Supporting Interactive Invocation of Remote Services within an Integrated Programming Environment

Bruce Quig
Faculty of Information Technology
Monash University
Australia
bquig@infotech.monash.edu.au

John Rosenberg
Faculty of Information Technology
Monash University
Australia
johnr@infotech.monash.edu.au

Michael Kölling
Mærsk Institute
University of Southern Denmark
Denmark
mik@mip.sdu.dk

ABSTRACT

Building distributed systems is an inherently difficult and complex task. Modern middleware architectures assist developers by providing abstractions that hide transport layer functionality. This paper argues that the development of such systems can be aided by the availability of appropriate, integrated tools. We discuss ways in which the building of such systems can be supported by development tools, focusing particularly on interactive testing and debugging mechanisms.

A prototype system based on the BlueJ programming environment has been developed to support the development of Java RMI applications as a basis for further investigation. The tools developed as part of this prototype are presented and discussed.

Categories and Subject Descriptors

D.2.6 [**Software Engineering**]: Programming Environments - *integrated environments, interactive environments, programmer workbench;* D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms

Experimentation, Verification.

Keywords

Distributed Systems, testing, debugging, BlueJ.

1. INTRODUCTION

The evolution of modern computing applications has been influenced by a number of continuing trends. Computing devices continue to become more powerful and smaller in physical size. Networking infrastructure is also becoming more widespread, higher in capacity and more accessible. This has led to a continued interest in the use and development of distributed applications, potentially targeting a wide variety of networked computing devices.

Distributed applications have numerous potential advantages, such as increased reliability, performance and maintainability [1]. Building systems that realise these advantages, however, is an inherently difficult task. It requires developers to deal with the unique characteristics of distributed systems such as increased latency, bandwidth restrictions and fluctuations, security implications and requirements, and the increased risk of partial and complete failure of other network nodes.

Middleware architectures such as Java Remote Method Invocation (RMI), Common Object Request Broker Architecture (CORBA), and Microsoft's Distributed Component Object Model (DCOM) are commonly used to allow developers to concentrate on the application logic. These architectures manage the lower level details of distributed communication between application components. The development process however is still difficult and complex. Such systems continue to be considered difficult to design, build, configure, test and execute.

We believe that the process can be simplified and improved by the availability of appropriate integrated software development tools, particularly those that support easy prototyping, testing and debugging. This paper describes an integrated programming toolset based upon the existing BlueJ Programming Environment [2] and extensions to BlueJ to support distributed programming with Java RMI.

2. TOOLS FOR DISTRIBUTED SYSTEMS

Object-oriented (OO) systems are now commonly used to develop modern computing systems. The object-oriented paradigm is seen to provide a good conceptual fit with distributed systems [3]. The notion of objects as autonomous individuals, and messages as the communication mechanism between objects, closely matches the structure and behaviour of distributed systems.

There is a continuing trend towards the use of object-oriented middleware for building distributed systems [4]. Examples of these include CORBA, Java RMI, Microsoft's COM/DCOM and .NET Framework. These architectures, while providing their own use and design idioms, rely upon a similar set of abstractions ([5, 6]). The core activities of building these systems are similar.

Distributed object systems, like traditional OO systems, are peer-to-peer in nature. Objects can act as both consumers and providers of services. Within the context of a particular invocation, an object may be deemed to be either a client or a provider of a service. The following subsections look at the typical development stages for client and server objects

2.1 Developing Remote Service Objects

2.1.1 Define Interface

One of the first steps in developing distributed objects is the definition of interfaces as the means by which clients and servers

PPPJ 2003, 16-18 June, 2003, Kilkenny City, Ireland.

conform to a contract representing the allowed interactions. CORBA and COM/DCOM use a separate contract language for defining these interfaces. CORBA uses its own Interface Definition Language (IDL), as does COM/DCOM, which uses Microsoft IDL (MIDL). These definitions are then used to generate appropriate proxies through which distributed communication is marshalled. As a single language platform, RMI remote object interfaces can use an existing language Interface construct to define remote interfaces. RMI interfaces use the existing Java language Interface construct. Early versions mandated that remote interfaces needed to inherit a methodless parent interface Remote to flag their remote capabilities to both the system compiler and Java Virtual Machine (JVM). Latter versions only require the implementation of interfaces whose methods provide remote exception handling.

2.1.2 Generate Service Stubs

CORBA and DCOM implementations contain tools that perform the task of producing remote stubs. On the service side these act as a local proxy [7] for the client. Conversely on the client side the stub acts as a local proxy for the remote service. The stubs act to hide the distributed nature of the method invocations. In RMI the process is slightly different. RMI concerns itself only with Java objects on both the client and server. As it therefore deals with a single type system, there is no need for an intermediary definition language to specify remote interfaces. Stub generation is still carried out by a separate compiler; however the service implementation class file is used as the input source of the compilation process. Early versions of Java produced separate stubs for servers and clients (skeletons). As of Java version 1.2 a unified stub provides client and server capabilities.

2.1.3 Create Service Implementation

Service implementations are created typically using some form of inheritance or interface implementation. Depending on the capabilities of the implementation language, this may be direct inheritance of a base or abstract class. In languages such as Java that support a separate notion of an interface type, it is typically through the implementation of such a type.

2.1.4 Service Registration

For a service to be available to respond to requests it needs to undergo some form of registration so it can be successfully located. In CORBA the most basic form is the implementation repository which resides on each host. It associates service objects with the means to start up the server process. COM/DCOM uses the Windows registry as an implementation repository. RMI uses Activation interfaces in conjunction with the RMI daemon to start up processes.

It is common for services to be placed in some form of repository for access by clients. Examples of these are Naming services that allow the lookup of services by matching with an identifying name, and Trading services where lookup is based on service type. The CORBA standard [8] includes specifications for both Naming and Trading services [9]. RMI provides a simple naming registry (rmiregistry) as part of the standard Java software development kit (SDK) distribution.

2.2 Developing a client to consume services

A number of the steps involved in creating a client object are the same as in producing remote services as outlined in Section 2.1. The definition of interfaces is the same as for services. The client will use the same interface or type definition as provided by the remote service object. The service stub generation process is also similar if not identical to that of services. In some cases mechanisms exist for the dynamic discovery of these. CORBA provides a Dynamic Invocation Interface (DII). Whilst RMI does not have an equivalent, similar functionality can be achieved through the use of Java's reflection API with RMI.

Clients need to gain a reference to the remote object. This may be a persistent reference that is available to the client or can be through some form of registry or broker such as a naming, trading or directory service. Once the client has access to a reference to the remote object it can send requests to the remote service in a transparent manner similar to the way it would to a local object.

The above description of both services and clients provides a simplistic overview of their respective roles. In practice the roles of each may be merged. As mentioned earlier, objects that take on a role as a client may also provide remote services. These may be in the form of callback mechanisms where an object forwards a remote reference to itself as part of its communication with other objects, or by making different remote

2.3 Development Issues and Tool requirements

One of the advantages of distributed object-based systems is that they are designed to reduce complexity by providing transparency to the underlying communication protocols and mechanisms. However, there are still a range of challenges and difficulties faced by developers.

There is a growing interest in highly iterative development methodologies such as extreme programming (XP) and Scrum [10]. These are also referred to as lightweight or agile development processes. At the heart of these processes is the notion of developing complex systems by a finely grained iteration of developing prototypes that are constantly evolved, tested, refactored and enhanced. In such systems the roles of unit testing and debugging are extremely important.

At the same time, the field of ubiquitous computing [11], also known as pervasive computing, is quickly gaining importance and attention. In ubiquitous computing systems, small devices communicate over wireless networks, forming highly dynamic and flexible distributed applications.

2.3.1 Difficult to create services in isolation

While remote services may be part of a larger overall application they are potentially difficult to test and debug in isolation. To support lightweight development processes, there are now a number of unit test frameworks available such as JUnit [12] for Java-based applications. To test remote services it is currently necessary to develop client code to use the remote services and to test its functionality. It is therefore necessary to either write extra client code in the form of unit tests or simultaneously develop clients of these services to test their correctness. This needs to be repeated at every iterative step in development, discouraging developers from adopting a fine-grained iterative approach to development.

Additionally if these objects do not perform correctly, it is currently difficult to debug them without using some form of driver program that can then be run using a debugger. It would be

easier to test and debug services if they could be tested and debugged at an object level via direct interaction mechanisms. The Blue programming environment [13], and its successor BlueJ [2], are examples (at a non-distributed level) of systems which support such interaction.

2.3.2 Clients need to understand services

In an environment where distributed object systems are deployed there is likely to be a need to re-use existing services just as would be done in local OO systems. This leads to the inevitable situation in which developers are using services written by others. It is therefore an important capability to be able to browse repositories of existing objects and retrieve data about them. Even more useful than retrieving data may be the ability to interact with them. Valuable information about services could be gained if it were possible to dynamically interact with these services. By this we mean the ability to directly call methods of these remote objects and receive the results of these invocations. This facility would also be useful in the testing and management of pervasive computing systems, which often incorporate types of objects introduced dynamically after initial system development.

3. PROTOTYPE SYSTEM

We have built a prototype system to address these issues. Our design goals were to:

- provide visualization of relevant system abstractions: repositories, services and interfaces;
- allow direct interaction & manipulation of these entities;
- provide direct visual feedback of changes in state;
- allow users to avoid the writing of unnecessary code;
- ensure tools appear as an integrated toolset; and
- be supportive of contemporary development methodologies, but non-prescriptive.

The design and implementation of the prototype system was greatly influenced by earlier work on the development of the Blue system [13] and its successor, BlueJ [2].

Our wider aim is to investigate development tools applying to a range of distributed object technologies with architectural similarities. In the first instance we have selected to support Java RMI as a vehicle for researching appropriate tools. The reasons for choosing RMI over other architectures include widespread availability and platform support, relative maturity, a relatively simple programming model, and its use as a basic infrastructure component of a number of other Java-related, distributed computing models such Jini and Enterprise JavaBeans (EJB).

From a pragmatic point of view, it also allowed us to leverage our existing work with BlueJ, a Java-based development environment designed to support teaching Java in university undergraduate courses. Our prototype system is an extended version of BlueJ that adds support for developing applications using Java RMI.

3.1 Java RMI

Remote Method Invocation (RMI) is a distributed object model [14] specifically designed for the Java programming language. It is essentially an object-oriented version of Remote Procedure Calls (RPCs). Objects residing in one Java Virtual Machine (JVM) can invoke methods upon remote objects that reside within different JVMs. These JVMs may reside on different host machines.

The uniform nature of Java bytecode across platforms allows RMI to send bytecode from one JVM to another, enabling objects to be passed by value either as parameters or return types of method requests. This allows the distributed interaction between objects to appear to be fairly transparent. There are some subtle semantic differences, however, in the behaviour of remote method invocations and local ones. In local Java method invocations, parameters that are primitive types are passed by copy. Object parameters are also passed by copy, it is however a copy of a reference to the particular object not a copy of the actual object [15].

In remote method invocations, non-remote objects are passed by copy. Java uses serialization to send a copy of the object as a stream to the target JVM. Non-remote objects parameters are therefore required to conform to the `java.io.Serializable` interface. Remote objects passed as parameters are passed as a copied reference to the remote object. Primitive types are passed by copy as in non-remote calls.

As a middleware platform, RMI does not offer the level of supporting services found in some other middleware architectures such as CORBA. It does not provide inbuilt support for services such as transactions, persistence and event notification. It does, however, provide the architectural basis upon which a number of more sophisticated and specialized architectures are built. Jini [16] and Enterprise JavaBeans [17] leverage RMI capabilities and provide additional services.

3.2 BlueJ

The BlueJ environment was developed as part of a university research project about teaching object-orientation to beginners. Special emphasis has been placed on visualisation and interaction techniques to create an environment that encourages experimentation and exploration.

BlueJ combines a number of unique features with more conventional development tools to assist university students in learning how to develop object-oriented programs using the Java programming language. The overriding principle behind BlueJ is to allow manipulation of OO programs using graphical representations of the key OO abstractions such as classes and objects.

This emphasis on the fundamental abstractions allows students to gain a greater understanding of the underlying principles of OO development. While BlueJ is aimed at programmers with little programming experience, the underlying principle of using graphical representations of the system as a means of managing complexity is also relevant to professional software development.

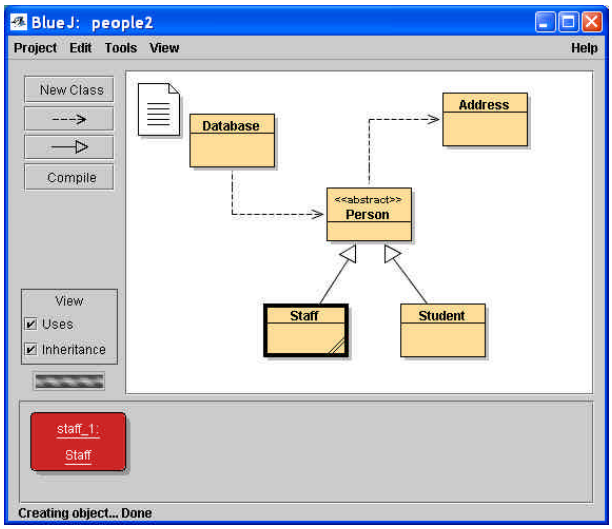

Figure 1 The BlueJ Development Environment

3.2.1 Class Diagram

The main project view within BlueJ is at a class level. It represents a Java package (Figure 1). The diagram uses a subset of Unified Modelling Language (UML) notation to represent classes and their relationships. These relationships can be interactively manipulated by the user, but are also automatically generated by the environment. Users can edit the source of these classes via the graphical screen representation. Right or double-clicking allows an integrated text editor to be opened. Class and project level compilation is also integrated into the environment.

3.2.2 Object Bench

A key component of BlueJ is its Object Bench, which is used to allow testing and debugging of classes individually by allowing the creation of object instances without any form of static main method to bootstrap an instance of a class. This means that testing and debugging can occur from the moment that the first method in the first class is created.

The Object Bench appears in the user interface below the class diagram (Figure 1). Right-clicking a class displays a popup menu with static methods and constructors of that class. If a constructor is selected an instance of this class is dynamically created using this constructor.

By right-clicking on the object it is possible to browse the available methods that can be invoked on this object. These include inherited and redefined methods. Selection of a method invokes that method on the instance. Parameters can be provided, including passing other objects from the object bench. Possible return results are presented via a dialog. If the return value is an object type it is possible to place that object onto the Object Bench for further testing and inspection.

Another important aspect of the Object Bench is the ability to inspect the internal state of objects. Double-clicking an object allows access to an Object Inspector window, which displays the internal state of the objects including all static and private fields. This allows users to note and test for the side effects of method invocations. If the instance fields of the object are also objects they can be inspected in turn.

3.3 Prototype for Distribution

The distribution prototype is based upon the BlueJ system and builds upon its base functionality. A main goal was to extend the direct interaction model that BlueJ supports for local objects to distributed objects. At the same time, the new prototype should follow the same interface construction guidelines that have led to the design of the BlueJ interface: that the main concepts of the problem domain should be graphically represented and available for interaction while incidental complexity should be avoided by automating underlying bookkeeping tasks.

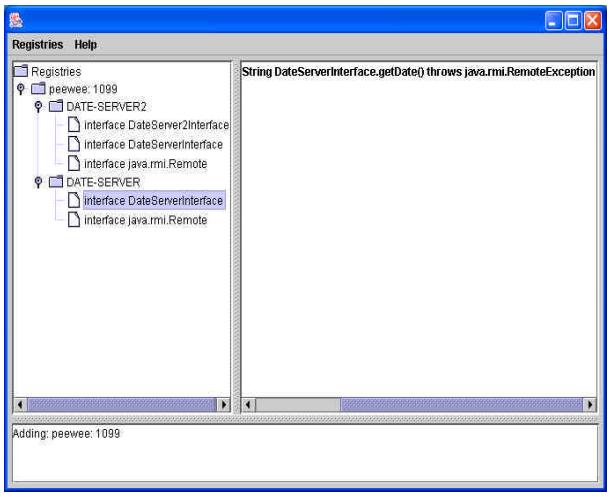

Figure 2 Remote Object Browser

The ability to test and debug at an object level provides a strong foundation for adding support for distributed browsing and interaction mechanisms. We have added remote object browsing capabilities that integrate with the Object Bench in a consistent and relatively transparent manner. The resultant prototype provides a set of specific, integrated tools that support partial functionality of an integrated programming environment for distributed objects. Its main aim is to act as a testing ground for integrated distributed object development tools.

One of the main visible components that have been added is the remote object browser (ROB). The browser allows connection to one or more RMI Naming services that can be operating locally or remotely.

Once connected to a naming service, the browser shows in tree form connected registries, services registered and service information (Figure 2). Services are listed by their registration name. Expanding a service allows it to show the interfaces that it implements. Selection of any of these will list the methods that this interface provides in the right hand pane.

The remote object browser is integrated with a remote interaction and invocation system (RIIS). Right-clicking a registered service brings up an option to “Get” this service object onto the Object Bench. Then very similar functionality for testing and interaction is available for these remote objects as for local objects. This includes direct interaction with object methods, leading to a much more direct access for testing purposes without the need to develop test clients.

In distributed terms, getting an object means downloading a remote stub that acts as a local proxy for the remote object. This

stub is then acts as the object's representation on the Object Bench and marshals the calls.

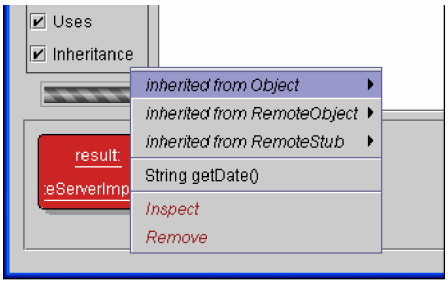

Figure 3 Remote Object Stub on Object Bench

Once a remote object has been selected for retrieval it appears on the Object Bench with a similar appearance and behaviour to local objects. Right-clicking the object lists available methods which may be invoked (Figure 3). Methods on the remote object can be directly executed by selecting them from the popup menu, and method results are displayed in a result dialog (Figure 4). If the return type is an object, it is possible to select it and place it on the Object Bench. It is then possible to directly interact with the returned object, including the inspection of their internal state. Any objects on the object bench can be composed by passing the object as a parameter to methods of other objects.

Figure 4 Result of Remote Method call

The browser can work with any remote registry with no prior knowledge of the services, classes and interfaces that may be available. The `java.rmi.server.codebase` property is used to annotate the stream that transports the interface stub to the client with the remote codebase site from which class definitions can be downloaded. The browser can also work with locally based class files.

Uses of this tool include the use as a testing and debugging mechanism when developing new remote services, as well as investigating existing services while developing client objects.

4. RELATED WORK

4.1 Toad

Toad is a development environment produced by IBM as a research project [18]. It is not a complete set of development tools and does not include conventional elements such a source code editor or source compiler. Toad comprises a range of post-production static and runtime development tools including a class file bytecode browser, code coverage tools, distributed system monitoring tools and an RMI registry browser. The RMI registry browser allows users to browse systems and make remote invocations upon remote methods. Return results of invocations are cached and can be used as parameters of further invocations.

Toad does not provide the ability to inspect the state of objects that have been returned, or the ability to directly interact with them at a local level. It also provides monitoring tools that monitor a number of protocols including the Java Remote Method Protocol (JRMP) used by RMI. The dynamic components of TOAD such as the RMI components are no longer under development. Whilst providing some similar functionality to our prototype system, the emphasis of Toad is quite different. It aims to provide post-production support for monitoring, understanding and optimising applications.

4.2 NetBeans

The open source NetBeans Integrated Development Environment (IDE) provides a plug-in module that supports RMI development [19]. It provides an RMI Registry Browser that allows remote registries to be browsed. It provides information on interfaces and methods supported. Interaction with these services is limited to the generation of code snippets that bind and then invoke particular methods on services and with downloading the interface class file for that service. These are the two main requirements in developing a client that can then invoke this method on this service. It also provides an Activation Browser that shows services that use RMI's activation framework. The RMI module's website lists the dynamic invocation of methods from RMI Registries as a future possible enhancement to the module.

5. CONCLUSIONS AND FUTURE WORK

We have presented two tools of an integrated development environment for the development of distributed systems. They are a remote object browser and a direct invocation and interaction system for remote objects.

The tools presented provide a useful base toolset that supports iterative development processes such as Extreme Programming for the construction of distributed software systems. The debugging and testing capabilities do not replace the more formalised unit testing strategies mandated by XP, but rather provide a complimentary toolset. In fact we see great potential to integrate the two approaches. For example, recording the interactions with remote objects could be used to create reproducible test fixtures using existing unit test frameworks such as JUnit.

The combination of browsing, interaction and inspection makes it very easy to test remote objects early and often during the development process. The tool in its current state, while useful, could potentially become more productive when also integrated with additional tools. The types of tools that it could interoperate with and leverage include tools to assist with interactive and dynamic client code generation, service refactoring, service wrapping, service configuration and system performance monitoring and logging.

As system development moves from static distributed systems to more dynamic and flexible ubiquitous computing systems, we envisage the importance of tools of this nature to increase. In addition to initial development and testing, such tools can support ongoing system monitoring and maintenance.

There are a number of planned enhancements to the environment. Firstly we wish to investigate more thoroughly additional complimentary tools to support RMI based systems as described

above. These include adaptive source code generation, activation support and service configuration tools.

There is also potential for this prototype to provide a base upon which more specialised tools for technologies that build upon RMI such as Jini and EJB could be developed. The dynamic and temporal nature of Jini based systems for instance, provides a number of interesting challenges to system as well as tool developers.

6. REFERENCES

[1] Coulouris, G., Dollimore, J. and Kindberg, T. *Distributed Systems, Concepts and Design*. Addison Wesley, 1994.

[2] BlueJ. BlueJ Programming Environment, http://www.bluej.org

[3] Schmidt, D.C. and Vinoski, S. Introduction to Distributed Object Computing. *The C++ Report*, *7* (1).

[4] Emmerich, W. *Engineering Distributed Objects*. J. Wiley & Sons Ltd, 2000.

[5] Watkins, D. and Thompson, D. Comparisons between CORBA and DCOM: Architectures for Distributed Computing. in Chen, J., Li, M., Mingins, C. and Meyer, B. eds. *TOOLS Asia 24*, 1997.

[6] Plasil, F. and Stal, M. An Architectural View of Distributed Objects and Components in CORBA, Java RMI and COM/DCOM. *Software-Tools & Concepts, Springer Verlag*, *19* (1). 14-28.

[7] Gamma, E., Richard, H., Johnson, R. and Vlissides, J. *Design patterns : elements of reusable object-oriented software*. Addison-Wesley, Reading, Mass., 1995.

[8] OMG. The Common Object Request Broker Architecture and Specifications., Object Management Group, 2000.

[9] OMG. Trading Object Service Specification 1.0, Object Management Group, 2000.

[10] Beedle, M., Devos, M., Sharon, Y., Schwaber, K. and Sutherland, J. SCRUM: An extension pattern language for hyperproductive software development. in Harrison, N., Foote, B. and Rohnert, H. eds. *Pattern Languages of Program Design 4*, Addison-Wesley, 1999.

[11] Weiser, M. The computer for the 21st Century. *Scientific American*, *265* (3). 94-104.

[12] JUnit. JUnit, Testing Resources for Extreme Programming, http://www.junit.org

[13] Kölling, M. and Rosenberg, J., Blue - A Language for Teaching Object-Oriented Programming. in *27th SIGCSE Technical Symposium on Computer Science Education*, (1996).

[14] Sun Microsystems Java Remote Method Invocation Specification.

[15] Gosling, J., Joy, B. and Steele, G.L. *The Java language specification*. Addison-Wesley, Reading, Mass., 1996.

[16] Arnold, K. *The Jini Specification*. Addison Wesley, Reading, Mass., 1999.

[17] Matena, V. and Hapner, M. Enterprise JavaBeans 1.0 Architecture Specification, Sun Microsystems Inc., 1998.

[18] IBM. Toad Development Enviroment, http://www.haifa.il.ibm.com/projects/systems/cot/toad/

[19] NetBeans RMI. NetBeans RMI Module Home Page, http://rmi.netbeans.org/