicit support for migration. Programmers define classes called *complets* and instances of these classes can be referenced remotely or migrated between address-spaces. The granularity of distribution is at the complet level.

The mechanism for creating distributed applications in FarGo is similar to that employed when using the Do! framework. Classes that support remote access or migration must extend special interfaces and a special compiler is used to generate versions of these classes that are accessible remotely using Java RMI.

Programmers define event handler methods that are called when particular system events occur, such as migration or JVM shutdown. For example, these methods can perform housekeeping tasks then migrate objects to new address-spaces when the local JVM terminates. Programmers can also define event handlers using a scripting language that supports particular FarGo primitives e.g. *move*, which causes migration to occur.

FarGo allows types that represent different migration policies to be imposed onto complet references. These policies can control the migration policies applied to both referenced objects and reference holders. By altering the types of references dynamically, programmers can define migration policies. There are five types of remote reference supported by FarGo:

- *Link references* ensure referential integrity if the referenced complet migrates.
- *Pull references* indicate that if the reference holder migrates to another address-space then the referenced complet should follow.
- *Duplicate references* indicate that if the reference holder migrates to another address-space then a copy of the referenced complet should follow.
- *Stamp references* indicate that if the reference holder migrates to another address-space then it should rebind to a complet of the same type as the previously referenced complet. This allows complets to rebind to physical resources such as disks or displays after migration.

- *Bi-directional pull references* indicate that if either the reference holder or referenced complet migrates to another address-space, the other should follow.

FarGo gives programmers explicit control over the migration. Migration policy is defined by converting references in the application into one of the above types dynamically. FarGo does not abstract over the distributed nature of applications and requires programmers to construct classes with explicit support for remote access. It does not consider initial complet placement policy, which renders its approach to policy specification inappropriate for use with middleware systems that do not support migration. Control over the initial placement policy is useful even in systems supporting migration, as migration has non-zero cost. It is preferable to avoid unnecessary migration operations by instantiating objects in suitable locations, rather than migrating them as required.

3.6 Limitations of Existing Middleware Systems

This chapter has described both first generation and second generation middleware systems. As described in Chapter 1, there are five main problems inherent in first generation middleware systems, namely:

1. Design decisions must be made early in the design process.
2. Applications are inflexible to dynamic changes in their distribution.
3. Middleware systems are complex and error-prone.
4. An unnatural encoding of application-level semantics may be forced.
5. Programmers have little control over distribution policies.

The described second generation systems each tackle some of the problems, though none circumvents them all. Each limitation is now examined in turn to explain how it is manifested in the systems that exhibit it, and to address the reasons why it presents a problem to programmers. Table 3.1 relates these five problems to the described middleware systems and shows whether each system offers a full solution, partial solution or no solution to each of the problems.

Middleware System	Forcing Early Design Decisions	Brittleness with Respect to Change	Complexity	Distorted Application Level Semantics	Support For Placement Policy
Sun-RPC	✘	✘	✘	✘	✘
XML-RPC	✘	✘	✘	✘	✘
Web Services	✘	✘	✘	✘	✘
JBoss Remoting	✘	✘	✓	✘	✘
CORBA	✘	✘	✘	✘	✘
Java RMI	✘	✘	✘	✘	✘
DCOM	✘	✘	✘	✘	✘
.NET Remoting	✘	✘	✘	✘	✘
Emerald	✓	✓	✓	✓	✘
JavaParty	✘	✓	✓	✘	!
J-Orchestra	✓	✓	✓	!	!
Do!	✘	✘	✓	✘	!
Pangaea	✓	✓	✓	!	!
XRMI	!	✓	✘	✘	✘
Coign	✓	✘	✓	!	✓
JavaSymphony	✓	!	!	✘	✓
ProActive	!	✓	!	✘	✘
FarGo	!	✓	!	✘	✓

✓	The system offers a solution to this problem
✘	The system offers no solution to this problem
!	The system offers a partial solution to this problem

Table 3.1: The problems exhibited by existing middleware systems.

The problems are now each examined in more detail in the context of the systems described in this chapter.

3.6.1 Forcing Early Design Decisions

Only instances of classes that support remote access may be separated into different address-spaces from their reference holders, constraining the ways in which applications can be distributed. To change an application's distribution, programmers may be forced to introduce support for distribution into classes without it and vice versa. Programmers must determine whether the additional application complexity inherent in

unnecessarily supporting remote access outweighs the cost of removing it in terms of programmer effort.

First generation middleware systems require programmers to decide which classes will support remote access at application design-time. Programmers must specify the interfaces between distribution boundaries. Only the application classes that implement these interfaces can be accessed remotely. This support is hard-coded at the source level and so changes to an application's distribution may result in source-level changes. Programmers must know enough about how application objects will be distributed at run-time to be able to determine which application classes need to support remote access. CORBA tackles this problem by providing a dynamic invocation mechanism that allows programmers to create invocation requests explicitly. This mechanism can be used to expose objects to remote access without static type information but forces programmers to implement middleware level functionality at the application level.

Without the ability to expose objects to remote access dynamically, application distribution is inflexible. It is not possible to introduce support for remote access into every application class using existing systems because of the semantic restrictions placed on remote classes. For instance, application classes cannot pass remote references to instances of pre-defined library classes that do not support remote access.

The second generation systems that employ custom compilers, such as JavaParty [62] and Do! [67], still force programmers to make early design decisions. These systems simplify the process of creating distributed applications through automated generation of distribution-related code but it is the programmers that must determine which classes will support remote access.

Tools such as J-Orchestra [65] and Pangaea [70] are designed to transform a single non-distributed application into a distributed version that pushes itself out into the distributed system at run-time. They perform static code analysis and code transformations to help programmers choose suitable distributions. The distributed version of an application is generated automatically and in this respect these systems tackle this middleware system limitation. However, both transform only local applications and are unsuitable tools for the creation of general distributed applications because programmers cannot include multiple entry points.

Using these systems, programmers define initial application distributions using the provided tools. Both systems support changes to application distribution using migration, however it is not possible to migrate arbitrary application objects. If fundamental changes

are made to application distributions then the applications must be re-transformed, limiting the effectiveness of these systems in dynamically changing applications.

ProActive [81] and JavaSymphony [77] allow programmers to expose objects to remote access dynamically. However, both adopt the active object [80] model which associates a thread with each remotely accessible application object. The conversion of existing application objects into active objects alters the threading semantics of the application. Further, active objects may not have shared access to any non-active objects. Programmers may need to alter the structure of the application to ensure that this strict separation of active object closures is preserved.

3.6.2 Brittleness with Respect to Dynamic Change

In addition to forcing decisions early in the design process, which results in static inflexibility to change, distributed applications created using existing middleware systems also exhibit brittleness with respect to dynamic change. Brittleness and inflexibility to change occurs in middleware systems that do not support object migration. None of the first generation middleware systems are capable of migrating objects between address-spaces and so objects are fixed in the address-spaces in which they are instantiated. This hinders the adaptability of applications to changing execution environments, for example, objects on heavily loaded machines cannot migrate to other machines. It also has implications for long running systems because applications cannot be re-distributed as machines join and leave the distributed system. Several of the second generation systems provide object migration mechanisms, including JavaParty [62], J-Orchestra [65], ProActive [81], JavaSymphony [77], FarGo [82] and Pangaea [70].

3.6.3 Complexity

The creation of distributed applications using first generation middleware systems can be a difficult and error-prone task due to the complexity of these systems. Programmers using first generation middleware systems (CORBA [8], Java RMI [9], Microsoft COM [32] and Microsoft .NET remoting [10]) must ensure that the application classes supporting remote access adhere to particular semantic rules. These rules are specific to the middleware system in use. For instance, Java RMI forces application classes to implement certain interfaces, places restrictions on the types that may be used in method signatures and forces programmers to handle distribution-related error conditions explicitly. CORBA and COM require that programmers define IDL interfaces

and implement ancillary classes. Microsoft .NET remoting forces application classes to extend certain base classes. The Observer/Observable example described previously in the context of Java RMI illustrates the problems that result from these restrictions.

Configuration of the middleware system can be a complex process as programmers must ensure that all aspects of the infrastructure are running, that remotely accessible objects are registered and that necessary application code has been distributed to all address-spaces or made accessible through a centralized code repository. Some first generation systems, such as CORBA [8], Microsoft COM [32] and Web Services [46], do not generate ancillary distribution-related code such as skeletons and proxy classes automatically. They provide tools that programmers must employ to generate ancillary code for the required classes. Programmers must either distribute code among all address-spaces or must configure the code distribution infrastructure explicitly. Several additional steps, and consequently potential points of failure, are introduced into the software development process.

The complexity inherent in creating and configuring distributed applications using first generation systems is addressed by many of the second generation systems. Systems such as JavaParty [62], J-Orchestra [65] and Pangaea [70] employ customized compilers or code transformation techniques to generate the distribution-related code automatically.

3.6.4 Distorted Application Level Semantics

The semantic limitations forced on application classes in order to support distribution affect application semantics. Inheritance relationships between classes are affected and it is difficult to make application classes remotely accessible if their super-classes do not meet the necessary requirements. This causes an unnatural or inappropriate encoding of application semantics because classes are forced to support remote access for the benefit of their sub-classes, entangling application logic and distribution. This is particularly a problem for application classes that need to extend pre-compiled classes without support for remote access.

First generation middleware systems decide statically which parameter-passing semantics should be applied when remote methods are called. In Java RMI [9], only classes that implement the *java.rmi.Remote* interface and handle network related errors explicitly in application logic can be exposed to remote access or passed by-reference. All other objects that are passed as arguments or return values to remote methods must be instances of classes that implement the *Serializable* interface. Parameter-passing

semantics are affected by static design level decisions and are tightly coupled with application distribution.

Microsoft .NET remoting [45] adopts semantics that are similar to Java RMI. Instances of classes that extend the *MarshalByRefObject* class are passed by-reference and all other objects that are passed to remote methods must be instances of *Serializable* classes. The .NET remoting framework incrementally improves on Java RMI by applying these semantics consistently to objects. However, parameter-passing semantics are still fixed statically and are dependent on the distribution of the application.

In CORBA and COM, arguments are marked in IDL with the passing semantics to be applied. Further, CORBA component classes are defined statically as either pass-by-reference or pass-by-value. CORBA and COM allow only components and data structures that have been explicitly described to be passed across address-space boundaries. Web Services and other RPC systems allow pass-by-value semantics and permit only objects of pre-determined types to be passed.

Several of the second generation middleware systems build on Java RMI, namely JavaParty, J-Orchestra, Do! and ProActive. However, these second generation systems strive to preserve local Java method calling semantics and so fix parameter-passing semantics statically.

In general, reusability and application semantics are restricted for the following reasons:

- *Some systems allow no programmer control over parameter-passing semantics at all.*

  Systems that allow no control over passing semantics lack flexibility as programmers cannot employ the most suitable parameter-passing mechanisms on a per-application basis. With control over passing semantics, programmers can manage the trade-offs between different parameter-passing mechanisms to reduce network traffic, introduce resiliency or permit caching.

- *When programmers can decide parameter-passing semantics, they cannot do so dynamically.*

  Application programmers have limited dynamic control over inter-address-space parameter-passing semantics. Within a single application, it may be required that objects are transmitted by-value or by-reference depending on the circumstances and in most existing middleware systems this would require

that different classes be created. Complexity is introduced into applications due to the limitations of the middleware system.

- *The parameter-passing semantics and application distribution are tightly bound.*

  The parameter-passing semantics and application distribution are tightly coupled. Reuse of large-grained components, composed of instances of multiple classes, is hindered because concrete class implementations must be developed in the context of some planned deployment environment. Various physical considerations dictate the nature of the implementation, such as the available computational resources, network connectivity, latency or bandwidth. These considerations influence the implementation of classes limiting reuse [85]. For example, in a poorly connected environment, it may be appropriate that pass-by-value semantics are adopted in order that the called methods can continue to perform computation over arguments, even if the network connection to the caller is lost transiently. Conversely, in a well-connected environment, it may be appropriate to adopt pass-by-reference semantics to allow shared access to arguments and ensure coherency.

3.6.5 Lack of Support for Object Placement Policy

Some of the second generation middleware systems support the creation of objects in remote address-spaces or the migration of objects between address-spaces. Each system provides a mechanism to define the distribution policy that controls these operations. Some of the systems that support this functionality, e.g. ProActive [81], defer these policy decisions to the programmers, who trigger object migration directly. Others, e.g. Coign [73], Pangaea [70], J-Orchestra [65], FarGo [82] and JavaSymphony [77], provide a policy mechanism that allows control over object placement.

Coign makes extensive use of instrumentation and component classification to re-distribute client/server COM applications. The component classification schemes in Coign [73] classify components before they are created. Coign then performs component placement based on this classification, providing completely automated distribution of applications based on profiling. However, it does not allow programmers to exploit the classification schemes to distribute applications into more than two address-spaces or to define placement policies directly. Further, it does not support migration or the definition of migration policy.

Pangaea and J-Orchestra both perform static analysis of applications and allow placement policies to be associated with constructor calls in source code. This allows specification of policies at a finer granularity than class though the placement policies that can be associated with each constructor call are basic. Objects can be placed in a particular address-space or can be distributed among a group of address-spaces in a round-robin manner. This approach to policy specification cannot be reused as a general mechanism because it is tightly bound to the capabilities of these systems, i.e. policies may only be associated with constructor calls. It provides no scope for alternative approaches to imposing object identity, such as Coign-style classification.

FarGo allows programmers to associate migration policies with references. References are associated with policy information describing how the referenced objects behave when migration occurs. FarGo allows dynamic control over aspects of migration policy but only pre-defined migration policies can be used. FarGo's approach to object placement policy does not consider initial object placement, limiting its reusability.

JavaSymphony allows programmers to define constraints when performing remote object instantiation. These constraints can be considered an object placement policy as they define metrics that the target address-space of the instantiation must meet.

Without the separation of distribution policy from implementation, programmers cannot re-distribute existing applications without altering the application classes themselves and must re-implement placement policy on a per-application basis. None of the existing systems provides a flexible mechanism for the specification of placement policy.

3.6.6 JChord Case Study

The JChord case study described in the previous chapter has a number of requirements that render current systems inappropriate for its implementation, namely:

- *JChord was initially developed as a local application.*
- *JChord must adapt to changing requirements.*

  The semantic limitations placed by current systems on classes that support remote access would force design level changes to the JChord application in order to introduce distribution. Application logic would be affected, risking the addition of errors into a tested and stable application. Further, the inflexibility to change inherent in existing systems would not permit JChord to

adapt to changing research goals in the required manner without extensive re-engineering.

Systems such as J-Orchestra and Pangaea that transform existing applications into distributed versions do not provide sufficient flexibility. The distributed version of JChord has multiple entry points and exposes certain aspects of its distribution to programmer control.

- *Each JChord node presents a multiplicity of interfaces to clients.*

  Current systems permit exposed objects to implement multiple interfaces statically. It may be necessary to make changes to application source in order to present particular interfaces to clients, making it difficult to expose instances of pre-compiled classes with the desired interfaces.

- *References to remote JChord nodes must cache some of the state of the remote nodes locally for efficiency and for use during failure, to identify the failed nodes.*

  None of the systems described above provide support for smart proxies.

- *It must be possible to create nodes on remote machines to automate ring deployment.*

  None of the first generation systems provide support for remote object instantiation, though several of the second generation systems do, including JavaSymphony, ProActive and Do!.

- *The Data Store point-of-presence needs to alter parameter-passing semantics dynamically when accessing the individual Data Store objects.*

  None of the described systems permit dynamic control over parameter-passing semantics meaning that it is difficult to implement the Data Store in the desired manner using these systems. Some systems provide no control at all over these semantics or decide them as a consequence of application distribution.

- *It must be possible to migrate objects that are remotely referenced by Data Store objects from one address-space to another.*

  Migration is supported by several second generation systems, though always with restrictions. JavaParty [62], and therefore also Pangaea [70], do not ensure referential integrity throughout the closure of the migratory objects. J-Orchestra [65] permits only transformed objects to migrate. FarGo [82] allows

only complets created with explicit support for migration to be moved. ProActive [81] and JavaSymphony [77] permit migration though force applications to adopt the active object model which does not permit shared access to non-active objects.

- *Distribution policies to control the deployment of JChord ring nodes and the migration of stored objects must be defined.*
  In the systems that support remote object instantiation and migration, policies to control distribution are inflexible and non-adaptive.

3.7 Conclusion

This chapter has investigated and evaluated current middleware systems, describing first generation and second generation systems. All of these systems exhibit limitations, indicating the need for a third generation middleware system that provides flexibility throughout the creation, maintenance and evolution of distributed applications.

Chapter 4

Requirements of a Third Generation Middleware System

This chapter builds on the work in previous chapters to define the requirements of a third generation middleware system that allows programmers to create flexible distributed applications.

4.1 Introduction to Chapter 4

This thesis describes the design and implementation of a third generation middleware system that aids the creation, maintenance and evolution of distributed applications. This system will separate functional and non-functional considerations, and hide the complexity of distribution where appropriate. Applications will be flexible with respect to change and programmers will be able to control both parameter-passing semantics and object placement dynamically.

This chapter defines the requirements that a third generation system must meet, based on the taxonomy of existing systems defined in the previous chapter. This combination of requirements is unique to a third generation system, though several of the individual requirements are not. The chapter concludes by showing how such a third generation middleware system meets the requirements of the JChord case study described in Chapter 2.

4.2 Requirements

There are twenty requirements in total, which have been divided in four sub-groups, as indicated by the following four sub-sections.

4.2.1 Server-Side Functionality

A third generation system must abstract over the complexity inherent in distributed application creation and the configuration of the underlying infrastructure. Programmers must not be required to create distribution-related ancillary code so the creation of such code should be automated and hidden from the programmer. To avoid the overhead of superfluous code, ancillary code must only be generated for classes that require support for remote access.

Requirement 1: *All the ancillary code required to perform inter-address-space communication, namely proxy classes, skeletons, serializers and deserializers, must be created automatically.*

To allow application logic to be separated from the non-functional considerations of distribution, it must be possible to expose instances of arbitrary classes to remote access. Programmers must not be forced to decide statically which classes of object will participate in inter-address-space communication.

Changes to application requirements may force classes that are not remotely accessible to become so or vice versa. Programmers must not be forced to re-engineer classes in either case. Since source code may not be available for all application classes, it must be possible to expose pre-compiled classes to remote access, thus promoting code reuse.

Requirement 2: *It must be possible to expose instances of arbitrary classes to remote access without modifications to their source code.*

It must be possible to control which particular objects are remotely accessible dynamically in order that objects can be exposed to remote access at any time.

Requirement 3: *It must be possible to expose objects to remote access dynamically.*

It must be possible to control which of the methods provided by application classes are remotely accessible. Different objects of the same class must be able to expose different subsets of the available methods. A single object must be able to expose different sets of methods to different remote clients, allowing the clients to have multiple views over the object.

Requirement 4: *It must possible to control which methods are accessible remotely on a per-object basis, allowing remote clients to have multiple views over a single object.*

Programmers must be able to retain the access semantics of non-distributed applications after distribution is introduced. It must be possible to control whether the local protection semantics provided by the implementation language (such as the *public*, *protected*, *private* and default modifiers in Java) are preserved when remote methods are called.

Requirement 5: *The local protection mechanisms of the implementation language must be preserved.*

Since the exposure of arbitrary objects to remote access can result in the transmission of arbitrary objects as arguments or return values, it must be possible to pass instances of any class across address-space boundaries by-value. Further, since the cost of remote method calls is many magnitudes greater than the cost of local method calls, support for pass-by-value semantics allows programmers to avoid the cost of unnecessary inter-address-space communication.

Requirement 6: *It must be possible to pass any objects by-value as arguments or return values to remote method calls.*

Since arbitrary objects can be passed by-value, the middleware infrastructure in a given address-space may receive serialized instances of unknown classes from other address-spaces. The middleware system must be able to obtain and load the appropriate classes dynamically from a code repository. The code repository should be scalable and exhibit resilience to failure.

Requirement 7: *A scalable resilient code distribution scheme must be provided.*

4.2.2 Client-Side Functionality

It must be possible to call methods directly on remote references and to pass them as arguments and return values. Reference equality semantics should also be preserved. The middleware system must provide a remote reference scheme that permits references to remotely accessible object to be used interchangeably with local references.

Requirement 8: *It must be possible to hold remote references to arbitrary objects and to treat local and remote references in the same manner.*

The middleware system must provide a name service that allows programmers to assign names to remotely accessible objects. The middleware system must associate automatically generated names with remotely accessible objects if none are provided by the programmers.

Requirement 9: *It must be possible to associate names with remotely accessible objects, either implicitly or explicitly, and to obtain remote references to objects based on those names.*

To allow programmers to retain non-distributed application semantics in distributed applications, it must be possible to preserve pass-by-reference semantics across address-space boundaries if desired.

Requirement 10: *It must be possible to pass any objects by-reference as arguments or return values to remote method calls.*

Static members do not exhibit the same semantics as instance members and so it must be possible to preserve the static semantics found in non-distributed applications.

Programmers must be able to decide whether to preserve non-distributed static semantics on a per-application basis.

Requirement 11: *It must be possible to control the semantics of static members on a per-application basis and to preserve non-distributed static semantics in distributed applications if required.*

The addition of inter-address-space communication into applications unavoidably introduces new failure modes related to network errors. The middleware must provide a failure model that handles errors in a consistent manner and allows programmers to specify whether distribution-related errors should be propagated back into applications. In the event of distribution-related errors, the middleware system must offer the programmers a choice between fast-failure and continued execution on a best-effort basis.

Requirement 12: *The programmer must be able to control whether distribution-related failures are propagated to the application or handled internally by the middleware system.*

It is desirable that remotely accessible application objects are local to the reference holders that make greatest use of them, so that the cost of remote method calls is not incurred more often than necessary. A mechanism that permits the instantiation of objects in remote address-spaces ensures that application objects can be grouped into address-spaces according to the needs of the application, rather than based on the initial application distribution.

Requirement 13: *It must be possible to instantiate objects directly in remote address-spaces.*

Support for object migration allows applications to adapt to dynamically changing execution environments by reconfiguring their distributions.

Requirement 14: *It must be possible to perform the migration of objects from one address-space to another without loss of application consistency.*

4.2.3 Controlling Transmission Policy

The middleware system must allow control over the parameter-passing semantics employed when remote methods are called. A framework that allows programmers to define parameter-passing semantics both statically and dynamically must be provided.

Requirement 15: *It must be possible to control parameter-passing semantics dynamically.*

Parameter-passing semantics should not be restricted to pass-by-reference or pass-by-value only, as these two mechanisms represent two ends of a spectrum. Remote references must be able to cache fields and methods of objects locally, in order that the objects can be passed partially by-value and partially by-reference. When reference holders access cached fields or methods, then no remote method calls take place and the cached copies are used instead.

Requirement 16: *Remote references must be capable of caching fields and methods locally.*

4.2.4 Controlling Distribution Policy

With the provision of remote instantiation and object migration, it must be possible to control the object placement policies applied when these operations are performed.

Requirement 17: *It must be possible to create policies to control the placement of objects when instantiation and migration operations are performed.*

Applications must be able to delegate to the policy framework when they require object placement decisions to be made. The creation of application logic can therefore be separated from the determination of application distribution. To allow programmers to focus on creating application logic, the mechanisms providing control over object placement policy must allow the separation of functional and distribution-related concerns.

Requirement 18: *It must be possible to define object placement policies independently of application logic.*

It must be possible to construct new placement policies from existing policies by reusing and recombining aspects of those policies. Reuse of existing policies can simplify the policy creation and testing process. It must be possible to create policies of arbitrary complexity.

Requirement 19: *It must be possible to define arbitrarily complex object placement policies in terms of reusable policy components.*

Policies must be able to use information about the application context in which the instantiation and migration operations are performed to aid policy decisions. This permits, for example, placement policies to be decided using profiling tools external to the middleware system, which can measure system profiling metrics (such as CPU load or free memory) or application profiling metrics (such as the number of method calls performed on particular objects).

Requirement 20: *It must be possible for policies to use application context to aid policy decisions.*

4.3 Meeting the Requirements of JChord

The previous chapter showed that existing systems were unsuited to the implementation of the JChord and Data Store applications. The requirements of the case study are re-examined here to show that a third generation system meeting these requirements is a more suitable choice of implementation platform.

- *JChord was initially developed as a local application.*
  By allowing instances of arbitrary classes to be exposed to remote access and by supporting remote references, distribution can be introduced with minimal changes to application logic. This reduces the likelihood of new errors in application logic. The adaptive failure model allows the distributed JChord application to be developed without concern for distribution-related failure. Explicit error handling code can be introduced later, as required.
- *JChord must adapt to changing requirements.*
  By exhibiting flexibility to static changes in the application distribution, the third generation middleware system allows JChord, in its remit as a research tool, to adapt easily to modifications in application requirements caused by changes in research direction.
- *Each JChord node presents a multiplicity of interfaces to clients.*
  The object-oriented principles of encapsulation can be preserved as the set of methods that each node exposes to remote clients can be controlled and the local protection semantics of the original JChord application retained.
- *References to remote JChord nodes must cache some of the state of the remote nodes locally for efficiency and for use during failure, to identify the failed nodes.*

Control over parameter-passing semantics and support for smart proxies, allows JChord to replicate ring state, pass immutable objects by-value and cache immutable state in remote references.

- *It must be possible to create nodes on remote machines to automate ring deployment.*

  This functionality is provided by any system that meets the above requirements.

- *The Data Store point-of-presence needs to alter parameter-passing semantics dynamically when accessing the individual Data Store objects.*

  A system meeting these requirements permits flexibility in parameter-passing mechanisms and allows dynamic control over them. This dynamic flexibility allows objects to be stored both by-reference and by-value.

- *It must be possible to migrate objects that are remotely referenced by Data Store objects from one address-space to another.*

  This functionality is provided by any system that meets the above requirements.

- *Distribution policies to control the deployment of JChord ring nodes and the migration of stored objects must be defined.*

  The control permitted over distribution policies allows the deployment of JChord rings according to flexible policies that are separated from the application logic.

4.4 Conclusion

This chapter has stated the requirements that must be fulfilled by the third generation middleware system that is designed and implemented in this thesis. These requirements define the functionality that the system must provide in order to aid the creation, maintenance and evolution of distributed applications. Complexity is hidden where appropriate yet the system allows programmers fine-grained dynamic control over the parameter-passing semantics employed when remote methods are called and the placement of objects in the distributed systems.

The remainder of the thesis is structured as follows. Chapter 5 describes the design of a middleware system that meets these requirements and provides qualitative evaluation of this system. Chapter 6 examines the implementation details of this third

generation middleware system and provides quantitative evaluation of this implementation. Chapter 7 concludes the thesis by summarizing and evaluating the contribution of the described research.

Chapter 5

The RAFDA Run-Time (RRT)

This chapter describes the design of the RAFDA Run-Time (RRT), a reflective third generation middleware system that permits application logic to be designed and implemented completely independently of distribution-related concerns. This simplifies the software engineering process to aid the creation, maintenance and evolution of distributed applications.

5.1 Introduction to Chapter 5

The RAFDA Run-Time (RRT) is a middleware system that meets the requirements identified in the previous chapter. The RRT conceals inter-address-space communication by default but allows programmers to expose and control all aspects of middleware behaviour. The RRT provides programmers with flexible control over its behaviour and can be used for quick application prototyping or to create fully featured distributed applications.

Throughout this chapter, the functionality provided by the RRT is illustrated using the JChord case study, to demonstrate the benefits of the RRT over traditional approaches to middleware. This chapter contains code examples that illustrate the use of the RRT. These code examples are all written in Java, although the RRT does not take advantage of any features unique to Java and the techniques described here are applicable in other languages. The RRT can be downloaded from *http://www-systems.dcs.st-and.ac.uk/rafda/rrt.html*.

5.2 Overview of the RRT

The RRT permits arbitrary application objects to be exposed to remote access through standard Web Services [46]. The RRT provides:

- Full DOM functionality to RRT-based clients, making it a suitable tool for the creation of new distributed applications and for the introduction of distribution into existing non-distributed applications.
- RPC functionality to clients using other Web Services technologies, allowing programmers to provide service-oriented functionality that supports conventional Web Services calling semantics.

The RRT allows specific application objects to be exposed via Web Services. Programmers need not decide statically which application classes support remote access. Instances of any classes from any applications, including previously compiled classes and those with native members, can be exposed to remote access as Web Services without the need to access or alter source code. Using the RRT, programmers can adopt a methodology for developing and deploying distributed applications that permits application logic to be designed and implemented completely independently of

distribution concerns [86]. This eases the development process and permits the alteration of distribution decisions late in the development cycle.

The RRT aids the creation of tools such as debuggers or application probes that need to access object state from other address-spaces. For example, programmers can introduce remote observers to observable objects or can attach object browsers to arbitrary application objects, permitting them to be browsed remotely.

5.2.1 RRT Infrastructure

The primary purpose of the RRT is to abstract over the inter-address-space communication in distributed applications. This is achieved by allowing instances of arbitrary classes to be exposed to remote access and by permitting clients to obtain remote references to these exposed objects.

Applications access the functionality provided by the RRT system by calling methods on infrastructure objects called *RRT instances*. There is an *RRT instance* in each address-space in the distributed system, analogous to a CORBA ORB. Each RRT instance provides three interfaces to application programmers. The first, called *IRafdaRunTime*, provides server-side operations to application objects collocated with the RRT instance, allowing programmers to expose objects or access frameworks that control transmission policy and distribution policy. The second, called *IRafdaRunTimeRemote*, provides client-side functionality to application objects that are remote with respect to the RRT instance, allowing programmers to obtain remote references to existing objects or to perform object migration. The third, called *IRafdaRunTimeConfig*, is used to control the behaviour of an RRT instance.

Figure 5.1 shows the RRT instances present in two address-spaces. The large circles represent objects in the distributed application. Each RRT instance is represented by a shaded box with the *IRafdaRunTime*, *IRafdaRunTimeRemote* and *IRafdaRunTime-Config* interfaces shown. Each RRT instance is accessible locally via the *IRafdaRunTime* and *IRafdaRunTimeConfig* interfaces and remotely via the *IRafdaRunTimeRemote* interface.

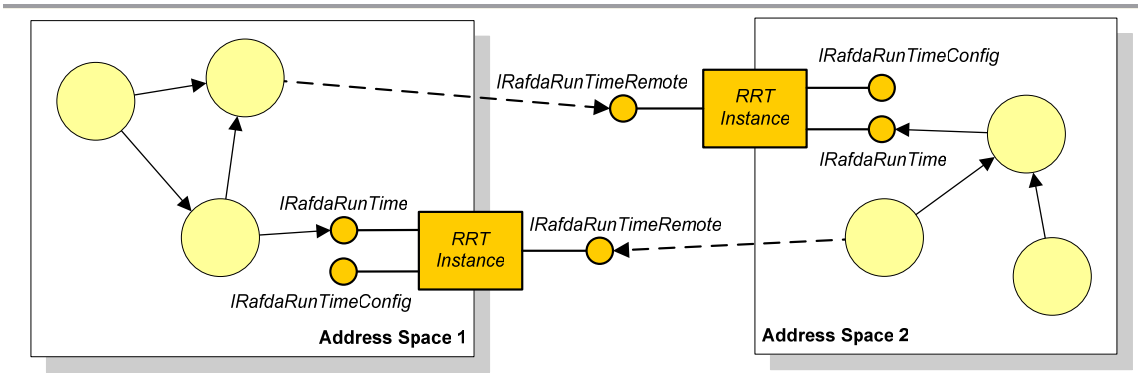


Figure 5.1: RRT instances exposing different interfaces to local and remote objects.

Figure 5.2 shows a subset of the functionality provided by the *IRafdaRunTime* interface. The *expose()* method is used to expose an object to remote access as a Web Service. The remote type argument is used to control which of the methods provided by the object will be remotely accessible. Remote types are discussed in detail later. The service name argument associates a name with the exposed object that can be used by clients to obtain remote references to the object.

```
public interface IRafdaRunTime {
      void expose(Object objectToExpose,
            Class remoteType,
            String serviceName);
      /* Other IRafdaRunTime methods omitted */
}
```

Figure 5.2: A subset of the *IRafdaRunTime* interface.

Figure 5.3 shows a subset of the functionality provided by the *IRafdaRunTimeRemote* interface. The *getRemoteReference()* method is used to obtain a remote reference to an exposed object using its name.

```
public interface IRafdaRunTimeRemote {
      Object getRemoteReference(String serviceName);
      /* Other IRafdaRunTimeRemote methods omitted */
}
```

Figure 5.3: A subset of the *IRafdaRunTimeRemote* interface.

The RRT provides a bootstrapping mechanism that allows programmers to obtain references to RRT instances, both local and remote. The *RRT* class shown in Figure 5.4 provides this functionality; it is assumed that this *RRT* class is available in every address-

space. The *get()* method returns a reference to the *IRafdaRunTime* interface of the local RRT instance, the *getRemote()* method returns a remote reference to the *IRafdaRunTimeRemote* interface of the RRT instance connected to the specified socket address and the *getConfig()* method returns a reference to the *IRafdaRunTimeConfig* interface of the local RRT instance.

```
public class RRT {
      public static IRafdaRunTime get() {…}
      public static IRafdaRunTimeRemote getRemote(
            InetSocketAddress isa) {…}
      public static IRafdaRunTimeConfig getConfig() {…}
}
```

Figure 5.4: The RRT class used by applications to obtain references to RRT instances.

5.2.2 Introducing Distribution into Applications

This section describes a simple example that illustrates how connectivity between address-spaces can be achieved. The JChord case study introduced in Chapter 2 is used throughout this chapter to illustrate the functionality provided by the RRT. Figure 5.5 shows the *Chord* abstract class, which implements the basic functionality of a Chord node implementation and declares several abstract methods that are implemented by a particular Chord node implementation.

```
public abstract class Chord {
      private Key key = null;
      public abstract Chord lookup(Key key);
      public abstract void addNode(Chord node);
      public abstract Chord getSuccessorNode();
      public Key getKey() {return key;}
      protected void setKey(Key key) {this.key = key;}
      public void printKeyInfo() {
            System.out.println("Key = " + key);
            System.out.println("Successor Key = " +
                  getSuccessorNode().getKey());
      }
}
```

Figure 5.5: The *Chord* abstract class.

Figure 5.6 shows the *JChordNode* class, which extends the *Chord* abstract class. The implementation details have been omitted as they are not important in this example. The *Chord* and *JChordNode* classes were designed to allow non-distributed simulations of Chord peer-to-peer networks and have not been written with concern for distribution. These classes do not extend any special base classes or implement any special interfaces.

```
public class JChordNode extends Chord {
	public JChordNode(Key key) {…}
	public Chord lookup(Key key) {…}
	public void addNode(Chord node) {…}
	public Chord getSuccessorNode() {…}
	public String getLog() {…}
	public void stop() {…}
	public void start() {…}
	public static String getVersion() {…}
}
```

Figure 5.6: The *JChordNode* implementation class.

An instance of *JChordNode* can be exposed to remote access as shown in Figure 5.7. Once an instance of *JChordNode* has been created, the *expose()* method provided by the local RRT instance is called. In this example, the exposed object's own class has been specified as the remote type, indicating that all methods should be exposed to remote access. The object has been exposed using the name *"JCNode"*. When this application is run, the RRT instance binds to the default port (5001) on the local host and exposes the *JChordNode* instance as a Web Service that can be remotely accessed using any Web Services technology.

```
public class JChordServer {
	public static void main(String[] args) {
		JChordNode jchordNode = new JChordNode(new Key());
		IRafdaRunTime rrt = RRT.get();
		rrt.expose(jchordNode,
			JChordNode.class,
			"JCNode");
	}
}
```

Figure 5.7: Exposing an instance of *JChordNode* to remote access.

In the client-side address-space, an application can obtain a remote reference to the exposed object directly from the RRT instance that exposes it. In Figure 5.8, the client obtains a remote reference to that RRT instance based on its socket address. It is assumed that this RRT instance is running on a machine called *"host.rafda.org"*, connected to the default port (5001). The client calls *getRemoteReference()* on the remote RRT instance, specifying the service name *"JCNode"*. A remote reference to the exposed object is returned, which the client casts into class *JChordNode*. This remote reference can be used as though it were a local reference to a local instance of *JChordNode*.

```
public class JChordClient {
      public static void main(String[] args) throws Exception {
            InetSocketAddress isa =
                  new InetSocketAddress("host.rafda.org", 5001);
            IRafdaRunTimeRemote remoteRRT = RRT.getRemote(isa);
            JChordNode node = (JChordNode) remoteRRT.
                  getRemoteReference("JCNode");
            System.out.println(node.getLog());
      }
}
```

Figure 5.8: Obtaining and using a remote reference to the exposed *JChordNode* instance.

Inter-address-space connectivity has thus been achieved without taking any special steps when creating the functional application classes *Chord* and *JChordNode*. Clients can access remote instances of the *JChordNode* class in the same manner as local instances. Thus, programmers have complete separation of functional concerns from those related to application distribution.

Thus far, only a subset of the functionality provided by the *IRafdaRunTime* and *IRafdaRunTimeRemote* interfaces has been shown. The following sections describe all the features provided by the RRT, both server-side and client-side, to allow programmers fine-grained control over application semantics. The functionality provided by the RRT is described in the context of the requirements stated in the previous chapter.

5.3 Server-Side Functionality

This section describes the server-side functionality of the RRT provided through the *IRafdaRunTime* interface. Using this interface, programmers can expose objects, perform migration, access the transmission policy manager that controls parameter-

passing semantics and access the distribution policy manager that controls object placement policy. The complete *IRafdaRunTime* interface is shown in Figure 5.9.

```
public interface IRafdaRunTime {
	/* Exposing objects to remote access */
	void expose(Object objectToExpose,
		Class remoteType,
		String serviceName);

	/* Migration */
	Object makeMigratable(Object object, Class remoteType);

	/* Automatically exposing objects to remote access */
	void associateClassWithRemoteType(
		Class applicationClass,
		Class remoteType);

	/* Policy Managers */
	ITransmissionPolicyManager getTransmissionPolicyManager();
	IDistributionPolicyManager getDistributionPolicyManager();

	/* Utility methods */
	IRafdaRunTimeRemote getExposingRRT(Object object);
}
```

Figure 5.9: The *IRafdaRunTime* interface.

The purpose of each of these methods is briefly summarized:

- The *expose()* method exposes objects to remote access.
- The *makeMigratable()* method converts application objects into functionally identical versions with support for object migration.
- The *associateClassWithRemoteType()* method controls which methods of a particular class are exposed to remote access when automatic exposure is performed. This method is described in Section 5.6.5.
- The *getTransmissionPolicyManager()* and *getDistributionPolicyManager()* methods allow access to the transmission policy framework and distribution policy framework respectively.
- The *getExposingRRT()* method is a utility method used to obtain a remote reference to the RRT instance that exposes a particular object.

This functionality is now examined in more detail in the context of the requirements defined in the previous chapter.

5.3.1 Exposing Objects as Web Services

Any object in a running application can be exposed at any point in its lifetime. A Web Service is created when an object is exposed, and the exposed object is the underlying object on which incoming requests to this service are performed. The attachment of a Web Service to an application object occurs transparently from the perspective of the application and so does not affect the execution semantics of the underlying application. Figure 5.10 shows a conceptual diagram in which an application consisting of objects labelled *A*-*E* exposes some of those objects as Web Services (objects *A*, *B* and *E*).

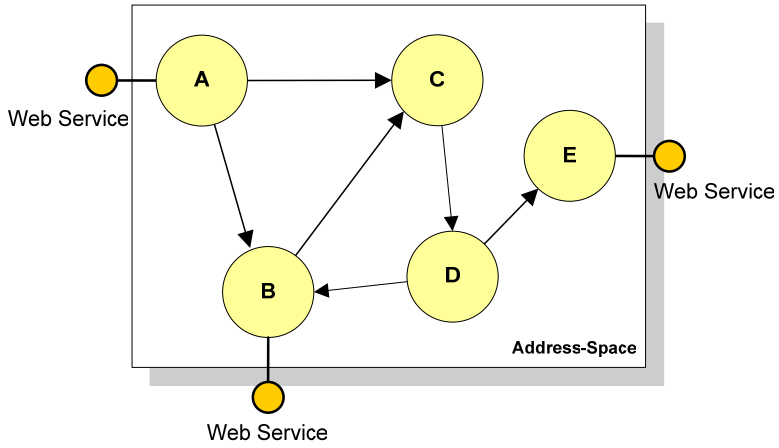


Figure 5.10: An application with some exposed objects.

Using standard Web Services has several advantages over a proprietary approach. Web Services provide interoperability across programming languages, architectures and operating systems. The underlying protocol, SOAP [87], is simple, standard, well supported and firewall-friendly (from the perspective of the application programmers).

Consider the first three requirements of a third generation middleware system as defined in the last chapter:

1: *All the ancillary code required to perform inter-address-space communication, namely proxy classes, skeletons, serializers and deserializers, must be created automatically.*

2: *It must be possible to expose instances of arbitrary classes to remote access without modifications to their source code.*

3: *It must be possible to expose objects to remote access dynamically.*

These requirements have been fulfilled by the functionality shown in the introductory example.

5.3.1.1 Remote Types

In the example shown in Figure 5.7, exposing the *JChordNode* instance meant exposing all of its method to remote access. Programmers need fine-grained control over the set of remotely available methods in order to allow information hiding. Further, it is often useful to allow a single object to present multiple views to remote clients, much as a conventional application class can implement multiple interfaces in order to present different encapsulated views over its functionality. The 4th requirement states:

4: *It must possible to control which methods are accessible remotely on a per-object basis, allowing remote clients to have multiple views over a single object.*

Every exposed object is associated, either implicitly or explicitly, with a *remote type* that controls which of its methods may be called remotely. A remote type is the distributed equivalent of an interface in a non-distributed application class and is used to control method visibility. Remote types provide multiple views over exposed objects to remote clients. From the perspective of clients, exposed objects are instances of their associated remote types. Different instances of a single class can be exposed with different remote types and a single object can be exposed multiple times with different remote types.

The RRT allows an object to be exposed using a particular remote type if and only if that remote type is *structurally compliant* with the exposed object's class, meaning that every method in the remote type has a counterpart with an identical signature in the exposed object's class. Therefore, the remote type associated with an exposed object need not be a super-class of the object's class nor an interface implemented by it. Programmers can expose instances of classes without the overhead of source level modifications.

5.3.1.2 Local Protection Mechanism

When determining which methods an object exposes to remote access, consideration must be given to the local protection mechanisms provided by the

implementation language, which control how instances of different classes can interact with each other. In Java, this protection is provided by the *public*, *protected*, *private* and default modifiers. For instance, in a Java class, public methods may be accessed from instances of any other class but private methods may only be accessed by instances of the same class.

If local protection semantics are preserved in a distributed application, the private methods of an exposed object must be accessible to remote objects of the same class but not to remote objects of other classes. As a consequence, methods cannot be universally defined as either accessible or inaccessible remotely.

The 5th requirement defined in the previous chapter states:

5: *The local protection mechanism must be preserved.*

When the RRT is used both server-side and client-side, local protection semantics are preserved. When an exposed object is accessed from clients using conventional (non-RRT-based) Web Service technologies, non-public methods may not be called by default. Programmers can override this behaviour to allow non-public access by altering the RRT configuration via the *IRafdaRunTimeConfig* interface, described at the end of this chapter. By allowing non-public access, programmers can attach debuggers or probes to existing applications without restricting which methods can be called.

5.3.1.3 Exposing Objects

The *expose()* method provided by the *IRafdaRunTime* interface is shown in Figure 5.11. It takes three arguments:

- A local reference to the object to be exposed.
- The remote type, as an instance of *java.lang.Class*, representing either a reified Java interface or a reified Java class.
- A service name.

```
void expose(Object objectToExpose,
      Class remoteType,
      String serviceName);
```

Figure 5.11: The *expose()* method.

The specified object is exposed, with the methods defined in the remote type made accessible to remote clients. Methods that the remote type inherits from its super-classes or super-interfaces are also exposed to remote access. If a Java class is specified as remote type, its method implementations are ignored and the class is treated as an interface. If the object to expose is remote with respect to this RRT instance or if the specified remote type is not structurally compliant with the object's class, meaning the remote type contains one or more methods for which there are no counterparts of the same signature in the object's class, the RRT throws an *IllegalArgumentException*. This exception is an unchecked exception, meaning that applications calling the *expose()* method need not statically define handlers, though if an unchecked exception occurs and is not caught, the RRT instance will immediately terminate.

The current RRT implementation does not permit remote types to be final classes or to contain final methods. Exposure will fail if exposure using such a remote type is attempted. The RRT provides a class loader that can be used to change application classes and methods such that they are non-final to overcome this limitation. However, the class loader cannot transform system classes dynamically, meaning that system classes that are final or contain final methods cannot be used as remote types. Implementations of the RRT in other languages may not exhibit this problem. A detailed description of the RRT prototype implementation is provided in the next chapter.

The following two figures show an example in which a *JChordNode* instance (Figure 5.6) is exposed three times with different remote types. The three remote types used are the *Chord* abstract class (Figure 5.5), the *IMonitor* interface (shown in Figure 5.12) and the *IManage* interface (also shown in Figure 5.12). The *IMonitor* interface provides methods used to monitor the running peer-to-peer system and the *IManage* interface provides methods used to manage nodes remotely.

```
public interface IMonitor {
      String getLog();
}
public interface IManage {
      void stop();
      void start();
}
```

Figure 5.12: The *IMonitor* and *IManage* interfaces.

The *JChordNode* class (Figure 5.6) does not implement either of these interfaces, though both interfaces are structurally compliant with the *JChordNode* class. Figure 5.13 shows an application in which an instance of *JChordNode* is exposed three times, once with each of the interfaces.

```
public class JChordServer {
      public static void main(String[] args) {
            JChordNode jchordNode = new JChordNode(new Key());
            IRafdaRunTime rrt = RRT.get();
            rrt.expose(jchordNode, IManage.class, "Manage");
            rrt.expose(jchordNode, IMonitor.class, "Monitor");
            rrt.expose(jchordNode, Chord.class, "Chord");
      }
}
```

Figure 5.13: Exposing an instance of *JChordNode* with multiple remote types.

Each time the object is exposed, a new Web Service is created. This Web Service is accessible via two URLs. One is based on the service name supplied by the programmer. However, this service name may subsequently be rebound to another service. In order to impose identity on services, a second URL is created for each service based on a randomly generated Universally Unique ID (UUID). This UUID-based URL is guaranteed to bind to the same Web Service for the service's lifetime. The service-name-based URL is an alias to this UUID-based URL. Web Service URLs take the form:

```
http://<machineName>:<port>/<serviceName or UUID>
```

For example:

```
http://host.rafda.org:5001/Manage
http://host.rafda.org:5001/b9d1052f-83f1-42f3-bf85-72fe6e17b169
```

The names of the methods provided by a Web Service attached to an object match the corresponding Java method names where possible. However, Java allows method names to be overloaded but Web Services do not. Figure 5.14 shows an interface that contains three methods to perform application profiling. Two methods are called *startProfiling()* and one is called *stopProfiling()*. The method name *"startProfiling"* is therefore overloaded whereas the method name *"stopProfiling"* is not.

```
public interface Profiler {
      void startProfiling(Chord ring, int time);
      void startProfiling(String profileName);
      void stopProfiling(String profileName);
}
```

Figure 5.14: An interface with an overloaded method.

If an object is exposed with a remote type that contains multiple methods with the same name, a naming scheme is employed to ensure that the names of the associated methods presented by the Web Service are unique within that service. A unique name for each overloaded method is constructed by appending the type signature of the method to its Java method name.

Figure 5.15 shows how the naming scheme is applied to this interface when it is used as a remote type. Since the *stopProfiling()* method is not overloaded, the name of the method provided by the Web Service is the same as the Java method name. Unique names for the two *startProfiling()* methods are generated based on their signature types. The *$* character is reserved in the Java Language Specification [17] for use in automatically generated code only. Therefore, the generated method names can be relied upon not to clash with existing application method names.

```
void startProfiling(Chord ring, int time)
      becomes startProfiling$Chord$int
void startProfiling(String profileName)
      becomes startProfiling$java.lang.String
void stopProfiling(String profileName)
      remains stopProfiling
```

Figure 5.15: Naming scheme for overloaded methods.

5.3.1.4 Limitations

The RRT does not expose fields to remote access directly. Fields can only be remotely accessed via get/set methods. If programmers perform direct field access on remote objects, application semantics are unpredictable. However, it is generally considered bad software engineering practice to access state in other objects directly, as encapsulation is lost. Thus, this limitation is not considered serious. Proposed future work

could transform applications dynamically to ensure that all direct field access operations are changed into accessor method calls, thereby completely removing this limitation.

5.3.1.5 Exposed Object Lifetime

When creating applications using garbage collected languages such as Java, the memory used by objects is reclaimed when those objects are no longer referenced directly or indirectly from the running application [78]. Once objects are exposed to remote access, they may at some time be referenced only remotely from outwith their local address-spaces. The local garbage collector in each address-space cannot detect these remote references and so may collect the objects. The RRT infrastructure can hold local references to exposed objects in order to prevent their collection. The RRT offers three approaches to controlling the lifetime of exposed objects:

1. Always allow the local collector to collect exposed objects when they are no longer referenced locally, even if remotely referenced.
2. Allow the local collector to collect exposed objects that have not been remotely accessed within a particular lease time.
3. Never allow the local collector to collect exposed objects.

These three approaches allow programmers to trade off safety for completeness in garbage collection terms. Programmers determine which approach is adopted on a per-RRT-instance basis via the *IRafdaRunTimeConfig* interface.

The first approach is complete but unsafe as it allows objects to be collected while still remotely referenced. It is particularly suitable for applications like probes or debuggers in which clients do not wish to hold references to objects once they are no longer live within the application. When exposed objects are collected, the Web Services associated with these objects are shut down. Extant remote references to collected objects become invalid. Any attempts to perform remote method calls on these references will fail.

The second approach is particularly suitable when the RRT is employed as a traditional DOM as it allows the system to perform lease-based distributed garbage collection [88]. It is assumed that any objects that are not remotely accessed within a programmer-defined lease time are not remotely referenced and so may be collected. It balances completeness and safety by ensuring that any objects that are frequently accessed remain live in the distributed applications, even if they are not referenced locally. Programmers can increase the lease time to decrease the risk that remotely

referenced application objects will be collected with the increased risk that the available memory will fill with garbage objects that cannot be collected because they hold valid leases.

The third approach is safe but not complete. Exposed objects will never be collected until they are removed from remote access manually. This approach is particularly suitable when the RRT is used as a conventional application container providing services to remote clients. In this case the availability of services is the primary concern and services must remain live even if not accessed for long periods.

5.3.1.6 Accessing the RRT via a Web Browser

Each RRT instance can be accessed using a conventional web browser to show all available Web Services. This allows application programmers to gain a global view of all remotely accessible objects in the RRT instance and to inspect the state of the running application.

The list of available Web Services is shown, with the following information:

- The remote type.
- The service name (URL).
- The class of the exposed object.
- A string representation of the exposed object.

Figure 5.16 shows the results of attaching a web browser to the RRT instance that is running the application shown in Figure 5.13, which exposes an instance of *JChordNode* with three different remote types. Note that all classes are in a Java package named *jchord* and that the RRT instance is bound to port 5001 on a machine called *"host.rafda.org"*. Since each Web Service is accessible via a URL based on the service UUID and a URL based on the service name, the three calls to the *expose()* method have resulted in six entries in the table shown.

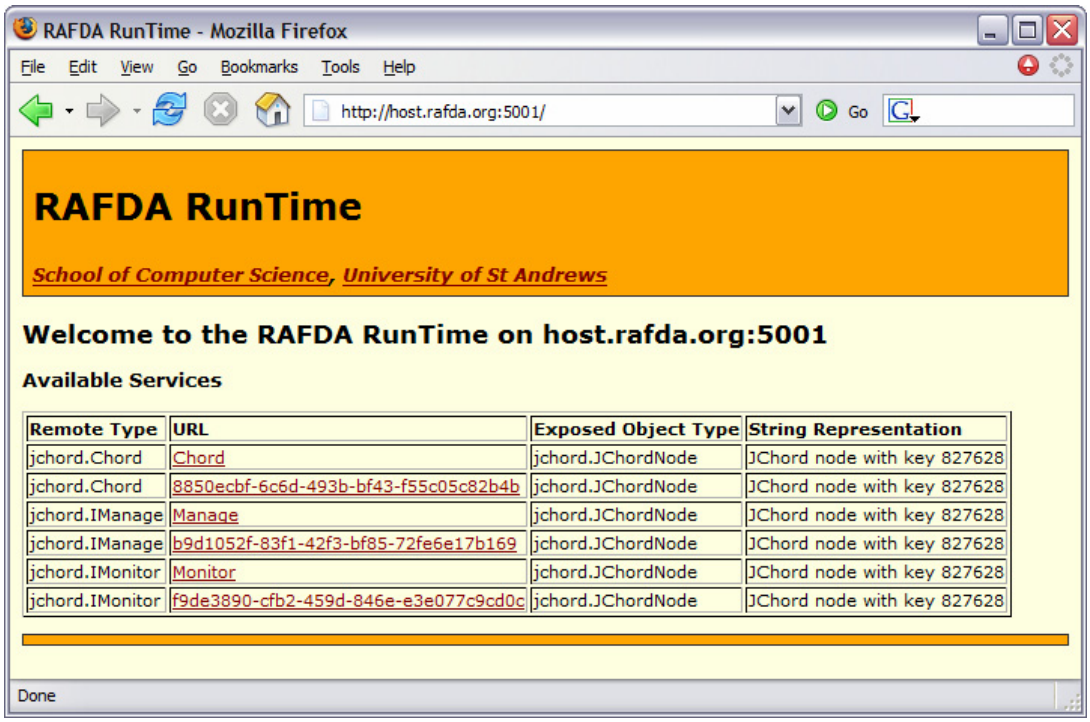


Figure 5.16: A web page generated by an RRT instance showing the objects it exposes.

The links in the URL column refer to service-specific pages that provide:

- A list of the methods provided by the remote type.
- A list of the methods and fields provided by the exposed object's class.
- The current state of these fields in the exposed object.

By default, RRT instances show information only about the remote types. The information about the underlying exposed object is not available unless this functionality is explicitly enabled in the RRT configuration.

Figure 5.17 shows the page associated with the service named "*Manage*". This page shows the methods specified by the remote type (*IManage*). Only these methods may be accessed remotely. The methods implemented by the exposed object's class and the current state of the exposed object are shown in this case.

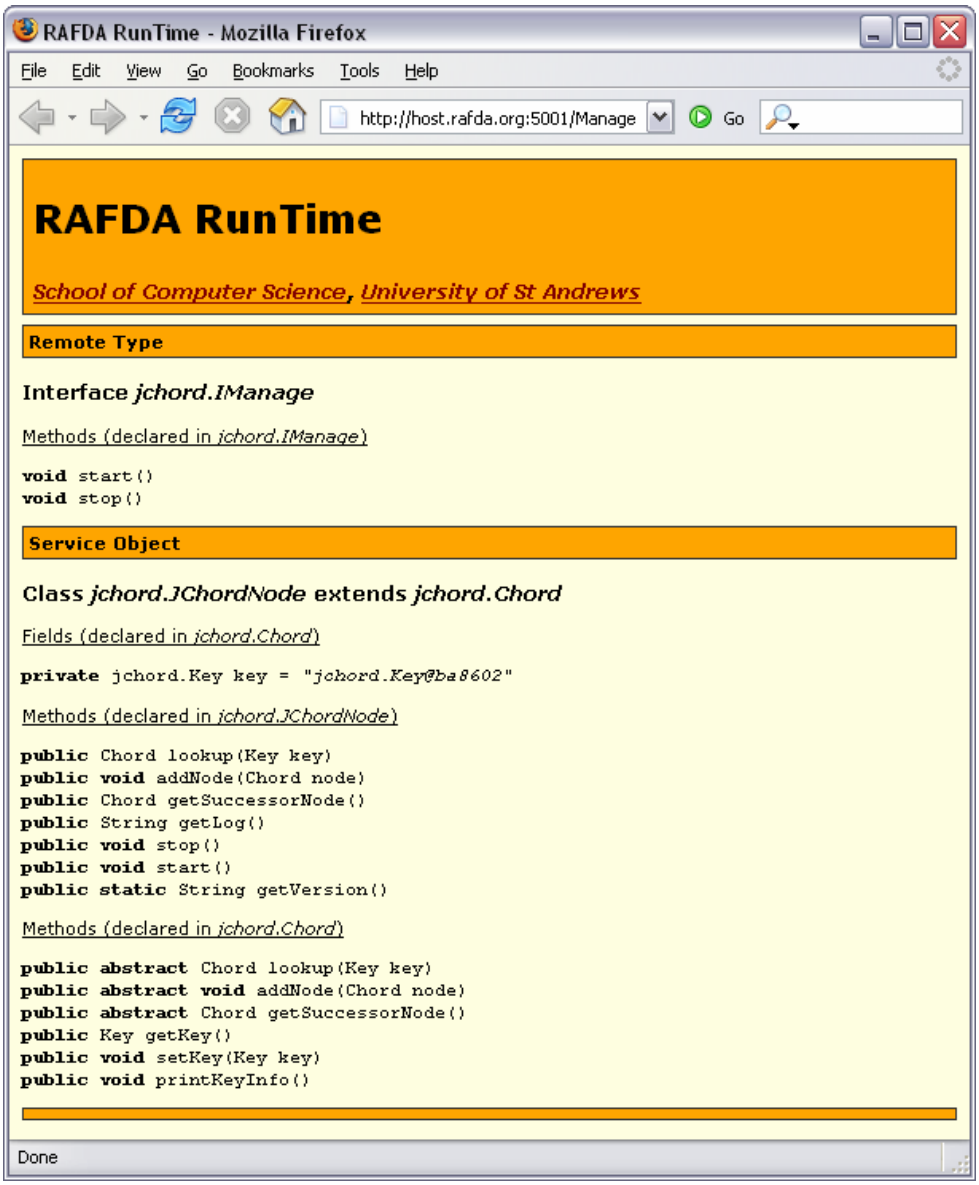


Figure 5.17: Detailed information about the *Manage* service.

5.3.1.7 Security

As described so far, the RRT exposes application objects to remote access and provides information to clients connecting through web browsers without concern for security. Though the RRT is primarily intended for use in environments where programmers have complete control over all machines in the distributed system and connecting network, several security features are provided.

A firewall built into each RRT instance can be configured to allow connections only from trusted addresses. When active, only clients on trusted machines can perform

remote calls on exposed objects or otherwise interact with the RRT instance. Similarly, only web browsers running on trusted machines are allowed to connect.

The RRT permits programmers to provide custom socket implementations, such as encrypted sockets, which are used when performing inter-address-space communication. When encrypted sockets are used in conjunction with the firewall, RRT-based applications can be deployed securely on a trusted subset of machines on an un-trusted network. This functionality can also be exploited to simulate network connections with varying bandwidths, latencies or failure rates.

5.3.2 Passing Arbitrary Objects By-Value

The RRT permits the transmission of arbitrary objects across the network by-value to implement conventional Web Service semantics when communicating with clients using traditional Web Services technologies. This mechanism is also useful when both client and server are RRT-based as it can be used to cache and replicate application objects. The RRT must therefore be capable of serializing and deserializing instances of any classes. The 6th and 7th requirements from the previous chapter state:

6: *It must be possible to pass any objects by-value as arguments or return values to remote method calls.*

7: *A scalable resilient code distribution scheme must be provided.*

Each RRT instance provides two approaches to serialization/deserialization. Programmers may choose which is adopted via the RRT configuration. The approaches are:

1. Perform all serialization/deserialization using a generic, reflection-based serializer/deserializer.
2. Perform serialization/deserialization using per-class custom serializer/ deserializers, which are generated and compiled dynamically by the RRT. The RRT employs generative techniques to create serializers/deserializers that are tuned to work with the classes in each particular application.

The approaches offer different trade-offs [79]. The generic serializer/deserializer can serialize and deserialize instances of any class, using reflective techniques to access the internal state of objects. Each generated per-class serializer/deserializer can serialize and deserialize only instances of one particular class but does not employ reflection at

serialization/deserialization time. The advantage of the latter approach lies in the cost difference between accessing fields in objects directly and accessing them using the reflection tools, which is typically an expensive operation.

There is a one-time cost incurred by generating and compiling the per-class serializer/deserializers. In applications that serialize/deserialize a large number of instances of the same class, the one-time cost of creating the per-class serializer/ deserializers is outweighed by the lower cost of each serialization/deserialization operation. By default, per-class serializer/deserializers are discarded when the RRT instance terminates. Programmers can alter the RRT configuration so that RRT instances cache the per-class serializer/deserializers for future use, thereby avoiding the cost of re-generation.

When deserializing data into an object, the RRT needs access to the code associated with the serialized object in order to instantiate it. If the associated class file cannot be loaded, the RRT obtains the code automatically from the RRT instance that serialized the object, which necessarily has access to the class. Code is lazily distributed throughout the distributed system as necessary, negating the need to perform code distribution manually. The risk of programmer-related errors caused by failure to distribute code correctly is removed.

5.3.3 Summary

This section has described the server-side functionality of the RRT provided via the *IRafdaRunTime* interface. The RRT allows the exposure of arbitrary objects to remote access by clients in remote address-spaces. Remote types allow programmers control over the methods that objects expose to remote access. This provides multiple views over exposed objects to clients, allowing programmers to preserve encapsulation in distributed applications. The RRT conceals the distributed nature of applications unless programmers explicitly expose the inter-address-space communication. Programmers benefit by using the RRT due to the simplified software engineering process, separation of concerns and the flexibility of the RRT to adapt to the requirements of different distributed applications.

The RRT provides several approaches to exposed object lifetime management, which can allow the local garbage collector to reclaim remotely accessed objects if they are no longer referenced locally, provide DOM-style lease-based garbage collection or

retain the semantics of service-oriented application containers by permitting no collection of exposed objects.

The RRT is capable of serializing and deserializing instances of arbitrary classes, allowing these objects to be transmitted across the network by-value. Thus, conventional Web Services semantics can be preserved and the RRT can cache or replicate application objects across multiple address-spaces.

5.4 Client-Side Functionality

The server-side functionality described in Section 5.3 is sufficient to allow the exposure of objects to access by clients using the RRT and other Web Services technologies. This section fully describes the RRT client-side functionality and shows its advantages over other technologies. Full Distributed Object Model functionality is provided when the RRT is used both server- and client-side, allowing the creation of isomorphic distributed versions of non-distributed applications.

Conventional Web Services technologies do not provide remote reference schemes and so objects may only be passed across address-space boundaries by-value. Without the ability to pass-by-reference, programmers are limited in terms of the applications that can be created. The 8th, 9th and 10th requirements express the necessity of support for remote references and pass-by-reference semantics. They state:

8: *It must be possible to hold remote references to arbitrary objects and to treat local and remote references in the same manner.*

9: *It must be possible to associate names with remotely accessible objects, either implicitly or explicitly, and to obtain remote references to objects based on those names.*

10: *It must be possible to pass any objects by-reference as arguments or return values to remote method calls.*

The 9th requirement has already been partially met as the RRT allows service names to be associated with objects at exposure. In order to fully meet all three requirements, the RRT introduces a remote reference scheme that is synergistic with existing Web Services technologies. Using the remote reference scheme built into the RRT, remote references to exposed objects can be passed across address-space

boundaries as arguments or return values to remote method calls. When methods are invoked on remote references, the calls are propagated across the network and performed on the exposed objects.

To allow any object to be passed by-reference, including those that are not yet exposed, the RRT provides automatic exposure of objects on demand. This ensures that any application object that is passed by-reference is remotely accessible. Each RRT instance differentiates between RRT-based clients and those that use conventional Web Services technologies. It ensures that remote references are never transmitted to clients that are not RRT-based.

Each RRT instance provides functionality to remote clients through the *IRafdaRunTimeRemote* interface shown in its entirety in Figure 5.18. This interface captures the functionality that RRT instances provide to application objects in remote address-spaces.

```
public interface IRafdaRunTimeRemote {
      Object getRemoteReference(String serviceName);
      Object instantiateAndExpose(Class classToInstantiate,
            Object[] constructorArguments,
            Class remoteType,
            String serviceName);
      void migrate(Object objectToMigrate);
      IDistributionPolicyManager getDistributionPolicyManager();
}
```

Figure 5.18: The *IRafdaRunTimeRemote()* interface.

This purpose of each of these methods is briefly summarized here, then described in detail later in this section:

- The *getRemoteReference()* method was introduced in Section 5.2. It is used to obtain a remote reference to an object exposed by this RRT instance.
- The *instantiateAndExpose()* method instantiates an object in this RRT instance and exposes it immediately to remote access.
- The *migrate()* method migrates an object to this RRT instance.
- The *getDistributionPolicy Manager()* method allows remote clients access to the distribution policy framework on this RRT instance.

Clients obtain references to the *IRafdaRunTimeRemote* interface using the *RRT* class introduced previously (Figure 5.4).

5.4.1 Accessing Remote Objects

The *getRemoteReference()* method allows programmers to bootstrap distributed applications; using this method, application objects in separate address-spaces can obtain references to each other, based on service names. The returned remote references will bind to particular services for their lifetimes, even if the specified service names are rebound to different services. This approach ensures that the rebinding of service names to different objects does not alter existing references as a side-effect.

Returning to the JChord case study, the non-distributed JChord implementation allows researchers to simulate JChord rings in a single address-space. Figure 5.19 shows an application that creates a non-distributed simulation of a JChord ring. The application performs two distinct tasks. An initial node is created, then multiple peers are added to the ring through repeated calls to the *addNode()* method of the initial node.

```
public class LocalRing {
	public static void main(String[] args) {
		JChordNode jchordNode = new JChordNode(new Key());
		jchordNode.addNode(new JChordNode(new Key()));
		jchordNode.addNode(new JChordNode(new Key()));
		jchordNode.addNode(new JChordNode(new Key()));
	}
}
```

Figure 5.19: A JChord ring created in a single address-space.

A distributed version of this application can be created by creating two entry points into the application, one of which is used to create a ring while the other is used to join an existing ring. Figure 5.20 shows the former. The initial JChord node is created and exposed to remote access with the name *"initialNode"*.

```
public class NewRing {
	public static void main(String[] args) {
		JChordNode jchordNode = new JChordNode(new Key());
		RRT.get().expose(jchordNode,
			JChordNode.class,
			"initialNode");
	}
}
```

Figure 5.20: Creating the initial node.

The other entry point, which creates a node and then joins it to an existing ring by calling the *addNode()* method of the initial node, is shown in Figure 5.21. A truly distributed version of the original JChord application has been created without making changes to any of the underlying *JChordNode* implementation classes.

```
public class JoinRing {
      public static void main(String[] args) throws Exception {
            InetSocketAddress isa = new
                  InetSocketAddress("host.rafda.org", 5001);
            IRafdaRunTimeRemote remoteRRT = RRT.getRemote(isa);
            JChordNode jchordNode = (JChordNode) remoteRRT.
                  getRemoteReference("initialNode");
            jchordNode.addNode(new JChordNode(new Key()));
      }
}
```

Figure 5.21: Joining an existing JChord ring.

When the *addNode()* method is called in the original application shown in Figure 5.19, Java semantics dictate that the argument is passed by-reference. The RRT allows programmers control over the parameter-passing mechanisms applied to arguments when remote method calls are performed but adopts pass-by-reference by default. This distributed JChord implementation takes advantage of the default pass-by-reference semantics in order to retain local calling semantics when remote calls are performed. When *addNode()* is called, the *JChordNode* argument will be passed by-reference thereby concealing distribution from the programmers.

5.4.2 Static Members

The semantics of static members (that is, methods and fields) differ from instance methods and fields as static members are associated with classes rather than objects. The 11th requirement in the previous chapter states:

11: *It must be possible to preserve non-distributed static semantics in distributed applications and to control the semantics of static members on a per-application basis.*

In local applications, there is one unique copy of each static field in each class that is shared by all objects in the application. The RRT allows static members to be handled in one of two ways in distributed applications:

1. A copy of each static field is stored in each address-space and is only ever accessed locally. This approach has the benefit that no remote calls need ever be performed to access static methods. Static methods execute locally and access only their local copy of the fields.
2. A single copy of each static field in stored in the entire distributed system. This approach preserves non-distributed static semantics but can incur the cost of remote method calls when accessing static members.

Using the first approach, each class is loaded in each address-space and stores a complete copy of all static fields. All static methods execute locally when called. Using the second approach, exactly one RRT instance in the distributed system is designated the root RRT instance. The root is responsible for managing static member access in the distributed system and the application programmer must define which particular RRT instance is the root in the configuration of every RRT instance in the distributed system.

The second approach requires that the RRT use a class loader to transform application classes automatically. When the distributed application is run, the programmer specifies a command line argument to indicate whether the class loader is employed. If an application is run without using the class loader, the first approach is adopted. If the class loader is employed, then the second approach is adopted. The use or otherwise of the command line argument implicitly indicates which approach should be employed. As a consequence of using Java as the underlying implementation language, this dynamic transformational approach is limited with respect to system classes because these classes cannot be altered dynamically. Non-distributed static functionality may therefore only be preserved for application classes.

The manner in which the RRT handles static members differentiates it from other middleware systems, all of which adopt only one of the above approaches. Neither of the approaches is suitable for all distributed applications so the RRT allows programmers to adapt middleware behaviour to the particular requirements of each application.

5.4.3 Failure

The RRT provides a failure model that offers programmers a number of approaches to handling errors. The introduction of distribution into applications brings

new types of failure mode. For certain types of distributed application any such failure is immediately terminal, much as a failure within a single machine is terminal for local applications. For other types of distributed application, the programmer may need to handle errors occurring due to network failure explicitly or to continue execution on a best-effort basis. The RRT allows programmers to decide on a per-application basis whether to handle such errors manually or defer responsibility to the RRT.

There are two kinds of failure that can occur in distributed applications, namely:

- Distribution-related exceptions that occur as a direct result of the distributed nature of the application, such as network failure or remote machine failure.
- Application exceptions that occur for reasons orthogonal to the distributed nature of the application.

The 12th requirement from the previous chapter states:

12: *The programmer must be able to control whether distribution-related failures are propagated to the application or handled internally by the middleware system.*

Application exceptions are always thrown back to clients as they are not the concern of the RRT. Distribution-related exceptions are either handled directly by the RRT instance or propagated back to clients. Programmers control which approach is utilized in the RRT configuration, which is summarized in Appendix C.

There are three approaches to handling distribution-related errors that are open to developers:

1. Configure the RRT to handle all distribution-related exceptions internally. If failure occurs, default values are returned. No application-level exception handlers need to be defined in this case.
2. Configure the RRT to propagate all distribution-related exceptions to the clients but do not define application-level exception handlers. If failure occurs, the uncaught exception causes the RRT instance to terminate immediately.
3. Configure the RRT to propagate all distribution-related exceptions to the clients and define application-level exception handlers statically at any points in the application where failure can occur. If a distribution-related exception occurs, it is handled in a programmer-defined manner.

If the RRT is configured to handle all distribution-related errors internally, it logs failures and returns default values to the clients (null, 0, etc.). The client code need not create any special handlers for distribution-related exceptions, provided that programmers accept that remote method calls may return default values. This approach is suitable for application prototyping or for applications in which it is assumed that network failure will not occur (a reasonable assumption on a LAN).

Figure 5.22 shows part of an application in which the RRT handles all distribution-related errors internally. The application obtains a remote reference to an object exposed using the *IMonitor* interface (Figure 5.12). When the remote call is made to *getLog()*, no exception handler is defined and a null value is returned if the remote call fails.

```
IRafdaRunTimeRemote remoteRRT = RRT.getRemote(
      new InetSocketAddress("host.rafda.org", 5001));
IMonitor monitoredNode = (IMonitor) remoteRRT.
      getRemoteReference("Monitor");
/* 'log' will be set to null if a distribution-related
 * exception occurs when getLog() is called */
String log = monitoredNode.getLog();
```

Figure 5.22: An application in which the RRT handles distribution-related errors.

Distribution-related exceptions are wrapped in *unchecked exceptions*. In Java, methods do not need to declare statically that they throw unchecked exceptions and callers are not forced to define handlers. If the RRT is configured to propagate distributed-related exceptions, but clients do not define handlers then the Java Virtual Machine terminates if an exception occurs. Programmers are not forced to handle errors explicitly. In applications that consider distribution-related failures to be terminal, the application logic is not permeated by distribution-related code. The code in an application adopting this approach is unchanged from that shown in Figure 5.22.

In order to handle distribution-related exceptions, programmers must define handlers that catch instances of *RafdaRuntimeException* as shown in Figure 5.23. The *RafdaRuntimeException* wraps the distribution-related exception, which can be extracted with a call to *getCause()*.

```
IRafdaRunTimeRemote remoteRRT = RRT.getRemote(
      new InetSocketAddress("host.rafda.org", 5001));
IMonitor monitoredNode = (IMonitor) remoteRRT.
      getRemoteReference("Monitor");
String log = null;
try {
      /* Distribution-related exception could occur here */
      log = monitoredNode.getLog();
} catch (RafdaRuntimeException rre) {
      /* Handle the exception */
      Throwable cause = rre.getCause();
      cause.printStackTrace();
}
```

Figure 5.23: An application that handles distribution-related errors.

By providing a multiplicity of approaches to handling failure, the RRT simplifies application prototyping as programmers can ignore the possibility of distribution-related exceptions during initial development. The RRT offers programmers the flexibility to introduce error handling code into applications only where it is deemed necessary.

5.4.4 Creating Objects in Remote Address-Spaces

The RRT provides a mechanism that allows programmers to instantiate application objects in arbitrary address-spaces in the distributed system, thereby implementing the 13th requirement, which states that:

13: *It must be possible to instantiate objects directly in remote address-spaces.*

The RRT provides the *instantiateAndExpose()* method in the *IRafdaRunTimeRemote* interface, which allows applications to be deployed across the distributed system from a single starting point. This method, shown in Figure 5.24, permits programmers to instantiate arbitrary classes in remote address-spaces then expose the instantiated objects to remote access. It takes the following parameters:

- The class of the object to instantiate.
- Constructor arguments.
- The remote type with which the newly created object should be exposed.

- The service name with which the newly created object should be exposed.

A remote reference to the exposed object is returned to the caller. The remote RRT instance automatically determines which constructor to use based on the types of the constructor arguments. If no constructor that takes the specified constructor arguments is found, an *IllegalArgumentException* is thrown.

```
Object instantiateAndExpose(Class classToInstantiate,
      Object[] constructorArguments,
      Class remoteType,
      String serviceName);
```

Figure 5.24: The *instantiateAndExpose()* method used to create objects in remote address-spaces.

Figure 5.25 shows how this method can be used to distribute JChord nodes across a distributed system. Initially, an array of remote references to the available RRT instances in the system is created. The *instantiateAndExpose()* method is called on each remote RRT instance instructing it to create an instance of the *JChordNode* class with a single constructor argument of type *Key*.

Once created, each *JChordNode* instance is exposed with remote type *Chord* and service name *"Node"*. A remote reference to the newly created node is returned and passed as argument to the *addNode()* method of its predecessor, in order to connect the deployed nodes together into a ring. The programmer can deploy a complete ring from a single application entry point, mimicking local application behaviour.

```
public void deployRing() {
      /* Get references to the RRTs in the distributed system */
      IRafdaRunTimeRemote remoteRRTs[] = new IRafdaRunTimeRemote[] {
            RRT.getRemote(
                  new InetSocketAddress("host1.rafda.org", 5001)),
            RRT.getRemote(
                  new InetSocketAddress("host2.rafda.org", 5001)),
            RRT.getRemote(
                  new InetSocketAddress("host3.rafda.org", 5001)),
            RRT.getRemote(
                  new InetSocketAddress("host4.rafda.org", 5001))
      };
      /* Create a node in each RRT instance and add it to the ring */
      Chord[] remoteNodes = new Chord[4];
      for (int x = 0; x < remoteRRTs.length; x++) {
            remoteNodes[x] = (Chord) remoteRRTs[x].
                  instantiateAndExpose(
                        JChordNode.class,
                        new Object[] { new Key() },
                        Chord.class,
                        "Node");
            if (x > 0)
                  remoteNodes[x - 1].addNode(remoteNodes[x]);
      }
}
```

Figure 5.25: Deploying a JChord ring using the *instantiateAndExpose()* method.

There are security implications for the remote RRT instances involved in this process. No sandboxing mechanisms are provided to restrict the operations that remotely instantiated objects can perform. This functionality is intended for use in trusted distributed systems, in which no byzantine RRT instances are present. Such a system can be constructed using the firewall and secure socket functionality described earlier. However, support for remote instantiation is not available by default and must be activated explicitly via the RRT configuration.

5.4.5 Migrating Objects to Remote Address-Spaces

The RRT allows the migration of objects between address-spaces, permitting application distribution boundaries to be modified to take advantage of changes in the

underlying distributed system or in the application itself. For example, as new machines are added to the distributed system, it may be desirable to re-distribute application objects to reduce the overall load on each machine in the system. Similarly, if objects that interact heavily are collocated then the number of (expensive) remote method calls that need to be performed can be reduced. Application objects can be migrated between address-spaces in order to collocate objects with their working sets as the application executes. Thus, the RRT implements the 14th requirement from the previous chapter, which states:

14: *It must be possible to perform the migration of objects from one address-space to another without loss of application consistency.*

Migration is completely transparent to reference holders but requires that programmers perform an explicit preparatory step. This step is performed dynamically by calling the *makeMigratable()* method, which is provided by the local *IRafdaRunTime* interface and shown in Figure 5.26

```
Object makeMigratable(Object object, Class remoteType);
```

Figure 5.26: The *makeMigratable()* method used to introduces support for migration into objects.

The programmer supplies a reference to an arbitrary local object and a remote type as arguments to this method. If, in the future, the object is migrated to a remote address-space, it will be exposed in that address-space using this remote type. The *makeMigratable()* method returns a wrapper that encapsulates the application object but is an instance of the remote type. This wrapper introduces a layer of indirection between local reference holders and the wrapped application object. All local references to the application object must be updated, by the programmer, to refer to the wrapper, therefore providing location transparency. If the application holds any direct references to the original application object, rather than the returned wrapper, migration will not proceed correctly resulting in the loss of application coherency.

Once an object has been wrapped by passing it as an argument to the *makeMigratable()* method, it may be migrated directly by programmers, passed by-migrate as an argument or return value, or migrated as the result of distribution policy evaluation. Object migration is performed by calling the *migrate()* method (Figure 5.27)

provided by the *IRafdaRunTimeRemote* interface. Passing objects by-migrate and the use of distribution policy are discussed in Sections 5.5 and 5.7 respectively.

```
void migrate(Object objectToMigrate);
```

Figure 5.27: The *migrate()* method used to migrate objects between address-spaces.

The Data Store built on JChord that was introduced in Chapter 2 is used to illustrate migration. To implement the Data Store service, multiple Data Store objects are created in the distributed system, each of which holds references to a sub-set of the stored objects. Figure 5.28 shows a distributed system in which a Data Store object in *host2.rafda.org* (labelled *DS*) is storing a database object (labelled *DB*). The database object is referenced by multiple application objects, *A*, *B* and *C*.

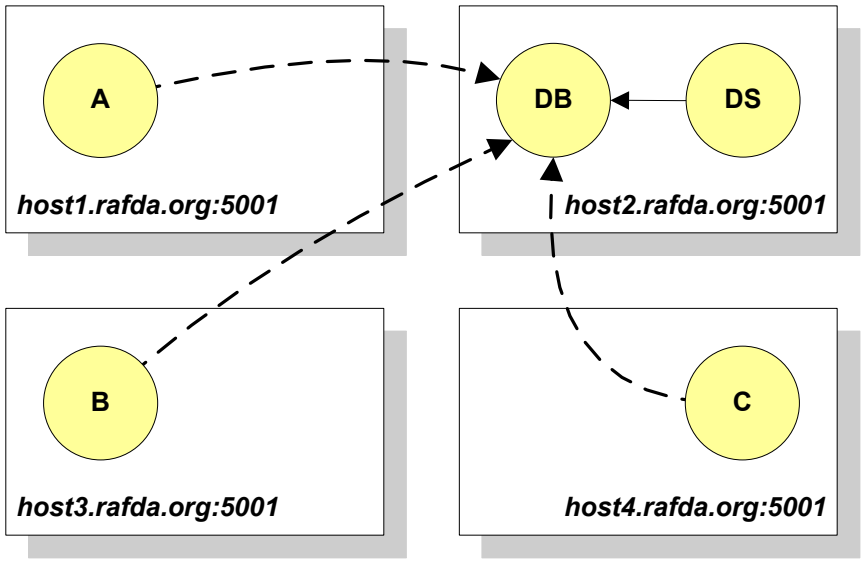


Figure 5.28: A stored database object that is referenced by multiple application objects.

At some point, the Data Store object determines that resources are running low on *host2.rafda.org* and determines that the database object should be migrated to another machine, namely *host1.rafda.org*. Figure 5.29 shows the code executing within the Data Store object that performs this migration. It is assumed that the database object is held in the *db* field and is of class *Database* (not shown). Initially, support for migration is introduced into the database object by calling *makeMigratable()* and the *db* field is updated to refer to the migratory version of the database object. A remote reference to the RRT instance in *host1.rafda.org* is obtained and the migration performed.

```
IRafdaRunTime rrt = RRT.get();
db = (Database) rrt.makeMigratable(db, Database.class);
InetSocketAddress isa = new InetSocketAddress("host1.rafda.org", 5001);
IRafdaRunTimeRemote remoteRRT = RRT.getRemote(isa);
remoteRRT.migrate(db);
```

Figure 5.29: Migrating the database object to another address-space.

Figure 5.30 shows the application after the migration has completed. Once the one-time preparatory step that wraps the database object has been taken, the object can migrate transparently with respect to its reference holders.

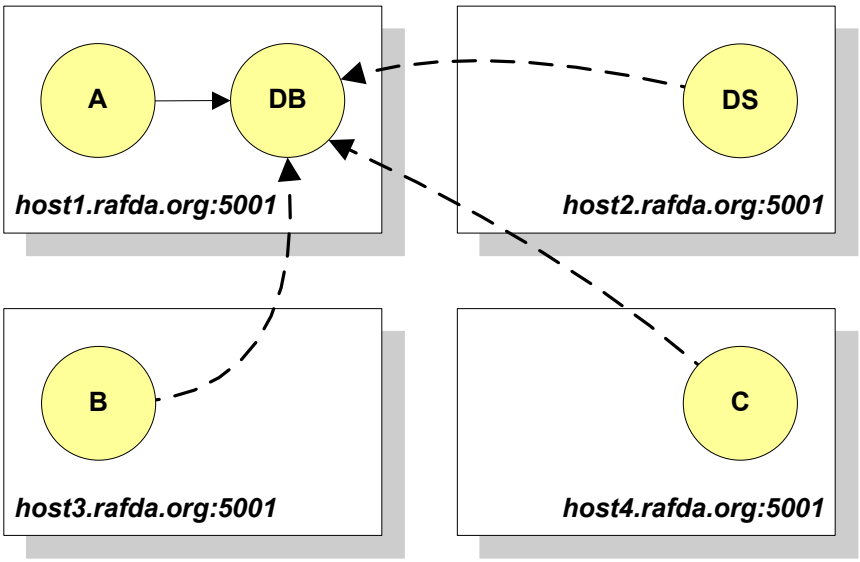


Figure 5.30: The database application after migration.

5.4.6 Summary

The section has described the main client-side functionality of the RRT, which allows the creation of distributed applications by permitting clients to obtain remote references to exposed objects. Remote references can be treated in the same manner as local references and may be passed between address-spaces as arguments or return values. Application distribution and logic are separated so programmers need not consider at design time which class of object will participate in inter-address-space communication.

Multiple failure models are provided, giving programmers freedom to handle distribution-related errors, or to allow the RRT to handle them in a best effort fashion. Control over static members provides a choice between the preservation of local

semantics and the introduction of per-address-space copies of static fields. The RRT provides support for remote object instantiation and object migration, allowing applications to adapt dynamically to changes in the underlying distributed systems or in the applications themselves.

The purpose of this client-side functionality is to aid the software engineering process so that distribution can be treated as a non-functional concern. The RRT conceals inter-address-space communication by providing remote calling semantics that reflect local calling semantics. An isomorphic distributed version of an existing application can be created with minimal programming effort. The RRT allows programmers fine-grained control over the behaviour of the RRT in order to expose and control the distributed nature of an application to an arbitrary extent. Programmers have the flexibility to configure the middleware to meet the requirements of a particular application instead of adapting the application to accommodate the limitations of the middleware system.

5.5 Summarizing the Limitations of the RRT

The current Java-based implementation of the RRT exhibits some limitations, which are described throughout this chapter. These limitations are summarized here.

1. Remote types may not be system classes that are final or contain final methods.
2. Non-distributed static semantics cannot be preserved for system classes.
3. Arrays may not be passed by-reference.
4. Accessor methods must be defined for all fields.

Limitations 1 and 2 could be resolved through the use of a tool that transforms system classes ahead of application run-time. This tool would perform the same transformations as the class loader, which cannot itself perform the transformations as the Java Virtual Machine does not permit alterations to system classes dynamically. Limitations 3 and 4 could be resolved for application classes by the RRT class loader or ahead of application execution using this offline transformation tool.

5.6 Controlling Transmission Policy

The transmission policy active in an application determines the parameter-passing semantics employed when remote methods are called. The RRT allows programmer control over the parameter-passing mechanisms that apply to objects:

- Pass-by-value, which passes duplicate copies of arguments across address-space boundaries.
- Pass-by-reference, which passes remote references to arguments across address-space boundaries.
- Pass-by-migrate, which migrates arguments across address-space boundaries.

Hybridisation, whereby some object state is cached at a client whilst other state is remotely accessed, is supported. Using the transmission policy framework, programmers can employ the most advantageous parameter passing semantics for the circumstances of each application. This provides programmers with control over application semantics and promotes reuse of library classes in distributed contexts since the transmission policy can be specified independently of class implementation. Library classes need make fewer assumptions about the environment in which they are to be deployed since programmers have the freedom to apply any parameter-passing policy to instances of any class, increasing the likelihood that any given class will be reusable in another context.

5.6.1 Defining Transmission Policy

When interacting with clients using standard Web Services technologies, RRT instances adopt pass-by-value semantics, as required by standard Web Services. However, when interacting with RRT based clients, any of the three parameter-passing mechanisms can be employed. The transmission policy framework described here provides a mechanism to allow the programmer to specify dynamically how objects are marshalled when passed as arguments and return values when remote methods are called. This transmission policy framework allows programmers to define transmission policy, which controls parameter-passing semantics, and caching policy which controls caching and hybridization of the by-reference and by-value mechanisms. The framework supports four types of rule governing transmission policy and two types of rules governing caching of methods and fields in remote references.

5.6.1.1 Transmission Policy Rules

Programmers can specify four types of *transmission policy rule*:

- *Method policy rules* are associated with methods and specify how all the arguments to those methods are marshalled. For example, a method policy rule might specify that all arguments must be passed by-reference when calling a particular method.

- *Return policy rules* are also associated with methods and control how the return values are marshalled. For example, a return policy rule might specify that the return value from a particular method must be passed by-value. The method policy rule and return policy rule associated with a single method are independent of each other.
- *Argument policy rules* are associated with individual method arguments to indicate how they are marshalled to allow programmers fine-grained control over transmission policy. For example, an argument policy rule might specify that the second argument must be passed by-migrate when calling a particular method.
- *Class policy rules* are associated with classes rather than methods and indicate how instances of particular classes are marshalled. For example, a class policy rule might specify that all instances of a particular class must be passed by-value.

Policy rules apply to all objects that are marshalled in the local address-space for transmission to remote address-spaces, namely arguments passed when calling remote methods, return values passed when local methods are called by remote clients, and any objects in the closures of these arguments/return values. When an RRT instance is marshalling objects, it queries the transmission policy framework to determine whether to pass the objects by-reference, by-value or by-migrate.

Method policy rules, return policy rules and argument policy rules are specified with depth constraints that indicates how deep into the closure of the arguments/return values the rules apply. The depth of an object in an argument/return value's closure is based on the shortest path from that argument/return value to the object. A depth of zero indicates infinite depth.

5.6.1.2 Caching Policy Rules

There are two types of *caching policy rule* available to programmers;

- *Method caching rules* are associated with methods and specify that these methods should be cached by remote references. Calls on cached methods are executed locally with respect to the reference holder rather than propagated across the network.

- *Field caching rules* are associated with fields and specify that these fields and their accessor methods should be cached by remote references. Any calls to the accessor methods are executed locally with respect to the reference holder and modify only the local cached copies of the fields.

Using these rules, individual objects can be marshalled not just by-value or by-reference but as a combination of both. The spectrum ranges from standard remote references at one end to remote references with all fields and methods cached, which are effectively by-value copies, at the other. Caching state in remote references allows them to remain partially usable even when connectivity is lost. The RRT does not provide automatic coherency control over cached fields, meaning caching is particularly appropriate for use with fields that are immutable within the context of the current application.

The transmission policy framework meets the 15th and 16th requirements, that:

15: *It must be possible to control parameter-passing semantics dynamically.*

16: *Remote references must be capable of caching fields and methods locally.*

5.6.2 Transmission Policy Manager

There is a *transmission policy manager* in each address-space in the distributed system, through which programmers control transmission policy. Figure 5.31 shows the *ITransmissionPolicyManager* interface, which provides methods to control transmission policy rules and caching policy rules. Only the *set()* methods that allow programmers to define rules are shown. The interface also provides a series of *get()* methods that allow the currently active rules to be accessed but these methods are omitted for brevity. This interface is implemented by each transmission policy manager instance. Programmers can obtain a reference to the local transmission policy manager using the *getTransmissionPolicyManager()* method in the *IRafdaRunTime* interface. The *PassingMechanism* class, which defines constants to enumerate the parameter-passing mechanisms supported by the RRT, namely pass-by-reference, pass-by-value and pass-by-migrate, is also shown in Figure 5.31.

```
public interface ITransmissionPolicyManager {
      /* Setting transmission policies */
      void setMethodPolicy(Method methodIdentifier,
            PassingMechanism passingMechanism,
            int depth,
            int priority);
      void setReturnPolicy(Method methodIdentifier,
            PassingMechanism passingMechanism,
            int depth,
            int priority);
      void setArgumentPolicy(Method methodIdentifier,
            int argumentNumber,
            PassingMechanism passingMechanism,
            int depth,
            int priority);
      void setClassPolicy(Class classIdentifier,
            PassingMechanism passingMechanism,
            int priority);

      /* Caching */
      void setFieldToCache(Field fieldIdentifier,
            Method getMethodIdentifier,
            Method setMethodIdentifier);
      void setMethodToCache(Method methodIdentifier);

      /* File-based policies */
      void getPolicyFromFile(File policyFile,
            boolean replaceCurrentPolicy);
      void writeCurrentPolicyToFile(File policyFile);
}

public class PassingMechanism {
    public static final PassingMechanism BY_REFERENCE = …;
    public static final PassingMechanism BY_VALUE = …;
    public static final PassingMechanism BY_MIGRATE = …;
}
```

Figure 5.31: The *ITransmissionPolicyManager* interface and *PassingMechanism* class.

5.6.2.1 Setting Transmission Policy Rules

The *setMethodPolicy()* method is used to control how arguments passed to particular methods should be marshalled using method policy rules. Programmers identify a method using an instance of the *java.lang.reflect.Method* class and indicate an associated parameter-passing mechanism, a depth indicating how far into the closures of each argument this rule applies, and a priority. Priority plays a role in resolving conflicts between policy rules that specify contradictory parameter-passing policies. Conflict resolution is discussed later in Section 5.6.6.

The *setReturnPolicy()* method is used to specify return policy rules that control how the return values of particular methods should be marshalled. It takes the same set of arguments as *setMethodPolicy()*.

The *setArgumentPolicy()* method is used to specify argument policy rules that control how one particular argument of a method should be marshalled. It takes the same arguments as *setMethodPolicy()* plus an extra parameter which identifies the particular argument to which this policy applies.

The *setClassPolicy()* method is used to specify class policy rules that control how instances of particular classes are marshalled when passed across the network as arguments or return values.

5.6.2.2 Setting Caching Policy Rules

The *setFieldToCache()* method is used to indicate that a particular field in a particular class should be cached in remote references to instances of that class. Programmers identify fields using instances of the *java.lang.reflect.Field* class. In addition they must identify the accessor methods of the specified field, which are also cached. Calls to these accessor methods are not propagated across the network but instead access the locally stored copy of the field.

The *setMethodToCache()* method is used to indicate that a particular method in a particular class should be cached in remote references to instances of that class. Any calls to cached methods will be performed locally with respect to the caller.

These rules allow a hybridization of pass-by-reference and pass-by-value that permits objects to be passed partially by-reference and partially by-value. Immutable state within objects can be cached in remote references to reduce the need for remote method call.

5.6.3 File-Based Policy Rules

Policy rules can be defined directly in application source code using the *ITransmissionPolicyManager* interface. This provides a mechanism allowing appropriate default policy to be specified directly in classes but does not provide complete separation of transmission policy from the functional application. To address this need, the transmission policy framework allows policies to be stored in a policy file, which describes one or more policy rules, meaning:

- Transmission policy can be completely separated from source code.
- Transmission policies can be reused in multiple applications.

The *getPolicyFromFile()* method reads rules from the specified file. The second argument indicates whether the currently active rules are deleted before the policy rules defined in the file become active. The *writeCurrentPolicyToFile()* method writes the complete set of currently active policy rules back to file.

Policy rule files contain transmission and caching policy. Programmers can modify the policy files directly in order to change the policy associated with applications without the need to recompile application source. A simple example policy file in shown in the following section. Appendix B shows a fuller example and defines the XML schema of the policy files.

5.6.4 Using Transmission Policy in JChord

In the JChord implementation, keys are immutable and small. Therefore it is desirable that keys are always passed by-value in order that the application need not perform remote method calls to access key state. Figure 5.32 shows a class policy rule of priority 0 associated with class *Key* that defines this policy. This rule could be defined when the *Key* class is loaded, by specifying it in initialization code.

```
RRT.get().getTransmissionPolicyManager().setClassPolicy(Key.class,
      PassingMechanism.BY_VALUE,
      0);
```

Figure 5.32: Setting pass-by-value transmission policy for keys.

It is also desirable that keys are cached in remote references to instances of *JChordNode*, since they are immutable. The *Chord* class (Figure 5.5) defines a *key* field along with two accessor methods, *getKey()* and *setKey()*. The *printKeyInfo()* method is

also cached as it makes use only of keys. Figure 5.33 shows a caching policy rule that caches the *key* field in instances of any class that extends the *Chord* class, and a method caching policy rule that caches the *printKeyInfo()* method.

```
/* Caching the key field */
Method getKeyMethod = Chord.class.getDeclaredMethod(
      "getKey",
      null);
Method setKeyMethod = Chord.class.getDeclaredMethod(
      "setKey",
      new Class[] { Key.class });
Field keyField = Chord.class.getDeclaredField("key");
RRT.get().getTransmissionPolicyManager().setFieldToCache(
      keyField,
      getKeyMethod,
      setKeyMethod);
/* Caching the printKeyInfo method */
Method printKeyInfoMethod = Chord.class.getDeclaredMethod(
            "printKeyInfo", null);
RRT.get().getTransmissionPolicyManager().setMethodToCache(
            printKeyInfoMethod);
```

Figure 5.33: Setting the caching policy for keys.

It would be possible to create a version of the *Key* class that was always passed by-value using traditional middleware. However, this key implementation could not be reused in a different context in which it was mutable. Another by-reference version of the key would be necessary for such applications.

This following example illustrates the use of transmission policy rules and caching rules in the context of the Data Store application implemented as part of the JChord case study. Chapter 2 described the behaviour of the Data Store service and explained that each Data Store object presented the *IDataStoreInternal* interface to remote clients. This is implemented by exposing a JChord node and a Data Store object in each RRT instance in the distributed system. Each Data Store object is exposed using the *IDataStoreInternal* interface shown in Figure 5.34 as remote type with the service name *"DataStore"*.

```
public interface IDataStoreInternal {
      public void put(Key key, Object data);
      public Object get(Key key);
}
```

Figure 5.34: The *IDataStoreInternal* interface exposed by the Data Store service.

When an object is stored in the Data Store, a key is generated and associated with the object. This key has a dual role:

- It identifies the address-space in which the object is stored. A JChord lookup of the key maps to the single live JChord node that is collocated with the Data Store instance holding the object.
- It identifies the object within the Data Store instance in that address-space.

The Data Store service is accessed via a point-of-presence (POP) that is implemented by the *DataStorePOP* class shown in Figure 5.35. This class implements the *IDataStorePOP* interface shown previously in Figure 2.14 and takes advantage of the transmission policy framework.

The *store()* method is used to insert objects into the store. Objects can be passed to the store by-reference or by-value. When the *store()* method is called, an object to store and a Boolean indicating whether the object will be passed by-reference or by-value are supplied. Every Data Store object is collocated with a JChord node. The method begins by generating a key for the application object then performing a JChord lookup on the key. The application object will be stored by the Data Store object collocated with the looked up JChord node.

The *store()* method gets a reference to the RRT instance that exposes this JChord node using the *getExposingRRT()* method provided by the *IRafdaRunTime* interface. A remote reference to the Data Store object that is exposed by this RRT instance is then obtained.

The application object and generated key are passed as arguments to the *put()* method provided by that remote Data Store object, via the *IDataStoreInternal* interface. Using the transmission policy framework, an argument policy rule is created to control the parameter-passing semantics applied to the stored object.

```
public class DataStorePOP implements IDataStorePOP {
      private JChordNode localJChordNode = …;
      public synchronized Key store(Object objectToStore,
            boolean storeByRef) {
            /* Generate and lookup the key for the object to store */
            Key key = generateKeyForData(objectToStore);
            Chord node = localJChordNode.lookup(key);
            /* Get a remote reference to the Data Store object
             * collocated with that JChord node */
            IRafdaRunTimeRemote remoteRRT = RRT.get().
                  getExposingRRT(node);
            IDataStoreInternal store = (IDataStoreInternal) remoteRRT
                        .getRemoteReference("DataStore");
            /* Decide the parameter passing semantics */
            PassingMechanism passingMechanism = null;
            if (storeByRef) {
                  passingMechanism = PassingMechanism.BY_REFERENCE;
            } else {
                  passingMechanism = PassingMechanism.BY_VALUE;
            }
            /* Set an argument policy controlling how the object will
             * be passed to the Data Store object's put() method */
            Method putMethod = null;
            try {
                  putMethod = IDataStoreInternal.class.
                        getDeclaredMethod( "put",
                              new Class[] { Key.class, Object.class });
            } catch (Exception e) {e.printStackTrace();}
            IRafdaRunTime localRRT = RRT.get();
            localRRT.getTransmissionPolicyManager().setArgumentPolicy(
                  putMethod, 1, passingMechanism, 0, 0);
            /* Add the data to the store */
            store.put(key, objectToStore);
            return key;
      }
      public Object retrieve(Key key) {…}
      public Key generateKeyForData(Object data) {…}
}
```

Figure 5.35: The *DataStorePOP* class, which dynamically changes transmission policy.

Using conventional middleware, in which the parameter-passing semantics applied to objects are decided statically, it would not be possible to implement the Data Store in this flexible manner. The programmer would be forced to apply the same passing mechanism to all instances of a single class. Using the RRT, class reuse is promoted and code to handle non-functional considerations does not pervade application classes.

The transmission policy framework promotes the reuse of application classes in different contexts by allowing the creation of a single implementation to which widely varying parameter-passing mechanisms can be applied. Programmers can create classes without concern for the application context in which the classes will be deployed, provided greater separation of concerns than is possible using traditional middleware systems.

An example of a transmission policy file is shown in Figure 5.36. This file defines the class policy shown in Figure 5.32 and the caching policies shown in Figure 5.33. A more complete example which includes all types of rule is shown in Appendix B. To allow the framework to identify methods uniquely, the name, class and argument types of each method are specified. This enables the framework to differentiate between overloaded methods.

```
<?xml version="1.0" encoding="UTF-8" ?>
<transmissionPolicy>
  <classPolicy>
    <className>Key</className>
    <paramPassingMechanism>byvalue</paramPassingMechanism>
    <priority>0</priority>
  </classPolicy>
  <cachedField>
    <className>Chord</className>
    <fieldName>key</fieldName>
  </cachedField>
  <cachedMethod>
    <method>
      <className>Chord</className>
      <methodName>getKey</methodName>
    </method>
  </cachedMethod>
  <cachedMethod>
    <method>
      <className>Chord</className>
      <methodName>setKey</methodName>
      <argumentType>Key</argumentType>
    </method>
  </cachedMethod>
  <cachedMethod>
    <method>
      <className>Chord</className>
      <methodName>printKeyInfo</methodName>
    </method>
  </cachedMethod>
</transmissionPolicy>
```

Figure 5.36: An example transmission policy file.

5.6.5 Automatic Exposure

If an RRT instance needs to pass a remote reference to an object across the network but that object has not been exposed, then that RRT instance performs automatic exposure. By default, the RRT exposes objects using their own classes as remote types, with automatically generated service names. However, the concept of remote types stems

from the fact that it is not always desirable to expose all methods of a given object to remote access. Programmers can therefore associate particular remote types with particular application classes, meaning that the RRT will employ the specified remote types when automatically exposing any instances of the specified application classes. This association between application class and remote type is created using the *associateClassWithRemoteType()* method shown in Figure 5.37, which is provided by the *IRafdaRunTime* interface.

```
void associateClassWithRemoteType(
       Class applicationClass,
       Class remoteType);
```

Figure 5.37: The *associateClassWithRemoteType()* method.

5.6.6 Resolving Policy Rule Contention

Contention can occur when two or more rules contradict each other. Consider the policy rules shown in Figure 5.38, which indicate that the parameters to the *addNode()* method are passed by-value to a depth of 0 and that instances of *JChordNode* are passed by-reference. These rules contradict each other when an instance of *JChordNode* is passed as an argument to the *addNode()* method. The transmission policy framework requires meta-rules to determine which transmission policy rules to adopt.

```
Method addNodeMethod = JChordNode.class.getDeclaredMethod(
       "addNode",
       new Class[] { Chord.class });
RRT.get().getTransmissionPolicyManager().setMethodPolicy(
       addNodeMethod,
       PassingMechanism.BY_VALUE,
       0,
       0);
RRT.get().getTransmissionPolicyManager().setClassPolicy(
       JChordNode.class,
       PassingMechanism.BY_ REFERENCE,
       0);
```

Figure 5.38: Policy rule contention.

When programmers specify transmission policy rules they must associate priorities with the rules. When contention occurs, the highest priority rule that applies is chosen over all others. As a consequence, the transmission policy framework does not permit the following:

- Two method policy rules of the same priority to be associated with the same method.
- Two return policy rules of the same priority to be associated with the same method.
- Two argument policy rules of the same priority to be associated with the same method and argument.
- Two class policy rules of the same priority to be associated with the same class.

If a programmer specifies a rule of the same type and priority as an existing rule, the existing rule is discarded and the new rule adopted. Consequently, when an object is marshalled during a call to a particular method, there cannot be two applicable rules of the same type with the same priority. There can however be two conflicting rules of the same priority but different types.

In Figure 5.38, the two conflicting rules are of different types and were specified with the same priority of 0. An order of precedence is imposed on policy rules based on their types to allow the framework to choose between rules of different types with the same priority. Argument rules are a specialization of method rules, so are defined to have a higher precedence. Contention cannot exist between return policy rules and method/argument rules as the former apply policies to return values and the latter to arguments. Return policy rules are defined (arbitrarily) to be of lower precedence than method/argument rules. Class policy rules are defined (again arbitrarily) as having the lowest precedence.

The overall ordering of rules is summarized in Figure 5.39. The rules of higher priority and precedence will be chosen and followed before the rules of lower priority and precedence. This approach to rule priority and precedence ensures that the temporal order in which rules are specified is not relevant, which is important given that policy rules may be defined dynamically in arbitrary application classes at any time during execution. If no policy rules are associated with the object to be marshalled then the default by-reference policy is chosen.

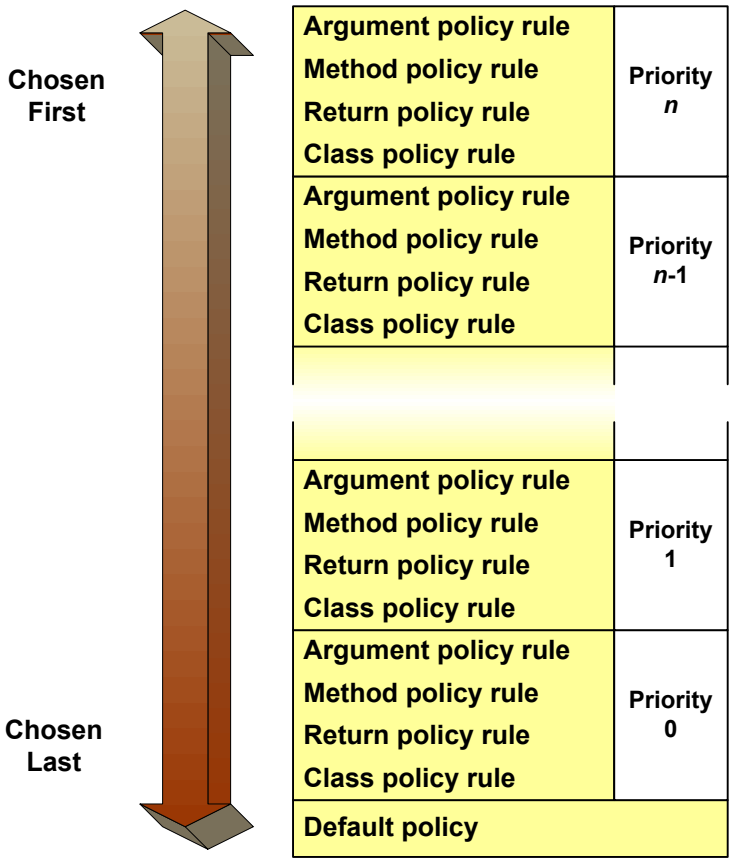


Figure 5.39: Policy rules ordered by dominance.

5.6.7 Summary of the Transmission Policy Framework

The transmission policy framework allows dynamic control over the parameter-passing mechanisms employed when calling remote methods. It separates the specification of the parameter-passing semantics applied to objects from the creation and implementation of their classes. Policies may be defined dynamically on a per-class, per-method, per-return-value or per-argument basis. Application semantics are not driven by decisions made statically. Programmers can also specify caching policies that control which fields and methods are cached in remote references, allowing hybridization of pass-by-value and pass-by-reference.

Since transmission policies can be associated with classes on a per-application basis, a greater degree of class reuse is possible. The most appropriate passing mechanisms for the circumstances can be applied to instances of arbitrary classes. Classes can be reused in distributed and non-distributed contexts as policy is controlled independently of source code. A distributed application can be optimized using

hybridized remote references that cache particular fields and methods in order to take advantage of the programmers' knowledge of the distributed nature of the application.

5.7 Controlling Distribution Policy

The RRT provides mechanisms through which remote instantiation and object migration can be performed. These operations can be invoked directly by programmers, who specify the target address-spaces in which objects are instantiated or to which migration occurs. In addition to these mechanisms, the RRT provides a *distribution policy framework* that allows programmers to specify distribution policies that determine how application objects are partitioned among the address-spaces in the distributed system automatically at run-time.

Control over distribution policy is useful as it introduces location transparency into applications. Applications perform all object instantiation and migration operations through the framework. The framework determines in which address-spaces in the distributed system the objects are created or to which address-spaces the objects are migrated using the active distribution policy. A single application can be distributed in multiple different ways without changes to its underlying source code. By allowing programmers to defer distribution decisions to a separate policy framework, the RRT promotes the separation of application logic from distribution

Changes to the policy that controls remote instantiation and migration allow programmers to re-configure application partitions to adapt to changes in the distributed system (e.g. to take advantage of extra machines added to a distributed system) or in the application itself (e.g. to collocate an object with its current working set). Applications can evolve dynamically and fine-tune their distributions to maximize performance.

The last four requirements defined in the previous chapter state:

17: *It must be possible to create policies to control the placement of objects when instantiation and migration operations are performed.*

18: *It must be possible to define object placement policies independently of application logic.*

19: *It must be possible to define arbitrarily complex object placement policies in terms of reusable policy components.*

20: *It must be possible for policies to use application context to aid policy decisions.*

The distribution policy framework meets these requirements as follows:

- It permits the separation of application logic from application distribution.
- It allows the creation of reusable, modular object placement policies.
- It permits arbitrarily complex policies.
- It permits remote instantiation and migration to be performed consistently with the object placement policies that apply in remote address-spaces.

Distribution policies are defined at object granularity, rather than a coarse-grained component level. However, the latter can be achieved by associating distribution policies with the objects at the edges of the components.

Programmers define distribution policies that, when evaluated, return references to the RRT instances in the distributed system that meet that particular policy, for example, a "round robin" policy initialized with a set of RRT instances returns references to them in a round robin order while a "machines with no more than 50% CPU usage" policy returns references to machines that meet this criterion. Each application class is associated with exactly two policies, one to control remote instantiation of the class and one to control the migration of instances of the class.

5.7.1 Architectural Overview

The distribution policy framework is made up of five conceptual parts:

- *Policy objects* are stateful objects that capture particular distinct distribution policies, for example, "round robin" or "machines with less than 50% CPU load". Policy objects are associated with application classes, but may define policies at finer-than-class granularity based on programmer supplied information describing the application context of the instantiation or migration operation. A single policy object can define the distribution policy for multiple application classes, allowing policy to be defined at greater-than-class granularity also.
- *Factories* perform all policy-based object instantiation. They can instantiate objects locally or remotely in conjunction with factories in remote address-spaces. Factories query policies to determine in which address-spaces objects should be instantiated.

- *Migration controllers* perform all policy-based object migration. They query policies to determine to which address-spaces objects should migrate.
- *Distribution policy managers* allow applications to obtain references to factories or migration controllers when they need to perform remote instantiation or migration.
- *Feedback* on policy decisions is provided to policy objects by the policy objects in other address-spaces with which they interact. This allows policy objects to adapt their behaviour based on responses received from their remote counterparts.

Figure 5.40 shows the structure of the distribution policy framework and the relationships between these components. There is a single distribution policy manager in each address-space, through which applications can access factories and migration controllers. Each application class is associated with a single factory and migration controller, though a single factory/migration controller may be associated with multiple application classes. Each application class is associated with two policy objects; one that controls the policy applied during remote instantiation (used by the class's factory) and one that controls the policy applied during migration (used by the class's migration controller).

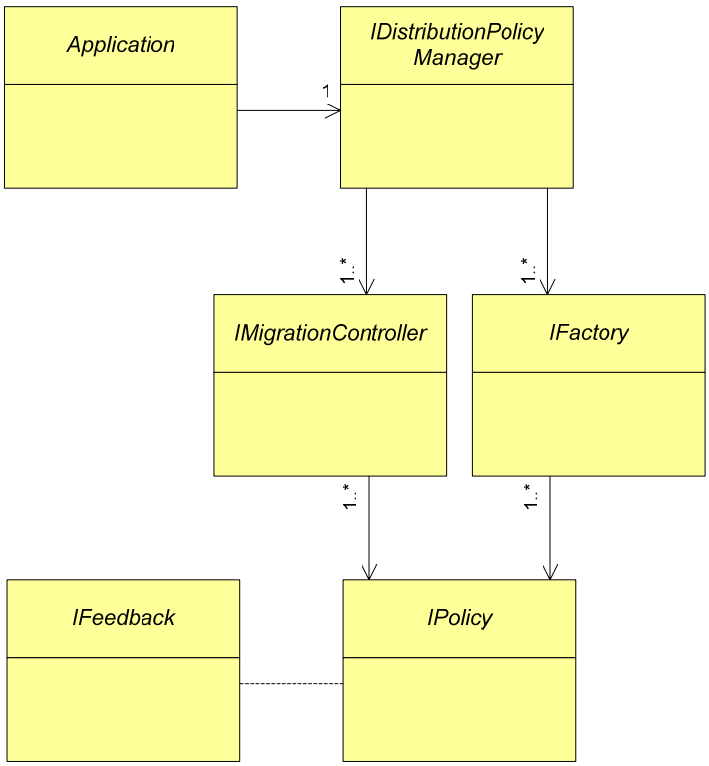


Figure 5.40: The structure of the distribution policy framework.

5.7.2 Evaluating Distribution Policies

In order to instantiate or migrate objects according to the active distribution policy, programmers use the distribution policy manager, which is accessible via the *getDistributionPolicyManager()* method provided by the *IRafdaRunTime* interface. This method returns a reference to the local distribution policy manager, which meets the *IDistributionPolicyManager* interface shown in Figure 5.41.

```
public interface IDistributionPolicyManager {
      /* Factories */
      IFactory getFactory(Class applicationClass);
      void setFactoryPolicy(Class applicationClass, IPolicy policy);
      void associateFactoryWithClass(
            Class applicationClass,
            Class factoryClass);

      /* Migration Controllers */
      IMigrationController getMigrationController(Object object);
      void setMigrationControllerPolicy(Class applicationClass,
            IPolicy policy);
      void associateMigrationControllerWithClass(
            Class applicationClass,
            Class migContollerClass);
}
```

Figure 5.41: The *IDistributionPolicyManager* interface.

The role of each of these methods is briefly described:

- The *getFactory()* method returns a reference to a factory that can create instances of the specified class.
- The *setFactoryPolicy()* method associates a particular policy object with an application class to control the policy applied when this class is instantiated using a factory.
- The *associateFactoryWithClass()* method allows the programmer to override the default factory implementation with a customized factory implementation.
- The *getMigrationController()* method returns a reference to a migration controller that can be used to migrate the specified object.

- The *setMigrationControllerPolicy()* method associates a particular policy object with an application class to control the migration policy applied to instances of this class.
- The *associateMigrationControllerWithClass()* method allows the programmer to override the default migration controller implementation with a customized migration controller implementation.

5.7.3 Defining Distribution Policies

Policy objects are instances of *policy classes* that implement the *IPolicy* interface shown in Figure 5.42. The *Feedback* class, through which policy objects exchange feedback, is also shown.

```
public interface IPolicy {
	IRafdaRunTimeRemote getRRT(Object context);
	Feedback evaluatePolicy(Object context);
	void handleFeedback(Feedback feedback);

}
public class Feedback {
	private boolean positive = false;
	private Object instantiatedObject = null;

	public Feedback(boolean positive) {
		this.positive = positive;
	}
	public Feedback(Object instantiatedObject, boolean positive) {
		this.instantiatedObject = instantiatedObject;
		this.positive = positive;
	}
	public boolean isPositive() {return positive;}
	public Object getInstantiatedObject() {return instantiatedObject;}
}
```

Figure 5.42: The *IPolicy* interface and *Feedback* class.

The *IPolicy* interface provides the following methods:

- The *getRRT()* method, which makes the policy decisions. This method returns the RRT instance that best meets the captured policy at that moment. It takes an object that describes the application context in which the instantiation or

migration operation is performed. This argument can contain any arbitrary data that the programmer wishes to supply to the policy object. If context is to have any benefit, programmers must create policy objects that understand it. The distribution policy framework does not analyse the context directly; it propagates the context from the application to the policy objects.

- The *evaluatePolicy()* method returns feedback that is used to determine whether the local address-space is a suitable choice for an instantiation or migration. This method is called by factories and migration controllers before they perform operations on behalf of remote clients.
- The *handleFeedback()* method is used to pass feedback received from remote policy evaluation back into the local policy object.

The *Feedback* class provides the following two methods:

- The *isPositive()* method is used to indicate whether the returned feedback is positive or negative.
- The *getInstantiatedObject()* method is used by factories to access newly instantiated objects, which are returned from remote factories as part of the feedback.

Programmers can associate two policy objects with each application class, one to determine placement during instantiation and one to determine migration policy, using the *setFactoryPolicy()* and *setMigrationControllerPolicy()* methods provided by the *IDistributionPolicyManager* interface. If no policy objects are associated with an application class, then instances of this class will always be instantiated locally by factories and will never be migrated.

The *setFactoryPolicy()* method is used to indicate the policy object that must be evaluated by factories that create instances of the specified application class. The *setMigrationControllerPolicy()* method is used to indicate the policy object that must be evaluated by migration controllers that migrate instances of the specified class.

A simple “single RRT policy” that always returns a reference to a particular RRT instance is shown in Figure 5.43. It ignores any application context provided by programmers and returns positive feedback. When queried, this policy will always return a reference to the RRT instance with which it was initialized.

```
public class SingleRRTPolicy implements IPolicy {
      private IRafdaRunTimeRemote remoteRRT = null;

      public SingleRRTPolicy (IRafdaRunTimeRemote remoteRRT) {
            this.remoteRRT = remoteRRT;
      }
      public IRafdaRunTimeRemote getRRT(Object context) {
            return remoteRRT;
      }
      public Feedback evaluatePolicy(Object context) {
            return new Feedback(true);
      }
      public void handleFeedback(Feedback feedback) {}
}
```

Figure 5.43: A single RRT policy.

This policy may be used to ensure that all instances of a particular class are instantiated on a single machine or are migrated to a single machine. For example, the code fragment in Figure 5.44 specifies that all instances of *JChordNode* created by the code running in this RRT instance should be instantiated on the RRT instance bound to port 5001 of machine *"host.rafda.org"* by associating an instance of the "single RRT policy" class with the *JChordNode* factory.

```
IDistributionPolicyManager dpm = RRT.get().
      getDistributionPolicyManager();
SingleRRTPolicy singleRRTpolicy = new SingleRRTPolicy(
      RRT.getRemote(new InetSocketAddress("host.rafda.org", 5001)));
dpm.setFactoryPolicy(JChordNode.class, singleRRTpolicy);
```

Figure 5.44: Using the single RRT policy.

Figure 5.45 shows a simple round robin policy. Again, this policy does not make use of application context but exhibits more complex behaviour than the single RRT policy. It is initialized with a set of RRT instances (perhaps obtained by querying another policy) and returns references to these RRT instances in round robin order. This policy is used to distribute objects of a particular class evenly among a number of RRT instances.

```
public class RoundRobinPolicy implements IPolicy {
	private IRafdaRunTimeRemote[] remoteRRTs = null;
	private int current = -1;

	public RoundRobinPolicy(IRafdaRunTimeRemote[] remoteRRTs) {
		this.remoteRRTs = remoteRRTs;
	}
	public IRafdaRunTimeRemote getRRT(Object context) {
		current = (++current) % remoteRRTs.length;
		return remoteRRTs[current];
	}
	public Feedback evaluatePolicy(Object context) {
		return new Feedback(true);
	}
	public void handleFeedback(Feedback feedback) {
	}
}
```

Figure 5.45: A round robin policy class.

Figure 5.46 shows how a round robin policy can be associated with both the *JChordNode* factory and the *DataStore* factory. It is notable that both factories are associated with the same instance of the round robin policy class. The policy object is not concerned whether it is accessed by the *JChordNode* factory or *DataStore* factory; it simply returns the specified RRT instances in round robin order. For example, if two instances of *JChordNode* are created, then two instances of *DataStore*, the former will be created on *"host1"* and *"host2"* while the latter will be created on *"host3"* and *"host4"*. This many-to-one relationship between factories/migration controllers and policy objects helps avoid a preponderance of policy objects in the system and allows a single policy to be applied to instances of multiple classes.

```
IDistributionPolicyManager dpm = RRT.get().
      getDistributionPolicyManager();
IRafdaRunTimeRemote remoteRRTs[] = new IRafdaRunTimeRemote[] {
      RRT.getRemote(new InetSocketAddress("host1.rafda.org", 5001)),
      RRT.getRemote(new InetSocketAddress("host2.rafda.org", 5001)),
      RRT.getRemote(new InetSocketAddress("host3.rafda.org", 5001)),
      RRT.getRemote(new InetSocketAddress("host4.rafda.org", 5001)) };
RoundRobinPolicy roundRobinPolicy = new RoundRobinPolicy(remoteRRTs);
dpm.setFactoryPolicy(JChordNode.class, roundRobinPolicy);
dpm.setFactoryPolicy(DataStore.class, roundRobinPolicy);
```

Figure 5.46: Using the round robin policy.

Policies of greater complexity could be created in several ways:

- Policy objects could make use of profiling tools external to the RRT to base policy decisions on system or application metrics.
- Multiple policies could be composed together. For example, the round robin policy could be initialized using the output of another policy object.
- Policy objects that aggregate the output of other policy objects could be created, such as policies which determine the union or intersection of several policy objects, effectively acting as filters over this output.

Context can be employed to aid the policy decisions. Context indicates any additional information that the programmer wishes to supply to aid policy decisions, for example, the identity of the method or class that is performing the given operation or meta-data associated with a class or object. This allows policies to differentiate between instances of the same application class, based on their application context.

Feedback allows policies to exchange arbitrary information. By default, feedback is positive or negative but programmers are free to extend the *Feedback* class to provide an arbitrarily rich explanation of a policy decision. Feedback is not used directly by the distribution framework; rather it is a vehicle for information interchange that can be employed by policy objects, much as context allows information to be passed from the application to the policy objects.

The *JChordLookupPolicy* class shown in Figure 5.47 illustrates the use of context and feedback. This policy expects the application to provide a JChord key as context and returns a reference to the RRT instance exposing the JChord node associated with this key. When the *evaluatePolicy()* method is called, this policy casts the context into a key

and confirms whether the JChord node associated with the supplied key is the local JChord node. If so, positive feedback is returned otherwise negative feedback is returned.

The *handleFeedback()* method expects always to receive positive feedback, since a single key should always map to the same JChord node unless the underlying ring has changed. If negative feedback is received, the policy performs some ring maintenance to confirm the local JChord node's ring state information is up-to-date. The code shown in this class omits error-checking and assumes that the context object is the key.

```
public class JChordLookupPolicy implements IPolicy {
	private JChordNode localJChordNode = …;

	public IRafdaRunTimeRemote getRRT(Object context) {
		/* Assume context is the key associated with the object */
		Key key = (Key) context;
		Chord nodeAssociatedWithKey = localJChordNode.lookup(key);
		return RRT.get().getExposingRRT(nodeAssociatedWithKey);
	}
	public Feedback evaluatePolicy(Object context) {
		/* Assume context is the key associated with the object */
		Key key = (Key) context;
		Chord nodeAssociatedWithKey = localJChordNode.lookup(key);
		if (nodeAssociatedWithKey.equals(localJChordNode)) {
			return new Feedback(true);
		} else {
			return new Feedback(false);
		}
	}
	public void handleFeedback(Feedback feedback) {
		if (!feedback.isPositive()) {
			/*
			 * Indicates that the underlying ring has changed.
			 * Perform some checks to ensure the local ring state
			 * information is up-to-date
			 */
		}
	}
}
```

Figure 5.47: The *JChordLookupPolicy* class.

5.7.4 Factories

A default factory implementation is provided with the distribution policy framework though programmers can create custom implementations of factories in order to provide arbitrarily complex factory behaviour. This allows programmers control over the manner in which factories evaluate policies. Factories must trade off the cost of evaluating distribution policy against the benefits. For instance, it may be advantageous to spend time evaluating policy carefully for long-lived objects that are instantiated rarely but not for objects that are instantiated often.

Factories must implement the *IFactory* interface shown in Figure 5.48. No semantic restrictions are placed on programmers though the *instantiateObject()* method is intended for use by the application to instantiate objects. The method takes five arguments, namely the class of the object to create, an array of constructor arguments, application context information, a remote type and a service name. The context argument holds additional information that the policy can use to aid the placement decision.

The *instantiateObjectForRemoteFactory()* method is intended for use by remote factories to perform instantiation in the local address-space. It takes the same arguments as the previous method but returns a feedback object, rather than a remote reference.

```
public interface IFactory {
      Object instantiateObject(
            Class applicationClass,
            Object[] constructorArguments,
            Object context,
            Class remoteType,
            String serviceName);
      Feedback instantiateObjectForRemoteFactory(
            Class applicationClass,
            Object[] constructorArguments,
            Object context,
            Class remoteType,
            String serviceName);
}
```

Figure 5.48: The *IFactory* interface.

Programmers must implement both of these methods to create a factory implementation but are free to alter factory behaviour as required. Custom factories are

associated with application classes using the *associateFactoryWithClass()* method, which is provided by the *IDistributionPolicyManager* interface. The distribution policy framework instantiates the custom factories as required.

5.7.4.1 The Default Factory

The default factory is employed when the distribution policy framework needs to instantiate classes for which no customized factory implementations exist. To create instances of a class, the default factory queries the policy associated with that class and instantiates the object in the RRT instance specified by the policy. Figure 5.49 shows a sequence diagram describing the flow of control during a particular remote object instantiation. The general case is discussed after this example. The components of the distribution policy framework are marked in bold.

The sequence diagram shows the following objects, divided between two RRT instances, which are marked by large dotted rectangles:

- The application object performing the instantiation.
- The **distribution policy managers** in the local and remote RRT instances.
- The **factories** in the local and remote RRT instances associated with the class to be instantiated.
- The **policy objects** in the local and remote RRT instances that are associated with the class to be instantiated.
- The created object.

The sequence diagram contains the following steps:

1. The application needs to instantiate a new object. It obtains a reference to a **factory** that can create instances of the required class from the local **distribution policy manager**.
2. The **distribution policy manager** returns a reference to a suitable **factory**.
3. The application instructs the **factory** to create an instance of a particular class.
4. The **factory** queries the **policy** associated with that class to determine in which RRT instance to instantiate the new object.
5. The **policy** determines that RRT instance 2 is the best choice.
6. The **factory** asks the **distribution policy manager** in RRT instance 2 for a reference to a **factory** that can create instances of the required class in that

address-space. A reference to a remote **distribution policy manager** can be obtained using the *IRafdaRunTimeRemote* interface.

7. The **distribution policy manager** returns a remote reference to a suitable remote **factory**.
8. The local **factory** asks the remote **factory** to instantiate the object.
9. The remote **factory** evaluates its **policy** to determine if it deems RRT instance 2 a suitable target in which to instantiate the object.
10. The **policy** returns positive **feedback** indicating that RRT instance 2 is a suitable target.
11. The remote **factory** instantiates the new object.
12. The remote **factory** returns **feedback**, which contains a remote reference to the new object.
13. The local **factory** passes the **feedback** into its **policy**.
14. The local **factory** returns the remote reference to the application.

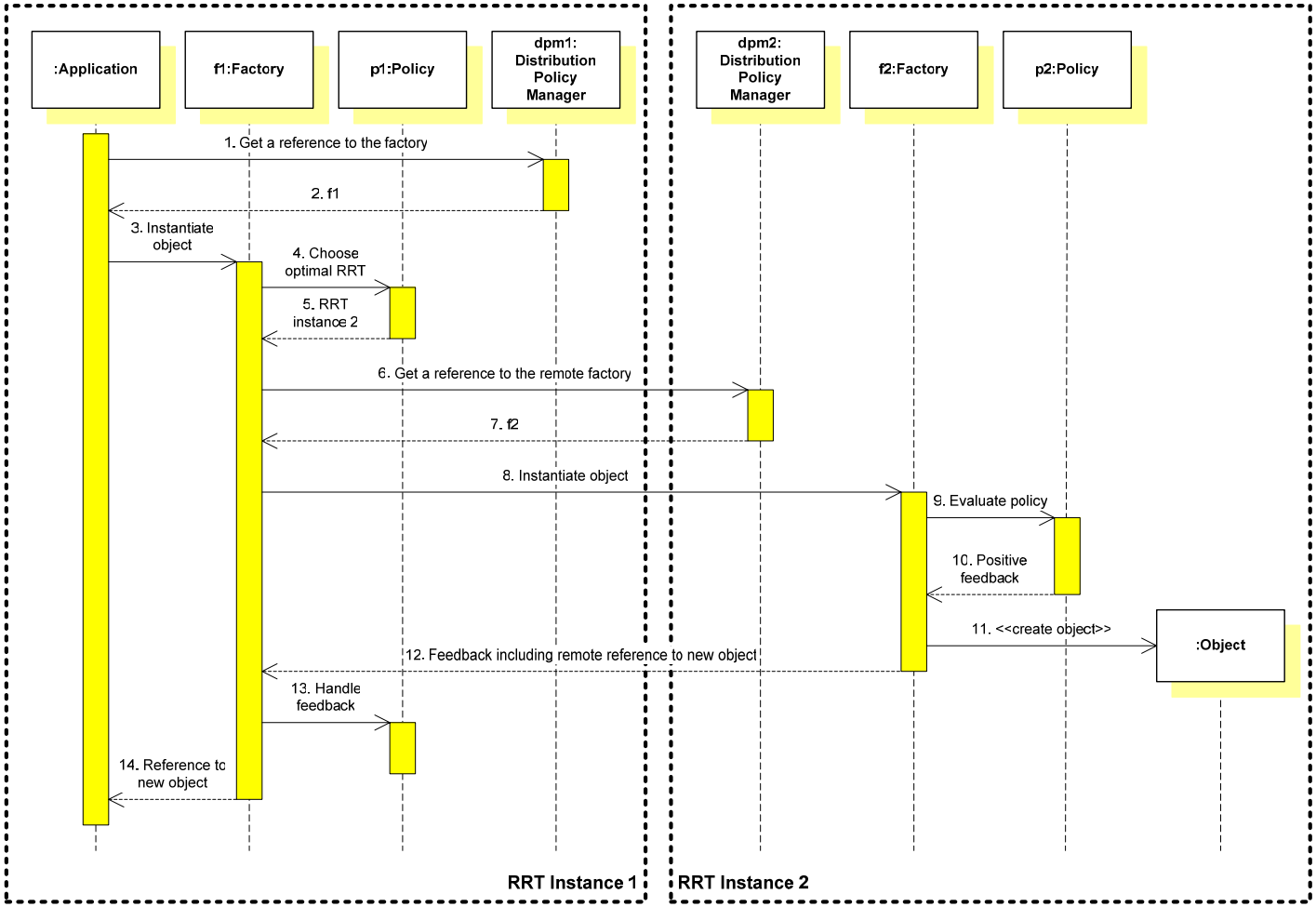


Figure 5.49: Sequence diagram showing a remote object instantiation.

The sequence diagram shows the flow of control during a particular remote instantiation. In other circumstances, the policy in the local RRT instance may determine that the object should be instantiated locally. Alternatively, the factory in the remote RRT instance may refuse to instantiate the object because its policy returns negative feedback. The following describes the flow of control during remote instantiation in general terms.

1. The programmer obtains a reference to a **factory** that can create instances of the required class via the **distribution policy manager**.
2. The programmer instructs the **factory** to instantiate the object.
3. The **factory** queries the **policy** to determine which RRT instance is the optimal choice in which to instantiate the object. The **policy** returns a reference to that RRT instance.
    3.1. If the **policy** has specified that the local RRT instance is the optimal choice, the **factory** instantiates the object immediately and returns a reference to the programmer.

    OR

    3.1. If the **policy** has specified that a remote RRT instance is the optimal choice, the **factory** attempts to instantiate the object in the remote RRT instance specified by the **policy**.
    3.2. The local **factory** uses the **distribution policy manager** in the remote RRT instance to obtain a reference to a remote **factory** that is capable of instantiating instances of the required class.
    3.3. The local **factory** instructs the remote **factory** to instantiate the object.
    3.4. The remote **factory** checks with its **policy** to determine whether the instantiation should be performed based on the **feedback** provided by its **policy**.
    3.5. The remote **factory** returns the **feedback** to the local **factory**. If the instantiation was successful, this **feedback** contains a remote reference to the newly created object.
    3.6. The local **factory** passes the **feedback** into its **policy**.
        3.6.1. If instantiation was successful, the remote reference to the new object included in the **feedback** is returned to the programmer.

        OR

3.6.1. If instantiation was not successful, the local **factory** queries its **policy** again to determine which other RRT instances the **policy** deems as suitable targets for instantiation.

3.6.2. The **factory** attempts remote instantiation, as described above, at each RRT instance in turn, until the object is instantiated successfully. If all possible RRT instances are tried without success, the **factory** instantiates the required object locally.

This approach shares the responsibility for choosing the target address-space among the policies in all participating address-spaces, though the local policy retains responsibility for ensuring that the operation completes. Remote policies have the power to veto the local policy to ensure that instantiation does not occur in their address-spaces.

5.7.5 Deploying a JChord Ring using the Framework

When remote instantiation was described earlier in Section 5.4.4, a ring of JChord nodes was deployed by instantiating and exposing nodes in remote address-spaces. The example in Figure 5.50 shows how the code to perform those instantiations (Figure 5.25) can be modified to deploy the nodes according to an active distribution policy, rather than explicitly. Initially, a round robin policy object is created and associated with instances of the *JChordNode* class. Association of a policy object with an application class can be performed at any point during the application initialization, not necessarily immediately before the distribution policy framework is employed, as in this example.

A reference to a factory that can create instances of *JChordNode* is obtained from the local distribution policy manager. The application performs four instantiation operations by calling the *instantiateObject()* method four times, specifying the *Chord* class as remote type and the service names *"Node0"*, *"Node1"*, etc. Aside from the node created first, each instance of *JChordNode* that is created is passed as an argument to the *addNode()* method, in order to connect it to the ring. Location transparency has been introduced into the JChord deployment application. By altering the distribution policy associated with the application, the application can be deployed in different ways.

```
public void deployRingAccordingToPolicy() {
      /* Create an array of references to available RRT instances */
      IRafdaRunTimeRemote remoteRRTs[] = new IRafdaRunTimeRemote[] {
                  RRT.getRemote(new InetSocketAddress(
                        "host1.rafda.org", 5001)),
                  RRT.getRemote(new InetSocketAddress(
                        "host2.rafda.org", 5001)),
                  RRT.getRemote(new InetSocketAddress(
                        "host3.rafda.org", 5001)),
                  RRT.getRemote(new InetSocketAddress(
                        "host4.rafda.org", 5001)) };
      /* Create the round robin policy object */
      RoundRobinPolicy roundRobinPolicy =
            new RoundRobinPolicy(remoteRRTs);
      /* Associate the round robin policy with the JChordNode class */
      IDistributionPolicyManager dpm = RRT.get().
            getDistributionPolicyManager();
      dpm.setFactoryPolicy(JChordNode.class, roundRobinPolicy);
      /* Get a reference to a suitable factory */
      IFactory nodeFactory = dpm.getFactory(JChordNode.class);
      Chord[] remoteNodes = new Chord[4];
      for (int x = 0; x < 4; x++) {
            /* Perform remote instantiation using the factory */
            remoteNodes[x] = (Chord) nodeFactory.instantiateObject(
                  JChordNode.class,
                  new Object[] { new Key() },
                  null,
                  Chord.class,
                  "Node"+x);
            /* Add each newly created node to the ring */
            if (x > 0) remoteNodes[x - 1].addNode(remoteNodes[x]);
      }
}
```

Figure 5.50: Deploying a JChord ring using the distribution policy framework.

5.7.6 Migration Controllers

Migration controllers decide when and to where migration will occur. Applications must poll migration controllers to evaluate migration policy, otherwise no

policy evaluation occurs and no migration occurs. Programmers decide on a per-object basis how often to poll the migration controller.

Migration controllers implement the *IMigrationController* interface shown in Figure 5.51. Although programmers are free to decide the semantics of these custom implementations, the *migrateObject()* method is intended for use by applications to evaluate migration policy and perform any required migration. The *isSuitableTargetAddressSpace()* is intended for use by remote migration controllers to evaluate the policy in the local address-space.

```
public interface IMigrationController {
	void migrateObject(
		Object object,
		Object context);
	Feedback isSuitableTargetAddressSpace(
		Object object,
		Object context);
}
```

Figure 5.51: The *IMigrationController* interface.

Programmers associate custom migration controller implementations with application classes using the *associateMigrationControllerWithClass()* method, which is provided by the *IDistributionPolicyManager* interface. Custom migration controllers are instantiated as required.

5.7.6.1 The Default Migration Controller

The default migration controller implementation is used to evaluate the migration policy for any classes that are not associated with customized migration controllers. Figure 5.52 shows a sequence diagram describing the flow of control during a particular migration operation. The general case is discussed after this example.

The sequence diagram shows the following objects, divided between two RRT instances, which are marked by large dotted rectangles:

- The application object performing the migration.
- The **distribution policy managers** in the local and remote RRT instances.
- The **migration controllers** in the local and remote RRT instances associated with the migratory object's class.

- The **policy objects** in the local and remote RRT instances that are associated with the migratory object's class.
- The migratory application object.

The sequence diagram contains the following steps:

1. The application polls the framework to determine whether an object should migrate. It asks the local **distribution policy manager** for a reference to a **migration controller** associated with this object.
2. The **distribution policy manager** returns a reference to a suitable **migration controller**.
3. The application asks the **migration controller** to evaluate the migration **policy** for the object and perform the migration if necessary.
4. The **migration controller** queries its **policy** to determine in which RRT instance the object should be located.
5. The **policy** determines that RRT instance 2 is the best choice.
6. The **migration controller** asks the **distribution policy manager** in RRT instance 2 for a reference to a **migration controller** in that address-space which can evaluate the **migration policy** associated with this particular object.
7. The **distribution policy manager** returns a remote reference to a suitable remote **migration controller**.
8. The local **migration controller** instructs the remote **migration controller** to evaluate whether RRT instance 2 is a suitable target for migration.
9. The remote **migration controller** evaluates its **policy**.
10. The **policy** returns positive **feedback** indicating that it deems RRT instance 2 a suitable target.
11. The remote **migration controller** returns the **feedback**.
12. The **migration controller** passes the **feedback** into its **policy**.
13. The local **migration controller** performs the **migration**.
14. The **migration controller** method returns.

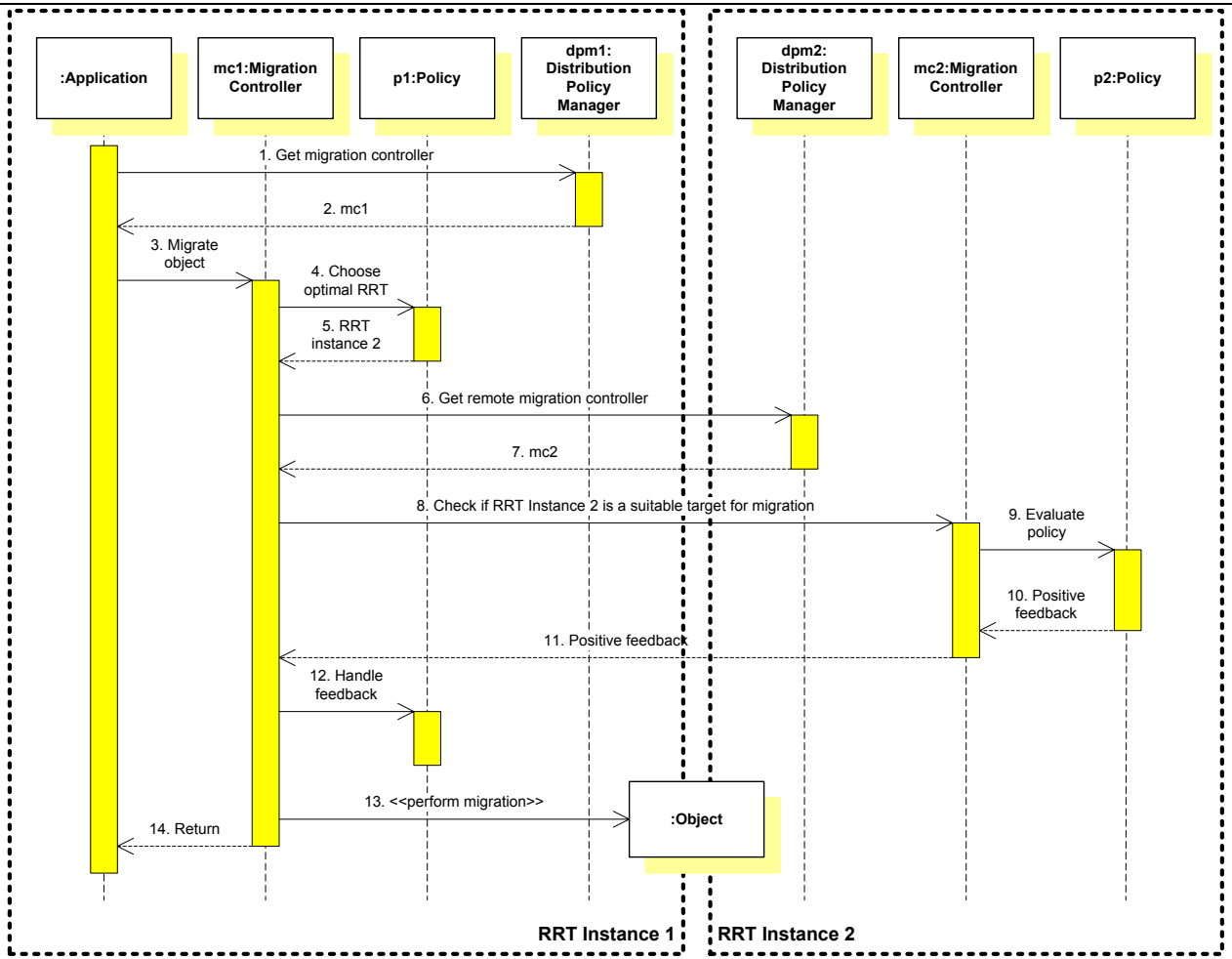


Figure 5.52: Sequence diagram showing a migration operation.

The sequence diagram shows the flow of control during a particular migration operation. Under different circumstances, the policy in the local RRT instance may determine that migration should not occur at all. Alternatively, the migration controller in the remote RRT instance may return negative feedback. The following describes the flow of control during the evaluation of migration policy in general terms.

1. The programmer obtains a reference to a **migration controller** associated with the migratory object via the **distribution policy manager**.
2. The programmer polls the **migration controller** to determine whether migration should occur.
3. The **migration controller** queries the **policy** to determine in which RRT instance the object should be located.
    4.1. If the **policy** has specified that the local RRT instance is the optimal choice, then no migration occurs.

    OR

    4.1. If the **policy** has specified a remote RRT instance as the optimal choice, the **migration controller** evaluates the policy in that RRT instance.

    4.2. The local **migration controller** uses the **distribution policy manager** in the remote RRT instance to obtain a reference to a remote **migration controller** associated with the migratory object.

    4.3. The local **migration controller** requests that the remote **migration controller** evaluate its policy.

    4.4. The remote **migration controller** evaluates its **policy** to determine whether the migration should be performed. The remote **migration controller** receives **feedback** from its **policy**.

    4.5. The remote **migration controller** returns the **feedback** to the local **migration controller**.

    4.6. The local **migration controller** passes the **feedback** into its **policy**.

        4.6.1. If the **feedback** is positive then the **migration controller** migrates the object to that RRT instance.

        OR

        4.6.1. If the **feedback** is negative then the **migration controller** queries its **policy** again to determine which other RRT instances the **policy** deems as suitable targets for migration.

4.6.2. The **migration** controller re-evaluates the remote **policy** at each RRT instance in turn, as described above, until a suitable target is found. If all possible RRT instances are tried without success, then no migration is performed.

Like the default factory implementation, the default migration controller offers a balanced approach to policy evaluation. Ultimate responsibility for determining whether migration takes place remains with a single policy but policies in remote address-spaces can participate in the decision.

5.7.7 Migrating Objects in JChord Automatically

Object migration was described in Section 5.4.5. The example in that section shows a database object migrating between address-spaces to free resources without loss of referential integrity. The following example modifies that original example by associating a round robin distribution policy with that migratory database object's class. Initially the database object exists in *host2.rafda.org* (Figure 5.28). A round robin policy (Figure 5.45) associated with the remaining machines is created as shown in Figure 5.53. This round robin policy is associated with the database (*db*) object's class.

```
/* Configure the distribution policy framework */
IRafdaRunTime localRRT = RRT.get();
IRafdaRunTimeRemote remoteRRTs[] = new IRafdaRunTimeRemote[] {
      RRT.getRemote(new InetSocketAddress(
            "host1.rafda.org", 5001)),
      RRT.getRemote(new InetSocketAddress(
            "host3.rafda.org", 5001)),
      RRT.getRemote(new InetSocketAddress(
            "host4.rafda.org", 5001)) };
RoundRobinPolicy roundRobinPolicy = new RoundRobinPolicy(remoteRRTs);
IDistributionPolicyManager dpm = localRRT.
      getDistributionPolicyManager();
dpm.setMigrationControllerPolicy(db.getClass(), roundRobinPolicy);
```

Figure 5.53: Associating a round robin policy with the database object.

When the database application detects resources running low in *host2.rafda.org* it polls the distribution policy framework to determine the RRT instance to which the object will migrate, by calling the *migrateObject()* method. The application does not explicitly

choose the RRT instance to which the database object migrates. Instead, it performs a call into the distribution policy framework as shown in Figure 5.54, thereby delegating the policy decision. The round robin policy will cause the migration to *host1.rafda.org* to occur, as shown in Figure 5.30.

```
IRafdaRunTime localRRT = RRT.get();
IDistributionPolicyManager dpm = localRRT.
      getDistributionPolicyManager();
IMigrationController dbmc = dpm.getMigrationController(db);
dbmc.migrateObject(db, null);
```

Figure 5.54: Evaluating the migration policy associated with the database object.

5.7.8 Summarizing the Distribution Policy Framework

The distribution policy framework controls the placement of objects in the distributed system when remote instantiation and migration operations are performed. Arbitrarily complex distribution policies can be created and associated with application classes. Policy objects can make use of the application context in which operations occur to aid policy decisions and can exchange feedback to allow cooperation between multiple address-spaces.

The framework uses factories to perform all policy-based remote instantiation and migration controllers to control object migration. Default implementations of factories and migration controllers are provided though programmers can create customized versions on a per-class basis to allow factory and migration controller behaviour to be defined on a per-application basis, allowing complete customization of the distribution policy framework. Programmers can use the default implementations for convenience but can obtain full control over the semantics of policy-based remote instantiation and migration if required.

The distribution policy framework separates application logic from distribution by introducing location transparency into remote instantiation and migration. Unlike existing approaches to the specification of distribution policy, the framework provides a flexible and expressive approach to defining application distribution dynamically and does not make assumptions about the granularity at which policies will be applied. Further, no limitations are placed on the kinds of distribution policy that can be specified.

5.8 Configuring the RRT

The *IRafdaRunTimeConfig* interface shown in Figure 5.55 allows control over RRT behaviour. Each configurable property of the RRT has a unique name that is used by programmers to get and set its value. The complete configuration can also be read from or written to file.

```
public interface IRafdaRunTimeConfig {
      void setProperty(String property, String value);
      String getProperty(String property) throws Exception;
      void writeConfigurationToFile(File configurationFile);
      void readConfigurationFromFile(File configurationFile);
}
```

Figure 5.55: The *IRafdaRunTimeConfig* interface policies.

The complete set of configurable aspects is listed in Appendix C along with a description of valid values for each. Examples of the configurable properties include:

- The network interface or port to which an RRT instance is bound.
- Control over code generators.
- Firewall configuration.
- Control over the approach to memory management adopted by each RRT instance.

The code fragment shown in Figure 5.56 sets properties that cause the RRT instance to bind to the network interface associated with the host name *"host.rafda.org"* and port 12345, with a socket timeout of 60 seconds (60000ms). The configuration can be altered at any time with a few exceptions, for example, the network interface or port to which an RRT instance binds cannot be changed after an object has been exposed.

```
IRafdaRunTimeConfig rrtConfig = …;
rrtConfig.setProperty("networkInterface", "host.rafda.org");
rrtConfig.setProperty("port", "12345");
rrtConfig.setProperty("socketTimeout", "60000");
```

Figure 5.56: Setting properties to control RRT behaviour.

5.9 Conclusion

The RAFDA Run-Time (RRT) is a middleware system providing a rich feature set that meets the requirements of a third generation middleware system that were defined in the previous chapter. The RRT allows programmers to separate the design and implementation of application logic from distribution-related concerns. As a result, the development effort required when creating distributed applications or when introducing distribution into existing application is reduced. Distributed applications created using the RRT are more maintainable and more easily evolvable than applications created using traditional systems.

Chapter 6

Implementing the RAFDA Run-Time

A prototype implementation of the RRT is described and evaluated quantitatively in this chapter. Particular attention is given to the manner in which the RRT attaches to arbitrary application objects dynamically, provides remote references that are type-compatible with local references, supports remote instantiation, implements migration, offers flexibility in parameter-passing semantics and provides efficient implementations of the policy frameworks.

6.1 Introduction

This chapter describes the implementation of the RRT prototype. The previous chapter introduced the *IRafdaRunTime*, *IRafdaRunTimeRemote* and *IRafdaRunTime-Config* interfaces (shown in Figure 5.9, Figure 5.18 and Figure 5.55 respectively) through which programmers access the RRT. The RRT prototype addresses several difficulties inherent in implementing the design described in the previous chapter, the main ones being:

- Attaching the middleware system to arbitrary application objects dynamically.
- Creating remote references that can be used interchangeably with local references.
- Support for remote instantiation of objects.
- Support for object migration.
- Allowing the middleware system to alter the parameter-passing mechanisms applied to arguments and return values dynamically.
- Creating optimized implementations of policy frameworks.

The RRT prototype has been implemented using Java and so all code examples are in this language. While the RRT prototype does not employ any unique features of Java, some of the implementation details, such as the special steps taken to handle static members, are specific to a Java implementation of the RRT.

6.2 Overview of the RRT Implementation

Figure 6.1 shows an overview of the RRT architecture. It illustrates the flow of control when a remote method call is performed by object *A* on object *B*. Circles represent objects and rectangles represent components of the RRT. The large dotted rectangles in each address-space represent RRT instances.

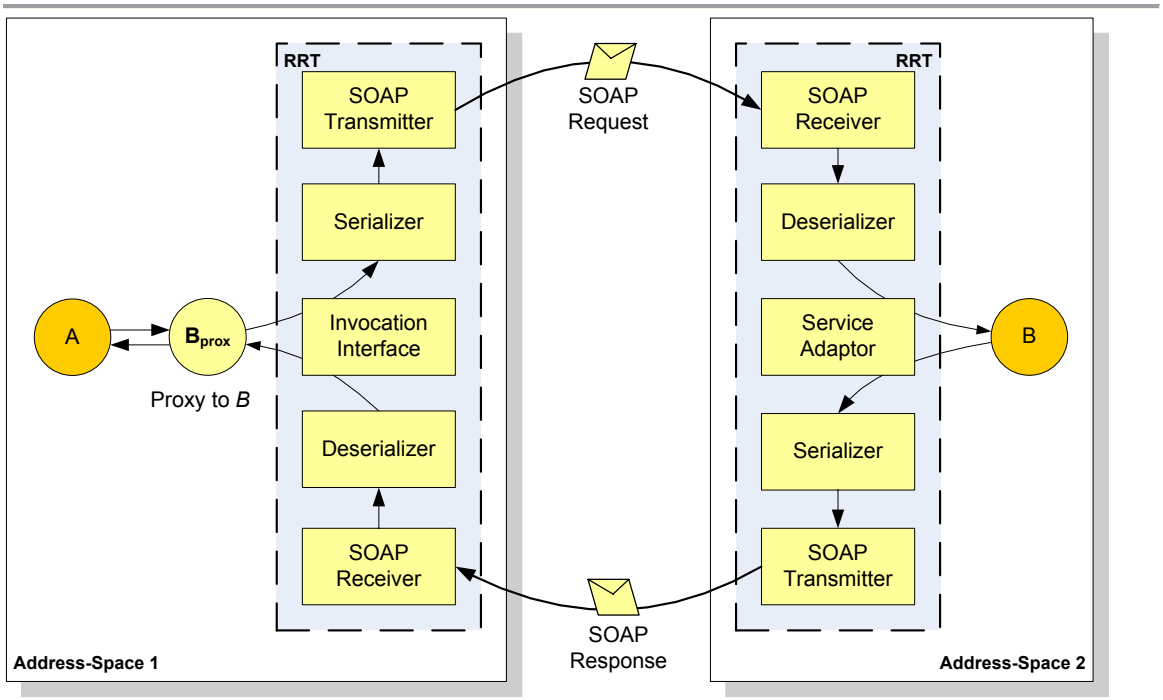


Figure 6.1: Flow of control through the RRT when a remote method call is performed.

Object *A* holds a reference to a proxy object associated with object *B*. When object *A* invokes a method on the proxy, the proxy forwards the call into the invocation interface provided by the client-side RRT instance. The client-side RRT instance marshals the method call and arguments, serializing arguments as required, and constructs a SOAP request. The request is passed to the server-side RRT instance, which deserializes the arguments and un-marshals the method call. The call on object *B* is performed by a *service adaptor*, which allows the server-side RRT instance to attach to any application object.

The return value is passed to the service adaptor, serialized then passed back across the network to the client-side RRT instance in a SOAP response. The client-side RRT instance deserializes the return value and passes it to the proxy object. The proxy object returns this value to object *A*.

Figure 6.2 shows a UML class diagram showing the structure of the RRT prototype implementation. The RRT implementation classes and interfaces are shaded. The RRT implementation can automatically generate ancillary code required to implement inter-address-space communication and these classes are represented by un-shaded boxes.

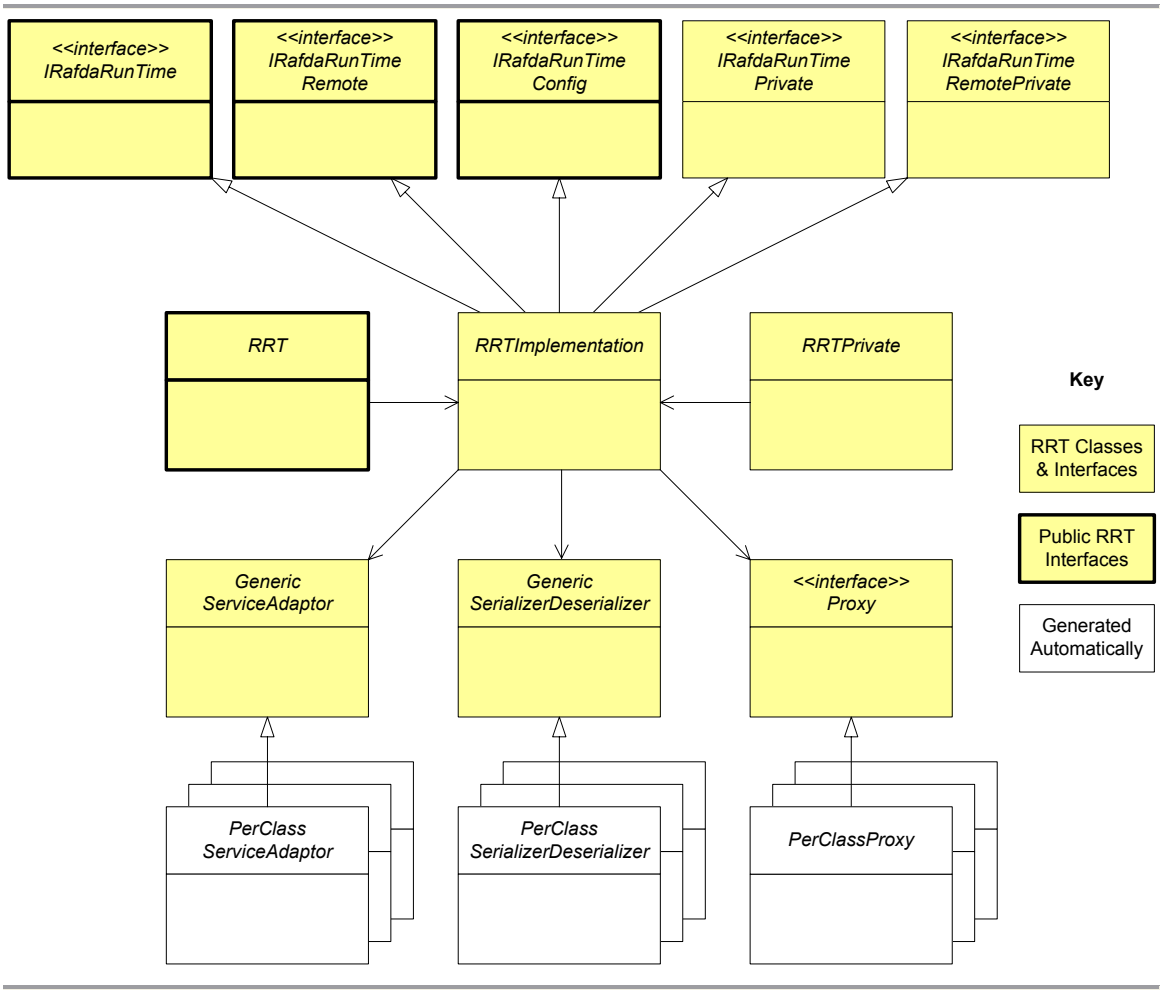


Figure 6.2: The structure of the classes and interfaces in the RRT implementation.

The main RRT implementation class, called *RRTImplementation*, implements five interfaces. The *IRafdaRunTime*, *IRafdaRunTimeRemote* and *IRafdaRunTimeConfig* interfaces, which allow programmers to access RRT instances, were introduced in the previous chapter. These are the only publicly accessible interfaces and are marked in bold. The other two interfaces, called *IRafdaRunTimePrivate* and *IRafdaRunTimeRemote-Private*, define methods that are for internal use.

The *IRafdaRunTimePrivate* interface shown in Figure 6.3 defines a series of methods that are used by automatically generated client-side code to access the local RRT instance.

```
interface IRafdaRunTimePrivate {
      Object invokeRemoteInstanceMethod(RafdaIOR rafdaIOR,
            String methodIdentifier,
            Object[] arguments);
      Object invokeRemoteStaticMethod(Class applicationClass,
            String methodIdentifier,
            Object[] arguments);
      boolean isHandlingStaticMethods(Class applicationClass);
      PassingMechanism evaluateTransmissionPolicy(
            Class argumentClass,
            Method methodIdentifier,
            int argumentNumber,
            int depth,
            boolean isReturnValue);
}
```

Figure 6.3: The *IRafdaRunTimePrivate* interface.

This interface provides the following methods:

- The *invokeRemoteInstanceMethod()* method which allows proxy objects to perform remote method calls.
- The *invokeRemoteStaticMethod()* and *isHandlingStaticMethods()* methods, both of which are used to preserve non-distributed static method semantics.
- The *evaluateTransmissionPolicy()* method which determines the parameter-passing semantics to apply to objects that are passed across address-space boundaries.

The *IRafdaRunTimeRemotePrivate* interface, shown in Figure 6.4, provides functionality to remote RRT instances.

```
interface IRafdaRunTimeRemotePrivate {
      byte[] getClassCode(String className);
      RafdaIOR migrateObject(Object objectToMigrate);
}
```

Figure 6.4: The *IRafdaRunTimeRemotePrivate* interface.

This interface provides the following methods:

- The *getClassCode()* method which is used to perform code distribution.
- The *migrateObject()* method which implements object migration.

The functionality provided by both interfaces is examined in more detail throughout this chapter.

The *RRT* class introduced in the previous chapter in Figure 5.4 is used by programmers to access RRT instances. The *RRTPrivate* class shown in Figure 6.5 is used only by RRT instances and proxy objects to obtain references to the *IRafdaRunTime-Private* and *IRafdaRunTimeRemotePrivate* interfaces of other RRT instances. The *RRTPrivate* class is not accessible to programmers.

```
class RRTPrivate {
      public static IRafdaRunTimePrivate get() {…}
      public static IRafdaRunTimeRemotePrivate getRemote() {…}
}
```

Figure 6.5: The *RRTPrivate* class.

The class diagram in Figure 6.2 contains five further classes and an interface:

- *GenericServiceAdaptor* provides a generic implementation of the service adaptor functionality which allows the RRT to attach to arbitrary application objects. The RRT can generate per-class implementations of service adaptors automatically. These per-class service adaptors extend the generic implementation class. They are represented on the diagram by the class *PerClassServiceAdaptor*.
- *GenericSerializerDeserializer* provides a generic serializer/deserializer, which is capable of serializing and deserializing instances of arbitrary application classes. The RRT can generate per-class implementations of serializer/deserializers automatically. These per-class implementations extend the generic implementation class. They are represented on the diagram by the class *PerClassSerializerDeserializer*.
- All proxy classes (represented by the *PerClassProxy* class) are generated on a per-class basis and implement the *Proxy* interface.

6.3 Implementing Server-Side Functionality

Each RRT instance is an instance of class *RRTImplementation*. Each RRT instance is exposed to remote access twice, using the *IRafdaRunTimeRemote* and *IRafdaRunTimeRemotePrivate* interfaces as its remote types. The basic connectivity that is provided by the RRT via remote method invocation is exploited to simplify the

implementation of the RRT itself. Clients can access remote RRT instances as easily as any other remote objects.

Self-exposure allows the RRT implementation to be easily maintained and extended. New functionality could be introduced to RRT instances by declaring new methods in *IRafdaRunTimeRemote* and implementing them in *RRTImplementation*, without the need to modify the underlying protocols or perform custom inter-address-space communication. Changes to the RRT implementation itself can be made as easily as changes to a distributed application created using the RRT as a middleware system.

6.3.1 Identifying Exposed Objects

Remotely accessible objects are associated with identifiers called RAFDA Interoperable Object References (*RafdaIORs*), from CORBA parlance, that uniquely identify the objects in the distributed system. *RafdaIORs* implement the remote reference functionality in the RRT and allow clients to identify remote objects. If an object is passed by-reference, it is the *RafdaIOR* associated with the object that is passed across the network. Each *RafdaIOR* identifies an exposed Web Service rather than an individual object so a single object that is exposed multiple times is associated with multiple *RafdaIORs*. The *RafdaIOR* associated with a particular object contains:

- The *InetSocketAddress* of the RRT instance exposing the object. When remote method calls are performed on the object, this address determines the RRT instance to which the SOAP requests are sent.
- A string representation of a 160-bit Universally Unique Identifier (UUID) that identifies the Web Service associated with the exposed object. UUIDs are generated randomly by the RRT such that no two services will ever have the same UUID.
- An instance of *java.lang.Class* capturing the remote type associated with the object, which was specified at exposure time. This remote type is used client-side during proxy generation and indicates which methods of those provided by the object's class will be remotely accessible. The automatic generation of proxy classes is described in Section 6.4.2.
- An instance of *java.lang.Class* capturing the class of the object. This is identified as the *real class* to differentiate it from the object's remote type. This class is also used during proxy generation.

- A list of the fields to be cached in any remote references to the object, which is used during proxy generation.
- A list of the methods to be cached in any remote references to the object, which is also used during proxy generation.
- The current values of any cached fields.

The *RafdaIOR* implementation class is shown in Figure 6.6.

```
public class RafdaIOR {
	private InetSocketAddress rrtInstance = null;
	private Class remoteType = null;
	private Class realClass = null;
	private String uuid = null;
	private Field[] cachedFields = null;
	private Object[] cachedFieldValues = null;
	private Method[] cachedMethods = null;

	public RafdaIOR(InetSocketAddress rrtInstance,
		Class remoteType,
		Class realClass) {
		this.uuid = …; // Generate UUID string
		this.rrtInstance = rrtInstance;
		this.remoteType = remoteType;
		this.realClass = realClass;
	}
	/* Getters and setters omitted */
}
```

Figure 6.6: The *RafdaIOR* class.

6.3.2 Service Adaptors

Service adaptors provide skeleton functionality in the RRT, allowing the infrastructure to attach to arbitrary application objects. When a remote call is performed on a remotely accessible object, the RRT instance exposing the object uses a service adaptor to access the object. Service adaptors are similar to servants that adopt the *tie* approach in CORBA [8]. There is a one-to-one correspondence between service adaptors and exposed services, meaning that there will be multiple service adaptors associated with

objects that are exposed with multiple remote types. Each service adaptor is an object that holds the following:

- A local reference to the exposed object.
- The *RafdaIOR* associated with the service, to allow the RRT to obtain the *RafdaIOR* associated with a particular object when passing that object across the network.
- A timestamp indicating when the service was last accessed by a remote client.
- A map between the names of the methods provided by the Web Service and instances of the *java.lang.reflect.Method* class, which allow reflective access to the exposed methods of the object.
- A Boolean indicating whether this service adaptor is acting as a tombstone. Tombstones are used during object migration and are described in Section 6.4.5.

The service adaptor permits only the methods defined in the remote type to be invoked on the exposed object. If a client attempts to call a method that is not provided by the remote type, the call will fail, even if the exposed object's class implements that method.

6.3.2.1 Generic Service Adaptor Implementation

The generic service adaptor implementation is shown in Figure 6.7. When an object is exposed, an instance of this class is created. A map from Web Service method names, which are used in SOAP requests, to instances of the *Method* class, which allow the methods to be called reflectively, is initialized. This map allows direct lookup of methods based on Web Service method names when handling remote calls, without the need for any processing of the method name. The map is populated based on the remote type and so it ensures that only methods defined in the remote type are accessible remotely. An RRT instance can invoke a method on the exposed objects using the *invokeMethod()* method, supplying the Web Service method name, any arguments and a Boolean flag indicating whether the caller is permitted access to non-public methods. Since the invoked method may throw exceptions, *invokeMethod()* throws an instance of *Throwable*, the super-type of all exceptions and errors in Java.

```
public class GenericServiceAdaptor {
	private Object exposedObject = null;
	private RafdaIOR rafdaIOR = null;
	private long timestamp = 0;
	private HashMap<String, Method> nameToMethodMap = null;
	private boolean isTombstone = false;

	public GenericServiceAdaptor(Object exposedObject,
		RafdaIOR rafdaIOR) {
		this.exposedObject = exposedObject;
		this.rafdaIOR = rafdaIOR;
		this.timestamp = System.currentTimeMillis();
		/* Code to populate nameToMethodMap omitted */
	}
	public Object invokeMethod(String wsMethodName,
		Object[] arguments,
		boolean callerHasNonPublicAccess) throws Throwable {
		/* Get hold of the java.lang.reflect.Method object
		 * associated with this Web Service method name */
		Method m = nameToMethodMap.get(wsMethodName);
		if (m != null) {
			/* If the caller has access, invoke the method */
			if (callerHasNonPublicAccess ||
				Modifier.isPublic(m.getModifiers())) {
				this.timestamp = System.currentTimeMillis();
				return m.invoke(exposedObject, arguments);
			}
		}
		throw new NoSuchMethodException("Unknown method " +
			wsMethodName + "().");
	}
	public Object getExposedObject() {return exposedObject;}
	public RafdaIOR getRafdaIOR() {return rafdaIOR; }
	public long getTimestamp() {return timestamp;}
	/* Methods used to implement migration */
	public void becomeTombstone() {isTombstone = true;}
	public boolean isTombstone() {return isTombstone;}
}
```

Figure 6.7: The *GenericServiceAdaptor* class.

Clients may only access non-public methods of exposed objects if the local protection semantics permit it. RRT instances can distinguish between clients that use the RRT and other clients by the HTTP headers supplied with SOAP requests. Clients using other technologies are not permitted to access non-public methods by default. Conversely, clients using the RRT are always permitted to access them. Each server-side RRT instance relies on its client-side counterpart to preserve local protection semantics. Each RRT instance trusts that no other RRT instance will permit a remote call that violates the local protection mechanism.

6.3.2.2 Automatically Generated Service Adaptors

Instances of the *GenericServiceAdaptor* class can be used to expose instances of any application class. However, the cost of Java reflection is incurred on every method call. In Java, it is more expensive to perform a method call using the reflection tools than it is to call the same method directly. The RRT can employ generative programming techniques to create customized service adaptors on a per-class basis. The per-class service adaptors call methods on exposed objects directly without using runtime reflection, with the reflective step moved to the time when the per-class service adaptors are generated. Per-class service adaptors all extend the *GenericServiceAdaptor* class and override *invokeMethod()* to create a version that is specific to a particular application class and remote type. The classes are generated and compiled dynamically by the RRT.

Dynamic compilation is performed using tools that are currently provided as part of the Java 2 SDK, Standard Edition (up to and including version 5.0 [40]) and the St Andrews Dynamic Java Compiler [89]. These tools are not guaranteed to be present in every Java implementation though dynamic compilation can be performed reliably under Microsoft Windows, RedHat Linux and MacOSX.

The example in Figure 6.8 shows a generated service adaptor that is associated with instances of class *JChordNode* (Figure 5.6) that have been exposed using class *Chord* (Figure 5.5) as remote type. The remote type dictates which of the exposed object's methods will be remotely accessible. Note that the *exposedObject* field must be typed according to the exposed object's real class. This is necessary because the exposed object's class does not necessarily implement or extend the remote type (though in this particular case it does); they need only be structurally compliant such that every method in the remote type has a counterpart with an identical signature in the exposed object's

class. Both the real class of the exposed object and the remote type with which is has been exposed must be known in order to generate the associated service adaptor.

```
public class JChordNode$Chord$ServiceAdaptor
      extends GenericServiceAdaptor {
      private JChordNode exposedObject = null;

      public JChordNode$Chord$ServiceAdaptor(
            JChordNode exposedObject,
            RafdaIOR rafdaIOR) {
            this.exposedObject = exposedObject;
            this.rafdaIOR = rafdaIOR;
            this.timestamp = System.currentTimeMillis();
      }
      public Object invokeMethod(String wsMethodName ,
            Object[] arguments,
            boolean callerHasNonPublicAccess) throws Exception {
            if (wsMethodName .equals("lookup")) {
                  return exposedObject.lookup((Key) arguments[0]);
            } else if (wsMethodName .equals("addNode")) {
                  exposedObject.addNode((Chord) arguments[0]);
                  return null;
            } else if (wsMethodName .equals("getSuccessorNode")) {
                  return exposedObject.getSuccessorNode();
            } else if (wsMethodName .equals("getKey")) {
                  return exposedObject.getKey();
            } else if (wsMethodName .equals("setKey") &&
                        callerHasNonPublicAccess) {
                  exposedObject.setKey((Key) arguments[0]);
                  return null;
            } else if (wsMethodName .equals("printKeyInfo")) {
                  exposedObject.printKeyInfo();
                  return null;
            }
            throw new NoSuchMethodException("Unknown method " +
                  wsMethodName  + "().");
      }
}
```

Figure 6.8: The per-class service adaptor associated with class *JChordNode* and remote type *Chord*.

The constructor in this generated class behaves differently to the inherited super constructor as it does not need to initialize the Web Service method name to *Method* object map required by the generic service adaptor. Also, the *setKey()* method declared in *Chord* is protected and so *invokeMethod()* must check whether the caller has non-public access before calling this method. The generated service adaptor class is created in the same Java package as the exposed object's class and so has access to methods with *public*, *protected* and default modifiers. However, it cannot access private methods of the exposed object directly. Generated service adaptors employ reflection to access private methods, negating their advantage over the generic implementation.

In order to allow generated service adaptors to access all methods directly, the RRT implementation provides a class loader that makes all application class members public at class load-time. If this class loader is employed, then the service adaptor will be able to access any of the exposed object's methods. This process is considered safe as the transformations are performed on code that has been verified by a standard compiler.

6.3.2.3 Generic vs. Generated Service Adaptors

Programmers specify in the RRT configuration whether the RRT should use the generic service adaptor or generate them on a per-class basis (see Appendix C for details). The two types of service adaptor offer different trade-offs [90]. The per-class versions are more efficient than the generic alternative in terms of per-call cost but incur the one-time cost of code generation and compilation. The generic implementation is more suitable for applications in which a large number of different classes of object are exposed and few remote calls are performed on each class of object. In such an environment, each generated per-class service adaptor would be used a small number of times, meaning that the cost of generation and compilation would outweigh the cost of incurring reflection on each method call. The generated per-class implementations are more suitable for use in applications in which exposed object lifetime is long or in which many remote calls are made to instances of each class. The cost of generation and compilation in these circumstances is amortized over many method calls.

By default, all generated code is discarded when the RRT instance that generated it terminates. However, programmers can configure the RRT to indicate that the generated code should be cached for future runs of the application. Currently, the RRT does not detect if the application classes associated with cached code have been modified since the code was created, necessitating another configuration option to indicate that all

cached code should be discarded and re-generated. Future work may address this problem by including a content hash in the cached code that is used to detect changes in the underlying application classes.

6.3.3 Service Adaptor Infrastructure

As described in the previous chapter, services are always accessible via URLs based on their UUIDs and, provided the services were not automatically deployed, are accessible via URLs based on programmer-specified service names. SOAP requests are passed to specific URLs and RRT instances must associate these URLs with service adaptors, in order to perform the calls. Each RRT instance holds a *service map* that is used for this purpose. The service map associates service URLs with service adaptors.

The RRT also holds an *object map* mapping from objects to service adaptors, which is used for server-side object management. Each entry in the object map associates an exposed application object with a secondary map, called the *remote type map*. The remote type map associates all the remote types with which an object is exposed to the corresponding service adaptors. The object map allows the RRT instance to find all the Web Services and corresponding remote types associated with a particular object, in order to determine whether an object is currently exposed or to obtain the *RafdaIORs* associated with a particular object. Figure 6.9 shows the service map and object map data structures present in an RRT instance and an application object *A*.

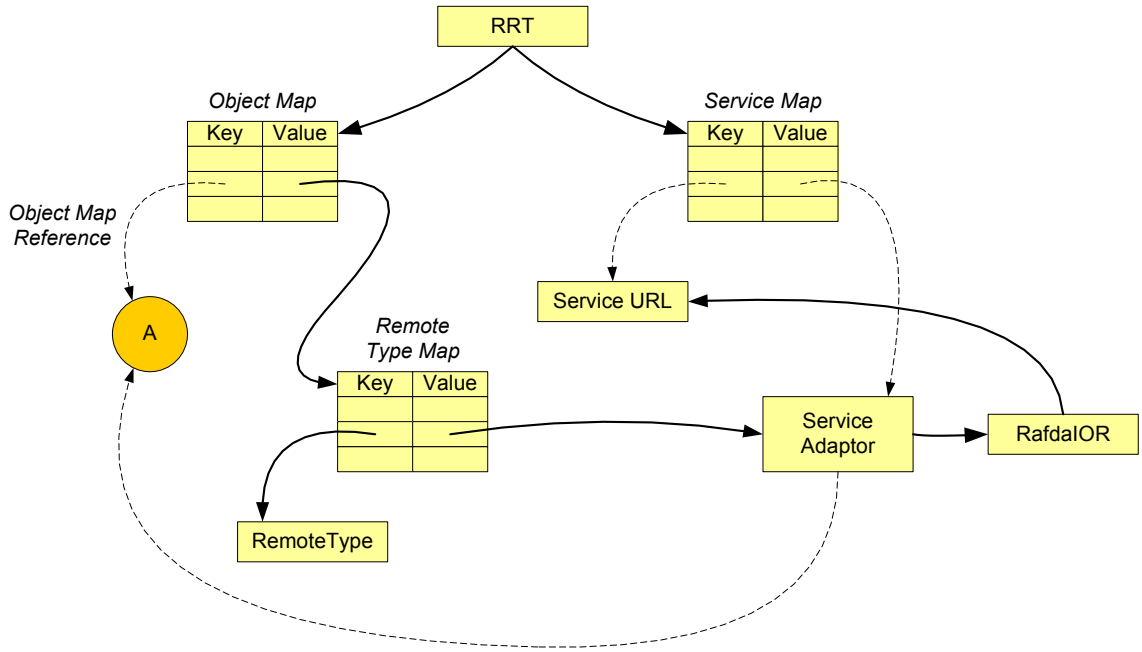


Figure 6.9: The Service Map and Object Map data structures, including an application object.

When an object is exposed with a particular remote type, a *RafdaIOR* is generated for the newly exposed service. A service adaptor is instantiated and initialized with the reference to the exposed object and the *RafdaIOR*. The last access time in the service adaptor is set to the current system time. Mappings are then created:

- Between the object and its service adaptor in the object map.
- Between the UUID-based URL and the service adaptor in the service map.
- Between the service-name-based URL and the service adaptor in the service map, if a service name has been specified.

The object map also controls whether the local garbage collector is allowed to collect exposed objects. As described in the previous chapter, the RRT provides programmers with three approaches to object lifetime management:

1. The RRT infrastructure allows exposed objects to be collected when they are no longer referenced locally by the running application, even if referenced remotely. To achieve this, the RRT holds only weak references to exposed objects, which are ignored by the local garbage collector when determining whether an object is referenced.
2. The RRT holds (conventional strong) references to exposed objects and continues to do so until they are removed from remote access manually.
3. The RRT holds (conventional strong) references to exposed objects. However, the references to any exposed objects that are not accessed remotely within a programmer-defined lease time are changed into weak references, allowing the local collector to reclaim them.

Figure 6.9 shows the references that exist when the RRT instance adopts approach 1. Conventional strong references are shown as solid arrows and weak references are shown as dashed arrows. In this example, both the service map and object map are implemented in Java using *weak maps*. Weak maps hold weak references to keys and so the keys may collected by the local collector, despite their presence in the weak map. The weak map detects when collection of keys has occurred and automatically removes the associated mappings, thereby releasing any references held to values associated with those keys.

The RRT infrastructure holds no strong references to the application object *A*. The object map holds a weak reference to the application object and as long as the application object is extant, the object map will hold a strong reference to the associated remote type

map. Consequently, the remote type map will not be collected, nor will the object's service adaptor, *RafdaIOR* or service URL.

The local collector is free to reclaim object *A*. If it does so, the weak object map will detect the collection of one of its keys (object *A*) and will release the reference it holds to the value associated with this key (object *A*'s remote type map). The remote type map is no longer referenced and so will eventually be reclaimed, along with the service adaptor, *RafdaIOR* and service URL object.

The two other approaches to memory management can be adopted on a per-RRT-instance basis through the RRT configuration. If the second approach to memory management is taken, in which application objects are never collected, the object map is implemented as a conventional map rather than a weak map. The reference held by the object map to the application object, marked *Object Map Reference* would be a strong reference. This ensures that the application object is always (strongly) locally referenced and so will never be collected until it is explicitly removed from remote access by the programmer.

The third approach to memory management, in which objects not accessed within the lease time are collected, is implemented using two object maps, one conventional and one weak. Initially, objects are placed into a conventional object map that strongly references them. Periodically, the RRT instance checks to see whether objects have been accessed within the lease time. Those that have not are moved from the conventional (strong) object map to the weak object map.

As with the first approach, the local collector may now reclaim the objects if necessary but the RRT instances can still access the corresponding service adaptors if any incoming remote calls to the objects arrive before collection. If remote calls occur, the objects are moved back into the conventional object map and their last access time is updated.

Using two separate object maps has advantages over an approach based on a single object map that holds both strong and weak references. Weak maps in Java have subtly different semantics to conventional maps holding weak references to keys. Even when a key is collected, a conventional map will continue to hold a strong reference to the associated remote type map, forcing programmers to remove the mapping manually. Using a weak map ensures that housekeeping is performed automatically by the local garbage collector when the application object *A* is collected. RRT instances do not need to take any special steps to detect the collection of exposed application objects.

6.3.4 Serializers and Deserializers

The RRT is capable of passing instances of arbitrary classes by-value. The implementation of appropriate serializers and deserializers that allow the transmission of arbitrary objects across the network is described here.

6.3.4.1 Serializers

A generic reflective serializer is provided. It is a generic object browser that examines the fields present in any arbitrary object and generates a corresponding SOAP encoded representation of the object using reflection. Using generative programming techniques, the RRT can generate and compile customized per-class serializers that directly access fields when serializing instances of a particular class. The per-class serializers do not use reflection when accessing public, protected and default fields though do when accessing private fields. The RRT class loader described previously in the context of service adaptors can again be employed to make all fields in application classes public to avoid the need for any reflection when serializing objects using the generated serializers.

The trade-offs between the generic and per-class implementations are similar to those faced when deciding which kind of service adaptor implementation to employ. The per-class generated serializers are more appropriate than the generic implementation when many instances of each class are serialized, allowing the one-time cost of generation and compilation to be amortized over many serialization operations. By default, the per-class serializers are deleted when the RRT instance that generated them terminates. The generated code may be cached across multiple runs of the application, in the same manner as per-class service adaptors, in order to avoid the cost of re-generation.

6.3.4.2 The Sub-type Problem

Conventional Web Services semantics dictate that return values are passed by-value. Each method provided by a Web Service has a statically defined return type and Web Services can return only objects that are of exactly this type. Instances of sub-types of the statically defined return types cannot be returned when using standard Web Services. The reason for this limitation with respect to sub-types is as follows.

Standard Web Services perform inter-address-space communication using the SOAP protocol. When an object is serialized into XML using SOAP, the middleware

needs to associate a type with this serialized object to permit deserialization. Using standard Web Services technology, each client holds a mapping between programming language types and XML namespaces. The XML namespace corresponding to the serialized object's type is included in the serialized object data. From this namespace, a deserializer can map the namespace back to a programming language type in order to instantiate the object.

Since there are an infinite number of possible sub-types of a statically defined return type, it is not possible to associate a unique XML namespace with every sub-type statically. Thus, conventional Web Services technologies cannot return arbitrary types because serializers cannot capture type information in the serialized object data.

Support for sub-typing is provided through an extension to Web Services semantics, which is incompatible with standard Web Services. The RRT employs the extended semantics when both client and server are RRT-based to allow full support for the transmission of sub-types. When the RRT is used in conjunction with conventional Web Services technology, standard Web Services semantics are adopted. The RRT determines whether to employ extended semantics on a per-class basis.

Transmission of sub-types is achieved using an approach similar to *autotyping* in Apache Axis [57]. A deterministic mapping from programming language class to namespace is adopted at serialization time. The namespace consists of two parts. The first part indicates that the RRT performed the serialization using this deterministic naming scheme and identifies the programming language in use. The second part contains the fully qualified name of the associated class. This approach allows the deserializer to determine the class of a serialized object directly from its namespace, negating the need for explicit mappings between namespaces and types. Figure 6.10 shows the namespace generated when an instance of Java class *JChordNode* in package *jchord* is serialized.

```
"uk.ac.stand.dcs.rafda/java:jchord.JChordNode"
```

Figure 6.10: Deterministically generated namespace.

6.3.4.3 Deserializers

The RRT implementation includes a generic deserializer that performs all object creation and initialization reflectively. To create objects, the generic deserializer uses the default constructor if one exists otherwise it uses one of the other constructors, specifying

default values for the initialization arguments. These default values are overwritten when the deserialized state is written into the newly instantiated object. This approach to object creation can have unexpected consequences if any of the constructors have side-effects. This problem could be solved in future work by generating a special RRT constructor for each application at load-time, to be used only for object instantiation during deserialization.

When performing deserialization, an RRT instance may need to instantiate a class for which it has no locally accessible code. A code distribution system that allows the deserializing RRT instance to query the serializing RRT instance in order to obtain the required code in binary form is provided. The received code can then be loaded dynamically and the instantiation performed.

Code is obtained from a remote RRT instance via the *getClassCode()* method (shown in Figure 6.11) provided by the *IRafdaRunTimeRemotePrivate* interface (shown in Figure 6.4). Given the name of a class, it returns an array of bytes that encodes that particular class. Application code is distributed throughout the distributed system lazily using this mechanism.

```
byte[] getClassCode(String className);
```

Figure 6.11: The *getClassCode()* method, used for code distribution.

Per-class deserializers can be generated to complement per-class serializers, in order to avoid the cost of reflection. Much like per-class serializers, per-class deserializers are beneficial in applications in which many instances of a particular application class are deserialized over the lifetime of the RRT instance. Generated deserializers cannot directly modify final or private fields though this limitation is overcome by reverting to reflective access where necessary. Programmers can employ the RRT class loader, to makes all fields non-final and public, in order to avoid the need for any reflective operations in generated deserializers.

6.4 Implementing Client-Side Functionality

As discussed at the beginning of this chapter, each RRT instance exposes itself to remote access using the *IRafdaRunTimeRemote* interface. Clients can obtain a reference to a remote RRT instance using the *getRemote()* method in the *RRT* class. Clients can

obtain references to remote objects, instantiate objects in remote address-spaces and migrate objects between address-spaces using this interface.

6.4.1 Proxy Objects

An application can call the *getRemoteReference()* method on a remote RRT instance to obtain a reference to an exposed object. The remote RRT instance returns the *RafdaIOR* associated with the exposed object. The *RafdaIOR* is not type-compatible with the application object it represents so the client cannot call methods directly on the *RafdaIOR*, therefore a type-compatible proxy object is created to encapsulate the *RafdaIOR*.

Proxy objects in the RRT adopt a dual role:

1. They act as conventional middleware proxies, namely as local handles on remote objects that propagate method calls across the network.
2. They act as wrappers that introduce a layer of indirection into applications to allow object migration to be performed transparently with respect to the objects' reference holders.

Each role is now examined in turn.

6.4.1.1 Conventional Proxy Behaviour

Each proxy object is associated with a single remote object. The proxy object appears to clients to be an instance of the remote type, irrespective of the real class of the remote object. Proxy classes are generated from remote types but contain only non-static methods. Static methods are handled as described later in Section 6.4.3. For each non-static method declared in the remote type, a type equivalent proxy method is created in the proxy class.

A proxy method propagates calls across the network using the remote invocation method provided by the local RRT instance. This remote invocation method, called *invokeRemoteInstanceMethod(),* is found in the *IRafdaRunTimePrivate* interface. This interface, shown in Figure 6.12, is not publicly accessible and is used only by proxy objects to perform remote method calls.

```
Object invokeRemoteInstanceMethod(RafdaIOR rafdaIOR,
      String wsMethodName,
      Object[] arguments);
```

Figure 6.12: The *invokeRemoteInstanceMethod()* invocation method.

When calling the *invokeRemoteInstanceMethod()* method, the proxy object passes the *RafdaIOR* of the remote object to be called, the name of the method to be called and any arguments. The local RRT instance marshals the method call according to the active transmission policy. Arguments may be passed by-reference, by-value or by-migrate.

- If an argument is passed by-reference, then the associated *RafdaIOR* is transmitted across the network. The argument is automatically exposed if necessary. Before serializing a *RafdaIOR*, the RRT examines the active transmission policy to determine if any fields or methods of the referenced object are cached. If so, the RRT updates the cached field and method information in the *RafdaIOR*. The *RafdaIOR* is then serialized and added to the SOAP message.
- If an argument is passed by-value, it is serialized and added to the SOAP message. Pass-by-value semantics may only be applied to objects that exist locally. If the transmission policy dictates that by-value semantics should be applied to an argument which itself is a remote reference, then the RRT passes that argument by-reference.
- If an argument is passed by-migrate, then the RRT initially checks whether the argument has been wrapped through a call to *makeMigratable()*, in preparation for migration. If so, the object is migrated immediately to the remote address-space then passed by-reference. If the argument does not support migration then it is passed by-reference.

The resulting SOAP message consists of the Web Service method name and a series of serialized arguments, some of which are remote references. The RRT instance sends the SOAP message to the corresponding remote RRT instance using the socket address stored in the *RafdaIOR*.

The server-side RRT instance receives and deserializes the SOAP request resulting in a Web Service method name along with a series of arguments, some of which are *RafdaIORs*. *RafdaIORs* are not type-compatible with the arguments they represent.

The RRT instance checks whether the *RafdaIOR* corresponds to a local object and, if so, will pass that object to the method call instead of the *RafdaIOR*. Alternatively, if a proxy object corresponding to the *RafdaIOR* already exists then that proxy object is passed to the method call. If no proxy object exists, one is created. The appropriate proxy class is generated and compiled automatically if necessary.

6.4.1.2 Wrapper Behaviour

Proxy classes in the RRT also play a role in object migration, which is implemented using Stub Scion Pair (SSP) Chains [91]. Instances of a particular proxy class can act as a wrapper to instances of the associated application class. By introducing this layer of indirection, the RRT can ensure referential integrity when a migratory object is copied from one address-space to another. All local and remote references to the original copy of the migratory object in the old address-space must be updated to refer to the copy in the new address-space. It is desirable, though not logically necessary, to change all remote references in the new address-space into local references to the copy in that address-space.

The substitution of an application object with a proxy object is difficult as there is no mechanism in Java, or in other typical object-oriented languages, to substitute one object directly for another i.e. to substitute an application object with a proxy object. Further, given an arbitrary object, it is not possible to determine which other objects reference it locally when using typical languages, making direct update of references in place difficult. The following example illustrates how wrappers can be used to allow substitution. Figure 6.13 shows an application in which object *A* in address-space 1 is exposed to remote access. Object *A* is locally referenced by multiple objects in address-space 1 and remotely referenced by multiple objects in address-spaces 2 and 3. There is one proxy object in each of address-spaces 2 and 3 that allow the reference holders to call methods in the remote object *A*. Application objects are darkly shaded whilst proxy objects are lightly shaded.

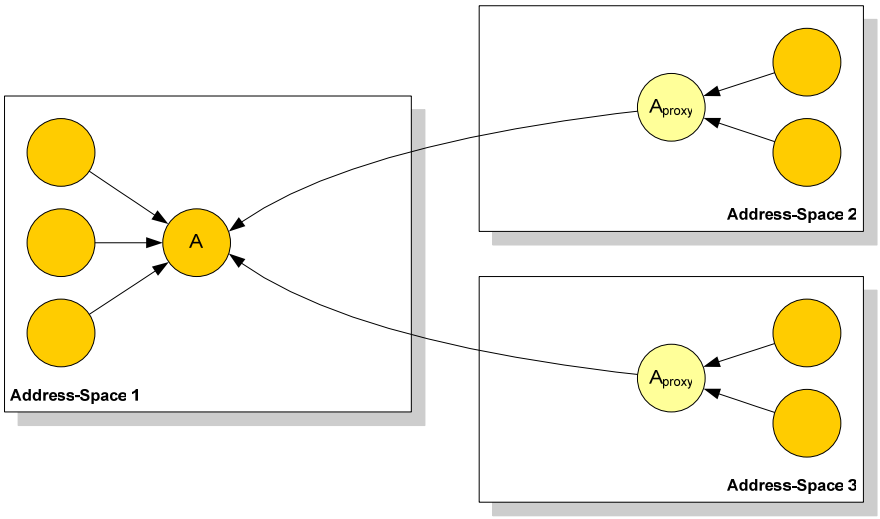


Figure 6.13: An example application in which both local and remote references to object *A* exist.

When the *makeMigratable()* method is called on object *A* to introduce support for migration, the object is wrapped using a proxy object. A reference to the wrapper returned to the application and all references to object *A* are updated (by programmers) to reference the proxy object. Figure 6.14 illustrates the resultant changes to the application.

Object *A* is referenced directly only by the wrapper (local proxy object). The proxy in address-space 1 forwards any calls performed by reference holders onto the wrapped object. The proxy objects in address-spaces 2 and 3 are conventional proxies, forwarding calls across the network. Any of the proxy objects in the system can act as wrappers meaning that the location of *A* is now transparent with respect to its reference holders, whether local or remote.

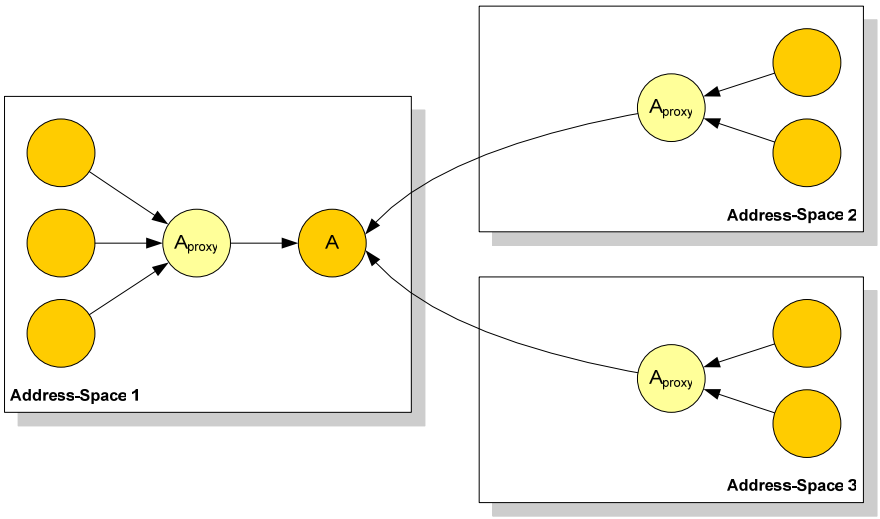


Figure 6.14: The application structure after *makeMigratable()* has been called on object *A*.

To migrate object *A* to address-space 2, it is first copied to that address-space as shown in Figure 6.15. The proxy in address-space 2 becomes a wrapper and the proxy in address-space 1 becomes a conventional proxy. The only references that need to be updated are those present in the proxy objects themselves, all of which are known and accessible to the RRT. In address-space 1, the service adaptor associated with the object begins to act as a tombstone so that other remote references can be updated lazily when they attempt to access object *A* in address-space 1. The details of migration implementation and the manner in which references are updated coherently are discussed later.

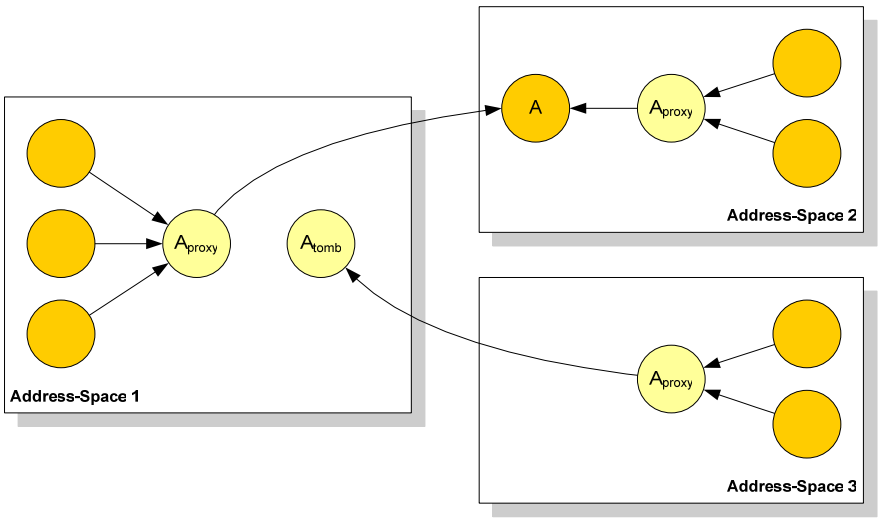


Figure 6.15: The application structure after object *A* has migrated from address-space 1 to 2.

Without the layer of indirection introduced by the wrapper object, it would not be possible to convert all local references in address-space 1 into remote references (by substituting object *A* with a proxy) or to convert the remote references in address-space 2 into local references (by substituting the proxy with the new copy of object *A*).

6.4.2 Implementing Proxy Classes

It has been shown that proxy objects adopt two distinct roles as:

- Conventional middleware proxies used to perform remote method calls.
- Wrappers used to introduce indirection into applications to allow the implementation of transparent object migration.

Proxy objects are instances of proxy classes, which are always generated automatically. No generic proxy implementation exists as it is not possible to create proxy objects that appear to clients to be instances of the associated remote types without using

code generation. Each proxy class is constructed based on the real class of an exposed object, the remote type with which the object is exposed and the lists of cached fields and methods included in the *RafdaIOR*.

Each proxy class implements the methods defined in the remote type. When acting as a conventional proxy, these methods are either un-cached, meaning that they propagate the call into the RRT infrastructure, or are cached, meaning that they execute locally. When acting as a wrapper, all method calls are forwarded onto the locally wrapped application object.

Each proxy object holds a reference to a wrapped application object or the *RafdaIOR* associated with a remote object but never both. When behaving as a wrapper, each proxy object also hold a reference to a reader/writer lock of class *ReadWriteLock*. This lock is used at migration time to lock access to the local object while it is copied to another address-space and the references are updated. The behaviour of this lock is described later in the context of migration.

Each proxy class implements the interface shown in Figure 6.16, which allows access to the locally wrapped object, the *RafdaIOR* associated with the proxy and the reader/writer lock.

```
public interface Proxy {
	Object getLocalObject();
	void setLocalObject(Object localObject);
	RafdaIOR getRafdaIOR();
	void setRafdaIOR(RafdaIOR rafdaIOR);
	ReadWriteLock getReadWriteLock();
}
```

Figure 6.16: The *Proxy* interface implemented by all proxy classes.

Since Java supports only single class inheritance, the inheritance hierarchy adopted by generated proxy classes differs depending on whether the remote types with which they are associated are Java interfaces or Java classes. When a remote type is a Java interface, the associated proxy class extends the real class of the exposed object and implements the remote type and *Proxy* interfaces as shown in Figure 6.17. Consequently, instances of the proxy class appear to clients to be instances of both the remote type and the real class. In addition to extending the application, the proxy class contains a field typed as the application class, in order to implement wrapper behaviour.

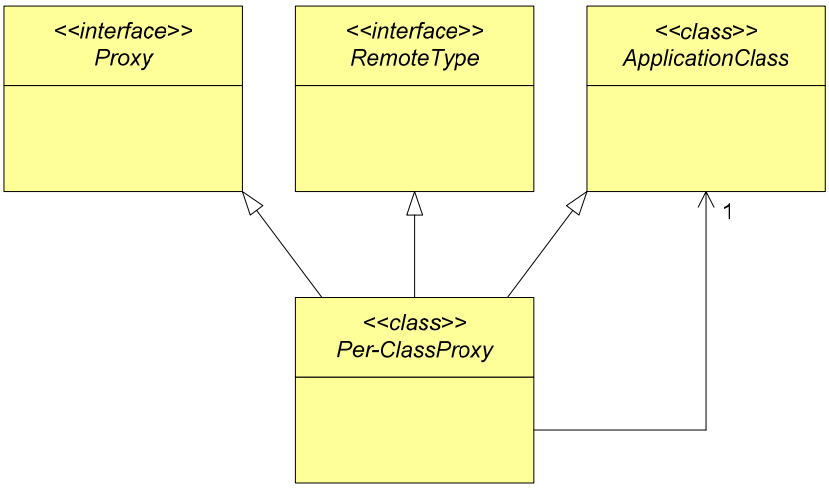


Figure 6.17: Proxy class derived from a remote type that is a Java interface.

When a remote type is a Java class, the associated proxy class extends the remote type and the *Proxy* interface only, as shown in Figure 6.18. Though the proxy class does not extend the application class, it still contains a field typed as the application class to allow it to wrap an instance of that class. Instances of the proxy class appear to be instances of the remote type, not instances of the exposed object's real class.

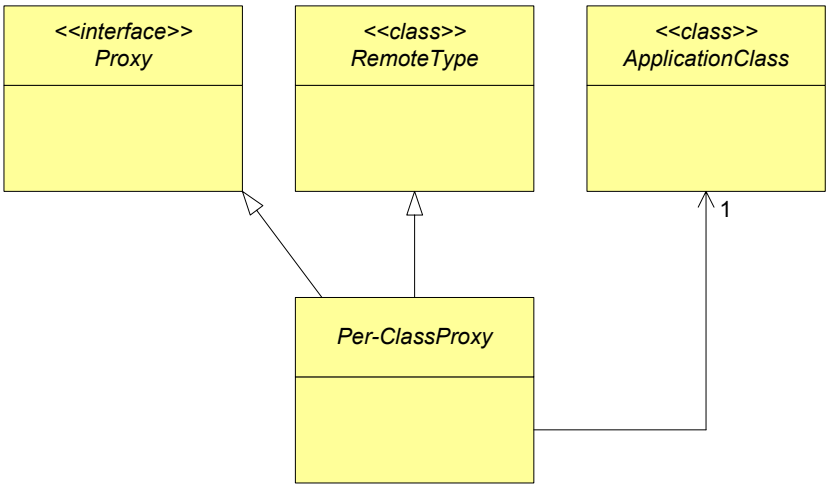


Figure 6.18: Proxy classes derived from remote types that are Java classes.

Figure 6.19 shows the proxy class generated when an instance of the *JChordNode* class is exposed to remote access using the *Chord* class as remote type. In this example, it is assumed that the *lookup()* method is un-cached and so behaves as a conventional proxy method but that the *getKey()* method is cached and so executes locally.

When called, the *lookup()* proxy method will either invoke the method locally on the wrapped object or invoke it remotely through the RRT instance, depending on whether the proxy object is acting as a conventional middleware proxy or a wrapper. The

getKey() method is cached so either accesses the method directly on the wrapped object (when behaving as a wrapper), or accesses a local implementation (when acting as a proxy with a cached method). It is important to note the distinction between cached methods and wrapper methods. The proxy class inherits cached methods from its super-class so they execute on the proxy object while wrapper methods execute on the wrapped object.

The reader/writer lock is used to implement migration and is described in detail later. Each method must ensure it holds a reader lock before accessing the locally wrapped object. The *try-finally* construct ensures that the lock is always released, even if exceptions are thrown.

```
public class JChordNode$Chord$Proxy extends Chord implements Proxy {
	private RafdaIOR rafdaIOR = null;
	private JChordNode localObject = null;
	private ReadWriteLock readWriteLock = new ReadWriteLock();

	/* Un-cached method */
	public Chord lookup(Key key) {
		readWriteLock.getReadLock();
		try {
			if (localObject != null) {
				return localObject.lookup(key);
			} else {
				IRafdaRunTimePrivate rrt = RRTPrivate.get();
				return (Chord) rrt.
					invokeRemoteInstanceMethod(
						rafdaIOR,
						"lookup",
						new Object[] { key });
			}
		} finally {
			readWriteLock.releaseReadLock();
		}
	}
```

```
	/* Cached method */
	public Key getKey() {
		readWriteLock.getReadLock();
		try {
			if (localObject != null) {
				return localObject.getKey();
			} else {
				/* Call inherited method implementation */
				return super.getKey();
			}
		} finally {
			readWriteLock.releaseReadLock();
		}
	}
	/* Other methods from Chord omitted */
	public Object getLocalObject() {return localObject;}
	public void setLocalObject(Object localObject) {
		this.localObject = (JChordNode) localObject;
	}
	public RafdaIOR getRafdaIOR() {return rafdaIOR;}
	public void setRafdaIOR(RafdaIOR rafdaIOR) {
		this.rafdaIOR = rafdaIOR;
	}
	public ReadWriteLock getReadWriteLock() {return readWriteLock;}
}
```

Figure 6.19: The proxy class associated with *JChordNode* instances exposed with remote type *Chord*.

6.4.3 Static Members

Static members (that is, methods and fields) are associated with classes rather than objects. The RRT provides two alternatives to handling static members. The first approach does not intercept static method calls, thereby allowing all of them to execute locally. The consequence of this is that each Java Virtual Machine holds a private copy of the fields. This approach has the advantages that it is efficient in terms of execution time as no remote calls occur and that the RRT is not required to take special steps to handle static members.

Using the second approach, programmers must identify a single RRT instance in the distributed system as the *root*. The root has responsibility for tracking which RRT

instances have responsibility for the static members of which classes. The root RRT must be set in every RRT instance in the distributed system via the RRT configuration.

The second approach requires the use of the RRT class loader, which can transform applications to allow interception of static method calls. The RRT instance in which the static members of a particular class are first accessed is assigned to be the *home RRT instance* for that class, in which its static fields will be stored. The root acts as arbiter to ensure that exactly one RRT instance becomes home instance for each application class. If static methods of a class are called by objects in the home RRT instance, they execute as normal, accessing the locally held copies of the static fields. If static methods of a class are accessed by objects in any other RRT instances, the calls are intercepted and the equivalent static methods remotely invoked on the home RRT instance.

The class loader transforms all static methods dynamically at class load-time. When called, each static method will check with the local RRT instance to determine whether the method executes locally or remotely. It then either executes the method as normal or performs a remote call to the equivalent method in another RRT instance. Figure 6.20 shows an example class called *JChordNodeSingleton*, which has not yet been transformed to support remote access to static members. It contains a static field called *singleton* that is accessed via the *getSingleton()* method.

```
public class JChordNodeSingleton {
	private static JChordNode singleton = …;

	public static JChordNode getSingleton() {
		return singleton;
	}
}
```

Figure 6.20: An untransformed class containing a static method.

Figure 6.21 shows the J*ChordNodeSingleton* class after transformation. The existing code is unchanged aside from the introduction of several lines at the beginning of the static method that use the *IRafdaRunTimePrivate* interface to determine whether the static member should be accessed remotely, and if so, to perform the remote call.

```
public class JChordNodeSingleton {
      private static JChordNode singleton = …;

      public static JChordNode getSingleton() {
            IRafdaRunTimePrivate rrt = RRTPrivate.get();
            if (!rrt.isHandlingStaticMethods(JChordNode.class)) {
                  return (JChordNode) rrt.
                        invokeRemoteStaticMethod(
                              JChordNode.class,
                              "getSingleton",
                              null);
            }
            return singleton;
      }
}
```

Figure 6.21: A class with a transformed static method.

The *isHandlingStaticMethods()* is used to determine whether the local RRT instance is the home of the specified class. If so, then the method executes locally as normal. If not, the *invokeRemoteStaticMethod()* method is called. This method propagates the call to that home RRT instance of the specified class. The root RRT instance can be used to determine the home RRT instance of any class.

The failure model adopted by the RRT when calling remote static methods is identical to that used when calling remote instance methods. The RRT can consume distribution-related exceptions or can propagate them back to the application.

6.4.4 Creating Objects in Remote Address-Spaces

The *instantiateAndExpose()* method defined in the *IRafdaRunTimeRemote* interface allows a programmer to create an instance of any arbitrary class in a remote RRT instance and expose it for remote access. This method is shown in Figure 6.22.

```
Object instantiateAndExpose(Class classToInstantiate,
      Object[] constructorArguments,
      Class remoteType,
      String serviceName);
```

Figure 6.22: The *instantiateAndExpose()* method.

The implementation of this method relies on the transmission policy framework and the self-exposure provided by the RRT instances. The Java reflection tools are used to instantiate and initialize an instance of the specified class. The remote RRT determines which constructor to use based on the types of the specified constructor arguments. The object is then exposed with the remote type and service name. Finally, a remote reference to the newly created object is returned to the caller.

This functionality is implemented in a few lines of code in the RRT. This is as a direct consequence of the RRT's exposure of itself to remote access and the control over parameter-passing semantics afforded by the transmission policy framework. By building on the flexibility provided by the RRT infrastructure, advanced middleware functionality can be easily provided.

6.4.5 Migrating Objects to Remote Address-Spaces

Migration is implemented by building on the transmission policy framework and the dual nature of proxy objects. When programmers call the *makeMigratable()* method provided by the *IRafdaRunTime* interface, a proxy object is created and used to wrap the supplied application object. A reference to this wrapper is returned and the programmer must ensure that all objects holding references to the application object are updated to refer to the wrapper, as shown previously in Figure 6.14. Once this has been achieved, the location of the application object is transparent to all its reference holders.

The RRT implements migration by copying the migratory object to the remote address-space then updating all references to it. In order to achieve this consistently, no clients can be permitted to call methods on the migratory object after the migration process has begun, until it is completed and all references have been updated. If updates to the old copy of the object were allowed after it had been duplicated into the remote address-space, those changes would not be reflected in the new copy of the object, and so would be lost.

To ensure application coherency, the RRT employs a locking mechanism that ensures migration will only be performed when no methods are executing on the object and that no method calls will be performed until the migration operation completes. A reader/writer lock is used to provide the required functionality. This lock allows multiple readers but only one writer to hold the lock simultaneously. When a writer tries to obtain the lock, it is blocked until all readers holding the lock release it. If a reader attempts to obtain the lock while a writer is blocked waiting for it, that reader is also blocked.

The reader/writer lock is used as follows. A read lock must be obtained in order to call a method on a wrapped object. A write lock must be obtained in order to migrate the wrapped object to another address-space. Thus, multiple methods may be executed simultaneously on the wrapped object but migration cannot proceed until all methods executing on the wrapped object complete. Any attempt to call a method on a wrapped object in the middle of a migration operation is blocked until that operation is completed and the writer lock released.

Thus migration of a particular object proceeds as follows:

- The local RRT instance obtains a write lock on the reader/writer lock in the service adaptor associated with the migratory object.
- The migratory object is copied to the new RRT instance and exposed using the remote type specified when *makeMigratable()* was called. If the migratory object was exposed in the old address-space using other remote types, it is re-exposed with those remote types also.
- The RRT permits at most one proxy object in a single address-space to be associated with the same object and remote type. If there is an extant proxy object associated with this migratory object in the remote address-space, it is used as a wrapper for the new copy of the migratory object.
- The wrapper in the old address-space is updated with the *RafdaIOR* of the object in the remote address-space and begins to act as a conventional proxy.
- All service adaptors associated with the object are updated to become tombstones, by calling their *becomeTombstone()* methods. If any clients attempt to access the object at the old address-space, the RRT detects that migration has occurred through the presence of the tombstone. The service adaptors hold the new *RafdaIOR* of the migrated object and return it to the caller by returning a response indicating that migration has occurred. The client updates its remote references with this *RafdaIOR* and reattempts the call.

The RRT provides a *migrate()* method through the *IRafdaRunTimeRemote* interface, allowing migration to be performed by programmers. This method is implemented using the RRT's support for pass-by-migrate semantics. The *migrate()* method in the RRT instance is an empty method that performs no computation but is associated with a method policy rule indicating that pass-by-migrate semantics should be

applied to an infinite depth when it is called. Thus, when called, it will immediately return but any arguments passed to it will be migrated to the remote RRT instance as a result of the active transmission policy.

6.4.6 Remote Method Call Cost

The cost of remote method calls in the RRT prototype was compared with the equivalent calls using other middleware systems. A test application was created then distributed using multiple different middleware technologies. The run-time cost of method calls was determined to allow comparisons between the RRT and existing middleware systems. The following Java-based systems were evaluated:

- The RRT.
- Java RMI (J2SE 1.5).
- CORBA (using the ORB supplied with J2SE 1.5).
- Apache Axis (version 1.2 final) deployed in Tomcat (version 5.5) [92].

Additional versions of the test application were created to execute under the Microsoft .NET framework, which permits programmers to choose between SOAP and a proprietary TCP-based transport protocol when performing remote method calls. The following .NET based systems were evaluated:

- Microsoft .NET framework using SOAP channels (version 1.0).
- Microsoft .NET framework using TCP channels (version 1.0).

Tests were run on a two machine network. The first machine, designated the *"server"*, was used to execute the server-side applications that exposed objects to remote access. It contained a 2.7GHz Pentium 4 with 512MB RAM. The second machine, designated the *"client"*, was used to execute the client-side applications that performed the remote calls. It contained a 1.2GHz Pentium 3 with 256MB RAM. The machines were connected using an isolated 100Mb/s Ethernet. Since the .NET framework executes only under the Windows operating system, all tests on both machines were run under Windows XP Service Pack 2, fully patched, with only default services running.

The first test evaluates the cost of a remote method call to a method that took no arguments, performed no computation and returned no results. The clock resolution provided by the test machines was 10ms, which is considerably greater than the average method call time. Therefore the test application performed 100 batches of 4000 method calls using each middleware system, resulting in a total run-time of between two and

twenty minutes wall clock time. The system clock was used to measure the time taken to perform each of the 100 batches of method calls. Apache Axis received special treatment as it ran around an order of magnitude slower than all other systems. Each batch performed only 400 method calls, rather than 4000, in order to achieve reasonable total test execution time.

This test determines the lower bound of call cost, since there are no arguments or return values to pass, meaning no marshalling is performed. Table 6.1 shows the average time in milliseconds for a remote method call and the minimum and maximum call times observed.

Middleware	Average	Min	Max
Java RMI	0.26	0.25	0.26
.NET (TCP)	0.44	0.44	0.47
CORBA	0.87	0.85	0.91
RRT	2.10	2.02	2.22
.NET (SOAP)	2.94	2.91	3.03
Apache Axis	12.60	11.87	14.43

Table 6.1: The time in milliseconds for a remote method call to an empty method.

The second test was run under the same conditions as the first test but introduced arguments that required serialization. The method called by this test application took ten arguments, all of which were passed by-value. The arguments were all instances of the same complex type, which contained a 10 character string, a 25 character string and an integer. In all tests the arguments were initialized identically.

Table 6.2 shows the time in milliseconds for a remote method call to this method, which required the middleware system to perform serialization. The table shows the average call time along with the minimum and maximum call times observed.

Middleware	Average	Min	Max
Java RMI	0.43	0.42	0.45
.NET (TCP)	0.86	0.86	0.88
CORBA	1.41	1.40	1.49
RRT	2.63	2.53	2.89
.NET (SOAP)	5.07	5.04	5.17
Apache Axis	20.88	16.24	24.24

Table 6.2: The time in milliseconds for a remote method call to a method with arguments.

The figures obtained from both tests are graphed in Figure 6.23.

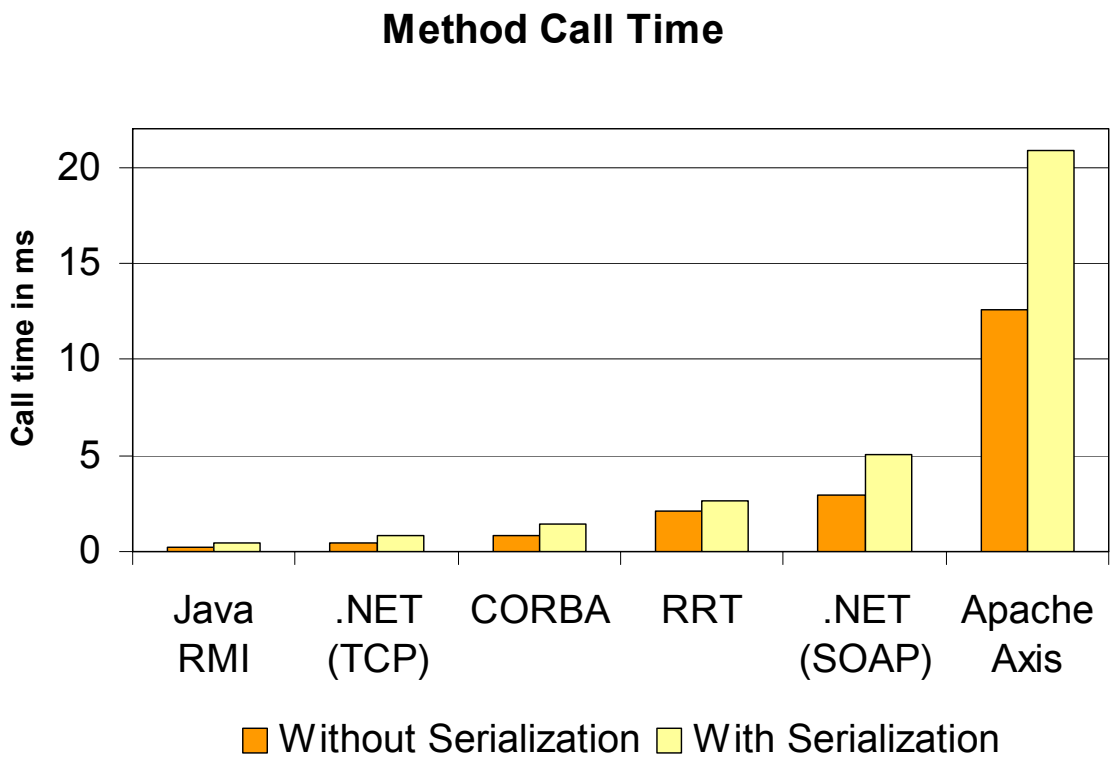


Figure 6.23: Method call time in milliseconds.

A clear difference can be seen between the middleware systems that use XML-based SOAP as their transport protocol (the RRT, Apache Axis and the .NET framework employing SOAP channels) and those that use binary protocols (Java RMI, CORBA and the .NET framework employing TCP channels). The RRT outperforms both its SOAP-based counterparts; the application employing the RRT ran more than 25% more quickly than the equivalent .NET application and around a factor of six times more quickly than the application employing Apache Axis. When serializing a large number of arguments, the RRT is again the quickest of the SOAP-based systems. During this test, the RRT used cached per-class serializers in order to optimize the serialization process, giving it a large advantage over the other systems, which do not generate such serializers.

The applications using Java RMI, CORBA and TCP-based .NET all executed two to five times as quickly as the RRT. It should be noted that there are many implementations of the CORBA specification and that the one tested is that supplied with the J2SDK 5.0. It is reasonable to suggest that commercial ORBs may be better tuned for performance than this implementation and that the call time could be reduced more in line with the other systems that employ binary protocols. While the middleware systems that employ binary protocols outperform the RRT, the binary approach has disadvantages in

that it does not provide the meta-data and opportunities for validation that XML does. SOAP can be considered the safer approach as the data is self-describing and less prone to problems with type safety [33].

SOAP-based systems offer a high degree of interoperability and a transport protocol with multiple advantages over binary approaches, as discussed above. Of the SOAP systems tested, the RRT prototype performed best, indicating that the advantage provided by the RRT's approach to application creation need not come at the cost of degraded performance.

6.5 Transmission Policy Framework

The transmission policy framework allows programmers to control the parameter-passing semantics employed when remote methods are called. There are four kinds of transmission policy rules that can be specified:

- *Method policy rules* are associated with methods. They specify how all the arguments to the methods are marshalled.
- *Return policy rules* are also associated with methods. They control how the return values are marshalled.
- *Argument policy rules* are associated with individual method arguments. They indicate how particular arguments within method signatures are marshalled.
- *Class policy rules* are associated with classes rather than methods. They indicate how instances of particular classes are marshalled.

The transmission policy framework needs to optimize lookup of policy rules, as these rules must be checked every time an object is marshalled. The RRT trades quicker lookup for increased rule addition and removal time since it is expected that rules will be looked up much more often than they are altered. The transmission policy manager uses two associative stores to hold rules. One contains class policy rules alone and the other contains all rules related to methods, namely, method policy rules, return policy rules and argument policy rules.

Programmers also use the transmission policy manager to manage caching policy rules, which indicate the fields and methods that are cached in particular remote references. Caching rules do not require any evaluation as there cannot be contention between separate rules. These rules are therefore recorded in simple associative stores keyed using classes, which are not examined in detail here.

6.5.1 Flow of Control during Policy Evaluation

To determine how a particular object should be marshalled, the RRT instance queries the transmission policy via the *evaluateTransmissionPolicy()* method provided by the *IRafdaRunTimePrivate* interface, which is shown in Figure 6.24. The object to be marshalled may be an argument, a return value or an object within the closure of an argument or return value.

```
PassingMechanism evaluateTransmissionPolicy(
      Class objectClass,
      Method methodIdentifier,
      int argumentNumber,
      int depth,
      boolean isReturnValue);
```

Figure 6.24: The *evaluateTransmissionPolicy()* method used during marshalling.

This method takes the following arguments:

- The class of the object being marshalled.
- The identity of the method being called.
- The identity of the argument being marshalled.
- The depth of object in the argument/return value closure.
- A Boolean indicating whether this is an argument or return value.

The algorithm that evaluates transmission policy for a particular object considers all policy rules that are associated with the specified class, method and argument. Rules that are not applicable at the current depth are ignored. The rule that is chosen by the transmission policy manager is called the *dominant rule*. The dominant rule defines which parameter-passing mechanism is applied to the specified object.

The dominant rule is chosen from all applicable rules based on rule priority and precedence (argument policy rules override method policy rules of the same priority, which override class policy rules of the same priority) as follows:

1. The class policy rule with the highest priority that is associated with the object's class is found. This rule becomes the provisional dominant rule.
2. Either the method policy rules or the return policy rules are evaluated depending on the *isReturnValue* Boolean. The highest priority rule that is valid at the current depth is found. If this rule is of equal or higher priority than the

current provisional dominant rule, then it becomes the provisional dominant rule.

3. The highest priority argument policy rule that is valid at the current depth is found. If this rule is of equal or higher priority than the current provisional dominant rule, then it becomes the provisional dominant rule.
4. The provisional dominant rule becomes the dominant rule. It dictates the parameter-passing mechanism to use.

The data structures used to store rules are now described in the context of the following policy rules:

- **Class policy rule** associated with the *JChordNode* class indicating pass-by-reference with a priority of 3.
- **Class policy rule** associated with the *JChordNode* class indicating pass-by-value with a priority of 0.
- **Class policy rule** associated with the *Key* class indicating pass-by-migrate with a priority of 1.
- **Class policy rule** associated with the *Key* class indicating pass-by-value with a priority of 0.
- **Method policy rule** associated with the *lookup()* method indicating pass-by-reference to depth 2 with a priority of 3.
- **Method policy rule** associated with the *lookup()* method indicating pass-by-value to depth 0 with a priority of 1.
- **Return policy rule** associated with the *lookup()* method indicating pass-by-migrate to depth 0 with a priority of 3.
- **Return policy rule** associated with the *lookup()* method indicating pass-by-reference to depth 4 with a priority of 0.
- **Argument policy rule** associated with the second argument of *lookup()* method indicating pass-by-value to depth 2 with a priority of 3.

6.5.2 Class Policy Map

The *class policy map* associates each class with a linked list of the class policy rules associated with that class. Since each rule has a priority that determines its precedence over other rules, the linked list of rules is sorted into priority order, such that the highest priority rule is at the head of the list as shown in Figure 6.25. Since the RRT

does not permit two rules of equal priority to be associated with a single class, there is always a single rule associated with each class that is of higher priority than all other rules associated with the same class.

Obtaining the class policy rule associated with a particular class requires a single map lookup to obtain the head of the list. Insertion, modification or deletion of rules requires a list traversal to locate the specified rule. If no rules are associated with a particular class then no entry appears in this map.

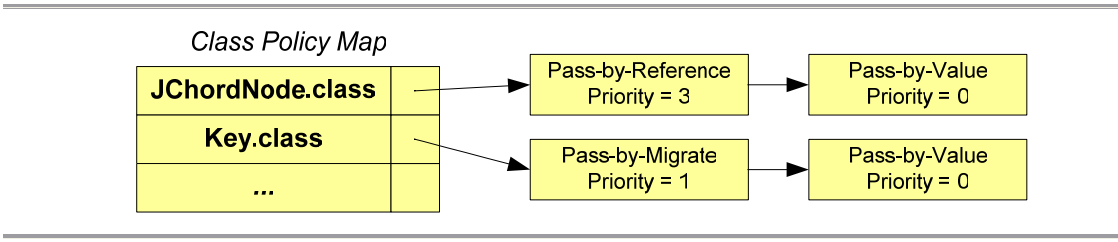


Figure 6.25: The transmission policy framework's data structure for storing class policy rules.

6.5.3 Method Policy Map

The *method policy map* associates a method with its method policy rules, return policy rules and argument policy rules. The method policy map associates each method with a secondary map called the *per-method map* as shown in Figure 6.26. The per-method map holds multiple linked lists of rules, sorted in priority order.

In the per-method map, method policy rules are associated with the key *"All"*, return policy rules are associated with the key *"Return"*, and argument policy rules are associated with keys based on their argument numbers, *"1"*, *"2"*, etc.

Lookup of a method policy, return policy or argument policy rule requires two map lookups to obtain the required linked list of rules. The first to obtain the per-method map associated with the required method and the second to obtain the link list of policy rules from this per-method map. The first rule in this linked list that is valid at the current depth is found by traversing the list.

For example, the policy rules shown are associated with the *lookup()* method. The method policy rules specify a pass-by-value policy to a depth of 2 with priority 3 then a pass-by-reference policy to a depth of 0 with priority 1. Up to a depth of 2, the pass-by-value rule at the head of the list is applied. Beyond this depth, the first rule is ignored as it is no longer valid and the next valid rule (dictating pass-by-reference be adopted) is followed.

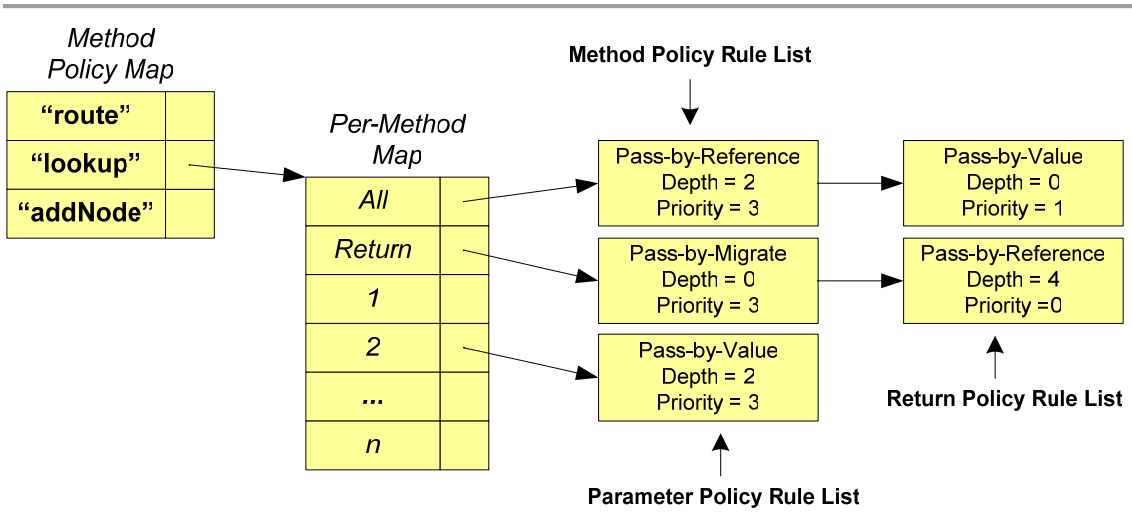


Figure 6.26: The data structure for storing method, return and argument policies.

6.5.4 Policy Evaluation Cost

The policy framework must be queried and the policy rules evaluated each time objects are marshalled, affecting remote method call cost. This cost is heavily dependent on the particular policy rules that are associated with the object to be marshalled. The transmission policy framework is an integral part of the RRT and so cannot be switched off under normal circumstances. To determine the cost of transmission policy evaluation, a special build of the RRT that employed only pass-by-reference semantics was created.

A test application that performed multiple calls to a remote method was created. This method took one argument and returned one return value, both by-reference. The test application was run using the specially built RRT with the transmission policy framework removed and again using the full RRT, using the test environment described in Section 6.4.6. In the former case, the special RRT was hard-coded to pass objects by-reference, and in the latter case, the transmission policy consisted of a method policy rule and a return policy rule stating that pass-by-reference semantics should be employed. The parameter-passing semantics were therefore the same for each run of the application.

The cost of a remote call when the policy evaluation phase was performed was around 2% to 3% greater than the cost of a remote call without the evaluation phase. The introduction of additional arguments has no effect on the proportionate cost of the policy evaluation phase as there is a one-to-one correspondence between the number of objects marshalled and the number of transmission policy evaluations performed.

From the perspective of the RRT prototype, this is considered a reasonable trade-off in execution time for the benefits afforded by the transmission policy in its implementation. Even incurring the cost of policy evaluation, the RRT prototype outperforms the nearest comparable rival by a considerable margin.

6.6 Distribution Policy Framework

The distribution policy framework allows the dynamic specification of policies to control object placement when remote instantiation and object migration operations are performed. The distribution policy manager is used to obtain references to factories (used for remote instantiation) and migration controllers (used for migration). Each factory and migration controller is associated with a policy object, which is an instance of a programmer-defined policy class that is used to determine the distribution policy.

Programmers can create custom re-implementations of the factories, migration controllers, policy objects and feedback objects provided by the distribution policy framework. Thus, implementation details concerning these components can be found in Chapter 5, though are briefly summarized here:

- The distribution policy manager, which is accessible through the *IRafdaRunTime* and *IRafdaRunTimeRemote* interfaces. There is one distribution policy manager per address-space that provides two associative stores, one of which maps classes to their associated factory objects and one of which maps classes to their migration controllers.
- Factories, which perform all policy-based object instantiation. The default factory class employs reflective techniques to instantiate objects in both the local and remote RRT instances. Programmers can create custom factory implementations in order to modify factory behaviour.
- Migration controllers, which perform all policy-based migration. A default migration controller class is provided but programmers are free to implement multiple different custom migration controllers.
- Policy objects, which make the object placement decisions.
- Feedback objects, which allow policies in multiple address-spaces to exchange information. Feedback classes are also customizable.

The overhead incurred by policy evaluation is heavily dependent on the policy implementation, which is provided by application developers. However, in order to gauge

whether there is any additional cost incurred by using the distribution policy framework mechanism instead of performing operations in remote RRT instances directly, the following test was performed using the previously described test environment. Two clients, each of which instantiated 5000 objects in a remote RRT instance, were created in order to determine the cost of remote instantiation.

One client performed remote instantiation directly in a particular remote RRT instance using the *instantiateAndExpose()* method provided in the *IRafdaRunTimeRemote* interface. The other performed the same remote instantiation operations but did so via the distribution policy framework and a "single RRT instance" policy. This policy performed minimal computation in order that the cost of instantiation through the framework could be directly compared to the cost of instantiation performed directly by programmers.

The cost of instantiating the first remote object (cold instantiation) was considerably greater in both clients than that of instantiating further objects (hot instantiation). In both clients, hot instantiation took the same time, around 19ms, indicating that the distribution policy framework introduces no significant overhead, outside of policy evaluation, to the cost of performing the remote operation. Cold instantiation however, was around five times slower when using the framework, caused by the need to initialize the distribution policy framework components. In both clients, cold instantiation was around two to three orders of magnitude slower than hot instantiation, as the cost of RRT initialization was also incurred.

It can be concluded that the cost inherent in using the distribution policy framework is heavily dependent on the implementation of the policy classes. The onus is therefore on the programmers creating these policy classes to do so efficiently.

6.7 Conclusion

This chapter has examined the implementation of the prototype RRT and evaluated it quantitatively. The RRT instances expose themselves to remote access through multiple interfaces to provide functionality to programmers and RRT instances, both local and remote. Functionality allowing programmers to obtain references to remote objects, perform remote instantiation and migrate objects between address-spaces has been implemented using the basic remote method invocation mechanism provided by the RRT.

Service adaptors allow the middleware system to attach to arbitrary application objects. Serializers and deserializers permit the transmission of arbitrary objects across the network by-value. Per-class customized implementations of the service adaptors, serializers and deserializers can be generated and compiled automatically by the RRT. These per-class implementations avoid the use of reflection and are optimized to work with the classes in each particular distributed application.

Proxy objects provide both conventional proxy behaviour, allowing remote method calls to be performed transparently, and wrapper behaviour, allowing the implementation of migration. Combined with *RafdaIOR* objects, the proxy objects provide a complete remote reference scheme, allowing any object to be accessed remotely or passed by-reference.

Though the RRT prototype has not been optimized for speed, the cost of remote method calls is lower than the equivalent industry-standard systems. This indicates that the benefits provided by the RRT in terms of flexibility and separation of concerns can be achieved without incurring additional expense over comparable systems.

The structure of the transmission policy framework was examined. It is designed to allow quick lookup of policy rules to optimize the object marshalling process. It was shown that the transmission policy framework is an integral part of the RRT that simplifies the implementation of middleware features such as remote instantiation and migration. Further, the additional cost incurred querying the transmission policy framework at marshalling time is low.

Finally, the implementation of the distribution policy framework was examined. Though this framework can be customized extensively by programmers, it was shown that the cost inherent in its use lies in the policy evaluation phase. Programmers trade policy complexity for speed of evaluation when creating distribution policies.

The RRT prototype is a complete implementation of the middleware system designed in Chapter 5. It is currently publicly available and is in use as a platform on which to carry out research into peer-to-peer systems [27], resilient Web Services [25] and automated application deployment.

Chapter 7

Conclusion

This chapter concludes the thesis by summarizing the work carried out and the contribution made. A synopsis of possible future work is also provided.

7.1 Conclusion

Middleware augments operating systems and network infrastructure to assist in the creation of distributed applications in a heterogeneous environment. Multiple approaches to middleware exist, though this thesis has focussed on a Distributed Object Model (DOM) approach. DOMs provide a programming model similar to that adopted when implementing non-distributed object-oriented applications and allow flexibility over the extent to which inter-address-space communication is visible to programmers.

Current middleware systems exhibit five main limitations:

1. Programmers are forced to make decisions early in the design process about which types of application component may participate in inter-address-space communication. Applications are therefore inflexible to static changes in their distribution.
2. Applications created using existing middleware systems are inflexible to dynamic changes in their distribution and cannot adapt to changes in the underlying distributed systems or in the applications themselves.
3. The creation of code to handle inter-address-space communication is complex, introducing additional points of potential failure into the software engineering process.
4. It is difficult to understand and maintain distributed applications because middleware systems may force an unnatural encoding of application-level semantics. Application classes are forced to meet particular semantic requirements, hampering the reuse in a distributed context of code written without support for distribution. Parameter-passing semantics are fixed statically and are inflexible, again limiting code reuse and preventing programmers from performing optimizations that take advantage of the distributed nature of applications.
5. It is difficult to control the way in which objects are distributed among available address-spaces. Programmers must adopt ad-hoc approaches to the definition of distribution policy if application logic and distribution are to be separated.

This thesis defined a taxonomy of current middleware systems. First generation RPC and DOM systems were described and shown to exhibit all five of these problems.

Second generation DOM systems that tackle these limitations with varying success were investigated. It was shown that all of these second generation DOMs exhibit some or all of these problems.

There are four roles in which programmers may employ middleware systems:

- To create new distributed applications.
- To introduce distribution into existing applications.
- To deploy services to remote clients.
- To integrate applications with legacy systems.

Each role makes different demands of a middleware system and none of the existing systems are flexible enough to be applied in all circumstances. Systems such as CORBA [8], Java RMI [9], JavaParty [62] and ProActive [81] expose the distributed nature of applications to programmers. Explicit support for remote access must be provided in application classes but it is difficult to introduce distribution into an existing application using these systems without extensive engineering effort. Systems such as J-Orchestra [65] and Pangaea [70] perform automatic application transformation to create distributed applications in which local calling semantics are preserved. By hiding inter-address-space communication completely, these systems cannot be employed to create applications that adopt different semantics, and so are suitable for developing only certain kinds of distributed application.

This thesis defined the requirements that must be fulfilled by a third generation middleware system, based on an evaluation of current systems. The design and implementation of the RAFDA Run-Time (RRT), a middleware system that meets these requirements, were described. The RRT allows programmers to trade-off the simplicity attained through the concealment of inter-address-space communication against the flexibility realized by exposing it to fine-grained programmer control. The RRT aims to provide intuitive distributed application semantics that reflect non-distributed semantics, allowing programmers to ignore whichever distribution-related aspects are of no concern.

The RRT allows inter-address-space communication to be concealed or exposed as required on a per-application basis. In order to conceal the distributed nature of applications, the RRT allows the exposure of arbitrary objects to remote access as Web Services. A remote reference scheme is introduced, allowing instances of any class to be passed across the network by-reference or by-value. This provides a similar degree of abstraction over the network as object-based Distributed Shared Memory systems. The

RRT ensures that non-distributed semantics are preserved in applications by default, unless programmers explicitly alter them. First generation systems cannot provide this degree of transparency because programmers are forced to meet stringent semantic requirements in order to support remote access.

Some second generation systems, such as JavaParty [62], ProActive [81] and Do! [68], simplify the engineering process by automatically generating distribution-related ancillary code but cannot support remote access to arbitrary objects. Consequently, programmers must adapt applications to the middleware.

The RRT provides fully transparent inter-address-space communication while allowing programmers, where appropriate, to take advantage of application-specific knowledge. The RRT allows inter-address-space communication to be exposed and controlled in several ways:

- Remote types can be associated with exposed objects in order to control which methods are exposed to remote access. Remote types provide multiple views over exposed objects and permit programmers to allow information hiding in distributed applications.
- Applications can be initially created without concern for failure, then extended to provide application-specific error handling as required.
- Multiple object lifetime policies are supported and managed on a per-RRT-instance basis. The RRT can ensure that exposed objects are never collected, are collected when no longer locally referenced, or collected when not accessed within a programmer-defined lease time.
- Objects can be instantiated in remote address-spaces.
- Objects can migrate between address-spaces.
- The parameter-passing semantics applied when remote methods are called can be controlled dynamically.
- Distribution policy can be associated with applications in a flexible manner.

The RRT is unique in its ability to provide abstraction over the network without removing control from programmers. Control over the parameter-passing semantics employed in remote calls is allowed to a degree that is not possible using traditional systems, due to limitations in the mechanisms that support remote access and the restrictions placed on application semantics. The transmission policy framework is used to control parameter-passing semantics and object caching policies independently of

application source. Transmission policy can be defined on a per-argument, per-method, per-return-value or per-class basis. This ensures that the functional logic of an application class is not affected by considerations related to the context in which instances of the class will be deployed. Components need not be designed for particular deployment environments, permitting reuse in a distributed context of components that were designed without support for remote access.

Caching policy rules allow programmers to treat the pass-by-reference and pass-by-value mechanisms as two ends of a spectrum. Remote references that cache fields and methods can be created, reducing the need to perform remote method calls and permitting remote references to remain partially usable even when connectivity is lost. Programmers can take advantage of application knowledge to trade-off by-reference and by-value semantics or to cache objects that are immutable in a particular application.

The RRT is novel both in its support for flexible parameter-passing mechanism and its provision of a policy specification mechanism that allows programmers to define application semantics dynamically.

Distribution policies can be defined through the distribution policy framework to govern the placement of objects when remote instantiation and migration operations are performed. By deferring placement decisions to the framework, programmers can create applications with flexible distribution boundaries. None of the first generation systems and few of the second generation systems support remote object instantiation or migration. Those that do, provide only partial solutions to the problem of policy specification, by limiting the expressiveness of policies and by allowing no flexibility in the granularity at which distribution policies are applied, e.g. per-class, per-constructor call.

Using the RRT, policies can be arbitrarily complex and may make use of application context information in order to apply policies at finer-than-class granularity. Policies can take advantage of tools external to the RRT, such as system or application profiling tools, to control application distribution in a completely flexible manner. The distribution policy framework provides location transparency, meaning that applications can adapt dynamically to changes in the underlying distributed systems or can modify their own distributions, for example, to minimize remote method calls.

The RRT has been evaluated using the JChord case study, which consists of a peer-to-peer overlay network implementation and a distributed object store built on top of this network. The JChord case study places a number of requirements on middleware

systems that cannot be fully met by conventional systems but are fulfilled by the RRT. JChord was originally developed as a non-distributed application and it was shown that the unique properties of the RRT allowed an isomorphic distributed version to be created without modifications to the JChord implementation classes. Distribution-related concerns such as error handling and the caching and replication of state were handled late in the development process. JChord is a research tool and its requirements change often, as research goals change. Using a conventional middleware system, changes to the distribution boundaries in JChord would require re-engineering of the application. Using the RRT, the programmer need only modify the JChord application logic and can rely on the middleware to accommodate these changes.

A prototype of the RRT design has been implemented and is currently publicly available. This prototype is in use as a development platform for research into peer-to-peer systems, resilient Web Services and automated application deployment. A mechanism to allow the RRT to connect to arbitrary application objects is required in order to expose objects to remote access. Serializers and deserializers that can handle instances of any application class are necessary, as are proxy objects that preserve the abstraction over the inter-address-space communication presented by the RRT.

The RRT employs generative programming techniques in three main areas:

- The creation of service adaptors that allow the RRT infrastructure to attach to application objects.
- The creation of serializers and deserializers that can handle instances of arbitrary classes.
- The generation of proxy classes to allow the implementation of remote references that are interchangeable with local references.

The RRT can automatically generate and compile per-class implementations of service adaptors, serializers, deserializers and proxy classes. Per-class implementations avoid the cost of reflection at run-time but incur the one-time cost of code generation. Generated code may be cached across multiple runs of the application to obviate the need for re-generation.

Each RRT instance in a distributed system exposes itself to remote access. RRT instances can therefore provide functionality to applications and other RRT instances in remote address-spaces using the same mechanisms that provide connectivity in distributed applications. The RRT implementation makes use of its own underlying

functionality to abstract over the network allowing the provision of sophisticated middleware functionality, such as remote instantiation of objects and migration, with minimal programmer effort. The RRT prototype is as extensible and maintainable as any other distributed application created using the RRT since code to perform inter-address-space communication does not permeate application logic.

7.2 Future Work

The RRT has succeeded in providing next generation middleware functionality and, as a consequence, has opened up interesting new research issues. Given mechanisms to provide transparent inter-address-space communication and control over object placement, the RRT could be used as a basis for developing support for dynamic re-distribution of applications in a fully automatic manner. Programmers could adopt an intentional programming model, in which they described the high level non-functional requirements of applications in terms of availability, response time, maximal permitted remote call time and so on. Using autonomic management tools in combination with meta-level policy rules that define how the programmers' requirements can be achieved, the RRT could automatically control and modify the distribution policies that were applied to the applications.

This model could be extended to investigate whether it is possible to capture a set of properties that is universally desirable in all distributed applications. It might be possible to define a set of definitive meta-level rules that could be used to completely remove the need for programmers to make distribution-related decisions. Programmers would provide only a non-distributed application to the middleware system, which would automatically discover machines in the distributed system and perform negotiation of resources. Each machine could be profiled to determine how best to distribute the application based on these profiles and meta-level rules. The use of compliant architectures [93, 94] that accommodate the needs of particular applications provide an alternative approach to dynamic profiling, as the distributed system could adapt to the needs of each application.

Part of the complexity inherent in implementing distributed applications results from the limitations of the industry standard programming languages used. These languages do not contain implicit support for distribution, hence the necessity of middleware systems. By subsuming the functionality provided by the RRT directly into

the programming language, the boundary between language and middleware system could be dissolved to provide a single programming model that is applicable to both distributed and non-distributed application development.

7.3 Finally

This thesis has described the requirements, design and implementation of a third generation middleware system. The hypothesis investigated states:

> *A middleware system that provides control over the extent to which inter-address-space communication is exposed to programmers aids the creation, maintenance and evolution of distributed applications.*

The RRT provides novel functionality that allows control over the extent to which inter-address-space communication is exposed. It is a middleware system that adapts to the needs of applications, rather than forcing distributed applications to adapt to the needs of the middleware system, with direct benefits for programmers. The RRT provides a solid foundation on which to develop the next generation of distributed applications.

Appendix A Glossary

*Terminology specific to the RAFDA Run-Time system is marked with a *.*

Deserializer	Creates objects from serialized representations.
Distributed Application	An application that runs in a distributed system.
Distributed System	A collection of distinct, spatially separate processes that communicate by exchanging messages [1].
Distribution Policy*	The policy controlling object placement when remote object instantiation and migration operations are performed.
DOM	Distributed Object Model.
Marshalling	The conversion of a method call into an invocation request. Part of the marshalling process includes object serialization.
Middleware System	Software that augments operating systems and network infrastructure to make the creation of distributed applications in a heterogeneous environment easier.
Migration	The movement of objects between address-spaces without the loss of referential integrity.
Pass-By-Migrate	Parameter-passing mechanism employed when remote methods are called in which arguments are migrated to the remote address-space.
Pass-By-Reference	Parameter-passing mechanism that passes remote references to arguments to the remote address-space.
Pass-By-Value	Parameter-passing mechanism that copies arguments to the remote address-space.
Proxy Object	A local handle on a remote object.
RAFDA*	Reflective Application Framework for Distributed Architectures.
Remote Reference	A reference to an object in another address-space.
RMI	Remote Method Invocation.
RPC	Remote Procedure Call.
RRT Instance*	The RRT infrastructure present in a single address-space.
RRT*	The RAFDA Run-Time middleware system.

Serializer	Creates serial data representations of objects.
Service Adaptor*	Provides skeleton functionality in the RRT allowing it to connect to arbitrary application objects.
Skeleton	The part of a middleware system that un-marshals incoming remote method calls and performs invocations on local objects.
Transmission Policy*	The policy controlling the parameter-passing mechanisms and caching semantics employed when remote methods are called.
Un-marshalling	The conversion of an invocation request into a method call. Part of the un-marshalling process includes object deserialization.
Wrapper	A proxy object associated with a local object.

Appendix B Policy File XML Schema

Transmission Policy Configuration File Schema

```
<?xml version="1.0" encoding="UTF-8" ?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="transmissionPolicy">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="argumentPolicy" minOccurs="0"
          maxOccurs="unbounded" />
        <xs:element ref="methodPolicy" minOccurs="0"
          maxOccurs="unbounded" />
        <xs:element ref="returnPolicy" minOccurs="0"
          maxOccurs="unbounded" />
        <xs:element ref="classPolicy" minOccurs="0"
          maxOccurs="unbounded" />
        <xs:element ref="cachedField" minOccurs="0"
          maxOccurs="unbounded" />
        <xs:element ref="cachedMethod" minOccurs="0"
          maxOccurs="unbounded" />
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="argumentPolicy">
    <xs:complexType>
      <xs:all>
        <xs:element ref="method" />
        <xs:element name="argumentNumber" type="xs:integer" />
        <xs:element ref="paramPassingMechanism" />
        <xs:element name="depth" type="xs:integer" />
        <xs:element name="priority" type="xs:integer" />
      </xs:all>
    </xs:complexType>
  </xs:element>
  <xs:element name="methodPolicy">
    <xs:complexType>
      <xs:all>
        <xs:element ref="method" />
```

```
        <xs:element ref="paramPassingMechanism" />
        <xs:element name="depth" type="xs:integer" />
        <xs:element name="priority" type="xs:integer" />
     </xs:all>
    </xs:complexType>
  </xs:element>
  <xs:element name="returnPolicy">
    <xs:complexType>
      <xs:all>
        <xs:element ref="method" />
        <xs:element ref="paramPassingMechanism" />
        <xs:element name="depth" type="xs:integer" />
        <xs:element name="priority" type="xs:integer" />
      </xs:all>
    </xs:complexType>
  </xs:element>
  <xs:element name="classPolicy">
    <xs:complexType>
      <xs:all>
        <xs:element name="className" type="xs:string" />
        <xs:element ref="paramPassingMechanism" />
        <xs:element name="priority" type="xs:integer" />
      </xs:all>
    </xs:complexType>
  </xs:element>
  <xs:element name="cachedField">
    <xs:complexType>
      <xs:all>
        <xs:element name="className" type="xs:string" />
        <xs:element name="fieldName" type="xs:string" />
      </xs:all>
    </xs:complexType>
  </xs:element>
  <xs:element name="cachedMethod">
    <xs:complexType>
      <xs:all>
        <xs:element ref="method" />
     </xs:all>
    </xs:complexType>
  </xs:element>
  <xs:element name="paramPassingMechanism">
```

```
    <xs:simpleType>
      <xs:restriction base="xs:string">
        <xs:enumeration value="byreference" />
        <xs:enumeration value="byvalue" />
        <xs:enumeration value="bymigrate" />
      </xs:restriction>
    </xs:simpleType>
  </xs:element>
  <xs:element name="method">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="className" type="xs:string" />
        <xs:element name="methodName" type="xs:string" />
        <xs:element name="argumentType" type="xs:string" minOccurs="0"
          maxOccurs="unbounded" />
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Transmission Policy Configuration File Example

This XML defines the following transmission policy rules:

- An argument policy rule associated with the first argument of the method *IDataStorePOP.store(Object objectToStore, boolean storeReference)* that specifies a by-migrate policy to depth 2 and is of priority 1.
- A method policy rule associated with the method *IDataStoreInternal.put(Key key, Object object)* that specifies a by-reference policy to depth 0 and is of priority 0.
- A return policy rule associated with the method *IDataStoreInternal.get()* that specifies a by-value policy to depth 4 and is of priority 3.
- A class policy rule associated with the *Key* class that specifies a by-value policy and is of priority 0.
- A caching rule that caches the *key* field with accessor *getKey()* and *setKey(Key k)* in instances of the *Chord* class.
- A caching rule that caches the *printKeyInfo()* method in instances of the *Chord* class.

```
<?xml version="1.0" encoding="UTF-8"?>
<transmissionPolicy>
  <argumentPolicy>
    <method>
      <className>IDataStorePOP</className>
      <methodName>store</methodName>
      <argumentType>java.lang.Object</argumentType>
      <argumentType>boolean</argumentType>
    </method>
    <argumentNumber>1</argumentNumber>
    <paramPassingMechanism>bymigrate</paramPassingMechanism>
    <depth>2</depth>
    <priority>1</priority>
  </argumentPolicy>
  <methodPolicy>
    <method>
      <className>IDataStoreInternal</className>
      <methodName>put</methodName>
      <argumentType>Key</argumentType>
```

```
      <argumentType>java.lang.Object</argumentType>
    </method>
    <paramPassingMechanism>byreference</paramPassingMechanism>
    <depth>0</depth>
    <priority>0</priority>
  </methodPolicy>
  <returnPolicy>
    <method>
      <className>IDataStoreInternal</className>
      <methodName>get</methodName>
    </method>
    <paramPassingMechanism>byvalue</paramPassingMechanism>
    <depth>4</depth>
    <priority>3</priority>
  </returnPolicy>
  <classPolicy>
    <className>Key</className>
    <paramPassingMechanism>byvalue</paramPassingMechanism>
    <priority>0</priority>
  </classPolicy>
  <cachedField>
    <className>Chord</className>
    <fieldName>key</fieldName>
  </cachedField>
  <cachedMethod>
    <method>
      <className>Chord</className>
      <methodName>getKey</methodName>
    </method>
  </cachedMethod>
  <cachedMethod>
    <method>
      <className>Chord</className>
      <methodName>setKey</methodName>
      <argumentType>Key</argumentType>
    </method>
  </cachedMethod>
  <cachedMethod>
    <method>
      <className>Chord</className>
      <methodName>printKeyInfo</methodName>
```

```
    </method>
  </cachedMethod>
</transmissionPolicy>
```

Appendix C RRT Configuration Options

RRT configuration is described in Chapter 5 and allows control over various aspects of system behaviour. These properties can be set using the *setProperty()* method of the *IRafdaRunTimeConfig* interface by specifying a property name and associated value. The following shows a full list of all configurable properties organized into groups, with permitted and default values shown.

System Configuration

firewallAllowedAddresses

Permitted Values: Semi-colon separated list of IP addresses and partial IP addresses

Default: Connections from any address permitted

Indicates a list of valid hosts from which incoming connections can be accepted.

networkInterface

Permitted Values: IP address/hostname

Default: Result of InetAddress.getLocalHost()

Indicates the network interface to which the RRT instance should bind.

port

Permitted Values: 1-65535

Default: 5001 upwards

Indicates which port the RRT instance should bind to when accepting socket connections. If no port is specified the RRT will use the first free port in the range 5001 upwards.

socketTimeout

Permitted Values: integer

Default: Default platform socket timeout

Indicates how long a socket will wait for a response from a remote RRT before determining that the host is off-line.

Handling Static Members

makeRootRRTInstance

Permitted Values: Boolean

Default: false

If true, this RRT instances acts as the root RRT instance that manages access to static members.

If false, this RRT instance is not the root RRT instance.

setRootRRTInstance

Permitted Values: Socket address

Default: None

Indicates the socket address of the root RRT instance.

Code Generation

autoGenerateServiceAdaptors

Permitted Values: Boolean

Default: false

If true, service adaptors are generated and compiled dynamically on a per-application-class basis.

If false, all objects are exposed using the generic reflective service adaptor.

autoGenerateSerializersDeserializers

Permitted Values: Boolean

Default: false

If true, per-class serializers and deserializers are generated and compiled dynamically on a per-application-class basis.

If false, all serialization and deserialization is performed using the generic reflective serializer/deserializer.

cacheGeneratedProxies

Permitted Values: Boolean

Default: false

If true, automatically generated proxy classes are cached locally for reuse during subsequent runs of the application.

If false, automatically generated proxy classes exist only for the lifetime of the JVM running the RRT instance.

cacheGeneratedServiceAdaptor

Permitted Values: Boolean

Default: false

If true, automatically generated service adaptors are cached locally for reuse during subsequent runs of the application.

If false, automatically generated service adaptors exist only for the lifetime of the JVM running the RRT instance.

cacheGeneratedSerializersDeserializers

Permitted Values: Boolean

Default: false

If true, automatically generated per-class serializer/deserializers are cached locally for reuse during subsequent runs of the application.

If false, automatically generated per-class serializer/deserializers exist only for the lifetime of the JVM running the RRT instance.

deleteAllCachedCode

Permitted Values: Boolean

Default: false

If true, all cached per-class service adaptors, serializers, deserializers and proxy classes are deleted at start-time.

If false, cached code is re-used

Access Control

allowNonPublicMethodAccess

Permitted Values: Boolean

Default: false

If true, all clients are allowed access to the non-public methods of exposed objects.

If false, clients may only access non-public methods if the local protection semantics permit it.

allowBrowsingOfExposedObjects

Permitted Values: Boolean

Default: false

If true, the service-specific web pages, accessible via a web browser, display information about the real classes of exposed objects and their current state.

If false, this information is not displayed.

allowRemoteInstantiation

Permitted Values: Boolean

Default: false

If true, remote RRT instances can create objects in this RRT instance.

If false, remote instantiation is not permitted.

allowMigration

Permitted Values: Boolean

Default: false

If true, remote RRT instances can migrate objects to this RRT instance.

If false, migration is not permitted.

throwDistributionRelatedExceptions

Permitted Values: Boolean

Default: false

If true, distribution-related exceptions occurring during remote method calls are wrapped in *RafdaRuntimeExceptions* and thrown back to clients.

If false, distribution-related exceptions occurring during remote method calls are not thrown back to clients. The RRT logs the exception and returns default values.

Memory Management

memoryManagement

Permitted Values: "none", "manual", "automatic"

Default: automatic

If “none”, the RRT holds weak references to exposed objects and so objects will be garbage collected when they are no longer referenced locally. Once objects are collected, the associated Web Services will be shut down and extant remote references will become invalid.

If “manual”, the RRT (strongly) references all exposed objects and will continue to do so until the programmer manually shut down the services.

If “automatic”, the RRT (strongly) references all the objects it exposes. It assumes that any exposed objects not remotely accessed within a programmer-defined lease time are no longer remotely referenced. These services will be shut down and any extant remote references will become invalid.

References

[1] Lamport, L, *Time, Clocks, and the Ordering of Events in a Distributed System.* Communications of the ACM, 1978. 21(7): p. 558-565.

[2] Message Passing Interface Forum, *MPI: A Message-Passing Interface Standard*. 1995.

[3] Geist, A, A, B, Dongarra, J, Weicheng, J, Manchek, R, and Sunderam, V, *PVM: Parallel Virtual Machine. A Users' Guide and Tutorial for Networked Parallel Computing.* 1994: MIT Press.

[4] Hapner, M, Burridge, R, Sharma, R, Fialli, J, and Stout, K, *Java Message Service*. 2002, Sun Microsystems.

[5] Birrell, A D and Nelson, B J, *Implementing Remote Procedure Calls.* ACM Transactions on Computer Systems, 1984. 2(1).

[6] White, J E. *A High-Level Framework for Network-Based Resource Sharing (RFC 707)*. in *Proc. National Computer Conference 76*. 1976.

[7] Sun Microsystems. *RPC: Remote Procedure Call Protocol specification: Version 2 (RFC 1057)*. 1988. http://www.faqs.org/rfcs/rfc1057.html

[8] Object Management Group, *Common Object Request Broker Architecture: Core Specification 3.0.3*. 2004.

[9] Sun Microsystems, *Java™ Remote Method Invocation Specification*. 1996-2005.

[10] Obermeyer, P and Hawkins, J, *Microsoft .NET Remoting: A Technical Overview*. 2001, Microsoft Corporation.

[11] Gelernter, D, *Generative communication in Linda.* ACM Transactions on Programming Languages and Systems, 1985. 7(1): p. 80-112.

[12] Freeman, E, Hupfer, S, and Arnold, K, *JavaSpaces: Principles, Patterns, and Practice*. 1999: Pearson Education.

[13] Li, K and Hudak, P, *Memory Coherence in Shared Virtual Memory Systems.* ACM Transactions on Computing Systems, 1989. 7(4): p. 321-359.

[14] Yu, W and Cox, A, *Java/DSM: A platform for heterogeneous computing.* Concurrency: Practice & Experience, 1997. 9(11).

[15] Bal, H and Kaashoek, F. *Object Distribution in Orca using Compile-Time and Run-Time Techniques*. in *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 93)*. 1993.

[16] Coulouris, G F and Dollimore, J, *Distributed Systems, Concept and Design*. 1988, Wokingham: Addison-Wesley.

[17] Gosling, J, Joy, B, and Steele, G, *The Java™ Language Specification*. 1996: Addison-Wesley.

[18] Ecma International, *C# Language Specification, 3rd edition (ECMA Standard 334)*. 2005.

[19] Stroustrup, B, *The C++ Programming Language (3rd edition)*. 1997: Addison Wesley Longman.

[20] Dearle, A and Kirby, G N C, *Reflective Application Framework for Distributed Architectures*. 2001, EPSRC GR/R51872.

[21] Rebón Portillo, Á J, Walker, S, Kirby, G N C, and Dearle, A. *A Reflective Approach to Providing Flexibility in Application Distribution*. in *2nd International Workshop on Reflective and Adaptive Middleware, ACM/IFIP/USENIX International Middleware Conference (Middleware 2003)*. 2003. Rio de Janeiro, Brazil: Pontifícia Universidade Católica do Rio de Janeiro.

[22] Stoica, I, Morris, R, Karger, D, Kaashoek, F, and Balakrishnan, H. *Chord: A Scalable Peer-To-Peer Lookup Service for Internet Applications*. in *ACM SIGCOMM 2001*. 2001. San Diego, CA, USA.

[23] Shapiro, M, *Structure and Encapsulation in Distributed Systems: the Proxy Principle.* IEEE Proc. 6th Intl. Conf. on Distributed Computing Systems, 1986: p. 198-204.

[24] Martin, P, Callaghan, V, and Clark, A. *High Performance Distributed Objects using Caching Proxies for Large Scale Applications*. in *International Symposium on Distributed Objects and Applications*. 1999.

[25] Norcross, S, Dearle, A, Kirby, G N C, and Walker, S M. *A Peer-To-Peer Infrastructure for Resilient Web Services*. in *IEEE International Workshop on Advanced Architectures and Algorithms for Internet Delivery and Applications (AAA-IDEA 2005)*. 2005. Orlando, Florida.

[26] Dabek, F, Zhao, B, Druschel, P, Kubiatowicz, J, and Stoica, I. *Towards a Common API for Structured Peer-to-Peer Overlays*. in *2nd International Workshop on Peer-to-Peer Systems (IPTPS '03)*. 2003. Berkeley, CA, USA.

[27] Kirby, G N C, Dearle, A, and Morrison, R, *Secure Location-Independent Autonomic Storage Architectures*. 2003, EPSRC GR/S44501/01.

[28] Open Group, *DCE 1.1: Remote Procedure Call*. 1997.

[29] ISO/IEC, *ISO Remote Procedure Call Specification*. 1991, ISO.

[30] Barkley, J, *Comparing Remote Procedure Calls*. 1993, NIST.

[31] Winer, D, *XML-RPC Specification*. 2003.

[32] Microsoft Corporation, *The Component Object Model Specification*. 1995.

[33] Lievens, D, *An Investigation into the Mechanisms Provided by CORBA to Preserve Strong Typing*. 2001, University of Glasgow.

[34] Pritchard, J, *COM and CORBA Side by Side: Architectures, Strategies, and Implementations*. 1999: Addison Wesley.

[35] Siegel, J, *CORBA fundamentals and programming*. 1996, New York: Wiley.

[36] MICO Project Group, *MICO (MICO is CORBA)*. 2004, ObjectSecurity Ltd.

[37] Baker, S, *CORBA Distributed Objects: Using Orbix*. 1997, Harlow, England ; Reading, Mass.: Addison-Wesley.

[38] IONA Technologies, *Orbix*. 2004.

[39] Sebesta, R, *Concepts of Programming Languages*. 6th ed. 2003: Addison Wesley Professional.

[40] Sun Microsystems, *Java 2 Platform Standard Edition 5.0*. 2004.

[41] Lindholm, T and Yellin, F, *The Java™ Virtual Machine Specification*. 1996: Addison-Wesley. 475.

[42] Birrell, A, Nelson, G, Owicki, S, and Wobber, E, *Network Objects.* Software - Practice and Experience, 1995. 25(S4): p. 87-130.

[43] Gamma, E, Helm, R, Johnson, R, and Vlissides, J, *Design Patterns*. 1995: Addison-Wesley Professional.

[44] Box, D, *Essential COM*. Addison-Wesley object technology series. 1998, Reading, Mass.: Addison Wesley.

[45] Thai, T and Lam, H Q, *.NET Framework Essentials*. 2001: O'Reilly.

[46] W3C, *Web Services Architecture*. 2004.

[47] Microsoft Corporation. *Shared Source Common Language Infrastructure 1.0 Release*. 2002. http://msdn.microsoft.com/net/sscli

[48] Mono Project. *Mono FAQ - General.* 2005. http://www.mono-project.com/FAQ:_General

[49] Sun Microsystems, *Enterprise JavaBeans Specification, Version 2.1*. 2003.

[50] OMG, *CORBA Component Model*. Vol. 3.0. 2004.

[51] Baker, S and Dobson, S, *Comparing service-oriented and distributed object architectures.* Proceedings of the International Symposium on Distributed Objects and Applications, 2005. LNCS 3760: p. p631-645.

[52] Vogels, W, *Web Services Are Not Distributed Objects.* IEEE Internet Computing, 2003. 7(6): p. p59-66.

[53] Web Services-Interoperability Organization (WS-I), *Basic Profile Version 1.1*. 2004.

[54] W3C, *SOAP Version 1.2 Part 0: Primer*. 2003.

[55] W3C, *SOAP Version 1.2 Part 1: Messaging Framework.* 2003.

[56] W3C, *SOAP Version 1.2 Part 2: Adjuncts.* 2003.

[57] Apache Software Foundation. *Apache Axis*. 2004. http://ws.apache.org/axis/

[58] Christensen, E, Curbera, F, Meredith, G, and Weerawarana, S, *Web Services Description Language (WSDL) 1.1*. 2001, W3C.

[59] JBoss Inc., *JBoss Enterprise Middleware System (JEMS)*. 2005.

[60] Hutchinson, N, Raj, R, Black, A, Levy, H, and Jul, E, *The Emerald Programming Language Report*. 1991, University of British Columbia: Vancouver BC, Canada.

[61] Jul, E, Levy, H, Hutchinson, N, and Black, A, *Fine-Grained Mobility in the Emerald System.* ACM Trans. on Computer Systems, 1998. 6(1): p. 109-133.

[62] Philippsen, M and Zenger, M, *JavaParty - Transparent Remote Objects in Java.* Concurrency: Practice and Experience, 1997. 9(11): p. 1225-1242.

[63] Smaragdakis, Y and J-Orchestra Group, *Application Partitioning without Programming (a White-Paper and Future Work Proposal)*. 2001, College of Computing, Georgia Tech.

[64] Smaragdakis, Y and Tilevich, E. *Automatic Application Partitioning: The J-Orchestra approach*. in *8th ECOOP workshop on Mobile Object systems*. 2002. Malaga.

[65] Tilevich, E and Smaragdakis, Y. *J-Orchestra: Automatic Java Application Partitioning*. in *European Conference on Object-Oriented Programming (ECOOP)*. 2002. Malaga.

[66] Liogkas, N, MacIntyre, B, Mynatt, E D, Smaragdakis, Y, Tilevich, E, and Voida, S, *Automatic Partitioning: A Promising Approach to Prototyping Ubiquitous Computing Applications.* IEEE Pervasive Computing, 2004(Special Issue on Building and Evaluating Ubiquitous System Software).

[67] Launay, P and Pazat, J-L, *A Framework for Parallel Programming in Java*. 1997, IRISA.

[68] Launay, P and Pazat, J-L, *Generation of distributed parallel Java programs*. 1998, IRISA.

[69] Spiegel, A. *Pangaea: An Automatic Distribution Front-End for Java*. in *Fourth IEEE Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS '99)*. 1999. San Juan, Puerto Rico: Springer-Verlag.

[70] Spiegel, A, *Automatic Distribution of Object-Oriented Programs*, in *FU Berlin, FB Mathematik und Informatik*. 2002.

[71] Busch, M, *Adding Dynamic Object Migration to the Distributing Compiler Pangaea.*, in *FB Mathematik und Informatik*. 2001, FU Berlin: Berlin.

[72] Chen, X. *Extending RMI to Support Dynamic Reconfiguration of Distributed Systems*. in *International Conference on Distributed Computing Systems*. 2002.

[73] Hunt, G C and Scott, M L, *The Coign Automatic Distributed Partitioning System*, in *Operating Systems Design and Implementation*. 1999. p. 187-200.

[74] Hunt, G C and Scott, M L, *Coign: Efficient Instrumentation for Inter-Component Communication Analysis*. 1997, Dept. of Computer Science, University of Rochester.

[75] Hunt, G C and Scott, M L. *A Guided Tour of the Coign Automatic Distributed Partitioning System*. in *2nd International Enterprise Distributed Object Computing Workshop (EDOC '98)*. 1998. San Diego, CA.

[76] Hunt, G C and Scott, M L. *Intercepting and Instrumenting COM Applications*. in *Proceedings of the 5th Conference on Object-Oriented Technologies and Systems (COOTS'99)*. 1999. San Diego, CA.

[77] Fahringer, T and Jugravu, A, *JavaSymphony: A new programming paradigm to control and to synchronize locality, parallelism, and load balancing for parallel and distributed computing.* Concurrency and Computation: Practice and Experience, 2002. 17(7-8): p. 1005 -1025.

[78] Fahringer, T. *JavaSymphony: A System for Development of Locality-Oriented Distributed and Parallel Java Applications*. in *IEEE International Conference on Cluster Computing, CLUSTER 2000*. 2000. Chemnitz, Germany.

[79] Testori, J, Winnisch, M, and Wohlmann, M, *JavaSymphony User Guide*. 2002.

[80] Lavender, R G and Schmidt, D, *Active Object - An Object Behavioral Pattern for Concurrent Programming*, in *Pattern Languages of Program Design 2*, J. Vlissides, J. Coplien, and N. Kerth, Editors. 1996, Addison-Wesley.

[81] Caromel, D, Klauser, W, and Vayssiere, J, *Towards Seamless Computing and Metacomputing in Java.* Concurrency Practice and Experience, 1998. 10(11-13): p. 1043-1061.

[82] Holder, O, Ben-Shaul, I, and Gazit, H. *Dynamic Layout of Distributed Applications in FarGo.* in *21st International Conference on Software Engineering (ICSE'99)*. 1999. Los Angeles, California.

[83] Abu, M and Ben-Shaul, I. *A Multi-Threading Model for Distributed Mobile Objects and Its Realization in FarGo*. in *21st International Conference on Distributed Computing Systems*. 2001. Mesa, Arizona.

[84] Holder, O and Gazit, H, *FarGo Programming Guide*. 1999, Electrical Engineering Dept, Technion - Israel Institute of Technology.

[85] Spiegel, A. *Objects by value: Evaluating the trade-off*. in *PDCN '98*. 1998. Brisbane, Australia: ACTA Press.

[86] Kirby, G N C, Walker, S M, Norcross, S, and Dearle, A. *A Methodology for Developing and Deploying Distributed Applications*. in *3rd International Working Conference on Component Deployment (CD 2005)*. 2005. Grenoble, France.

[87] Box, D, Ehnebuske, D, Kakivaya, G, Layman, A, Mendelsohn, N, Nielsen, H F, Thatte, S, and Winer, D, *Simple Object Access Protocol (SOAP) 1.1*. 2000, W3C.

[88] Birrell, A D, Eyers, D, Nelson, G, Owicki, S, and Wobber, E, *Distributed Garbage Collection for Network Objects*. 1993, DEC SRC.

[89] Kirby, G N C and Morrison, R, *Java Dynamic Compilation Package*. 1998, University of St Andrews.

[90] Kirby, G N C, Morrison, R, and Stemple, D W, *Linguistic Reflection in Java.* Software - Practice & Experience, 1998. 28(10): p. 1045-1077.

[91] Shapiro, M, Dickman, P, and Plainfossé, D. *Robust, Distributed References and Acyclic Garbage Collection*. in *11th ACM Symposium on Principles of Distributed Computing (PODC)*. 1992. Vancouver, Canada: ACM.

[92] Brittain, J and Darwin, I, *Tomcat: The Definitive Guide*, ed. B. McLaughlin. 2003: O'Reilly & Associates.

[93] Morrison, R, Balasubramaniam, D, Greenwood, R M, Kirby, G N C, Mayes, K, Munro, D S, and Warboys, B, *An Approach to Compliance in Software*

Architectures. IEE Computing & Control Engineering Journal, Special Issue on Informatics, 2000. 11(4): p. 195-200.

[94] Morrison, R, Balasubramaniam, D, Greenwood, R M, Kirby, G N C, Mayes, K, Munro, D S, and Warboys, B C, *A Compliant Persistent Architecture.* Software - Practice and Experience, Special Issue on Persistent Object Systems, 2000. 30(4): p. 363-386.