A Methodology for Developing and Deploying Distributed Applications

Graham N.C. Kirby, Scott M. Walker, Stuart J. Norcross and Alan Dearle

School of Computer Science, University of St Andrews,
North Haugh, St Andrews, Fife KY16 9SX, Scotland
{graham, scott, stuart, al}@dcs.st-and.ac.uk

Abstract. We describe a methodology for developing and deploying distributed Java applications using a reflective middleware system called RAFDA. We illustrate the methodology by describing how it has been used to develop a peer-to-peer infrastructure, and explain the benefits relative to other techniques. The strengths of the approach are that the application logic can be designed and implemented completely independently of distribution concerns, easing the development task, and that this gives great flexibility to alter distribution decisions late in the development cycle.

1 Introduction

This paper presents a methodology for developing and deploying distributed applications. This exploits many features of the RAFDA middleware system [1-4], the most significant of which is its ability to separate distribution concerns completely from the core application logic. The middleware allows any application object to be made remotely accessible. This means that any changes to distribution boundaries within the application do not require re-engineering of the application, making it easier to change the application's distribution topology. This separation of concerns simplifies the software engineering process to the programmer's advantage both when creating a new distributed application and when introducing distribution into an existing application.

In outline, the methodology involves three successive phases:

- The application is designed, implemented and tested without taking any account of how it will be distributed.
- Various mandatory details of distribution are defined, including how application objects should be partitioned across the network, which should be remotely accessible, and how they are initially connected.
- Other optional issues may be addressed—or may be ignored—including error handling of network-related failures, parameter passing semantics, and the insertion of monitoring probes.

Code written during the second and third phases is logically separated from the original application code written during the first phase; the original code executes unchanged, whether locally or distributed. Although the additional effort required to distribute the application is non-trivial, because the extra code resides in newly written classes rather than pervading the application logic, it is relatively straightforward to write, and to change at any time, including late in the development cycle.

2 Related Work

Industry-standard middleware systems—CORBA [5], Java RMI [6], Microsoft COM [7], Microsoft .NET remoting [8] and Web Services [9]—are complex, making the creation of distributed applications difficult and error-prone. Programmers must ensure that application classes supporting remote access correctly adhere to the particular rules of the middleware system in use, for example, extending certain base classes, implementing certain interfaces or handling distribution-related error conditions.

This affects inheritance relationships between classes and often prevents application classes from being remotely accessed if their super-classes do not meet the necessary requirements. At best, this forces an unnatural or inappropriate encoding of application semantics because super-classes are often required to be accessible remotely for the benefit of their sub-classes and, at worst, application classes that extend pre-compiled classes cannot be made accessible remotely at all.

The above systems all require programmers to follow similar steps in order to create the remotely accessible classes. Programmers must specify the interfaces between distribution boundaries then decide which classes will implement these interfaces. Thus classes are hard-coded at the source level to support remote accessibility; programmers must therefore know how the application objects will be distributed at run-time when defining classes—early in the design cycle.

The difficulties inherent in creating and configuring distributed applications are addressed by several second-generation middleware systems. These allow programmers to employ code transformation techniques to generate the distribution-related code automatically. J-Orchestra [10] and Pangaea [11] transform non-distributed applications into distributed versions based on programmer input. They perform static code analysis and employ tools to help programmers choose suitable partitions. Distributed versions of applications are automatically generated from the local versions and so the re-engineering process is simplified, making a trial and error approach to creating applications more feasible.

ProActive [12] and JavaSymphony [13] allow objects to be exposed to remote access dynamically. However, both subtly alter application threading semantics and force programmers to ensure referential integrity manually through their use of active objects [14]. This requires programmers to consider both application distribution and the middleware system's threading model at class creation time in order to ensure that thread safety is retained after objects are exposed to remote access or migrated to other address-spaces.

In all current middleware systems, the parameter-passing semantics employed during remote method calls are determined statically, often at design-time. Programmers cannot take advantage of run-time knowledge or application-specific information to alter these semantics dynamically. Generally, semantics are based on the remote accessibility of the application classes [6, 8] or defined in the classes explicitly [5].

3 The RAFDA Middleware System

By contrast with existing middleware systems, the RAFDA Run-Time [1-4] (RRT) permits arbitrary application objects to be dynamically exposed for remote access. Object instances are exposed as Web Services through which remote method invocations may be made. The RRT has four notable features that differentiate it from other middleware technologies:

1. The programmer need not decide statically which classes support remote access. Any object instance from any application, including compiled classes and library classes, can be deployed as a Web Service without the need to access or alter application class source code.
2. The system integrates the notions of Web Services, Grid Services and Distributed Object Models by providing a remote reference scheme synergistic with standard Web Services infrastructure, and extending the pass-by-value semantics provided by Web Services with pass-by-reference semantics. Specific object instances rather than object classes are exposed as Web Services, further integrating the Web Service and Distributed Object Models. This contrasts with systems such as Apache Axis [15] in which classes are deployed as Web Services.
3. Parameter passing mechanisms are flexible and may be controlled dynamically. Parameters and result values can be passed by-reference or by-value and these semantics can be decided on a per-call basis.
4. When objects are passed by-reference to remote address-spaces, the system deploys them automatically. Thus an object *b* that is returned by method *m* of deployed object *a* is automatically deployed before method *m* returns.

Although the RRT is written in Java and is designed to support Java applications, it does not rely on any features unique to Java.

4 Development and Deployment Methodology

The methodology is designed to support a separation between core application logic and the details of its distribution. It focuses specifically on the implementation and testing phases of the software engineering process. The steps involved are as follows:

1. Design and implement the application code, without taking any account of how it will be distributed.
2. Deploy, test and debug the (currently non-distributed) application within a single address-space.
3. Define how the application will be (initially) distributed.
4. Define how the new failure modes introduced by distributing the application should be handled (optional).
5. Define particular object transmission, caching and exception handling policies (optional).
6. Deploy, test and debug the application in multiple address-spaces on a single physical host.
7. Deploy, test and debug the application in a fully distributed setting.
8. Design and deploy probes to monitor the execution of the distributed application (optional).

For simplicity, these steps are described as a linear progression from start to finish. In practice the developer will often return to previous steps, as is common in many software engineering approaches. Indeed, it is a distinct benefit of this methodology that it is very simple to revisit and alter earlier decisions made regarding distribution policy. This is possible because the distribution policy and logical code structure are orthogonal to each other; furthermore the different policies, for example distribution policy and parameter-passing policies are also orthogonal to each other. In most middleware systems these orthogonal issues are conflated.

4.1 Implementation of Application Logic

The initial step is to design and implement the application logic, without taking any account of how the application will be distributed. The entire application at this stage will run within a single Address Space (AS). Interaction between components of the application, which may involve remote calls over the network in the final distributed version, is implemented using standard inter-object method calls.

This allows the developer to concentrate on the core logic, ignoring distribution issues[1]. In particular, the developer need not:

- (ever) write any networking code
- consider which application objects will communicate with remote objects
- extend or implement any special base classes or interfaces to enable remote communication

Although this is described as a single step in the methodology, it would typically represent most of the development effort.

The methodology will be illustrated in the context of developing JChord, an implementation of the Chord peer-to-peer protocol [16]. This employs a global ring topology to link all participating nodes, with additional inter-peer links to support resilience and efficient routing. Each node has a unique key; the node keys are used to order the nodes in the ring. The fundamental operation provided by the peer-to-peer network is *lookup()*, which maps a key to the node currently "in charge" of that key.

Although a Chord network may contain a large number of participating nodes, the intrinsic symmetry of the peer-to-peer model means that the software running on most of the nodes is identical. In the JChord implementation, four principal node types can be identified:

- the initial network node
- any other network node
- a diagnostic console node that receives events from network nodes
- a control node that is able to start and stop network nodes

The first and second node types differ only in the way that they are initialised: the initial node needs no configuration information, whereas all nodes subsequently joining the network must be configured with a reference to a node already in the network. **Fig. 1** shows an (extremely simplified) outline of a class *P2PNode* that implements a network node. At this stage the focus is on application logic rather than distribution, so although

[1] With the exception that all fields in any class that may be accessed remotely must be **private**. This is often regarded as good coding practice anyway.

instances of the class are likely to be remotely accessible, the class does not implement any special interface or extend any base classes.

The methods respectively: return the key of a node; get and set the successor node in the ring; lookup the node corresponding to a key; route a message to the node for a given key; start and stop the node; and set the diagnostic console to which events should be sent. The interfaces *IP2PNode* and *IConsole* are defined in **Fig. 5**.

```
public class P2PNode {
     private final Key key;
     private IP2PNode successor;
     public Key getKey(){…}
     public IP2PNode getSuccessor(){…}
     public void setSuccessor(IP2PNode successor){…}
     public IP2PNode lookup(Key key){…}
     public void route(Key key, Message msg){…}
     public void start(){…}
     public void stop(){…}
     public void setConsole(IConsole console){…}
}
```

Fig. 1. Outline of peer-to-peer node implementation

Fig. 2 shows the outline of a class *ConsoleNode* that implements a diagnostic console. The method *receiveEvent* allows a diagnostic event to be delivered to it by a network node.

```
public class ConsoleNode {
     public void receiveEvent(Event event){…} }
```

Fig. 2. Outline of console node implementation

A class *ControlNode* defines the control node type; details are omitted here. The completion of the implementation of these classes concludes the first development step. At this point it is possible to deploy and test the application in a single AS as described in step two. In contrast to most common middleware systems, the design and implementation thus far has not required the developer to consider distribution boundaries, extend base classes or implement particular interfaces. This eases the development task and retains flexibility with respect to how the resulting objects will be distributed.

4.2 Local Deployment and Testing

The next step is to deploy the application in a single AS and design a test suite for the core application logic, using conventional tools such as JUnit [17]. This may, of course, be integrated with the previous step for a test-driven development approach. Tests are run and any defects corrected.

The key point here is that although the entire application runs within a single AS at this stage, it is the real application code that is executing rather than a simulation. Few changes will be made to that logic during the later steps that introduce distribution, giving little scope for the introduction of further programming defects. In particular, there is no need to transform or translate the original code into a distributed form.

Fig. 3 shows a minimal JChord network in a testing configuration within a single AS. Three peer-to-peer node objects are linked in a ring;

each of these refers to a diagnostic console object; a control node object refers to one of the peer-to-peer nodes in order to control it.

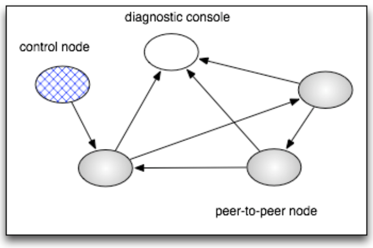


Fig. 3. JChord objects running in single AS

This configuration is created by a test program that instantiates the five objects and then establishes the connections among them. Testing checks that the ring is correctly formed, that *lookup()* and *route()* work as expected, that diagnostics are displayed by the console, etc.

The benefit of the methodology at this testing stage is that the developer can focus exclusively on verifying the application logic, ignoring issues of distribution.

4.3 Definition of Initial Application Distribution

Once a functional local version of the application has been produced in the previous steps, the developer defines its distribution. This involves:

- deciding how the application objects should be partitioned across the available ASs,
- deciding which objects should be made available for remote access (i.e. objects whose methods can be called by objects in remote ASs), and
- deciding the initial inter-AS object "wiring" (i.e. which pairs of objects located on different ASs should be connected by references)

These decisions feed into a number of coding activities. First, multiple entry points must be defined for the application, corresponding to each of the ASs on which part of the application will run. Thus whereas the initial version of the application may contain only a single class with a *main()* method, now a separate class with a *main()* is required for each entry point[2]. Execution of the application via the appropriate entry point on a particular AS results in instantiation of the appropriate application objects for that AS[3]. The partititioning for the JChord application is straightforward: each of the JChord objects described previously is placed in a separate AS, as illustrated in **Fig. 4**.

[2] Depending on the symmetry of the application, it is often possible for a particular entry point class to be used for multiple hosts.

[3] Support for remote object instantiation according to specified policies is under development.

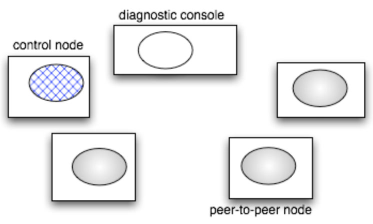


Fig. 4. JChord objects partitioned across ASs

Implementation of this partition involves writing an application entry point (a class with a *main()* method) for each of the four distinct node types. In each case the *main()* method creates an instance of the corresponding class (*P2PNode*, *ConsoleNode* or *ControlNode*). Where some configuration of the new object is required—for example, a *P2PNode* joining an existing network needs to be given references to an existing peer-to-peer node and to the console node—the configuration information is passed in the command line parameters.

Next, for each entry point, additional *deployment* code must be written to make the appropriate objects remotely accessible. Typically only a relatively small number of objects need be made remotely accessible; these will act as entry points. A deployed object may expose one or more *deployment interfaces*. Deployment interfaces are defined using Java classes or interfaces whose methods are structurally compatible with those defined in the object's actual class. The class need not have been defined as extending those classes or implementing those interfaces. This means that an interface through which a deployed object is exposed may be decided after the object already exists. The RRT provides the following API:

```
void deploy(Object objectToBeDeployed,
     Class interfaceToBeExposed, String deploymentName)
```

In the JChord application, three logically distinct interfaces can be identified: one exposing peer functionality to other peers, one supporting remote control of a peer from any other object, and one allowing peers to send events to the console. **Fig. 5** shows the definitions of the corresponding interfaces *IP2PNode*, *IManage* and *IConsole*.

```
public interface IP2PNode {
     public Key getKey();
     public IP2PNode getSuccessor();
     public void setSuccessor(IP2PNode successor);
     public IP2PNode lookup(Key key);
     public void route(Key key, Message msg); }

public interface IManage {
     public void start();
     public void stop(); }

public interface IConsole {
     public void receiveEvent(Event event); }
```

Fig. 5. JChord remote interfaces

As shown in **Fig. 6**, interfaces *IP2PNode* and *IManage* are both exposed by each peer-to-peer node; *IConsole* is exposed by the console; while the control node need not expose any remote interface. It should be emphasised again that the classes *P2PNode* and *ConsoleNode* were not declared as implementing any of these interfaces. This means that the decision as to what interfaces are exported can be made later in the development cycle than the definition of the functionality[4].

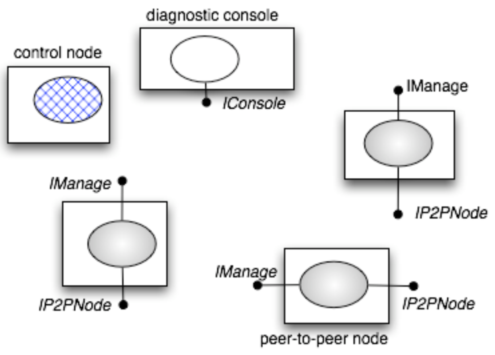


Fig. 6. JChord objects deployed for remote access

The code to deploy the appropriate remote interfaces is added to the *main()* method in the corresponding application entry point class. This is illustrated in **Fig. 7**, which shows the deployment of *IP2PNode* and *IManage* interfaces for a new *P2PNode*.

```
P2PNode p2pNode = new P2PNode();
// initialisation code omitted for brevity
RAFDARunTime.deploy(p2pNode, IManage.class, "Manage");
RAFDARunTime.deploy(p2pNode, IP2PNode.class, "P2P");
```

Fig. 7. Code to deploy remote interfaces

Finally, wiring code is needed to establish connections between objects on different ASs. Each connection consists of a remote reference held by an object, denoting another object in a remote AS. Since remote references are indistinguishable from local references, this is sufficient to allow methods on the remote object to be called. Each remote reference is obtained by a method call to the local middleware infrastructure, passing it a description of the remote AS identified by IP address and port, and a name or identifier for the required object. The RRT provides the following API for this purpose:

```
Object getObjectByName(SocketAddress rrt, String name)
```

Further connections can be established dynamically, through a remote method call returning a reference to another object. Thus the initial wiring code can be fairly minimal; only one connection into every AS is necessary to give connectivity between the different parts of the application.

Since these distribution policy decisions are specified independently of the main application logic, they can be altered easily. The partition of objects across ASs can be changed between successive builds of the appli-

[4] It also means that instances of library classes can be made remotely accessible even if the source code of those classes cannot be modified.

cation[5]. Furthermore, the deployment of objects for remote accessibility, and the inter-AS connections, can be changed dynamically.

Fig. 8 shows an initial configuration for the JChord application equivalent to that shown for single AS testing in **Fig. 3**.

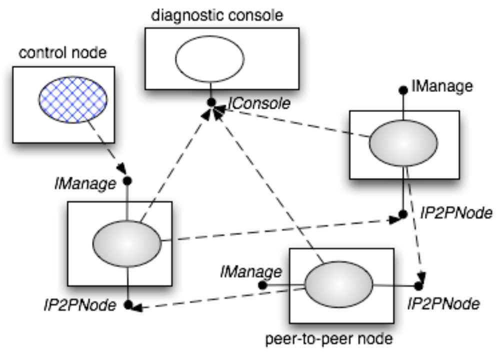


Fig. 8. Remote connections established between JChord objects

At this point, the necessary application components are extant in the appropriate ASs and available for remote access, but do not reference each other. The method *getObjectByName()*, described above, is used in order to establish remote references between the components. The only information that is required is the address of the RRT hosting each remote component, and the logical name. This code is added to each entry point class, taking details of the required network addresses from the command line parameters. **Fig. 9** sketches the code to set up the references for a new peer-to-peer node joining the ring, from the node to its successor node and to the console node. The final *start()* call starts the node, so that it accepts remote calls and periodically executes its fault tolerance algorithms (not described here).

```
public static void main(String[] args {
     P2PNode p2pNode = new P2PNode();    // As in Fig. 7
     ... // deployment code omitted
     SocketAddress successorAddr = ...   // extract from args
     SocketAddress consoleAddr = ...     // extract from args
     IP2PNode succ = (IP2PNode)RAFDARunTime.getObjectByName(
                    successorAddr, "P2P");
     p2pNode.setSuccessor(succ);
     IConsole cons = (IConsole)RAFDARunTime.getObjectByName(
                    consoleAddr, "Console");
     p2pNode.setConsole(cons);
     p2pNode.start();
}
```

Fig. 9. Setting up inter-AS references in entry point for P2PNode joining ring

For ease of management it may be preferable instead for the network addresses of the various connection end-points to be specified in a configuration file, copied to all participating hosts, rather than reading them from the command line.

[5] Support for dynamic object migration is under development.

4.4 Definition of Distribution-Related Error Handling (Optional)

Distributing a hitherto non-distributed application introduces new failure modes: a method call to a remotely accessible object may now fail due to network or remote host failures. If the developer wishes to specify in detail how such failures should be handled, this can be achieved by specifying appropriate exception handlers for remote method calls.

However, the RRT middleware can handle such errors automatically, in which case failure of a *void* remote method call will be invisible to the calling object, while failure of a remote method call that returns a result will lead to a default value (e.g. null, 0 etc) being returned. This capability is designed to increase distribution flexibility, in that code calling a method need not differ between local and remote calls. If used, however, the developer should be aware that remote calls may now return default values without warning. Automatic handling of distribution-related exceptions is disabled by default. This facility is especially useful in prototyping where different topologies can be easily explored without regard to application resilience.

To allow dynamic choice as to whether automatic handling is used, distribution-related exceptions are *unchecked*, achieved by sub-classing *RuntimeException*. The significance of this is that Java does not enforce the specification of handlers for code in which such exceptions may occur. Thus the developer has three choices:

- to enable automatic handling via a single API call
- to write no additional code at all
- to specify exception handlers in the normal way

In the first case, no network-related exceptions will be thrown, and default values will be returned from a remote call. In the second case, exceptions will be thrown and the calling application will fail. In the final case, exception handlers are written by the developer to catch network exceptions.

The first option is not appropriate for JChord, since network errors need to be detected and handled explicitly. With the second option, any error arising from network or remote node failure would throw an unchecked exception, which, not being caught, would terminate execution of the AS in which it occurred. This is unacceptable in the JChord application, which is designed to provide fault tolerance. If a peer-to-node is unable to communicate with its successor, for example, it should initiate action to locate a new successor.

```
try {
      IP2PNode nextButOne = successor.getSuccessor();
      ... }
catch (RafdaRuntimeException e) {
      // call to successor failed; initiate recovery actions
      Exception cause = e.getCause();
      ... }
```

Fig. 10. Handling a distribution-related error

Fig. 10 shows an example of exception handling code added for a call to *getSuccessor()* on a peer-to-peer node's successor, within the definition of the *P2PNode* class. Since the *successor* field holds a remote reference, calls performed on it may fail. Similar code is added for each remote call. The considerable developer effort required is the price paid for fault tolerance. Without it, the application would still function correctly on a reliable network, but would not be able to handle node or network failure.

4.5 Configuration of Middleware Policies (Optional)

The RRT middleware permits control of the following policies:

- whether parameters and result values for remote method calls should be passed by-reference or by-value (default: *by-reference*)
- whether particular fields of objects denoted by remote references should be cached locally, and if so whether methods of such objects that access cached fields should be executed locally (default: *not*)
- whether network-related errors should raise exceptions or be handled automatically (default: *raise exceptions*)

Default settings for these policies are designed such that the developer may omit this step and still obtain a functioning distributed application.

By default, all objects passed to and from a remote method call are passed by-reference. This preserves object identity and involves minimal change in application semantics between the initial local implementation and the distributed version. However, where it is known that an object's state will change infrequently, it may be desirable for it to be passed by-value so that future operations on it may be performed without the need for a remote call. This may improve efficiency and eliminate potential network-related errors. When an object is passed by-value, a copy is created in the receiving AS. The middleware does not currently provide any automatic coherency control, hence it is the responsibility of the application to maintain coherency of object copies in the event of update.

Parameter passing policy is controlled by the sending side. Thus the policy in effect within a particular AS controls the passing of parameters **to** remote calls to other ASs, and the returning of results to remote calls made **from** other ASs. The policy can be specified at various levels of granularity as appropriate: for all instances of a given class, for all parameters of a given method, or for specific method parameters. The RRT provides the following API for class-level control (others omitted here):

```
void setClassPolicy(Class c, int policy)
```

Field caching allows a reference transmitted to a remote AS to include copies of particular fields of the referenced object. Typically this is used in cases where fields are not expected to be updated. As with passing by-value, this may improve efficiency and eliminate potential network-related errors. Method caching allows a method call on a remote object to be evaluated locally, in cases where all the fields accessed by the method are locally cached. Again, the motivations are efficiency and fault-tolerance. Setting all fields to be cached would have the same effect as passing by-value, thus this mechanism may be viewed as giving finer control than the by-reference / by-value distinction. The RRT provides the following API:

```
void setFieldToBeCached(Field field)
void setMethodToBeCached(Method method)
```

In the JChord implementation, instances of classes *Key* and *Message* are candidates for being transmitted by-value, since they are immutable and likely to be relatively small. This is specified by further code added to the entry point classes, illustrated in **Fig. 11**.

```
TransmissionPolicyManager.setClassPolicy(
     Key.class, BY_VALUE, LOW);
// LOW priority allows this to be overridden
// by more specific policies
TransmissionPolicyManager.setClassPolicy(
     Message.class, BY_VALUE, LOW);
```

Fig. 11. Setting transmission policy for particular classes

The intention here is to set the transmission policy for these classes for the duration of the application execution. It is also possible to change the policy more dynamically. For example, the *route()* method might set the policy for *Message* instances to *BY_VALUE* for small messages, and to *BY_REF* for larger messages [3].

For this application it is also beneficial for each remote reference to a *P2PNode* to cache the value of the *key* field locally, and for calls to the *getKey()* method to be evaluated locally. This improves efficiency since keys are accessed frequently. The code to specify this is shown in **Fig. 12**.

```
TransmissionPolicyManager.setFieldToBeCached(
     P2PNode.class.getField("key"));
TransmissionPolicyManager.setMethodToBeCached(
     P2PNode.class.getMethod("getKey"));
```

Fig. 12. Setting field and method caching

A further benefit of this caching is that diagnostic code reporting failure of a peer node is able to access the peer's key even though the peer is inaccessible. Thus the exception handling block in **Fig. 10** can include:

```
console.receiveEvent(new Event(
     "successor failed - key: " + successor.getKey()));
```

4.6 Local Distributed Deployment and Testing

The initial testing of the distributed version of the application can be performed on a single host, by instantiating multiple ASs locally. Communication between the RRT instances in the various ASs will take place via the loopback network interface in the same way as for genuinely distributed ASs. This allows testing of the object partitioning, the deployment of selected objects for remote access and the initial inter-AS object wiring in a reliable context, before the introduction of potential time-outs and other failures in the fully distributed setting.

Fault tolerance to distribution-related errors can be tested to some extent by killing various AS processes, producing a similar effect to the abrupt failure of a remote host or network connection in a genuinely distributed deployment. Since such errors are always possible, the developer should verify at this stage that the parts of the application on the surviving ASs handle such events in an acceptable way. The RRT also allows the developer to specify the class of *Socket* used for inter-RRT communication, allowing the use of *Socket* implementations which emulate connections that are low bandwidth, high latency, etc.

For repeated testing, it is useful to write scripts containing the Java commands to instantiate a number of ASs. Each command includes the entry point class for that AS, and parameters such as descriptions of other ASs (specified by IP address and port) to be used by the application in performing initial inter-AS object wiring. The command also specifies a Java classpath that includes the RRT *.jar* file.

Testing at this stage can be further automated using tools such as JUnit. It then becomes necessary to be able to initialise an entire collection of ASs under control of a running test program. This may be achieved using Java's *Runtime.exec()* to create ASs running within new processes. The test code can then establish inter-AS remote references to objects in other ASs, and proceed to carry out application tests. The only difference in the form of these application tests from those performed during single AS testing is that remote calls are, naturally, restricted to use only the interfaces through which the remote objects have been deployed.

Section 4.3 described how a separate entry point class can be written for each distinct variety of node, with a *main()* that instantiates and configures an instance of the appropriate class. This approach presents the problem of orchestrating the deployment and execution of the appropriate entry points on appropriate hosts. To ease this, it may be preferable to combine the entry points into a single class, which reads details from a local configuration file as to which variety of node is required. The problem is then reduced to one of distributing a single application image to all hosts, and tailoring the configuration file appropriately on each.

4.7 Full Distributed Deployment and Testing

The final testing phase involves genuine distribution of the application. This requires no changes to the code or the tests developed in the previous step, but the deployment infrastructure must be adapted. Two actions are required: copying of the application code and the RRT release *.jar* file onto each host, and execution of the appropriate application entry point on each host. On a small scale this can be performed manually. For a more scalable solution these tasks can be automated using a deployment application written in Java. This uses an SSH library [18] to establish a secure connection to each of the remote hosts and create a process that copies the required files and runs a AS with the appropriate application entry point.

An interactive tool has been developed to support simple launching of a JChord ring with any number of nodes. Each node runs in a separate AS, created either locally or remotely via SSH. AS processes can be killed to simulate failure. The tool also provides an API.

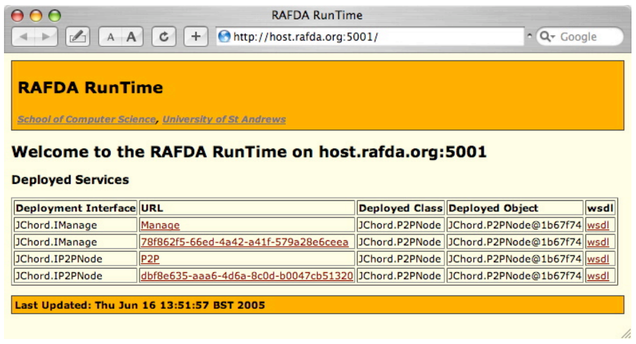
4.8 Monitoring (Optional)

It may be useful to monitor the state of a running distributed application, for the purposes of debugging or for gathering ongoing diagnostics. The RRT middleware offers two approaches:

- the RRT instance running on a particular host/AS may be queried via a web browser
- probe objects, tailored to the application, may be dynamically deployed within a particular AS

Each RRT instance runs a web server, which can be accessed using a conventional web browser to obtain information about deployed objects. Each deployed object is listed, showing the deployment interface, service class, service name and a string representation of the service object[6]. This interface can be used to verify which objects have been successfully deployed within a particular AS.

Fig. 13 shows the web interface provided by the instance of the RRT running in a particular AS. It lists the deployed interfaces, with the corresponding classes and objects. In this example each interface is accessible both via a logical deployment name and via a generated unique identifier.



RAFDA RunTime

School of Computer Science, University of St Andrews

Welcome to the RAFDA RunTime on host.rafda.org:5001

Deployed Services

Deployment Interface	URL	Deployed Class	Deployed Object	wsdl
JChord.IManage	Manage	JChord.P2PNode	JChord.P2PNode@1b67f74	wsdl
JChord.IManage	78f862f5-66ed-4a42-a41f-579a28e6ceea	JChord.P2PNode	JChord.P2PNode@1b67f74	wsdl
JChord.IP2PNode	P2P	JChord.P2PNode	JChord.P2PNode@1b67f74	wsdl
JChord.IP2PNode	dbf8e635-aaa6-4d6a-8c0d-b0047cb51320	JChord.P2PNode	JChord.P2PNode@1b67f74	wsdl

Last Updated: Thu Jun 16 13:51:57 BST 2005

Fig. 13. Web interface for RRT instance

Probe objects to monitor particular aspects of the application's execution can be installed and accessed remotely, either by another Java application via the RRT middleware, or by any Web Services client—by virtue of the fact that the RRT uses Web Services as its remote invocation mechanism.

Probes may be deployed by the application itself, or installed remotely under administrator control. In the latter case, the application must expose an interface that supports the integration of probes.

5 Conclusions

This paper has presented a methodology for developing and deploying distributed applications, exploiting many of the features of the RRT middleware. The strengths of the approach are that the application logic can be designed and implemented completely independently of distribution concerns, easing the development task, and that this gives great flexibility to alter distribution decisions late in the development cycle. The RRT middleware is available for download [1].

Plans for further development include support for policy-driven object placement, support for transparent object migration, a distributed naming service, improved resilience to transient network failures, an improved security model, and improvements in performance.

[6] Support for automatic generation of WSDL for each deployed object is under development, as is a facility to allow object method invocation from the web browser.

6 Acknowledgements

This work was supported by EPSRC grants GR/R51872 and GR/S44501/01 and by EC Framework V IST-2001-32360.

7 References

1. Dearle A., Kirby G.N.C., Rebón Portillo A.J., Walker S. Reflective Architecture for Distributed Applications (RAFDA). 2003. *http://www-systems.dcs.st-and.ac.uk/rafda/*
2. Rebón Portillo Á.J., Walker S., Kirby G.N.C., Dearle A. A Reflective Approach to Providing Flexibility in Application Distribution. In: Proc. 2nd International Workshop on Reflective and Adaptive Middleware, ACM/IFIP/USENIX International Middleware Conference (Middleware 2003), Rio de Janeiro, Brazil, 2003, pp 95-99
3. Dearle A., Walker S., Norcross S., Kirby G.N.C., McCarthy A. RAFDA: Middleware Supporting the Separation of Application Logic from Distribution Policy. University of St Andrews Report CS/05/3, 2005.
4. Walker S.M. RAFDA Run-Time (RRT) Beginner's Guide v1.0. University of St Andrews Report CS/05/4, 2005.
5. OMG. Common Object Request Broker Architecture: Core Specification, 2004
6. Sun Microsystems. Java™ Remote Method Invocation Specification, 1996
7. Microsoft Corporation. The Component Object Model Specification. 1995.
8. Obermeyer P., Hawkins J. Microsoft.NET Remoting: A Technical Overview. Microsoft Corporation, 2001.
9. W3C. Web Services Architecture. 2004. *http://w3c.org/2002/ws/*
10. Tilevich E., Smaragdakis Y. J-Orchestra: Automatic Java Application Partitioning. In: Proc. European Conference on Object-Oriented Programming (ECOOP), Malaga, 2002
11. Spiegel A. Automatic Distribution of Object-Oriented Programs. PhD thesis, 2002
12. Caromel D., Klauser W., Vayssiere J. Towards Seamless Computing and Metacomputing in Java. Concurrency Practice and Experience 1998; 10,11-13:1043-1061
13. Fahringer T., 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; 17,7-8:1005-1025
14. Lavender R.G., Schmidt D. Active Object - An Object Behavioral Pattern for Concurrent Programming. In: J. Vlissides, J. Coplien and N. Kerth (ed) Pattern Languages of Program Design 2. Addison-Wesley, 1996
15. Apache Axis. 2004. *http://ws.apache.org/axis/*
16. Stoica I., Morris R., Karger D., Kaashoek F., Balakrishnan H. Chord: A Scalable Peer-To-Peer Lookup Service for Internet Applications. In: Proc. ACM SIGCOMM 2001, San Diego, CA, USA, 2001, pp 149-160
17. JUnit, Testing Resources for Extreme Programming. 2005. *http://www.junit.org*
18. AppGate Network Security. MindTerm. 2005. *http://www.appgate.com/products/80_MindTerm/*