Aspect-Oriented Smart Proxies in Java RMI

by

Andrew Stevenson

A thesis
presented to the University of Waterloo
in fulfilment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2008

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Java's Remote Method Invocation (RMI) architecture allows distributed applications to be written in Java. Clients can communicate with a server via a local proxy object that hides the network and server implementation details. This loosely coupled architecture makes it difficult for client-side enhancements, such as method caching and validation, to obtain useful information about server state and implementation. Statically-generated custom proxies can provide a limited solution, but are troublesome to deploy and cannot change dynamically at runtime. This thesis presents a framework for Java RMI smart proxies using a distributed aspect-oriented platform. The framework allows server-controlled dynamic changes to Java RMI proxy objects on the client, without requiring changes to the client application code or development cycle. The benefits of this framework are demonstrated with three practical examples: method caching, client-side input validation, and load balancing.

Acknowledgements

I would like to first and foremost thank my supervisor, Steve MacDonald, for his guidance and support throughout my thesis writing. His editing efforts and revision comments are very much appreciated, and his ideas strongly influenced the direction and emphasis of my thesis. I would also like to thank my thesis proofreaders, Anne and Martin Wills, for their excellent editing and organizational feedback.

I am grateful to my official thesis readers, Ondřej Lhoták and Paul Ward, for volunteering their time to read my thesis and providing feedback to improve it. I also want to acknowledge my course professors at the University of Waterloo, from whom I learned a lot, and my fellow PLG students who provided an enjoyable social environment in which to do research. Finally I would like to thank my cousin, Kate Breedon, for her ongoing encouragement and enthusiasm.

Contents

List of Figures

Chapter 1: Introduction

In a distributed software environment, physically separated hosts work together to achieve results that a single host cannot achieve on its own. Within this topology, client applications consume the services made available by servers. Various design patterns are used to ease server-client communication for software developers, such as the Proxy design pattern [6] used in the Java Remote Method Invocation (RMI) specification [7]. Transparency to the client application and encapsulation of network code are two major benefits of such a proxy design, and are reasons for its widespread use and acceptance.

Software enhancements can be made to both clients and servers to help streamline the communication between them. Server enhancements have a distinct advantage over client enhancements because of the server's more extensive knowledge of the distributed system. A server is aware of each of its clients, and precisely how those clients are using its resources. This knowledge is valuable when determining how to allocate server resources in order to serve all clients as efficiently as possible. In contrast, the client has a limited view of the overall system. At best, it only has information about its own communication with the server. Although the client proxy is a logical extension of the server, it cannot easily benefit from the server's intelligence because of its physical isolation.

This paper proposes a framework that allows the intelligence and knowledge of the server to be used at the client. This results in "smart" clients and allows for better coordination between both ends of the communication channel. The framework uses a distributed aspect-oriented programming implementation, set in a Java RMI environment.

1.1 Loose Coupling vs. Server Knowledge

The Java RMI specification is designed to allow clients and servers to communicate while keeping their implementations as loosely coupled as possible. The loose coupling between client and server, elaborated in Chapter 2, reduces implementation dependencies in the

distributed software so that components can be modified without fear of breaking other components in the system. Java RMI hides internal implementation details among hosts and uses remote interfaces to achieve loose coupling, a property that is considered beneficial to software development.

Although loose coupling reduces dependencies between server and client, it is detrimental to implementing effective client-side communication enhancements because the implementation details of the server are hidden from the client. The more information the client has about server state and implementation, the better it can coordinate with server-side changes resulting in more efficient client-server communication. The increased efficiency between client and server is primarily realized by eliminating unnecessary remote method invocations from a client to the server. This has the secondary benefits of reducing network traffic, decreasing server load, and speeding up the client application.

Loose coupling and effective client-side enhancements are both desirable, but it appears one of them can improve only at the cost of the other. Giving clients internal server-side implementation details comes at the cost of breaking the loose coupling between client and server. Conversely, the current loosely coupled Java RMI architecture prevents clients from using server knowledge to enhance their communication channel with the server.

This thesis proposes a solution that keeps the client and server software components loosely coupled while allowing server state information and implementation knowledge to be shared by client-side enhancements. The goals in Section 1.2 describe desirable characteristics of the proposed framework, used as criteria to determine if the solution maintains the loosely coupled nature of Java RMI while still exposing server knowledge for client-side enhancements.

1.2 Goals

This thesis identifies four characteristics of a framework that accommodates both server knowledge sharing and the loose coupling of distributed components. A description accompanies each goal indicating how it supports either knowledge sharing or loose coupling. A solution that maintains the loose coupling of Java RMI while still providing server knowledge

to the client must satisfy all four goals.

1. **Make server context available to the client.** In general, the more information a program has about its environment the better decisions it can make and the more optimal its solution becomes. Servers have a wide view of a distributed software system, whereas a client's view is restricted. By making the server state and implementation details available to the client, the client's knowledge of its environment is substantially enhanced. Client-side enhancements such as cache policies and validation rules can take advantage of this knowledge to create a more intelligent and optimized client. Any solution that makes knowledge about internal server state or implementation details available to the client meets this goal.
2. **Obliviousness of the client application layer.** To help maintain a loose coupling between the client and server, the client application layer should be oblivious to the knowledge provided by the server. The application layer is concerned with solving problems and performing tasks in its domain and should not need to decide how to put server-side knowledge to proper use. The client application layer clearly requires remote services to achieve its objectives, but its core concern does not involve the communication efficiency between itself and the server.
3. **No changes to existing client application.** An existing client application that complies with the standard Java RMI specification should be able to apply the proposed framework without any changes to its source code or development process. The developers should not be required to use any new libraries in their source code or new compiling tools in their build process. This goal ensures that the coupling between server and client does not tighten by introducing new code in the client that depends on server state or implementation. This also agrees with the spirit of the second goal, that the client application layer does not need to do anything special and is oblivious to changes that occur in lower layers of the Java RMI architecture.
4. **Dynamic sharing of server context.** The first goal requires that server knowledge is shared with clients, but this knowledge sharing can conceivably occur at compile time,

resulting in client programs that must be halted and recompiled whenever the server wishes to impart new knowledge. A dynamic knowledge sharing framework allows server state information and implementation details to be transferred to the client at runtime, and automatically altering the behaviour of clients on the fly. This goal ensures a flexible and adaptable framework for a frequently changing server environment.

These goals combine to ensure any solution satisfying all four goals can share server knowledge with the client while still maintaining the loose coupling between server and client established by the Java RMI architecture. A solution that fails to fully meet any one of these goals severely limits its ability to both share server knowledge and remain loosely coupled.

1.3 Benefits

This thesis introduces a distributed dynamic aspect-oriented solution to satisfy the goals described in Section 1.2. In additional to satisfying these goals, the solution also has several separate benefits.

- **Communication performance/efficiency.** The biggest benefit of having server environment information available on the client is to improve the efficiency of client-server communication. This improvement is typically realized through the elimination of unnecessary remote calls from clients to the server, which speeds up the client application, reduces network traffic, and reduces the load on the server.
- **Client customization.** The proposed framework allows the server to customize enhancements on a per-client basis. Since each client may have a unique relationship with the server, the server can provide selective knowledge to each client individually to improve the entire distributed system. An example of this type of customization is shown in Section 4.2.3.
- **Easy to apply to existing Java RMI applications.** Goals 2 and 3 make the proposed solution easy to incorporate into existing distributed Java applications that use standard

Java RMI. The solution is non-intrusive to the client application because it does not require any client-side source code changes and it assumes that the client is oblivious to enhancements supplied by the server. Since the responsibility for client-side enhancements is shifted to the server, the server needs to include additional source code to manage the distribution of remote aspects.

1.4 Contributions

This thesis combines distinct methodologies and technologies, namely aspect-oriented programming and distributed programming, in a Java environment. It therefore provides several contributions to these areas, especially in the domain where these areas intersect.

- **Use aspects to capture client-side enhancements.** By viewing client-side communication enhancements as crosscutting concerns (described in Section 2.2.1) of the server, the theory and tools of aspect-oriented software development can be brought to bear on the problem. This insight influences many of the ideas presented in this thesis since the client-side enhancements are treated logically like server concerns even though they are physically separated from the server.
- **Server remotely weaving advice on client.** Remote aspects have been proposed to allow clients to advise server objects, but this thesis suggests using them in the opposite direction, server to client. By advising client objects from the server, the server is able to expose its context in a controlled way without breaking the loosely coupled remote communication architecture.
- **Dynamically modify Java RMI proxy object behaviour.** This solution uses a dynamic aspect-oriented framework to modify the behaviour of Java RMI proxy objects at runtime, effectively allowing the communication gateway from the client to the server to be controlled by the server itself. This level of control of the client proxy is rarely given to the server, opening up many potential uses. The proxy object is not "wrapped" in any additional control layer, but rather modified directly by remote aspects.

These novel features of the proposed framework build upon Java RMI client-server communication in several new directions. In particular, encapsulating client-side enhancements in aspects and remotely weaving them from server to client solves the knowledge sharing vs. loose coupling problem by combining typically unrelated methodologies in an innovative way.

1.5 Summary

This thesis combines several distinct paradigms and technologies to solve a specific problem relating to client-server communication in Java. It proposes a framework that makes server context available to the client while still abiding by the loose coupling imposed by the Java RMI architecture. The framework attempts to meet four important design goals, with an emphasis on client obliviousness, separation of concerns, and server information sharing.

Chapter 2 gives background information on the technologies involved: the RMI specification, especially the client proxy object; aspect-oriented programming, its terms and motivation; and Java Aspect Components (JAC) [16, 17], the Java-based dynamic distributed aspect-oriented platform that is used to implement these ideas.

Chapter 3 describes how the proposed solution works, from both a technical and abstract point of view. This chapter explains how aspects generated on the server can modify the client proxy object to support client-side enhancements with server knowledge. It shows how the JAC platform can be used to implement aspects representing client-side enhancements and how the server object can weave these aspects remotely on a client.

Chapter 4 shows three practical examples of communication concerns that can be enhanced with server knowledge. The first example, caching, allows return values to be cached on the client according to a server-defined cache policy. The second example, validation, handles erroneous remote method calls on the client according to validation rules specified by the server. The third example, load balancing, suggests an alternate method of balancing load across a server farm by using remote aspects to redirect client requests.

Chapter 5 describes alternate solutions to the loose coupling vs. server knowledge problem and how each solution meets or fails to meet the thesis goals in Section 1.2. The advantages and disadvantages of each solution are discussed in relation to the problem at hand and to the intended problem domain for that particular solution. Other related work in the area of distributed and aspect-oriented research is also discussed in this chapter.

Chapter 6 presents future work that can be done to improve the proposed framework's versatility and examines more advanced uses for the framework such as predictive caching. Lastly, this chapter summarizes and concludes the thesis.

Chapter 2: Background

This chapter contains background information about the technologies used in this thesis. The ideas in this thesis combine a Java RMI application with the aspect-oriented programming paradigm. Even readers familiar with Java RMI and aspect-oriented programming may benefit from this background since this chapter highlights characteristics pertinent to this thesis.

Section 2.1 explains Java RMI, the Java language protocol that allows Java applications to communicate remotely with each other. Java RMI is the problem domain in which the thesis problem in presented, so an understanding of the specification details is important. Section 2.2 describes the aspect-oriented programming paradigm and explains its goals, terminology, and benefits to software development. Understanding the aspect-oriented paradigm is important because it provides the basis for the suggested solution of this thesis. Section 2.3 introduces a dynamic, distributed aspect-oriented framework called Java Aspect Components (JAC), which allows the normally unrelated technologies of Java RMI and aspect-oriented programming to be combined, and thus supports the environment for the thesis research.

2.1 Java RMI

The Remote Method Invocation (RMI) specification for Java [7], developed by Sun Microsystems, allows Java programs to communicate with other Java programs running on different virtual machines that are typically separated by physical distance, but connected via a network.

Java RMI is designed to be a loosely coupled remote communication framework in three respects. First, the protocol used to transport method invocations over the network is loosely coupled to the Java application. Programmers are able to implement their own data marshalling and network transport classes for network protocols that may not be supported by the RMI implementation provided with the vendor's runtime library.

Second, the RMI specification attempts to make remote invocations as invisible to the Java application as possible. It achieves this by using a proxy object that resides in the client application. The client application can call methods on this proxy object like it does with any other local object. The proxy object is responsible for forwarding the method call to the server and returning the server's response to the client application. This results in a logical separation between the Java application logic and the underlying network code that provides the remote method invocation. The Java application can focus on its core concerns because it does not need to worry about lower-level remote communication considerations such as data marshalling and network transport protocols.

Third, RMI is designed with the goal that the client and server applications know as little as possible about the other's implementation details. In order for the client to invoke a method on the server, all it needs is the remote interface that the server has implemented and a proxy object for that server, as depicted in Figure 2.1 [7]. With this design, the server programmer is free to change the server's implementation without adversely affecting the client application.

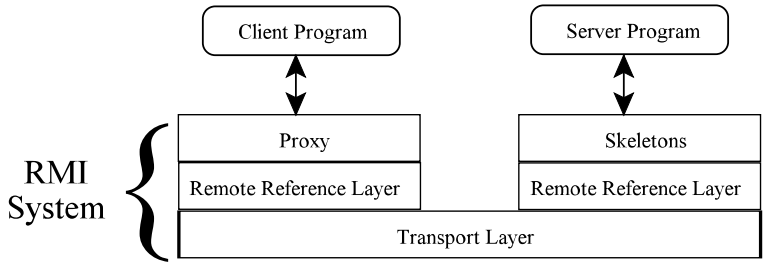

Figure 2.1 Java RMI allows remote communication while keeping the client and server loosely coupled.

2.1.1 RMI Proxy Object

Since this thesis proposes altering the proxy object at runtime, it is beneficial to examine the proxy object in more detail. An RMI compiler tool called `rmic` takes a server's implementation class and generates a corresponding stub class, as in Figure 2.2. The name of the generated stub class is the server's implementation class name concatenated with "_Stub". Both of these classes implement the remote interface that defines the services available for remote invocation.

```
// Stub class generated by rmic, do not edit.

public final class Calculator_Stub extends java.rmi.server.RemoteStub
    implements server.MathOperations
{
    // constructors
    public Calculator_Stub(java.rmi.server.RemoteRef ref) {
       super(ref);
    }

    // methods from remote interfaces

    // implementation of factorial(int)
    public int factorial(int $param_int_1) throws java.rmi.RemoteException
    {
       try {
           Object $result = ref.invoke(this, $method_factorial_0, new
java.lang.Object[] {new java.lang.Integer($param_int_1)}, -1108825836150346262L);
           return ((java.lang.Integer) $result).intValue();
       } catch (java.lang.Exception e) {
           throw new java.rmi.UnexpectedException("undeclared checked exception", e);
       }
    }
}
```

Figure 2.2 A stub class generated by `rmic`, based on the "Calculator" server implementation class.

Once the stub class is generated, it must be distributed to any clients who wish to use the server. RMI provides a naming registry to facilitate this distribution by binding a server's implementation object to a string name. When a client looks up this name in the RMI registry, an instance of the stub class for that server is returned to the client. This distribution process is presented in Figure 2.3 The stub instance is the proxy object that allows the client application to communicate with the server.

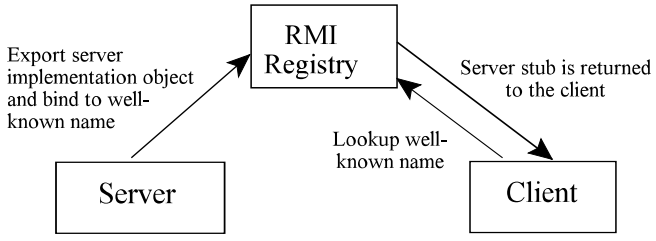

Figure 2.3 The RMI Registry service. A client can get a reference to the server using a well-known string name.

Without any further facilities, the proxy object behaviour is fixed and cannot be modified without re-coding and redistributing the server stub. As will be shown in this thesis, it is

possible to alter the proxy behaviour at runtime without sacrificing the benefits of loose coupling between the client and server.

Java version 1.3 introduced a dynamic proxy feature [21], allowing multiple arbitrary interfaces to be implemented in a single class at runtime. JDK 1.5 leverages this facility for the RMI architecture to create RMI client proxy objects dynamically if it cannot find the stub class created by `rmic`. This technique replaces statically-generated stubs with dynamically generated proxy objects. This thesis only considers `rmic`-generated proxies, though our approach should also work with dynamic proxies.

2.2 Aspect-Oriented Software Development

Aspect-oriented programming [11] is a programming paradigm that is intended to complement existing programming paradigms and address some of their limitations, namely the phenomena of code tangling and code scattering. Aspect-oriented programming in its modern form was conceived and developed by Gregor Kiczales and his team at Palo Alto Research Center, Inc., at the time known as Xerox PARC. A Java-style aspect-oriented language called AspectJ [1] arose from this research, and was quickly adopted as the standard aspect-oriented language because of its ease of use and similarities with the popular Java programming language.

Aspect-oriented programming is now part of a wider software engineering field called aspect-oriented software development, which encompasses other tools and software that aid in the development of aspect-oriented systems. These tools include source code and byte code weavers, aspect visualization software, and custom Java virtual machines for dynamic weaving, among others. The following sections explain the basic terms and concepts behind aspect-oriented software development.

2.2.1 Code Tangling and Code Scattering

The problems of code tangling and code scattering are the main justification for using an aspect-oriented approach. Both code tangling and code scattering refer to how code of similar function is distributed, and often duplicated, across different classes. Most methods in a large software system perform a variety of secondary tasks in addition to the core task to which they are assigned. In aspect-oriented parlance these tasks, or areas of interest, are called *concerns*. When a concern affects multiple unrelated classes, the concern is said to *crosscut* those classes. These crosscutting concerns are precisely what aspect-oriented programming is designed to encapsulate. Common examples of crosscutting concerns include logging, tracing, persistence, and profiling, and usually affect a wide range of classes or modules.

Code tangling occurs in a method when code for a secondary concern is intertwined with code for the core concern. This is undesirable because the intent of the method becomes obscured by code for secondary concerns. For example, take a deposit() method for a bank account that updates an instance variable, shown in Figure 2.4. This method is clean, simple, and its intent is obvious from a glance. Real banking systems must consider several secondary concerns for important banking operations. These secondary concerns may include synchronization, database transactions, logging, and security, as shown in Figure 2.5. The code for these concerns become tangled up with each other, making it difficult to understand code for the core concern.

```
public class BankAccount {
      private int balance;
      public void deposit(int amount) {
            balance += amount;
      }
}
```

Figure 2.4 `deposit()` method with core concern.

```
public class BankAccount {
      private int balance;
      public void deposit(int amount) {
            SQLServerDataSource ds = new SQLServerDataSource();
            ds.setDataSourceName("SQLServer");
            ds.setServerName("server");
            Connection con = ds.getConnection("user", "password");
            Statement stmt =con.createStatement();
            stmt.executeUpdate(getSQLStatement(this, amount));

            Log log = new BankAccountLog();
            log.addDepositInfo(this, amount);

            balance += amount;

            stmt.close();
            con.close();
      }
}
```

Figure 2.5 `deposit()` method with tangled database and logging concerns.

Code scattering is the phenomenon of crosscutting concern code being scattered throughout different classes in the program, as shown in Figure 2.6. For instance, if the programmer wants to log every public method in the program, they need to write logging code everywhere that a public method exists. This scattered logging code is likely to be the same throughout the program, and duplicate code should be avoided whenever possible. If the programmer wants to change the logging code they have to change it in every public method instead of changing it once in a self-contained module. Furthermore if the programmer creates a new public method or changes an existing method to be public, logging code must be added.

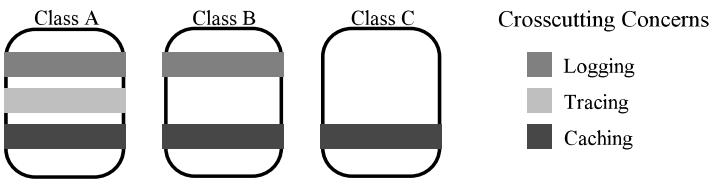

Figure 2.6 The code for the caching concern is scattered across multiple classes, and in Class A the code for multiple concerns are tangled together.

2.2.2 Aspects

The aspect-oriented paradigm builds upon existing paradigms by introducing a new structure called an *aspect*, which encapsulates the information for a single concern. An aspect contains two types of sub-structures: *pointcuts* and *advice*. Advice describes the concern's behaviour, and pointcuts indicate where to apply the advice in the program. Used together, pointcuts and advice can specify how a secondary concern alters classes and methods in the core program. Section 2.2.3 describes pointcuts in greater detail, and Section 2.2.4 covers the uses and types of advice.

2.2.3 Join Points and Pointcuts

In aspect-oriented software, a *join point* is an event that occurs during the execution of a program to which advice can be attached. Valid join points vary from language to language, but most aspect-oriented languages include method execution and field access in their join point model. The only join points that this thesis requires are method executions, so the pointcut examples in this thesis exclusively target method executions.

Pointcuts specify a subset of join points of interest, using a *pointcut language*. Pointcut languages typically use a regular expression-like syntax to specify this subset, and allow the programmer to include or exclude join points based on scope, access modifier, name, data type, number and type of arguments (for methods), and other distinguishing language elements.

2.2.4 Advice

Advice is code inserted at the join points defined by the pointcut. When this insertion occurs, the aspect is said to be *advising* or *decorating* the join point. Conceptually, a piece of advice

determines an aspect's behaviour or how it alters the program. For a logging aspect, the advice may contain code that prints messages to a file, or records the state of the program.

Most aspect-oriented programming languages support three types of advice: before, after, and around. *Before advice* is used to attach the advice before the join point. This means the advice code will run immediately before the join point code runs. Likewise, *after advice* runs after the join point code has finished.

Around advice can specify code to run both before and after the join point, and additionally can control whether the join point runs at all. A special method, typically named `proceed()` is reserved to trigger the join point execution, and this method can be used anywhere in the advice code. If the `proceed()` method does not get called in the advice then the join point will not run. In this way, *around advice* can effectively guard the execution of any join point it advises.

2.2.5 Code Weaving

A code weaver's purpose is to merge or *weave* an aspect's advice into the main program. It determines where to weave the advice by resolving the associated pointcuts into join points within the program. Once the join points are known, it weaves the advice before, after, or around the join point depending on the type of advice being woven.

Code weaving can be done at different stages in the compile-link-run process. A source code weaver transforms the source code directly, and then sends the transformed program to a normal compiler. A byte-code weaver can weave byte-code directly, such as a compiled Java class. Often the compiler is combined with the weaver so that both weaving and compilation is performed at the same time, as is the case with the compiler for AspectJ [13]. These are all examples of static weaving, because the weaving is performed at compile time.

In contrast, dynamic weaving however allows advice to be woven at runtime. Aspect-oriented frameworks that support dynamic weaving have the ability to deploy or remove aspects while the program is running. This offers a lot of flexibility to the system, but also introduces some dangers because developers have no way of knowing which aspects affect their code when

the program is running.

2.3 JAC

Java Aspect Components (JAC) [14] is a dynamic aspect-oriented software development framework implemented in the Java programming language. Unlike other common Java-based aspect-oriented programming languages such as AspectJ, JAC uses standard Java classes and does not extend Java with any new language constructs.

In JAC, any Java class that extends the `AspectComponent` class represents an aspect. The `pointcut()` method is used in the aspect class's constructor to define a pointcut expression and associate advice with that pointcut. There are several overloaded versions of this method to construct a pointcut and advice in different ways, but we will only consider one of these methods. The syntax for this `pointcut()` method is:

```
void pointcut(String objects, String classes, String methods, Wrapper advice);
```

The first three arguments are used to define a set of join points in the program, and the final argument contains the advice to be woven. The arguments are:

- *`objects`* – a regular expression string that identifies the objects to which the advice should be applied. JAC allows advice to be woven on a per-object basis so that one object can exhibit aspect behaviour but another does not, even if both objects are instantiated from the same class. An identifier for each object is created and maintained by JAC in an object repository. The mechanism for retrieving and using this identifier is described in further detail in the future work section of Chapter 6. For current uses, the string ".*" is used to represent all instances.
- *`classes`* – a regular expression string that identifies the classes to which the advice applies. Fully qualified class names must be used (i.e, class names must be prefixed with their package name). The string ".*" represents all classes.

- *methods* – a regular expression string that identifies the methods to be advised. The syntax for this parameter is "*methodName*(*arg1_type*,*arg2_type*, ...):*return_type*". The string ".*(.*):.*" represents all methods.
- *advice* – In JAC, the advice for an aspect is stored in a separate class that extends `Wrapper`. This parameter associates a wrapper object with a pointcut declaration.

2.3.1 JAC Wrappers

A JAC wrapper is a Java class that extends JAC's `Wrapper` class. The aspect developer should override the `invoke()` method inherited from Wrapper. The body of the `invoke()` method is the advice that will be woven at the join points specified by the pointcut. JAC only supports *around* advice, but *before* and *after* advice can be simulated by placing a `proceed()` call at the end or beginning of the `invoke()` method respectively.

The `invoke()` method has one parameter of type `MethodInvocation` that allows reflection on the join point being advised. Information about the join point receiver object method name, argument types and values, and return type can be obtained from the `MethodInvocation` object. Furthermore, the advice can alter the value of the arguments passed into the advised method, and can examine and alter the return value passed back to the application.

Consider an example to record successful and failed login attempts to a system. The advice wrapper class, shown in Figure 2.7, includes the `invoke()` method that contains the advice code. This code calls the advised join point method, `login(int)`, via `proceed()` which returns a boolean value that determines if the login was successful. The advice code in Figure 2.7 records "Successful login" or "Failed login" to a log depending on the result of the login method.

```
class LoggingWrapper extends Wrapper {
      Log log = new Log();

      public Object invoke(MethodInvocation mi) {
            // Before advice goes here

            Boolean loginOK = (Boolean) proceed(mi);

            // After advice follows
            if (loginOK)
                  log.add("Successful login");
            else
                  log.add("Failed login");

            return loginOK;
      }
}
```

Figure 2.7 An example of logging advice in JAC.

Where the advice wrapper in Figure 2.7 specifies the logging behaviour to execute, the pointcut declaration, shown in Figure 2.8, specifies that the advice should be executed whenever the `login()` method runs. A pointcut declaration in JAC is typically placed in the aspect class constructor so that the pointcut is defined when an aspect class is instantiated. The "`.*`" and "`ca.uwaterloo.*`" arguments narrow the scope of target join points to methods defined in objects instantiated from any class in the `ca.uwaterloo` package. Furthermore, the "`login(int):boolean`" argument specifies that the target join points should be methods named `login()` that take a single `int` parameter and return a `boolean` value. The `new LoggingWrapper()` argument associates the advice, from Figure 2.7, with the `login()` method join point.

```
public class LoggingAC extends AspectComponent {
      public LoggingAC() {
            pointcut(".*",     "ca.uwaterloo.*",
                               "login(int):boolean",
                               new LoggingWrapper());
      }
}
```

Figure 2.8 An example of a logging pointcut in JAC.

Although not strictly necessary, the wrapper class is often placed as an inner class to the aspect

class. Figure 2.9 combines the advice in Figure 2.7 and the pointcut in Figure 2.8 in a complete example of a logging aspect written in JAC.

```
public class LoggingAC extends AspectComponent {
      public LoggingAC() {
            pointcut(".*",     "ca.uwaterloo.*",
                               "login(int):boolean",
                               new LoggingWrapper());
      }

      class LoggingWrapper extends Wrapper {
            Log log = new Log();

            public Object invoke(MethodInvocation mi) {
                  // Before advice goes here
                  Boolean loginOK = (Boolean) proceed(mi);
                  // After advice follows
                  if (loginOK)
                        log.add("Successful login");
                  else
                        log.add("Failed login");

                  return loginOK;
            }
      }
}
```

Figure 2.9 A logging aspect in JAC.

2.3.2 Distributed Aspects

In addition to being a dynamic aspect-oriented language, JAC also has a robust distribution feature built into its framework [16]. The distribution feature allows aspects on one host to be easily woven into remote objects running at a remote host. The only requirement is that the underlying application must be run in a special JAC container that acts as a distribution server. When this container recognizes an incoming aspect sent from a remote host, it automatically weaves the aspect into its currently running Java application using the dynamic code weaving facilities of the JAC framework.

To identify hosts that can send and receive aspects remotely, JAC stores a network topology of all the hosts that comprise the distributed system. This topology can be set up in a

configuration file before the application is run, or hosts can be added to and removed from the topology at run time.

The JAC framework also comes with a variety of prepackaged aspects in categories like distribution, monitoring, persistence and transactions. The most relevant aspects to this thesis are the distribution aspects, composed of the deployment, consistency, broadcasting, and load balancing aspects. Chapter 4 presents an alternate load balancing implementation and briefly compares it to JAC's load balancing aspect.

2.4 Summary

Java RMI is a widely used communication technology for distributed Java applications that encourages loose coupling between clients and servers. It uses the proxy design pattern to help enforce loose coupling and make it easy for the client application developer to find remote server objects and call their methods.

Aspect-oriented software development is used to isolate and define cross-cutting concern of a software system, and help reduce the tangling and scattering of these concerns throughout the program. Aspects, advice, and pointcuts are aspect-oriented language constructs used to encapsulate these concerns and define their behaviour and scope.

Figure 2.10 depicts how this thesis combines Java RMI and aspect-oriented software development to improve communications between clients and servers. JAC lies at the heart of this fusion because it provides a platform for dynamic distributed aspect-oriented software development in Java.

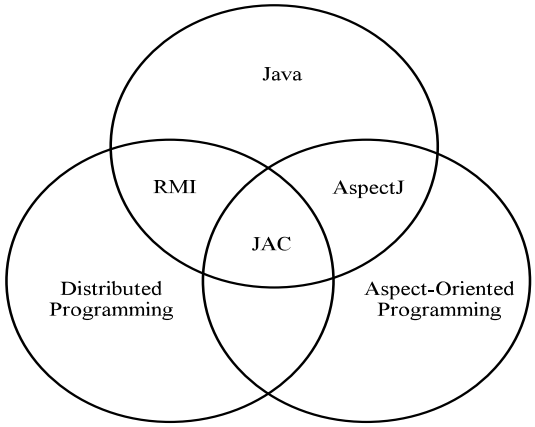

Figure 2.10 This thesis combines the technologies and ideas of Java, distributed programming, and aspect-oriented programming.

Chapter 3: A Dynamic Aspect-Oriented Solution

3.1 Introduction

Chapter 2 discussed the advantages of loose coupling in RMI, but the properties that make loosely coupled components so attractive can also be detrimental to efficient communication between client and server. One such property is the server's facility to hide implementation details from the client, which ensures the client software does not depend on the server's implementation and allows the server to change without adversely affecting its clients. However, if a client wants to optimize its communication with the server it is limited by its lack of knowledge about the server state and implementation. This thesis considers client-side enhancements that can benefit from knowledge of server-side implementation details.

The problem considered in this thesis is to provide server implementation details to clients so they can act intelligently, without compromising the benefits of loose coupling that RMI supports. This thesis proposes a solution that gives a client the knowledge it needs and still maintains the loose coupling of RMI: make the server responsible for developing and deploying the necessary client-side enhancements.

Shifting the development responsibility for client-side enhancements to the server eliminates the need for clients to know about the server's implementation details. Furthermore, the server can deploy these enhancements dynamically to its clients without clients needing to halt or recompile their applications. This solution maintains the advantages of RMI's loosely coupled architecture and provides the client with the server state and implementation knowledge it needs to communicate efficiently. Practical examples such as caching, validation, and load balancing are provided to demonstrate these ideas and their benefits. The caching example is used throughout this chapter and Chapter 4 explores all three examples in greater detail.

3.2 Aspect-oriented paradigm applied

The client-side enhancements mentioned in Section 3.1 can be considered crosscutting concerns of the server. However, instead of crosscutting just classes or modules these concerns crosscut hosts in the distributed system. Therefore a distributed, dynamic, aspect-oriented platform is required to represent and apply these client-side enhancements correctly, a role that JAC fills perfectly.

Aspects that encapsulate the client-side enhancements can be created on the server in JAC, then remotely woven on a client using the distribution support included in JAC's framework. These aspects can alter the behaviour of the client to improve the overall system's efficiency, but must do so in a way that is not intrusive to the client application layer. The RMI proxy object is the ideal place to weave aspects from the server for client-side enhancements for two reasons. First, altering its implementation does not require changes to the client application. Second, the RMI proxy object is the gateway from the client to the server, through which all remote calls to the server must flow. Behavioural changes on the client proxy object ensures that each remote call is affected by those changes.

3.3 Client

The client application consists of a Java application that uses standard Java RMI with standard `rmic`-generated stubs. In order for the client application to realize the benefits of remote aspects sent from the server, the JAC framework must be present on the client and the client application must be run inside a JAC container. It should be noted that running the client application inside a JAC container does not involve client calls to JAC libraries, JAC initialization code, nor any other references to JAC in the client application code.

The first step to running an application under JAC is to create an application descriptor. An application descriptor is simply a text file with a `.jac` extension that configures the application. For client applications, only the application name and launching class need to be specified as shown in Figure 3.1.

```
applicationName: RMI client
launchingClass: mypackage.RunClient
```

Figure 3.1 The application descriptor file for a Java RMI client application.

The application name is a string that uniquely identifies an application running in a JAC container, and is used by the server to weave aspects remotely on the client. The launching class is a fully qualified Java class name that contains a `public static void main(String[])` method and is launched when JAC starts the application. To run the client application in a JAC container, simply launch the JAC framework passing the path to the application descriptor as an argument. Figure 3.2 shows how to launch an RMI client application in a JAC container.

RMI client application launched normally:

```
java mypackage.RunClient arglist
```

RMI client application launched in a JAC container:

```
java -jar jac.jar -D containerName client.jac arglist
```

Figure 3.2 The differences between running the client application normally and running it in a JAC container.

The `-jar jac.jar` option runs the `org.objectweb.jac.core.Jac` class that, among other things, enables JAC's aspect weaving class loader. `-D` enables the distribution mode on the JAC container with `containerName` providing a string identifier for the JAC container, used by other remote JAC containers that wish to communicate with the local container. `client.jac` represents the path to an application descriptor file that, as shown in Figure 3.1, contains the name of the application and the class that JAC launches to start the application. `arglist` is a list of arguments passed to the launching class defined in the application descriptor.

The client must notify the server about its JAC container so the server can send remote aspects when appropriate. The server needs to know the client application name (as defined in the application descriptor), and the name and location of the JAC container. The location can be an IP address, computer name, or other network identifier used by the client host.

No other changes to the client application are needed. When the JAC container receives an aspect from the server, JAC will automatically and dynamically weave the advice into the

RMI proxy object. The client application does not need to be halted or recompiled for the weaving process.

If the aspect is received while the proxy object is waiting for a remote method to return, the behaviour does not manifest until a subsequent call from the client application is made to the proxy object. This avoids concurrency issues such as *after advice* executing without the execution of its corresponding *before advice*.

Client application developers are no longer burdened with writing client-side enhancement code since that responsibility now rests with the server. Developers can focus on the core concerns of their client application rather than a secondary concern, such as efficient communication with the server. In fact, the client application likely does not know which aspects are applied to its RMI proxy object at any given time, nor should it. The client application simply calls the RMI proxy object's methods when it needs a remote service, and the server manages all the client-side enhancements.

3.4 Server

Similarly to a client, the server is a standard RMI server implementation. The server implements a remote interface, and `rmic` generates stubs from the implementation class. The server implementation object is typically exported to an RMI naming registry so that clients can find the server easily through a name lookup.

The server requires the presence of the JAC framework and, like the client, runs inside a JAC container. An application descriptor that defines the server's application name and launching class is required. The server does not need to notify its clients about its JAC container or application name because clients do not need to send aspects to the server. In this thesis remote aspects travel in one direction only, from the server to the client. This unidirectional model is a characteristic of the solution proposed by this thesis, and not a restriction of JAC. JAC has the capability to weave aspects remotely in both directions, but this thesis only uses JAC for server-to-client weaving.

3.4.1 Client-side aspect

The contents of client-side aspects vary depending on the type of enhancements they represent, but these aspects all have several things in common. The pointcut, in particular, must be carefully defined to target only the join point in the client's RMI proxy object. If the pointcut matches join points in the client application code it can cause unexpected behaviour in the client.

Figure 3.3 shows a sample pointcut definition for a client-side aspect. The first argument identifies the Java objects where the aspect's advice is woven, since JAC is a dynamic aspect-oriented framework that has the ability to weave aspects on a per-object basis. The string ".*" indicates that all instantiations of the class specified in the second argument are considered as matching join points. The current framework dictates that this argument should always target all proxies, but selective proxy targeting is possible and further discussed in Chapter 6.

```
pointcut(    ".*",
             "mypackage.ServerImpl_Stub",
             "login(int,java.lang.String):boolean",
             new CachingWrapper());
```

Figure 3.3 A pointcut declaration targeting the `login()` method in the `ServerImpl_Stub` proxy object.

The second argument identifies the classes that contain matching join points and is set to the fully qualified server implementation class name concatenated with "_Stub". This class matches the `rmic`-generated stub used by the client, and ensures that the RMI proxy object is advised without inadvertently advising other objects in the client application. The third argument is a string that indicates the method signature of the target join point. This value is set to one or more of the method signatures defined in the remote interface. Since the `rmic`-generated stub class must implement the remote interface [7], these methods are guaranteed to exist. Multiple methods can be specified using a wildcard character or separated using the | (alternation) operator. The final argument associates the aspect's advice with the pointcut definition.

In addition to pointcuts and advice, the remote aspect may contain instance variables to support the behaviour of the aspect. These instance variables are typically used to store state that persists between invocations of the RMI proxy object's methods, such as a cache.

3.4.2 Weaving client-side aspects remotely

Once a client-side aspect is written, tested, and compiled at the server it can be woven remotely on a client. This is accomplished by calling the static `remoteWeaveAspect()` method in the `org.objectweb.jac.core.Jac` class. This method takes four string arguments, as shown in Figure 3.4.

```
void Jac.remoteWeaveAspect(String applicationName,
                           String clientContainer,
                           String aspectClass,
                           String aspectConfigPath);
```

Figure 3.4 The `remoteWeaveAspect()` method signature.

The `remoteWeaveAspect()` method has the following arguments:

- *`applicationName`* – The name of the application as defined in the client's application descriptor. If the client uses the application descriptor file from Figure 3.1, this argument would contain "`RMI client`".
- *`clientContainer`* – The client container name is determined when the client application is launched in the JAC container. This is the name that appears after the -D option in the Java application launch command line string.
- *`aspectClass`* – This argument is the fully qualified class name of the aspect being woven. For example, "`mypackage.CachingAC`".
- *`aspectConfigPath`* – This is the file path of the aspect's configuration file. This file can contain instructions to configure the aspect upon initialization. An empty file results in no custom configurations for the aspect.

3.4.3 Remote Weaving Condition

The server decides not only what client-side enhancements to provide, but also the condition that triggers the aspect to be remotely woven on a client. This condition is dependent on the type of client-side enhancement in question, but the remote methods that implement the server object are logical places to have this condition. When a client invokes a remote method, the server can identify the client, examine the remote method's arguments, and make an informed decision about which aspects to weave remotely on that client. A client's usage habits such as calling frequency, server resources used, time spent servicing the request, etc. can be collected over time to help further inform this decision on a per-client basis. Since the server has a wider picture of the usage tendencies of all its clients, it can provide client-side enhancements that benefit the distributed system as a whole. This idea is demonstrated in Chapter 4 with caching, validation, and load balancing enhancements.

3.5 Implementation Problems

Several unexpected problems were encountered during the development and testing of the client-side enhancements remotely woven by the server. The most significant problem is JAC's inability to weave the remote aspect on the Java RMI proxy object. The version of JAC used in the implementation of this thesis does not weave aspects on a Java RMI proxy object retrieved from an RMI registry. This unexpected behaviour is not accompanied by any error messages or other signs of failure, but rather runs as if the JAC system is not aware of the aspect or its associated advice. If print statements are placed in the advice code and the aspect is woven on the proxy object, the expected output messages do not appear when the appropriate proxy methods are called from the client application.

The solution to this problem requires the client application to reinstantiate the stub class retrieved from the RMI registry, illustrated in Figure 3.5. The `newInstance()` method of Java reflection is used to create a new instance of the proxy object, and the remote reference

from the original proxy object is used in the constructor. This essentially creates a clone of the original proxy object with an identical remote reference. JAC is able to weave aspects on this clone proxy, unlike the original proxy object. This workaround suggests that JAC is able to weave classes loaded with the default classloader, but is unable to weave classes loaded with the RMI classloader. However, when the classloaders of the original and clone proxy objects are examined they appear identical. Both proxy instances appear to share the classloader that allows JAC weaving to occur. JAC uses a custom classloader to achieve aspect weaving, but it is not clear why this classloader weaves for one instance of a class but not for another.

```
// Retrieve original stub from RMI registry
Registry registry = LocateRegistry.getRegistry();
RemoteInterface stub = (RemoteInterface) registry.lookup("ServerObj");

// Use Java reflection to construct a clone stub object
RemoteRef remoteRef = ((RemoteStub)stub).getRef();
Class stubClass = stub.getClass();
Constructor stubCons = stubClass.getConstructor(new Class[]{RemoteRef.class});
RemoteInterface newStub = (RemoteInterface) stubCons.newInstance(new Object[]
                                                        {remoteRef});
```

Figure 3.5 This workaround instantiates a clone of the RMI proxy object. Subsequent remote calls should use this proxy instead of the original proxy.

This workaround needs to be performed on the client prior to the invocation of remote interface methods on the proxy object. This violates the thesis goal of having no source code changes to the client application. The statically-generated stub distribution technique discussed in Section 5.1 can be used here to apply this workaround without intruding on the client application developer. This workaround can be placed into the server-side statically-generated stub and then distributed to the clients, allowing the workaround code to execute whenever the proxy object is instantiated on the client.

The difference between this workaround and the statically-generated proxy strategy from Section 5.1 is the ability for the client-side proxy object to be modified dynamically. The workaround code in Figure 3.5 bootstraps the dynamic aspect-oriented capabilities of JAC allowing dynamic remote advice weaving for the lifetime of the proxy object. In contrast, the server-side `rmic`-generated stub is static, and can only be distributed once to the client during its lifetime. Combining the distribution technique in Section 5.1 with the workaround code in

Figure 3.5 solves the proxy object weaving problem while still maintaining the non-intrusiveness goals of this thesis.

3.6 Summary

By using the distribution and dynamic weaving features of the JAC platform, a typical Java RMI application can be enhanced to allow remote aspect weaving from the server to the client. Incorporating this ability into an existing Java RMI application is unintrusive, since no additional library calls or source code changes are required in the client program. Instead, the client application must run inside a JAC container, achieved by creating a simple application descriptor file and running the client program with a modified command line. The server constructs aspects and remotely weaves them on clients, modifying the behaviour of the client-side proxy object.

This chapter presented the technical details and abstract elements of the proposed framework, but these concepts are more easily understood using concrete examples. Chapter 4 describes three practical uses of the ideas presented in this chapter, and reiterates the framework elements in the context of the three examples.

Chapter 4: Practical Applications of Client-Side Aspects

The dynamic aspect-oriented framework for client-side enhancements discussed in Chapter 3 is flexible and highly adaptable to a system's requirements, allowing the framework to efficiently address a variety of concerns involving client-server communication. These aspects can involve performance concerns on the network, client, or server; or can enforce business rules governing the interactions between client and server. This chapter explores three common examples of such concerns: method value caching, method argument validation, and load balancing.

Section 4.1 extends the discussion of method value caching, used as an example throughout Chapters 2 and 3. Section 4.2 introduces method argument validation, the ability to validate remote method arguments on the client before invoking an expensive remote method on a server. Finally, Section 4.3 shows how remote aspects can be used in a novel way to balance load across multiple servers.

4.1 Method Value Caching

Method value caching, henceforth *caching*, is remembering the return values of a method then retrieving them when the method is subsequently called using the same arguments. By retrieving return values that have already been computed, unnecessary method calls can be avoided. This is a valuable optimization in a remote object system since calling remote methods involves expensive network communication that can be avoided by returning cached values. It is especially valuable when the amount of data exchanged between client and server is large.

In this case the method is part of the remote interface exposed by the server, the arguments are sent from the client to the server, and the server computes and returns a value to the client. The RMI proxy object is a suitable place for a caching mechanism because it is a gateway for both the remote method arguments and the remote method's return value.

The caching solution in JAC has three parts: the client-side caching aspect, the client implementation, and the server implementation. The server implementation decides when to

weave the caching aspect on the client and can provide configuration options to the aspect to customize its behaviour. The caching aspect contains a pointcut and advice, the latter being the caching implementation. The client implementation can be a typical RMI client that calls methods on a remote object via an `rmic`-generated stub instance.

4.1.1 Client-side Caching Aspect

In JAC, aspects are Java classes that extend the `AspectComponent` class, and by convention end with "AC". The pointcut is declared in the constructor, as shown in Figure 4.1.

```
public class CachingAC extends AspectComponent {
      public CachingAC() {
            pointcut(".*", "Calculator_Stub", "factorial(int):int",
                  new CachingWrapper(this), null);
      }
}
```

Figure 4.1 The `CachingAC` class contains a pointcut declaration in its constructor.

For advice to be woven on the client's RMI proxy object, the `pointcut()` method's second argument must be the fully qualified name of that proxy object's class, as seen in Figure 4.1. Note the omission of a host argument in the pointcut definition, to identify where the join point resides. The pointcut does not need to know the join point host because the aspect is woven on a specific host provided by the server implementation.

Recall from Chapter 2 that the aspect's advice is contained in a separate class that extends the `org.objectweb.jac.core.Wrapper` class. This `Wrapper` class contains a method called `invoke()` that is overridden in its subclasses to support advice, as shown in Figure 4.2. The cache object itself is implemented as a Java hash table, mapping the integer argument of the `factorial()` method to its corresponding factorial value. The `CachingWrapper`'s constructor can simply delegate its work to the superclass's constructor. The body of the `invoke()` method contains the caching logic that checks to see if the integer argument already exists in the cache and retrieves its corresponding factorial value if it does. The `proceed()` method, which executes the join point method

`Calculator_Stub.factorial(int)`, is called when no matches are found in the local cache. This causes the invocation of the remote method, after which the return value is added to the cache with its argument as the lookup key.

```
public class CachingWrapper extends Wrapper {
    private Map<Integer,Integer> cache = new Hashtable<Integer,Integer>();

    public CachingWrapper(AspectComponent ac) {
       super(ac);
    }

    public Object invoke(MethodInvocation mi) throws Throwable {
      Object[] arguments = mi.getArguments();
      Integer arg = (Integer) arguments[0];
      Integer cachedAnswer = cache.get(arg);
      if (cachedAnswer == null) {
            // argument is not found in the cache, proceed with remote call
            Integer result = (Integer) proceed(mi);
            cache.put(arg, result);
            return result;
      }
      else {
            // argument is found in the cache, so return cached answer
            return cachedAnswer;
      }
    }
}
```

Figure 4.2 The `CachingWrapper` class contains advice for the caching aspect.

A caching mechanism is ideal for the `factorial()` method because a factorial is a function of its integer argument only, and does not depend on the state of the remote object. A cached factorial value never becomes stale, and thus no cache policy is needed to purge the cache. Furthermore, computing factorials for large integers is a computationally intensive task, so the client's performance increases with the use of a client-side cache. The advice in Figure 4.2 can be changed by the server to support a cache policy if needed, but this determination should be made by the server implementation, discussed further in Section 4.1.3.

4.1.2 Client Implementation

Any standard Java RMI client is able to make use of the framework proposed in this thesis. The key component that the client must have is an RMI proxy object, an instance of an `rmic`-

generated stub class, because this proxy object contains the join point targeted by the remote aspect's pointcut. When a remote aspect is sent from the server to the client, its advice is woven into the client's proxy object, giving control of remote method invocations to the advice code. Figure 4.3 shows the program control flow before the advice is woven and Figure 4.4 shows the program control flow after the advice is woven. The advice's caching code guards the call to the proxy object's `factorial()` method, preventing the remote method invocation if the factorial result already exists in the cache.

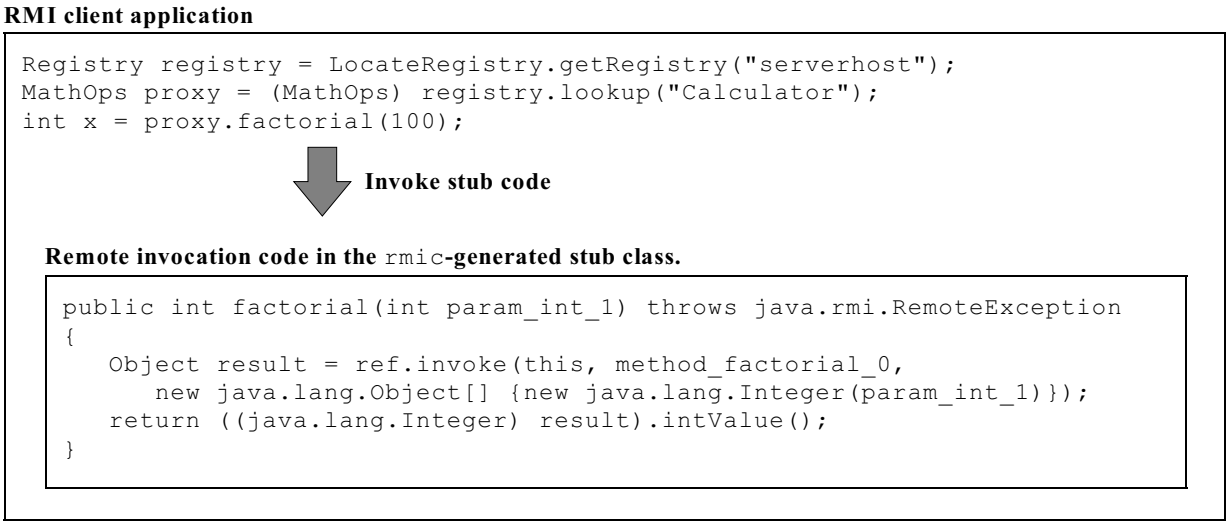

Figure 4.3 The program control flow before the remote advice is woven on the client.

RMI client application

```
Registry registry = LocateRegistry.getRegistry("serverhost");
MathOps proxy = (MathOps) registry.lookup("Calculator");
int x = proxy.factorial(100);
```

Call to factorial intercepted by aspect

Advice implementation from caching aspect.

```
public Object invoke(MethodInvocation mi) throws Throwable {
  Object[] arguments = mi.getArguments();
  Integer arg = (Integer) arguments[0];
  Integer cachedAnswer = cache.get(arg);
  if (cachedAnswer == null) {
    Integer result = (Integer) proceed(mi);
```

Invoke stub code in advice if result not cached

Remote invocation code in the `rmic`-generated stub class.

```
public int factorial(int param_int_1) throws java.rmi.RemoteException
{
   Object result = ref.invoke(this, method_factorial_0,
      new java.lang.Object[] {new java.lang.Integer(param_int_1)});
   return ((java.lang.Integer) result).intValue();
}
```

```
    cache.put(arg, result);
    return result;
  }
  else {
    return cachedAnswer;
  }
}
```

igure 4.4 The revised program control flow after the remote advice is woven on the client.

4.1.3 Server Implementation

The server application contains an object that implements a remote interface, and thus provides the service implementation for its clients. In a typical RMI application, this implementation is responsible for performing the computations and tasks associated with the service. Using the

`factorial()` method, shown in Figure 4.5, this task is to simply calculate the factorial value of a given integer.

```
public int factorial(int num) throws RemoteException {
	if (num == 0) return 1;
	int result = 1;
	for (int i = 1; i <= num; i++) {
		result *= i;
	}
	return result;
}
```

Figure 4.5 The primary task of the `factorial()` method is to calculate a factorial value.

To take advantage of server-provided client-side enhancements using distributed aspects, the server must weave these aspects on the client's proxy object. This remote weaving occurs in the service implementation method as a secondary task, in addition to the primary task of calculating the factorial. Figure 4.6 shows both of these tasks.

```
public int factorial(int num) throws RemoteException {
	Jac.remoteWeaveAspect("clientApp", RemoteServer.getClientHost(),
				"CachingAC", "caching.acc");

	if (num == 0) return 1;
	int result = 1;
	for (int i = 1; i <= num; i++) {
		result *= i;
	}
	return result;
}
```

Figure 4.6 The revised `factorial()` method with remote aspect weaving.

The static `remoteWeaveAspect()` method is used to weave the caching aspect on the client. This method takes four string arguments: the client application name, the client container name, the caching aspect class name, and the path to the caching aspect configuration file. The first two arguments must match the parameters set when the client's application is launched in a JAC container. The last two arguments indicate what aspect to weave and what configuration options to use when initializing the aspect on the client.

There is nothing to prevent the remote weaving command from being triggered by an event outside the service implementation. That is, the aspect can be woven into the client code at any time by any process. However, there are three benefits to having the remote weaving command occur in the server implementation, as is done in Figure 4.6.

First, it ensures that the remote aspect is woven when the client application is active and running, since the service implementation method is only executed as a result of a remote method invocation from the client. If the remote aspect weaving is triggered by an event outside the service implementation method, the server may try to weave a remote aspect when the client is not running or is unable to receive the aspect for other reasons. This may lead to a state where the server believes a client is using a caching aspect when the client is not.

Second, it prevents potential concurrency problems in the client application due to the synchronous invocation of remote methods required by the RMI specification [7]. When a client makes a call to the server its thread of execution blocks until a response from the server is received. By triggering the remote weaving command in the service implementation method, it is guaranteed that the client proxy object is blocked waiting for a response from the server, and thus can be modified without risking unexpected behaviour from race conditions, deadlock, or other concurrency problems. Advice is only woven on the proxy object after the currently blocked method in the proxy object has returned, so it is not possible to have a situation where the proxy's method executes up until the remote method invocation, then have *after* advice execute once the remote invocation returns without executing its corresponding *before* advice. This property is similar to the atomicity of database transactions: either the entire advice is woven into the join point method when it executes, or none of it is woven.

The third benefit of weaving the remote aspect from the service implementation method is the ability for the service implementation method to determine the calling client via `RemoteServer.getClientHost()`. This method returns a string representing the host of the client, typically an IP address, which can be used to locate and identify the client. Most importantly, this string is used to target a specific client when weaving an aspect remotely, as needed for the `remoteWeaveAspect()` method's second argument, the client container location. The server needs to know on which host to weave its caching aspect, and

`RemoteServer.getClientHost()` provides the only way to identify clients to the server. Since server methods outside the remote invocation call stack are unable to use `RemoteServer.getClientHost()` to capture client host information, having the remote aspect weaving trigger within the service implementation method allows this information to be utilized.

The service implementation method in Figure 4.6 weaves the caching aspect on the calling client every time the remote method is called. A more realistic solution is to guard the remote weaving command with a condition that evaluates to true or false, and determines whether the aspect is woven on the remote host or not. This *weaving condition* depends on the type of client-side enhancement being woven, the nature of the service used by the client, existing knowledge of the client, and knowledge of server resource usage.

There are several possible weaving conditions for a caching aspect. Ideally, a caching aspect for the factorial service is woven on a client for the duration of its execution. This means the server weaves a caching aspect on the client during the client's first remote method call to `factorial()`, and any subsequent calls to `factorial()` from the same client do not trigger remote aspect weaving. This can be implemented easily by storing the known client hosts in an initially empty list on the server, as in Figure 4.7. When a remote method call is invoked, the service implementation uses `RemoteServer.getClientHost()` to retrieve the calling client host, then checks whether the host exists in the known hosts list. If it does not appear in the list (an unknown host) then weave the caching aspect on the remote host and add the host to the known list. Otherwise it appears in the list (a known host) and thus already has the caching aspect woven, so no action is needed.

```
private List<String> knownHosts = new ArrayList<String>();

public int factorial(int num) throws RemoteException {
      if (!knownHosts.contains(RemoteServer.getClientHost())) {
            Jac.remoteWeaveAspect("clientApp", RemoteServer.getClientHost(),
                                  "CachingAC", "caching.acc");
            knownHosts.add(RemoteServer.getClientHost());
      }

      if (num == 0) return 1;
      int result = 1;
      for (int i = 1; i <= num; i++) {
            result *= i;
      }
      return result;
}
```

Figure 4.7 The `factorial()` method with a simple weaving condition for known hosts.

This solution is easy to implement but has two drawbacks. First, this solution cannot distinguish between multiple clients running from the same host because `getClientHost()` only returns the host information. Another identifier for the client is needed to support multiple clients on a single host. Second and more significantly, the list of client hosts that have caching aspects woven on them may become outdated. This happens every time the execution of a client application ends because all the advising code is lost. Dynamic aspect weaving in JAC is accomplished by modifying the class's bytecode in memory, so if the class is reloaded from its codebase location it loses all the advice woven on it and reverts to its original non-advised form. Unfortunately the server has no way of knowing when a client is closing or restarting, and thus cannot update its list accordingly. The result may be a client that can use a caching aspect to improve its efficiency, but never receives a caching aspect from the server because the server believes the client already has one.

A more complex but suitable approach is to base the weaving condition on the frequency of identical remote method calls from a particular client. The frequency of identical method calls can be based on a time threshold (i.e., in the last minute, this client has called the `factorial()` method twice with identical arguments); or based on a most-recent threshold (i.e. out of the most recent 100 invocations by this client, 70 of them have been identical); or some combination of both (i.e. in the last hour, identical calls have been made 70 times by this

client). Clearly, a more complex data structure than a simple list is needed to keep track of the calling history for each client.

When a client passes the calling threshold set in the weaving condition, a caching aspect is woven remotely on that client. The caching aspect causes identical remote method invocations from the client to fall well below the server's threshold, preventing caching aspects from being re-woven on clients that do not need it. If a client happens to lose its caching aspect, it eventually triggers the weaving condition, which causes another caching aspect to be woven on the client. Since this weaving condition technique does not store any state about the aspects woven on clients, it does not suffer from the disadvantages associated with outdated client state information.

In fact, this technique delivers a client-side caching aspect precisely when it is needed, as determined by the threshold set on the server. This is a perfect example of using server knowledge to get the optimal client-side caching solution. This solution is adaptable because the server is free to adjust the frequency threshold to its liking whenever it wishes, and it is dynamic because adjusting the frequency threshold automatically results in the remote weaving of caching aspects on clients that surpass the new threshold and thus trigger the weaving condition.

So far all discussion about the caching enhancement has used the factorial service example, but this is an atypical example because the factorial service is stateless. It is stateless because the factorial service implementation does not depend on the state of the remote object, nor does it use any server resources such as a database. A stateless service result can be computed solely by its arguments, making the caching strategy for such a service simple because the cached values never get stale.

A stateful service is a much more realistic study, but has the additional problem of cached data becoming stale over time. Many cache policies have been invented to purge caches periodically to keep their data fresh, but each cache policy is best suited to a different data access pattern.

Since JAC is a dynamic aspect-oriented platform, it has the ability to weave and unweave remote aspects into a client application at runtime. This facility can be leveraged to provide a dynamic caching policy, where the caching aspect's advice can be altered by the server

dynamically. This means the server does not have to have a priori knowledge of its service usage, but can continuously adapt its client-side cache policy to client calling trends and server usage tendencies.

4.2 Validation

A *validation* aspect, like the caching aspect, examines the arguments of a remote method, but instead of caching the method's arguments it validates them. The purpose of validating remote method arguments on the client is to prevent erroneous or invalid remote method calls to the server. Catching this invalid call on the client saves an expensive remote call to the server that will return with an error. The main benefits of client-side validation are improved efficiency and performance during client-server communication, and the ability for the server to provide custom validation for a specific client.

Section 4.2.1 introduces several validation techniques for method arguments. Sections 4.2.2 and 4.2.3 show the client-side validation aspect and server-side implementation, respectively. Section 4.2.3 also discusses two distinct validation objectives, performance and customization, and how their weaving conditions differ.

4.2.1 Introduction to Method Argument Validation

The purpose of the validation aspect is to examine arguments of method calls and either proceed with the method invocation if all arguments are valid, or throw an exception if the call contains an invalid argument. The type of validation performed is often associated with the preconditions of the method, and usually depends on the data types of the arguments, the information they intend to carry, and their relevance to the method itself.

The simplest examples of argument validation are restricting the range of a value, or ensuring data is well-formed. For instance, consider a remote method that allows a client to reserve a venue on a specified date with the month, day, and year being supplied as `int` arguments:

```
public void reserve(int month, int dayOfMonth, int year);
```

Realistically these arguments are encapsulated in a single `Date` object, but are separated here to demonstrate the validation of simple arguments with interdependencies. There is nothing preventing the client from supplying, intentionally or unintentionally, an invalid integer value for one of these arguments, so it falls to the validation aspect to check these arguments at runtime. For example, the `month` argument can be considered valid if and only if it is an integer between 1 and 12 inclusive. Similarly, the `dayOfMonth` argument can be restricted to integers between 1 and 31, and the `year` argument is valid only for positive integers.

In addition to validating each argument in isolation, the relationships and dependencies between arguments can also be validated. In the reservation example, the validation rules for the `dayOfMonth` argument can vary according to the `month` argument value, allowing a `dayOfMonth` value of 31 to be valid for March but invalid for April. The `year` argument can even be considered when calculating the number of days in February, to account for leap years.

The purpose of the method can also influence the validation performed at the client. In the aforementioned example, a reservation request for a date in the past is nonsensical and can be considered invalid, or the venue owner may only allow reservations during work days, in which case the validation logic can invalidate reservations that fall on weekends or statutory holidays. Although the month, day, and year arguments may combine to represent a valid date, the method call may still be considered invalid due to conditions imposed by the nature of the service. Figure 4.8 depicts the three validation techniques and how the arguments are validated for each.

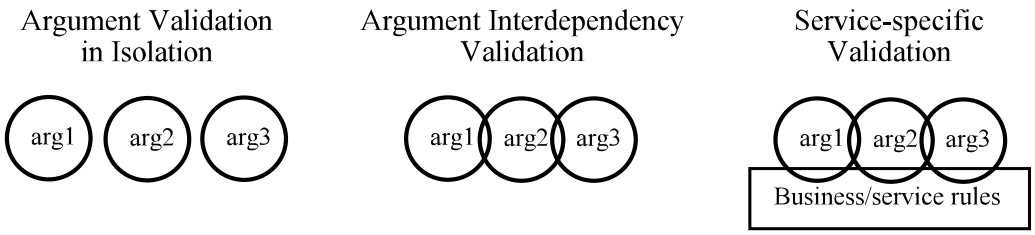

Figure 4.8 Validation techniques for method arguments in increasing complexity (from left to right). A precondition engine typically supports validation in isolation and interdependency validation for simple data types, whereas a validation aspect supports all three techniques with complex data types.

Other data types besides integers can be validated in a similar way. Regular expression tests can be applied to strings to ensure they are well-formed; objects variables can be checked for being null; and the correct size of arrays and collections can be enforced. Data structures with more complexity, such as graphs or tables, may require more advanced validation logic, especially if multiple data structures relate to each other in a nontrivial way. The flexibility and power of an aspect-oriented approach allows arbitrarily complex validation logic, and is not restricted to the limited argument validation of precondition verification engines [10, 12].

4.2.2 Client-side Validation Aspect

A validation aspect for the venue reservation example in Section 4.2.1 is similar to the caching example presented in Section 4.1, with the main difference being the advice. Figure 4.9 shows the validation aspect for this example, including the validation logic discussed in Section 4.2.1. The pointcut definition, in `ValidationAC`'s constructor, contains the method signature for the venue reservation service, "`reserve(int,int,int):void`". The aspect's advice, contained in the `invoke()` method of the `ValidationWrapper` class, validates the `reserve()` method's arguments. The validation advice first extracts the argument values from the `MethodInvocation` object provided by JAC, then it checks the month, day, and year values to ensure they fall into the appropriate ranges. It then verifies that the provided date

occurs in the future, because making a reservation in the past is disallowed. If the arguments pass all the validation requirements, the advice calls `proceed()` which follows through with the remote method invocation. If validation fails, then an `IllegalArgumentException` is thrown with a brief error message.

```
public class ValidationAC extends AspectComponent {
      public ValidationAC() {
            pointcut(".*", "Venue_Stub", "reserve(int,int,int):void",
                     new ValidationWrapper(this), null);
      }
}

public class ValidationWrapper extends Wrapper {
    public ValidationWrapper(AspectComponent ac) {
       super(ac);
    }

    public Object invoke(MethodInvocation mi) throws Throwable {
      Object[] args = mi.getArguments();
      int month          = (Integer) args[0];
      int dayOfMonth     = (Integer) args[1];
      int year           = (Integer) args[2];

      // Ensure arguments constitute a valid date
      if (month < 1 || month > 12)
            throw new IllegalArgumentException("Invalid month argument");
      if (dayOfMonth < 1 || dayOfMonth > daysInMonth(month, year))
            throw new IllegalArgumentException("Invalid dayOfMonth argument");
      if (year < 1)
            throw new IllegalArgumentException("Invalid year argument");

      // Retrieve today's month, day, year information
      Calendar today = GregorianCalendar.getInstance();
      int currentMonth = today.get(Calendar.MONTH);
      int currentDay = today.get(Calendar.DAY_OF_MONTH);
      int currentYear = today.get(Calendar.YEAR);

      // Ensure reservation date is in the future
      if (year       >= currentYear  &&
          month      >= currentMonth &&
          dayOfMonth >  currentDay)
            return proceed(mi);          // proceed with reservation request
      else
            throw new IllegalArgumentException("Past reservation");
   }

   // return total number of days in given month for the given year
   private int daysInMonth(int month, int year) {
      ...
   }
}
```

Figure 4.9 The validation aspect for a venue reservation service.

4.2.3 Server Implementation

In addition to providing a service for clients, the remote method implementation must weave the validation aspect on the appropriate clients to provide client-side validation for its services. Remote aspect weaving is achieved using JAC's `remoteWeaveAspect()` method, as shown in Figure 4.10. This method sends the `ValidationAC` aspect to the client host to be woven into the "`clientApp`" application. The `validation.acc` file is empty in this case because the `ValidationAC` class is non-configurable.

```
public void reserve(int month,int dayOfMonth,int year) throws RemoteException
{
      Jac.remoteWeaveAspect("clientApp", RemoteServer.getClientHost(),
                            "ValidationAC", "validation.acc");

      // reserve venue for the specified date
      ...
}
```

Figure 4.10 The `reserve()` method with remote aspect weaving.

Like the caching example, the server implementation can make use of a weaving condition that controls whether a validation aspect gets woven on the client or not. A weaving condition should be chosen based on the objective of the client-side enhancement. The objective of the caching aspect, for example, is to reduce the number of remote calls with identical arguments, thus a weaving condition based on the frequency of identical calls is best suited to meet this objective, as discussed in Section 4.1.3.

Likewise, the objectives for a validation aspect are to improve performance by reducing the number of erroneous remote method calls from the client, or to provide custom validation rules on a per-client basis. These objectives are not mutually exclusive since elements of both can be incorporated into a validation aspect to both improve communication efficiency and customize validation on the client.

4.2.3.1 A Weaving Condition to Improve Communication Efficiency

When discussing the execution speed of a Java program, the cost of calling a remote method is significantly higher than the cost of calling methods on objects in the local Java virtual machine. The reason for this difference is the extra steps needed to support a remote method call. Remote method arguments must be marshalled before being sent to the server, then unmarshalled into Java types once they reach the server. The marshalling process involves serializing the arguments into a byte stream so they can be sent over the network, which is then deserialized on the server as part of the unmarshalling process. Any objects returned to the client from the remote method must also undergo this marshalling/unmarshalling process in addition to the network communication costs.

Other factors can contribute to the overhead of remote method invocations. If a remote method argument or return value contains an instance of a class that the Java virtual machine does not recognize, the JVM must download that class from its codebase and then load the class before it can continue executing. This occurrence, while expensive, happens much less frequently than marshalling because unrecognized classes quickly become known to the Java virtual machine, and can be used immediately by subsequent remote calls because they are already loaded into the virtual machine.

Performance benefits are not only realized through execution speed, but also by the reduction of network traffic as a whole and to the server in particular. Ideally server resources are used to serve clients with valid requests instead of handling invalid requests that have no benefit to the client. A client-side validation aspect allows invalid requests to be handled by the client's proxy object, freeing server resources for legitimate requests. Figure 4.11 shows how requests are handled both before and after advice has been woven on the proxy.

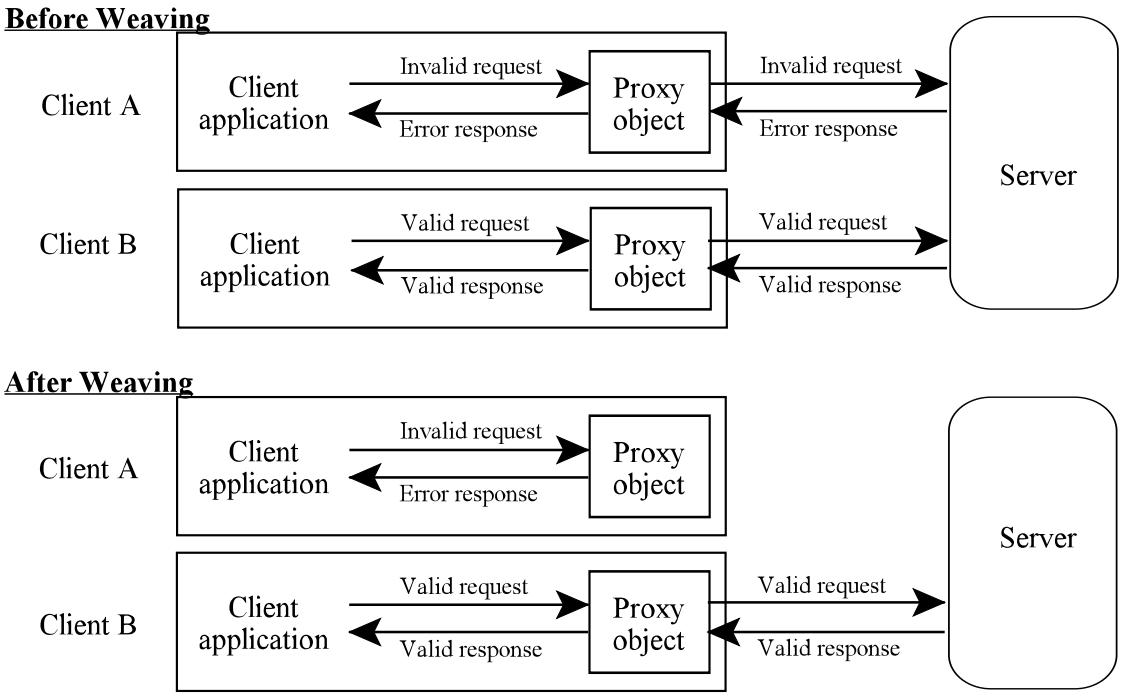

Figure 4.11 A client-side validation aspect reduces unnecessary network traffic, decreases server load, and decreases execution time for the client application.

Reducing the number of remote method calls to the server is obviously desirable, but is only feasible if the client can accurately mimic the server's response to a specific request. Validation and caching aspects both mimic responses from the server, but a validation aspect mimics responses that indicate bad or inappropriate data in the method arguments whereas caching mimics remote method return values in the case when the method completes successfully.

If the objective of the validation aspect is to reduce the number of remote method calls with improper arguments, then the weaving condition should be based on the frequency of remote method invocations from a particular client that contain this type of error. Once the frequency of erroneous remote calls from a client reaches a threshold set by the server, the weaving condition is triggered and a validation aspect is woven on that client. This weaving condition only targets clients that can benefit from a client-side validation aspect, and ignores clients that use the service with appropriate argument values.

A notable limitation of client-side validation is the difficulty involved if server state is required. Any validation that uses server resources such as a server-side object or database must be performed on the server. In the aforementioned reservation example, one cannot verify from the client whether a venue is already booked or not.

4.2.3.2 A Weaving Condition for Custom Validation Rules

Custom validation rules refer to the idea that each client's validation logic can be different, allowing the server to weave different validation rules to suit each client. A unique business relationship between the client and server drives the content of these customized validation rules, the same type of business/service rules mentioned in Figure 4.7.

Recall from the venue reservation example in Section 4.2.1 that the service-specific validation rules may prevent clients from reserving the venue on weekends or holidays. Suppose the venue owner wishes these rules apply to all his clients except a small handful who are allowed to reserve the venue on any day. The flexibility and adaptability of dynamic remote aspects allows the server to weave *any-day* validation aspects on the handful of privileged clients, and *work-day-only* validation aspects on the rest. Clearly the *work-day-only* aspect contains validation rules that invalidate reservations on weekends and holidays, and the *any-day* aspect contains no such rules.

The weaving condition for a custom validation aspect differs from the weaving conditions examined thus far. Instead of a frequency-based weaving condition, the validation aspect weaving is triggered by a client attempting to make a request that is against its rules. An unprivileged client that attempts to make a weekend reservation, for instance, triggers the weaving condition resulting in a *work-day-only* aspect being woven into its proxy object. Figure 4.12 highlights a customized weaving condition for the venue reservation example.

```
public void reserve(int month,int dayOfMonth,int year) throws RemoteException
{
       String clientHost = RemoteServer.getClientHost();
       if (isWeekendOrHoliday(month, dayOfMonth, year) &&
           !clientList.getClientInfoByHost(clientHost).isPrivileged()) {
             Jac.remoteWeaveAspect("clientApp", clientHost,
                                   "ValidationAC", "work-day-only.acc");
       }

       ...
}
```

Figure 4.12 A weaving condition that restricts unprivileged clients to work-day reservations.

Since the behaviour of a validation aspect can be customized for each client, there must be a mechanism to support different advice for a validation aspect. One solution is to have different aspects/advice for each supported custom configuration. The reservation example has two configurations (*any-day* and *work-day-only*) whose behaviour can be encapsulated in two separate aspects, say `AnyDayAC` and `WorkDayOnlyAC`. The server simply uses whichever validation aspect is appropriate for a particular client.

A second solution, alluded to in Figure 4.12, uses the aspect component configuration (`.acc`) file to externalize custom configuration options. This solution only requires a single validation aspect, `ValidationAC`, which is configured during its initialization using the contents of a specific `.acc` file. The configuration differences can then be expressed in two separate `.acc` files (`work-day-only.acc` and `any-day.acc`) rather than two aspects. In addition, the validation aspect needs to be enhanced slightly to support multiple configurations. Figure 4.13 shows an example of `work-day-only.acc`, while Figure 4.14 shows the corresponding methods required in `ValidationAC`.

```
restrictDayOfWeek "Saturday" ;
restrictDayOfWeek "Sunday" ;
restrictDate 1, 1 ;
restrictDate 12, 25 ;
restrictDate 12, 26 ;
```

Figure 4.13 The contents of `work-day-only.acc` restricts Saturdays, Sundays, January 1st, December 25th & 26th.

```
public class ValidationAC extends AspectComponent {
      // instance variables save the configuration
      private List<String> restrictedWeekdays;
      private List<Date>   restrictedDates;

      public void restrictDayOfWeek(String dayName) {
            restrictedWeekdays.add(dayName);
      }

      public void restrictDate(int month, int dayOfMonth) {
            restrictedDates.add(new Date(month, dayOfMonth));
      }
}
```

Figure 4.14 Two public methods, `restrictDayOfWeek()` and `restrictDate()`, must be added to the validation aspect to support the external configuration file.

As Figures 4.13 and 4.14 imply, the configuration commands and arguments in the `.acc` files must have matching method names and arguments in the aspect class they configure. The methods themselves are typically used to set instance variables that save the state of the configuration. These variables can then be read by the advice code to enforce the configuration, as in Figure 4.15. Note that getter methods for these instance variables may be needed to obtain them from the advice wrapper class. The code in Figure 4.15 assumes `ValidationWrapper` is an inner class of `ValidationAC` and has access to these variables.

Multiple configuration files can be created for different purposes, such as the *work-day-only* and *any-day* configurations. The *any-day* configuration file is empty because it does not restrict the days when the venue can be reserved. By referencing a specific configuration file name in the `remoteWeaveAspect()` method call, the server can easily customize the validation rules on a per-client basis.

```
public class ValidationWrapper extends Wrapper {
      public Object invoke(MethodInvocation mi) throws Throwable {
            // extract the date from the method arguments
            Object[] args = mi.getArguments();
            int month = (Integer) args[0];
            int day   = (Integer) args[1];
            Date d = new Date(month, day);

            // validate arguments using configuration rules
            if (restrictedWeekdays.contains(d.getDayOfWeek()))
                  throw new VenueNotAvailableException();
            if (restrictedDates.contains(d))
                  throw new VenueNotAvailableException();

            return proceed(mi);
      }
}
```

Figure 4.15 The advice code uses the saved configuration information to enforce the configuration rules.

4.3 Load Balancing

In the context of client-server communication, load balancing is the process of distributing client requests evenly over a set of physical servers, collectively known as a server farm. Load balancing prevents any one server from becoming overloaded with client requests and attempts to find an optimal use for server resources. A load balancer also balances server load dynamically if a physical server is added to or removed from the server farm.

Load balancing techniques are well established in modern servers, and a typical load balancing architecture is shown in Figure 4.16. The load balancer intercepts all client requests and delegates the request to a physical server in the server farm. When the server has prepared a response for the client, it sends the response to the load balancer who in turn sends it to the appropriate client.

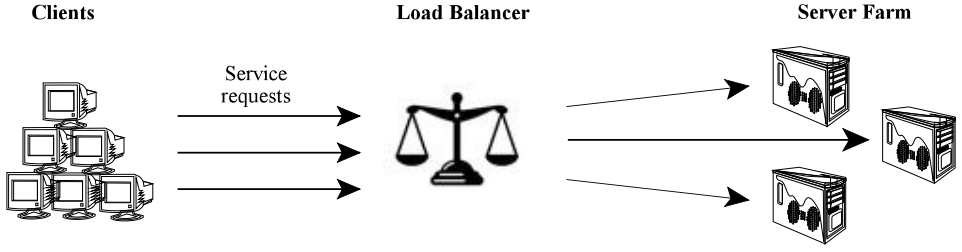

Figure 4.16 The traditional load balancing architecture. All requests and responses pass through the load balancer.

The distributed aspect-oriented framework proposed in this thesis provides a novel way to approach the load balancing problem. Instead of service requests and responses being routed through the load balancer, the routing can be relocated to the client's RMI proxy object via a remotely distributed, dynamically woven aspect.

This new architecture, depicted in Figure 4.17, requires the load balancer to configure and distribute client-side aspects that modify the client proxy. By dynamically modifying the server target of a client's remote method invocation, the load balancer controls which clients call which servers and can therefore implement a load balancing strategy.

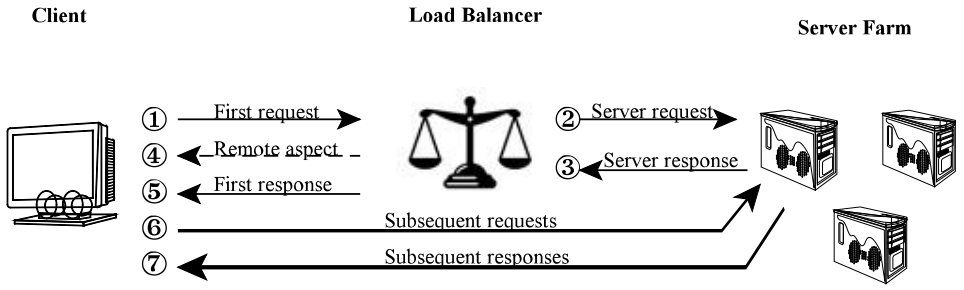

Figure 4.17 The revised load balancing architecture. Client-side aspects remotely woven by the load balancer redirect subsequent service requests from a client to a physical server in the server farm. The steps 1-7 indicate the chronological order of events.

Figure 4.17 shows how load balancing is achieved in seven chronological steps:

1. A client makes a request to the server, which is handled by the load balancer.
2. The load balancer uses its server farm status and load information to determine the physical server that is best able to serve the client and forwards the request to that server.
3. The server handles the request and returns a response to the load balancer.
4. The load balancer weaves a load balancing aspect on the client's proxy object that redirects service requests to the chosen server.
5. The load balancer forwards the server response from step #3 to the client. The load balancing aspect woven in step #4 does not take effect on the client until after this response is received.

6. The load balancing aspect causes subsequent requests to be sent directly to the chosen physical server, bypassing the load balancer entirely.
7. Responses are sent from the physical server directly back to the client, also bypassing the load balancer.

The main difference between these two architectures is the role of the load balancer. In the original architecture (see Figure 4.16) the load balancer is a server-side proxy that acts as a demultiplexer for service requests. In the revised architecture, the load balancer is used as a bootstrap mechanism to get a client communicating with the appropriate server. The advantages and disadvantages of this revised architecture are discussed in Section 4.3.3.

4.3.1 Load Balancing Aspect

The load balancing aspect differs from the caching and validation aspects in a significant way. The caching and validation aspects both decide whether to proceed with a remote method invocation or handle the remote call locally, but regardless of this decision the server target of the remote method never changes. Conversely, the load balancing aspect always proceeds with the remote method invocation, and may change the server target.

The easiest way to target a specific server is to include an instance of its client-side stub in the load balancing aspect. The aspect's advice then simply calls the remote method with this stub instead of the proxy object it is advising. Figure 4.18 shows a simplified load balancing aspect that redirects the client's remote method call using this technique.

```
public class LoadBalanceAC extends AspectComponent {
	public LoadBalanceAC() {
		pointcut(".*", "LoadBalancer_Stub", "factorial(int):int",
			new LoadBalanceWrapper(this), null);
	}
}

public class LoadBalanceWrapper extends Wrapper {
	private MathOps targetStub;

	public LoadBalanceWrapper(AspectComponent ac) {
		super(ac);
	}

	public Object invoke(MethodInvocation mi) throws Throwable {
		Object[] args = mi.getArguments();
		int num = (Integer) args[0];
		return targetStub.factorial(num);
	}
}
```

Figure 4.18 The load balancing aspect redirects remote method calls to a different server target.

The load balancing advice (the body of `LoadBalanceWrapper.invoke()` in Figure 4.18) first extracts the remote method arguments from the `MethodInvocation` object, then uses them in the remote method call to the new server. The value of `targetStub` is set by the load balancer when it instantiates and configures the load balancing aspect, but the methods supporting this configuration are omitted from Figure 4.18 for simplicity.

Another noteworthy element of Figure 4.18 is the `LoadBalancer_Stub` class reference in the pointcut definition. Recall from Figures 4.16 and 4.17 that clients know only about the load balancer and are ignorant of the physical servers in the server farm. When a client first uses the service it contacts the load balancer, and therefore must be using the load balancer's stub class.

A more complex load balancing strategy could involve the load balancing aspect having stubs to all the physical servers in the server farm, and alternating among them for different remote method calls. This effectively spreads the remote method calls from a single client among the available servers, balancing load by a somewhat different approach. JAC's library happens to contain a load balancing aspect that works in this way, and supports both a round-robin selection pattern and a random selection pattern. This aspect needs to be reconfigured

whenever a physical server is added to or removed from the server farm to keep the advice up to date with the server farm status.

4.3.2 Load Balancer

The load balancer is responsible for distributing load evenly across the available servers in the server farm. Technically the load balancer is a well-known, remotely visible object that implements the same remote service interfaces as the physical servers. For dynamic load balancing, the load balancer should be in constant contact with the server farm and have access to information about server status (online/offline, up-time, etc.) and load statistics (request frequency, duration of service, CPU usage, etc.).

Since the load balancer implements a service interface it must contain the same service methods as the physical servers. Figure 4.19 shows a possible service implementation for the load balancer, using the `factorial()` remote method in the `MathOps` service interface as an example. This implementation does not compute any factorial values whatsoever since that task falls to the server farm and not the load balancer.

```
public class LoadBalancer implements MathOps {
	public int factorial(int num) throws RemoteException {
		// assign a server to serve this client
		MathOps assignedServer = assign(RemoteServer.getClientHost());
		return assignedServer.factorial(num);
	}
}
```

Figure 4.19 When a client first uses the remote service, the load balancer assigns it a server then delegates the request to the assigned server.

Most of the work in Figure 4.19 is done in the `assign()` method, which finds the most suitable server for the client. Many factors can determine which server is most suitable but ideally the `assign()` method should choose a server with the least current load, using status information and statistics from the server farm to help it make an informed assignment. Consider a simple example where the most suitable server is the one currently serving the least number of clients, and the load balancer can query each physical server to retrieve the number of clients it is

currently serving via a call to `getClientCount()`. Figure 4.20 shows the `assign()` method implemented with this criteria.

This `assign()` method first iterates through a list of its servers and finds the one that is least busy. Then it retrieves the URL of a preset aspect configuration file for the least busy server and uses that configuration file along with the client host to weave a load balancing aspect on the client's proxy object. Finally, the least busy server is returned to the caller.

Any subsequent calls from the client are intercepted by the client-side load balancing aspect and redirected to the least busy server. As long as the load balancing aspect is advising the client's proxy, the client will continue to use the physical server assigned to it. If the client application resets or otherwise loses its load balancing aspect, the default behaviour of the proxy reestablishes itself and the load balancer once again gets called. This triggers another process of finding the least busy server and weaving the load balancing aspect back on to the client's proxy object.

In Figure 4.20, a separate thread can calculate the least busy server by periodically polling the load status of each physical server in the server farm and determining which physical server is least busy. This process can then update the `leastBusyServer` variable to reflect this knowledge. Synchronization issues involved with writing this variable at the same time it is being read should be taking into consideration, but are ignored in Figure 4.20 for simplicity.

```
public class LoadBalancer implements MathOps {
	public ServerImpl leastBusyServer;
	private List<ServerImpl> servers;
	private Map<ServerImpl,String> configurations;

	private MathOps assign(String clientHost) {
		// select aspect configuration that corresponds to chosen server
		String configURL = configurations.get(leastBusyServer);

		// weave aspect remotely on client
		Jac.remoteWeaveAspect("clientapp", clientHost,
					"LoadBalanceAC", configURL);

		return (MathOps)leastBusyServer;
	}
}
```

Figure 4.20 The load balancer assigns the least busy server to a client by weaving a load balancing aspect on the client.

4.3.3 Advantages and Disadvantages

By removing the load balancer from typical client-server communications an unnecessary relay can be avoided. It is advantageous to have the server that handles a client's requests communicate directly with the client since response times are delayed with each intermediate process added to the communication line.

Eliminating the load balancer from client-server communications also has disadvantages. One significant benefit of a standard load balancer is fault tolerance. If a server in the server farm unexpectedly goes offline, the typical load balancer can compensate immediately by redistributing its clients over the remaining physical servers. The load balancer proposed here is simply a bootstrap mechanism, and is not involved in subsequent client-server communications. If a server suddenly went offline, none of its clients would be redirected to other servers because the load balancer is no longer part of the communication.

Fortunately, there is an aspect-oriented solution to this problem. The load balancing aspect shown in Figure 4.16 can contain a try-catch clause around the retargeted remote method call to detect a `ConnectException`, `ServerException`, or any other `RemoteException` that indicates a non-responsive server. The exception handler for such an event can simply call `proceed()` to revert back to the proxy's standard behaviour. The proxy would then remotely invoke the load balancer causing it to find a more suitable server, and have the load balancer re-weave the load balancing aspect with a different configuration. Figure 4.21 shows the load balancing advice enhanced to support fault tolerance.

```
public class LoadBalanceWrapper extends Wrapper {
      private MathOps targetStub;

      public LoadBalanceWrapper(AspectComponent ac) {
            super(ac);
      }

      public Object invoke(MethodInvocation mi) throws Throwable {
            try {
                  Object[] args = mi.getArguments();
                  int num = (Integer) args[0];
                  return targetStub.factorial(num);
            }
            catch (RemoteException e) {
                  return proceed(mi);
            }
      }
}
```

Figure 4.21 Load balancing advice with fault tolerance. If `targetStub.factorial()` fails then try the original target.

A similar technique can be used by the physical servers to shed server load. If a server overloads, it can remotely unweave the load balancing aspect on some of its clients, causing those clients to stop calling the overloaded server and start calling the load balancer instead. The load balancer then remotely re-weaves a load balancing aspect on the clients to redirect them to a more capable server.

This technique can be streamlined by maintaining a session on the load balancer for each client, where we define a session as a sequence of individual remote method invocations that constitute a single, logical request. A server may hold state for the duration of a session, so it is best to direct the individual requests to the same server. When the session finishes the load balancer can designate a different server for the next session.

4.4 Overhead Measurements

This section is taken from a conference paper submission [20] *co-authored by Steve MacDonald.*

This section evaluates the overheads associated with the use of dynamic server-side aspects on load balancing. These overheads were measured using a set of microbenchmarks that consisted

of three processes (client, balancer, and server) running on three separate machines connected by a 100 Mbit per second network. The client invokes a remote method that is handled using the strategy in Figure 4.17. The remote method does not take any arguments, has no return value, and an empty method body, so the results presented here represent just the overhead introduced by the dynamic distributed aspects in JAC.

As a baseline, the performance of the first load balancing strategy given in Figure 4.16 was measured, where all requests must be forwarded through the balancer before being directed to an appropriate server. From the perspective of the client, requests take an average of 0.51 ms per round trip. Note that this time represents a lower bound since the balancer manages a single server and does not have to make any decisions to balance the load. The request is simply forwarded to the server.

The first assessment was the overhead of the first request by the client, which requires the balancer not only forward the request to the server but also weave advice into the client. From the perspective of the client, this first requests takes an average of 5.64 ms. Note that the advice was unwoven between successive calls to avoid any additional overhead that may accumulate by repeatedly weaving advice on the same join point. To reduce the overhead of this weaving, it was done by a separate thread that runs while the balancer forwards the method invocation to a server.

After the first request, the advice woven at the client proxy forwards all subsequent requests directly to the server without involving the balancer. This avoids the overhead of an extra exchange of messages, but incurs the overhead of invoking advice woven on the client proxy. Some aspect-oriented constructs can incur a significant performance penalty in ways that are not obvious [5]. In particular, around advice may include closure objects, which are expensive to create and use. The use of `proceed()` to invoke the code for a join point has unusual polymorphic properties that can be expensive to evaluate at runtime, and JAC uses Java Reflection which can be expensive. However, the balancing advice replaces the client proxy code rather than invoking it. Since `proceed()` is not called, some of this overhead should not be present.

To assess the overhead of the advice at the client proxy, the cost of a remote method invocation with an advised proxy and the cost of normal remote method invocation directly from

client to server were both measured. These two measurements indicate the overhead of applying aspects to the client proxy. From the perspective of the client, the cost of a remote method with a balancing aspect was measured to average 0.357 ms per requests, which was identical to a round trip with an unadvised proxy. The advice adds no appreciable overhead to the execution of the proxy on the client.

The dynamic aspect-oriented load balancing strategy, which allows requests to be sent directly to a server after the first request, cuts 0.153 ms from each request, a savings of 30%. However, this does come at the cost of weaving, which takes 5.64 ms in JAC. This requires clients to forward 44 requests to a given server to make up for this extra cost. Again, though, this represents a lower bound since the experimental setup did not include any scheduling decisions, computations, or significant data transfer for method arguments and return values. In a more realistic environment, this number is expected to be lower. In particular, given that remote methods are normally large-grained to make up for communication overhead, this weaving process can be overlapped with the execution of the method and may impose little overhead in overall execution. For requests that benefit greatly from server affinity, this approach may be best even with its overhead.

4.5 Summary

This chapter demonstrates several practical applications of client-side enhancements and some of the benefits when they are provided dynamically by the server. Caching, validation, and load balancing are already well-known and widely used techniques in distributed systems, but the dynamic aspect-oriented framework realizes these client-side enhancements in a new way by giving them access to changing server knowledge at runtime.

Despite the variety of practical uses for this framework, each example has the goal of reducing unnecessary network traffic between the client and server. The measurements performed on the overhead of JAC with the load balancing example indicate that performance savings are possible if sufficient remote method invocations are performed to recoup the overhead losses of remotely weaving aspects from the server on the client. Other framework uses

such as customizable business rules and client-side caching that adapts to constantly changing server state offers advantages besides performance.

Chapter 5 investigates other techniques and frameworks that can be used instead of a dynamic aspect-oriented solution, and compares their advantages and disadvantages to those presented in this thesis.

Chapter 5: Related Work

Recall from Chapter 2 that an RMI client has no knowledge about the internal state of the server because the RMI specification promotes a loose coupling between client and server. This lack of server knowledge limits the ability of the client to make informed and intelligent decisions that can lead to a more efficient communication channel with the server. To address this problem, this thesis presents methods and technologies that can create a "smart" client in a distributed Java application. This chapter explores other methods or technologies that provide a similar solution, and compares their advantages and disadvantages in light of the goals mentioned in Chapter 1.

5.1 Statically-generated Proxy

The most straightforward solution to the client-side enhancement problem is for the server developer to provide a statically generated proxy class to the client that contains code for a client-side enhancement. This code could be distributed in a library that the client application developer would have to explicitly invoke, or by enhancing the contents of the RMI stub that the client retrieves from the RMI registry.

The main advantage of the statically-generated proxy is its simplicity, since neither the client developer nor the server developer needs to learn and implement new software technologies, frameworks, or languages. In addition, RMI's loose coupling between client and server remains intact giving the server the freedom to change its implementation without worrying about its effects on the client.

Unfortunately this solution falls short of two thesis goals. First, the client application code ideally should not need to be aware of the communication enhancements provided by the server. Using server libraries or custom proxy classes in the client application intrudes on the client's core concerns and introduces extra code that becomes tangled with those concerns. Furthermore, client application developers are not oblivious of changes to the client-side enhancements because they may be required to use a new library or a new stub class in their code.

The second thesis goal is to apply client-side enhancements dynamically, without halting, recoding, or recompiling the client application. The statically-generated proxy technique fails this objective because the client application may need to be halted and possibly altered and recompiled in order to use new functionality provided by the server. Dynamic behaviour changes can be migrated from server to client through the use of a Strategy object [1] on the client, but this solution is difficult to manage, requires extra communication between server and client to coordinate the strategy, and can be intrusive to the client application.

One significant thesis goal that this model achieves is the use of server knowledge. Since the proxy class is written by the server developer, it is possible for the client-side enhancement to leverage information about the internal state and implementation details of the server. However once the proxy is distributed to various clients, the server's state or implementation may change, causing the server knowledge contained in the client-side enhancement to become outdated. This may be acceptable if the server's state or implementation rarely changes, but in a rapidly evolving server environment a more adaptable solution is needed.

The most significant drawback of a statically generated and manually distributed client-side enhancement is its inability to adapt quickly in a dynamic environment. Contrast this to a client-side enhancement that can be generated by the server on the fly and immediately delivered to clients and take effect instantly. The dynamic distribution of client-side enhancements clearly has the advantage and provides more flexibility and faster adaptability.

5.2 Fragmented Object Approach

Kapitza et. al. [8, 9] propose a related idea that uses object fragments, also implemented in the context of Java RMI. Their framework, FORMI, extends RMI call semantics and protocols to implement a fragment-oriented model. This model allows an object's state and functionality to be partitioned into *fragments*, with each fragment existing on a potentially different host in the distributed system. Each object fragment exposes the object's interface, but inter-fragment proxies are used to access object state and functionality that does not exist in the local fragment.

FORMI is relevant to this thesis because both distribute logically related functionality among different hosts, and thus share many common goals for an RMI application. In particular, Kapitza et. al. emphasize the use of fragments as smart proxies:

> "In another scenario, the fragment on Node 1 may act as a smart proxy...those can support caching mechanisms to reduce communication, or they may send method invocations to a group of replicas in order to balance load or mask faults." [8]

This thesis adopts a similar philosophy, as shown by the caching and load balancing examples in Chapter 4. Although both technologies use smart clients to improve communication, this thesis proposes a distributed dynamic aspect-oriented model to migrate functionality whereas FORMI uses a fragment-oriented model.

FORMI supports the dynamic migration of functionality among fragments, shown in Figure 5.1, analogous to the remote dynamic weaving described in Chapter 3. When one fragment exists on the server and another fragment on the client, the server can dynamically change the client fragment's implementation, achieving the same effect as a remotely woven aspect.

This dynamically adaptable solution satisfies one of the thesis goals because FORMI can make the server's implementation and state details available to the client. This additional server knowledge is a significant advantage over a standard RMI solution where the client is denied access to internal server information, an advantage that FORMI shares with other dynamic, distributed aspect-oriented frameworks like JAC.

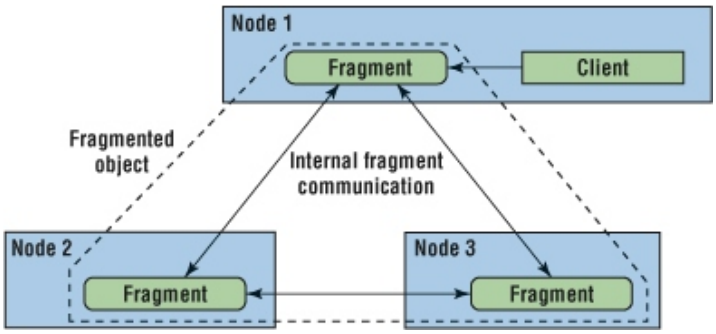

Figure 5.1 FORMI allows an object's functionality to be fragmented across multiple nodes. [8]

In addition to allowing client-side enhancements to migrate from server to client, FORMI ensures that the client application is oblivious to this migration by managing the fragment proxies and inter-fragment communications in the background. This satisfies the thesis goal of obliviousness because each host in the distributed software assumes its object fragment contains the object's entire state and behaviour locally. It should be noted that FORMI uses special interfaces in the application code to access fragments, so complete obliviousness is impossible because the client application source code would have to change from its vanilla RMI version to support FORMI.

More specifically, FORMI fragment stubs are generated through a special fragment factory that alters the marshalling mechanism of a typical Java RMI stub to provide an extra layer of indirection. The use of special FORMI libraries and interfaces to create and use fragments is intrusive to the client application code, and goes against the obliviousness goals set out by this thesis.

One goal that FORMI does achieve is the dynamic distribution of server knowledge to clients, because the implementation of a fragment can change at run time. A dynamic fragment is significantly more complex for the object developer to program than a static fragment, but it is still possible. However, the FORMI framework again imposes on the development of client applications by requiring fragment remote references to subclass FORMI classes and use a custom stub generator tool. Specialized development tools and libraries may make the FORMI approach longer to learn and implement for client application developers than the solution proposed here.

The separation between the application and inter-fragment communication layers allows FORMI to maintain the loose coupling between client and server established by RMI. The key to keeping the fragments loosely coupled lies in FORMI's ability to migrate functionality dynamically, and the transparency of the inter-fragment communication needed to make the migration work.

5.3 D: A Language Framework for Distributed Programming

D [13] is an aspect language framework to define concurrency and remote access strategies in distributed object-oriented programs. D regards thread concurrency and copy semantics for remote methods as crosscutting concerns of a distributed application, and attempts to encapsulate these concerns in two new languages, Cool and Ridl.

Since concurrency in Java is typically achieved with synchronized blocks embedded throughout the functional code, D views these blocks as a secondary concern tangled with the core concern, and attempts to externalize these concurrency rules into a separate aspect language called Cool. Accordingly, D acts upon a language called Jcore, a subset of Java whose major omission is the synchronized keyword and related structures. Cool is a small language that provides *coordinators*, structures that can group a class's methods into two types of sets: self-exclusive and mutually-exclusive. A method in the self-exclusive set prevents two different threads from calling that method at the same time. Conversely, if one thread is calling a method in a mutually-exclusive set, no other thread may call a different method in the same mutually-exclusive set.

In Java RMI, remote method arguments are passed by copying the data for each argument. For an object-based data structure like linked lists or trees, the entire structure is copied by deep copying the "head" object. Ridl is a language designed to give the programmer more control over which parts of the remote arguments should be copied.

The rules defined in both Cool and Ridl are enforced by an aspect weaver that pre-processes the Jcore source code and generates Java structures at the appropriate join points. D does not need a separate pointcut language because the join points are already well-defined for both Cool and Ridl. The aspect weaver outputs a valid Java program that can be compiled with a standard Java compiler.

D is useful for the distributed concerns of concurrency and remote access strategies, but it is not powerful enough to encapsulate a generic distributed crosscutting concern. Client-side enhancements could not be written in D because there is no way for the programmer to define a join point in the proxy object or the advice that should run there. D uses a preprocessor to weave

its aspects, requiring it to be done before compilation. An application programmed in D needs to be halted, changed, rewoven, recompiled, and restarted to alter its behaviour, ruling out a dynamic weaving ability outlined in the thesis goals. D is a good framework for the problems it is designed to solve, but it is unsuitable for client-side enhancements provided dynamically by the server.

5.4 Remote Pointcuts

The benefits of describing program crosscutting concerns using aspect-oriented programming quickly led programmers to wonder if this benefit could be extended to a distributed program. The obvious way to test this is to take an existing well-established aspect-oriented language like AspectJ and extend its functionality to support distributed crosscutting concerns, which is precisely what DJcutter [15] does.

DJcutter is based on the AspectJ language, and uses the AspectBench Compiler (abc) [1] framework to introduce the new pointcut predicate `hosts(hostId,...)`. The `hosts()` predicate takes one or more arguments, each identifying a host on the network. Used in conjunction with the standard AspectJ pointcut predicates, `hosts()` can narrow the pointcut scope to match join points on specific hosts. Figure 5.2 shows how RMI proxies on `client1` and `client2` could be targeted in DJcutter.

```
pointcut clientProxies():     execution(public * *_Stub.*(..)) &&
                              hosts(client1, client2);
```

Figure 5.2 Pointcut declaration in DJcutter for the execution the proxy methods on `client1` and `client2`.

The `execution(public * *_Stub.*(..))` predicate identifies execution join points for public methods in classes whose names end with "_Stub". The `hosts(client1,client2)` predicate, introduced by DJcutter, restricts the matching join points to those that reside on `client1` or `client2`.

All aspects in DJcutter are stored on a single host designated the *aspect server*. DJcutter automatically distributes aspects from the aspect server to the other hosts in the distributed

system, where classes are modified when they are loaded into the local Java virtual machine. The classes that are modified at load-time depend on the pointcuts of aspects known to the aspect server. When a join point event occurs that matches a join point declaration, a remote signal is sent to the aspect server, where the corresponding advice is executed. The advice for these remote pointcuts are always executed on the aspect server, not on the host where the join point originated. Using the example from Figure 5.2, when a proxy method is invoked on `client1` or `client2` the corresponding advice is executed on a third host, the aspect server.

There are three characteristics of DJcutter that are consistent with the goals of this thesis. The first is that DJcutter can be used by an aspect residing on the server to detect an event on the client such as an RMI proxy method invocation. Splitting the functionality of an aspect between two remote hosts is essential for this solution to work. The second characteristic of DJcutter that supports the thesis goals is its non-intrusiveness to the client application. The client application source code does not need to include new libraries, alter the way it uses RMI, or otherwise change in any way. It would need to run under a DJcutter container or framework, but this is a reasonable requirement. Third, DJcutter does not break the loose coupling between client and server provided by RMI because the internals of the server implementation remain hidden from the client. An additional advantage that DJcutter has over other approaches is its similarities to AspectJ, an existing well-established, popular, and well-supported aspect-oriented programming language.

There are two limitations of DJcutter that do not meet with thesis goals and make a solution for dynamic client-side enhancements impossible. The first and most significant limitation is DJcutter's inability to execute advice on a remote host. Since all advice is executed on a single host, the aspect server, writing advice to enhance the client proxy object is impossible with DJcutter. The second limitation is DJcutter's lack of dynamic (run-time) weaving. DJcutter only supports load-time weaving, meaning that all client-side enhancements would have to be ready on the aspect server prior to the client application loading. The lack of dynamic weaving prevents the server from weaving and unweaving client-side aspects at runtime, restricting the flexibility and adaptability of the distributed application.

Although DJcutter has some benefits for aspect-oriented programming in a distributed environment, it ultimately only supports distribution for one half of the aspect-oriented paradigm: remote pointcuts but not remote advice. Since the execution of remote advice is necessary to implement client-side aspects, DJcutter alone is not suitable to solve the problems presented in this thesis.

5.5 Aspects With Explicit Distribution (AWED)

One of the more comprehensive aspect-oriented languages with support for distributed aspects is AWED [14], an abstract language that describes a general syntax for distributed advice and pointcuts. Although a concrete implementation of AWED is provided in Distributed JAsCo [22], an extension to the aspect-oriented language JAsCo, this section focuses on AWED instead of Distributed JAsCo because AWED can conceivably be incorporated into any existing aspect-oriented language.

AWED has a very AspectJ-like grammar, in particular with its pointcut language and advice definitions. However AWED has three notable features that distinguish it from a non-distributed aspect-oriented programming language: remote pointcuts, distributed advice, and distributed aspect states. Remote pointcuts and distributed advice allow the programmer to specify a unique host for join point matches and advice execution respectively, and a distributed aspect is capable of maintaining a consistent state even if deployed across multiple hosts.

Remote pointcuts are realized by a new pointcut predicate `host()`, which takes one argument specifying the host or group of hosts where the join point is located. For example, `host(localhost)` indicates all join points on the local host, and `host("192.168.1.100")` indicates all join points on 192.168.1.100. Hosts can also be grouped together logically and assigned a group identification string, which can then be used to indicate all hosts within the group, such as `host("AllClients")`. The `host()` predicate can, as usual, be combined with other predicates using the `&&` and `!` operators to further narrow the scope of the join point set.

Similarly, distributed advice is implemented with the new pointcut predicate `on()`, taking one argument that indicates the host on which the join point event occurs. Like the `host()` predicate, this argument can be localhost, a host literal, or a group id. The `on()` predicate tells AWED where to execute the associated advice, and can be a completely different host than the join point's host. For instance, the `on()` predicate can be used with a group id to manage replication across hosts in a logical group. A special argument called `jphost` indicates the host of the join point matched by the current pointcut, and can be used by the `on()` predicate to indicate that the advice should execute on the same host as the join point.

For example, consider the pointcut declaration in Figure 5.3. The `execution(public * get*()) && host("AllClients")` part tells AWED to match all getter method join points on hosts that are members of the *AllClients* group, and the `!on(jphost)` part indicates that the associated advice should be executed on all hosts except the host where the aforementioned pointcut matches. In other words, whenever a host in the *AllClients* group runs a getter method, the advice executes on all the other clients.

```
pointcut clientGetters():       execution(public * get*()) && host("AllClients")
                                &&
                                      !on(jphost)
```

Figure 5.3 A pointcut in AWED demonstrating the use of the host() and on() predicates.

This separation of join point event and advice execution is a significant departure from traditional aspect-oriented languages where advice is thought to be woven directly before, after, or around a particular join point. This notion breaks down somewhat in a distributed environment where advice can execute on a different host than the join point, but the added flexibility provided by the separate `host()` and `on()` predicates makes for a powerful tool in distributed aspect-oriented software design.

Many aspect-oriented programming languages, following AspectJ's example, allow an aspect to declare variables that define its state, analogous to how static variables in Java define a class's state. This is straightforward in a non-distributed aspect-oriented environment because the state can be stored in local program memory, but when one aspect is deployed to multiple hosts, each with its own memory space, the aspect's state can become unstable if it is

desynchronized on different hosts. To solve this, AWED synchronizes the states of shared aspects automatically and provides field and aspect modifiers that let the programmer customize the scope of aspect state sharing (e.g. globally, locally, by group, etc.).

AWED meets or exceeds the all of the goals set out by this thesis, assuming that it is realized in a concrete language based on Java-like syntax. In the context of server-client communication and client-side enhancements, AWED allows client-side enhancement to use server-level knowledge without disrupting the loose coupling between client and server. It can accomplish this the same way JAC does, by executing advice remotely on the client's RMI proxy object. In fact, JAC's distribution features are a subset of those provided by AWED, especially in terms of remote pointcut flexibility. JAC can send an aspect to be woven at a remote host, but the aspect's pointcut and advice only apply to that host. AWED has the ability to specify both join point host and execution host, neither of which need to be the local host, and so offers a more versatile distributed solution.

Any disadvantages of an AWED implementation would likely arise from its rich distributed aspect functionality since it supports the distribution of pointcuts, advice, and aspect state. JAsCo, for example, uses a very different aspect-oriented design compared to AspectJ, and introduces several new language constructs like *hooks* and *connectors* to describe its crosscutting concerns. Any language that fully implements the features of AWED may take a while for server developers and client developers to learn.

A significant limitation of the JAsCo implementation investigated here is its inability to send advice among hosts. The `on()` predicate in AWED is implemented as `executionhost()` in JasCo, but is only able to execute advice already deployed on a remote host. A host must already know about all the advice code it will need to execute because there is no obvious transport mechanism among hosts for remote advice execution. Executing arbitrary advice code from a remote host is not possible with this implementation, as in the case of server-provided client-side enhancements.

Like JAC, the client must run their application in a AWED-compatible runtime environment and make the server aware of the client host for remote advice execution to work. The client application does not need to be altered or recompiled, and the dynamic execution of

advice from the server ensures that the client is oblivious to any changes made to the RMI proxy object.

5.6 Summary

Many different techniques are available to give clients access to server knowledge, but occasionally at the expense of loose coupling between the client and server. The approaches discussed in this chapter satisfy the thesis goals to varying degrees, and differ in their level of complexity.

The simplest solution involves distributing a statically-generated server-side proxy to the client, but this solution has many maintenance drawbacks. Other frameworks such as FORMI or D provide an alternate solution to distributed knowledge but still maintain the loose coupling between hosts in the distributed system. The tools and technologies most closely resembling JAC are other dynamic aspect-oriented programming platforms with explicit support for distribution, such as JAsCo and DJcutter.

Chapter 6: Future Work and Conclusion

This chapter presents future work based on the work in this thesis, and then presents some concluding remarks. The future work includes both technical enhancements for richer features and versatility as well as additional uses for the distributed dynamic aspect-oriented framework.

6.1 Future Work

The distributed aspect-oriented framework presented in this thesis is flexible, extendable, and useful for many domain specific problems. There are several different directions that further research into this area can take, with a few such ideas suggested in this chapter. Sections 6.1.1 and 6.1.2 discuss two innovative ways the framework can be adapted to solve generic computing problems, and Section 6.1.3 explains how JAC's per-object weaving feature could be leveraged to give finer control over proxy join points on the client.

6.1.1 Partial Client-side Execution of the Remote Method Implementation

Chapter 4 shows three practical uses of a distributed aspect-oriented framework, but its utility is clearly not limited to just caching, validation, and load balancing. For instance, parts of the remote method implementation can be pushed down to the client via a remote aspect in much the same way that validation logic is applied in the validation example. Remote implementation steps that require server resources obviously need to stay on the server, but independently executable steps that occur at the beginning or end of the remote method can be moved to the client by weaving those steps remotely on the client proxy object.

This technique can be used for problems that are difficult to solve in the worst case, but whose average case is relatively easy to solve. For example, consider a method that tests an integer for primality. Several efficient probabilistic primality tests such as Fermat [4] and Miller-Rabin [18] exist which can determine if a random integer is prime with a high degree of certainty.

The probabilistic primality test can be woven into the client proxy to be performed first, and if that test fails to determine the primality of the integer then the remote method can be called to use a deterministic primality test. Given a random positive integer, the client-side probabilistic primality test would determine the primality in most situations, and only rare integers such as pseudoprimes may require that the deterministic primality test on the server be used.

Any algorithms that contain base cases or check arguments for patterns that suggest easier ways to solve the problem can benefit from migrating these initial steps to the client. Since most realistic problems appear under these base cases, the majority of remote service requests can be handled at the client without performing a remote method invocation.

6.1.2 Predictive Cache

The caching example introduced in Section 4.1 remembers previously computed responses and uses them when possible. A predictive cache [3] in this context is a collection of responses computed by the server and woven on the client, based on the requests the server predicts it will receive from that client in the near future. The server is clearly not precognitive, so it must predict future requests based on the patterns of recent requests.

One obvious pattern to look for is an integer argument value that increments for each successive remote method invocation. The server can assume this pattern will continue, and spawn a new thread that computes the responses for the next 100 consecutive integers. These integer arguments and their corresponding responses can be stored in a cache and then sent to the client via a remote caching aspect. The client can then perform the next 100 requests very quickly because the responses are immediately available without needing to call the server.

For a concrete example, consider the `factorial(int)` method from the original caching example. A client that calls the server's `factorial()` method inside a loop like that in Figure 6.1 will generate the remote method calls `factorial(0)`, `factorial(1)`, `factorial(2)`, ... , `factorial(n - 1)`. If the server recognizes a pattern by `factorial(2)`, it can asynchronously compute all the values from `factorial(3)` to

`factorial(103)` and store those values in a cache. That cache can be included in the caching aspect sent to the client, who will start using the cached values instead of calling the server.

```
for (int i = 0; i < n; i++) {
      int fact = proxy.factorial(i);
      System.out.println(fact);
}
```

Figure 6.1 A loop in the client application making remote method invocations with a clear pattern.

Another pattern to look for are successive remote recursive method calls. Most recursive algorithms reduce the problem size by a known amount for each successive recursive call. The problem size may be represented by an integer, an array or list size, or a string length, but these cases should be easy to recognize and predict if the problem size decreases by a fixed amount for each recursive iteration.

The more interesting situation is when recursion is used to traverse a data structure such as a tree. If the server recognizes a breadth-first traversal pattern from a client it can compute the values for nodes in the tree X levels deep, where X is a parameter set by the server. Once these computed values are encapsulated in a caching aspect and sent to the client, the subsequent recursive service requests should be very quick because no remote method invocations are necessary. Similar predictive caches can be constructed for the various depth-first traversal patterns, also limited by a server parameter.

Like most predictive technologies, the predictive cache has disadvantages when its predictions are wrong. A client making a series of service requests triggering a pattern recognition on the server but who does not continue the pattern of requests is simply wasting the server's time and resources. The server can spend a lot of CPU cycles computing values that never get used on the client in this worst-case scenario. A slightly better scenario is when the server predicts the correct pattern but underestimates the number of responses to compute. The client will exhaust the server-generated cache and continue its request pattern, but reverting to slow remote method calls for each successive request. A well designed pattern trigger will once again recognize the request pattern, precompute the next batch of responses on the server, and send the new cache to the client by remote aspect.

The benefits of reduced remote method invocations needs to be balanced against the disadvantages of wasted CPU cycles on the server. There is ample room for further research by tweaking the server's limiting parameters and pattern recognition algorithms to optimize the benefits of predictive caching. Since the client application, like all programs, spends the majority of its execution time in loops or recursion, reducing the number of remote method invocations needed for successive related operations is extremely beneficial, and is something that cannot be solved by typical caching or any of the other examples described in this thesis.

6.1.3 Proxy-specific Advice Distribution

One limitation to the remote aspect weaving proposed thus far is its inability to distinguish between different proxy instances of the same class running on a client. JAC supports dynamic weaving on a *per-object* basis through the first parameter of the `pointcut()` method, a feature that suggests a specific proxy object can be advised separately from another proxy instance on the same client.

All the examples shown in this thesis supply an argument of "`.*`" for the first `pointcut()` parameter, meaning *all* proxy instances are advised. The join points selected by this pointcut can be narrowed to a single proxy object by replacing "`.*`" with a unique JAC object identifier. This identifier is a string with two parts, separated by a hash ("#") character. The first part is the name of the object's class in lower case without the package prefix, and the second part is an index that uniquely identifies the object to JAC.

An object's index can be retrieved using the static `getMemoryObjectIndex(obj)` method from the `ObjectRepository` class, where `obj` is the object in the JAC system. Suppose this method returns a value of 2 when it is called on an instance of the `Calculator_Stub` class. The corresponding identifier for this proxy object, used in the `pointcut()` definition, is "calculator_stub#2".

The problem is that proxy indices are only accessible from the client, but are needed at the server because the aspect, and thus the pointcut, originates on the server. The client somehow needs to make its proxy identification information available to the server to allow a finer

granularity of control over the client proxy objects that are affected by remote aspects woven from the server.

The first solutions that come to mind require the active cooperation of the client application, sacrificing the obliviousness of the client and one of the main goals of this thesis. The client would no longer be operating with the ignorance of server-woven aspects, but would be directly participating in the process in which proxies are advised. This sacrifice may be required to provide a proxy-specific advice weaving feature to the distributed system. Another possibility is making the index available through the RMI request itself. This solution may require custom RMI sockets on the client and server, again breaking the obliviousness and non-intrusive goals set out by this thesis.

The use of custom sockets opens up many other possibilities as well since the framework has access to the transport layer of the RMI stack. For instance, when remotely weaving on the client proxy object, the remote aspect can be piggybacked on the Java RMI response that the server returns to the client. Since a key objective of this framework is to reduce the quantity of messages sent between the client and server, combining remote aspects with the server response message is consistent with this objective. Server knowledge data can also be placed in the RMI layer unbeknownst to the client application developer, allowing the statically-generated proxy solution described in Section 5.1 to work without the corresponding complications involved with maintaining a Strategy object.

6.2 Conclusion

The information resources and services that our society takes for granted relies on a vast distributed software infrastructure composed of servers that provide services and clients that consume those services. Given the prevalence of client-server communication, research that investigates ways to improve or enhance these communications is important and widely applicable.

This thesis examined client-server communication in the context of Java RMI, and in particular investigated a solution to the problem of supplying sufficient server knowledge to the

client while still keeping the server and client loosely coupled. Treating remote communication efficiency as a software concern of the distributed system allows the tools and ideas of aspect-oriented software development to be part of the solution.

Remote communication efficiency differs from typical software concerns because it crosscuts not just program modules, but also physical computers in a network. An aspect-oriented programming framework with distribution support is needed to encapsulate and propagate this type of crosscutting concern in a distributed setting. Java Aspect Components (JAC) was chosen to fill this role because it is a Java-based aspect-oriented programming framework that supports remote advice execution and dynamic weaving.

The central idea behind the solution proposed in this thesis is to shift the burden of developing useful client-side enhancements from the client to the server. Aspects that represent client-side enhancements are created on the server and then remotely woven into the Java RMI proxy object on the client so they are executed when the client makes a remote call. The aspects are woven dynamically, which changes the behaviour of client proxy objects at runtime and allows for a highly flexible and adaptable solution. The server controls the content of the advice, the conditions under which the aspect is woven, and the specific clients it wishes to advise.

6.2.1 Thesis Goals and Benefits

In the introductory chapter, Section 1.2, four goals were specified that describe the desirable characteristics of a solution to the server knowledge sharing vs. loose coupling problem. The distributed dynamic aspect-oriented solution proposed in this thesis satisfies all four of these goals:

1. **Make server context available to the client.** This goal is achieved through the use of remote aspects. The server, which clearly has knowledge of its own state and internal implementation, produces the aspect's advice that modifies the client proxy object. The server context is made available on the client by weaving this advice remotely on the client.

2. **Obliviousness of the client application layer.** Client obliviousness is maintained because the remote aspects are woven dynamically on the Java RMI proxy object rather than into the client application itself. Since remote aspects in this solution only change the behaviour of proxies, the client application can continue using those proxies in exactly the same way without any idea that their behaviour may have changed. Dynamic weaving and unweaving of remote aspects allows the Java RMI proxy behaviour to be altered without the knowledge of the client application.

3. **No changes to existing client application code.** The distributed dynamic aspect-oriented solution is unobtrusive to the development cycle of the client application. No changes need to be made to the source code, referenced program libraries, or the build and compile tools. The only change required on the client is to run the client application inside a JAC container, which simply requires a slightly different command to run the application.

4. **Dynamic sharing of server context.** The proposed solution uses JAC, an aspect-oriented framework with built-in support for dynamic code weaving. This allows server context to be shared dynamically because the aspects that make use of server-side state and implementation knowledge can be woven on the client at runtime. The server has significant runtime control over the client proxy behaviour because it can also dynamically unweave aspects or weave multiple aspects on a single client to realize several client-side enhancements at once.

Since the purpose of dynamic client-side enhancements are to make client-server communications more efficient, the measurable benefits of this solution typically involve eliminating unnecessary remote method calls from the client to the server. This speeds up the execution of the client application because requests handled on the client are much faster than

invoking a remote method. Eliminating unnecessary remote calls also reduces overall network traffic and unburdens the server so it can serve other clients with relevant requests.

The degree of control given to the server by the framework has other benefits, such as the ability to provide custom enhancements to clients individually. A client's relationship with the server may be unique, so a custom aspect can be woven only on that client to reflect this unique relationship. Examples of client-side enhancement customization may include redirecting VIP clients to a faster, more powerful server; or making the validation rules more stringent for an untrustworthy client.

6.2.2 Relevance to Practical Problems

The flexibility of the distributed dynamic aspect-oriented solution is emphasized throughout this thesis, and the proposed framework can be adapted to many different domains to address a variety of problems. In Chapter 4, three practical client-side enhancements were introduced and implemented in the framework to demonstrate its potential uses.

The caching example is a widely used technique to reduce repetitive and redundant processing. When applied to client-server communications, it eliminates unnecessary remote method invocations to improve performance. The biggest problem with most client-side caching mechanisms is its lack of intelligent cache policy. The client does not know how the data on the server changes over time, making it difficult to select the most appropriate cache policy. The server, which does have this knowledge, needs to make the best cache policy available to the client to yield the optimal caching mechanism. The problem of making this cache policy available without compromising the loose coupling between client and server is solved by the framework proposed in this thesis.

The caching example may be extended to include predictive intelligence on the server that analyses client request patterns, predicts future requests, and pre-computes corresponding responses. These request-response pairs can be encapsulated in an aspect and woven, via a caching aspect, on the client. Any subsequent client requests that were correctly predicted can be retrieved from the local cache on the client proxy without requiring a remote method invocation.

Method argument validation is another well-known technique to eliminate unnecessary method calls. The validation rules are similar to method preconditions, ensuring that certain conditions about the arguments are met before the method is invoked. The problem with typical precondition engines is that the validation rules are bound to the method implementation and defined statically. The loose coupling between client and server allows the server implementation to change without affecting the client, but changing the server implementation may also change the method preconditions. Proper client-side validation must make use of the current server-side preconditions, so there needs to be a way of sending the server-side preconditions to the client without exposing the server implementation. Dynamically woven remote validation aspects provide the solution to this problem.

The load balancing example modifies the client in a slightly different way than caching and validation, and highlights the flexibility of this framework. Instead of controlling whether or not the remote method gets invoked, the load balancing example control *which* remote method gets invoked. More specifically, it alters the client proxy object to redirect the remote call to different servers. A load balancer uses this ability to distribute the load of client requests among resources in a server farm. The advantages of this load balancing mechanism over typical server-side load balancing is direct client-to-server communication. Typical load balancing requires all requests and responses to be routed through the load balancer, introducing additional overhead and communication traffic. The distributed dynamic aspect-oriented framework eliminates this middle-man in the communication channel, unbeknownst to the client.

Bibliography

[1] **ajc, the AspectJ compiler/weaver**, 2005. 12 Feb 2008. http://www.eclipse.org/aspectj/doc/released/devguide/ajc-ref.html

[2] P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, J. Lhoták, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. **abc: An Extensible AspectJ Compiler.** In *Proceedings of the 4th International Conference on Aspect-Oriented Software Development*, pages 87-98. ACM Digital Library, 2005.

[3] P. Cao, E. W. Felten, A. R. Karlin, and K. Li. **A study of integrated prefetching and caching strategies.** In *Proceedings of the 1995 ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems*, volume 23, issue 1, pages 188-197. ACM Digital Library, 1995.

[4] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. **Introduction to Algorithms**. Second Edition, MIT Press and McGraw-Hill, 2001.

[5] B. Dufour, C. Goard, L. Hendren, C. Verbrugge, O. de Moor, and G. Sittampalam. **Measuring the dynamic behaviour of AspectJ programs.** In *Proceedings of the 19th Annual Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 150-169. ACM Digital Library, 2004.

[6] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. **Design Patterns: Elements of Reusable Object-Oriented Software.** Addison-Wesley, 1994.

[7] jGuru. **Remote Method Invocation**, 2000. 05 Sept 2007. http://java.sun.com/developer/onlineTraining/rmi/RMI.html

[8] R. Kapitza , M. Kirstein, H. Schmidt, F. J. Hauck. **FORMI: An RMI Extension for Adaptive Applications**. In *Proceedings of the 4th Workshop on Adaptive and Reflective Middleware*. ACM Digital Library, 2005.

[9] R. Kapitza, J. Domaschka, F. J. Hauck, H. P. Reiser, and H. Schmidt. **FORMI: Integrating Adaptive Fragmented Objects into Java RMI.** In *IEEE Distributed Systems Online*, volume 7, issue 10, page 1. IEEE Educational Activities Department, 2006.

[10] M. Karaorman, U. Hölzle, and J. L. Bruno. **jContractor: A Reflective Java Library to Support Design by Contract.** In *Proceedings of the 2nd International Conference on Meta-Level Architecture and Reflection*, Lecture Notes in Computer Science, volume 1616, pages 175-196. Springer-Verlag, 1999.

[11] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin. **Aspect-oriented programming.** In *Proceedings of 11th European Conference on Object-Oriented Programming*, Lecture Notes in Computer Science, volume 1241, pages 220-242. Springer-Verlag, 1997.

[12] R. Kramer. **iContract - The Java™ Design by Contract™ Tool.** In *Proceedings of the Technology of Object-Oriented Languages and Systems*, pages 295-307. IEEE Computing Society, 1998.

[13] C. Lopes. **D: A Language Framework for Distributed Programming.** PhD thesis, College of Computer Science, Northeastern University, 1997.

[14] L. D. B. Navarro, M. Südholt, W. Vanderperren, B. De Fraine, and D. Suvée. **Explicitly distributed AOP using AWED**. In *Proceedings of the 5th International ACM Conference on Aspect-Oriented Software Development (AOSD'06)*, pages 51-62. ACM Press, 2006.

[15] M. Nishizawa, S. Chiba, and M. Tatsubori. **Remote pointcut: A language construct for distributed AOP.** In *Proceedings of the 3rd International Conference on Aspect-Oriented Software Development*, pages 7-15, 2004.

[16] R. Pawlak, L. Duchien, L. Seinturier, F. Legond-Aubry, G. Florin, and L. Martelli. **JAC: An Aspect-based Distributed Dynamic Framework**. Software Practice and Experience, volume 34, issue 12, pages 1119-1148. John Wiley & Sons, Inc, 2004.

[17] R. Pawlak, L. Seinturier, L. Duchien, and G. Florin. **JAC: A Flexible Solution for Aspect-Oriented Programming in Java.** In *Proceedings of the 3rd International Conference on Metalevel Architectures and Separation of Crosscutting Concerns*, Lecture Notes in Computer Science, volume 2192, pages 1-24. Springer-Verlag, 2001.

[18] M. Rabin. **Probabilistic algorithm for testing primality.** *Journal of Number Theory*, volume 12, pages 128-138, 1980.

[19] N. Santos, P. Marques, and L. Silva. **A framework for smart proxies and interceptors in RMI**. In *Proceedings of the 15th ISCA International Conference on Parallel and Distributed Computing Systems*, 2002.

[20] A. Stevenson and S. MacDonald. **Dynamic Aspect-Oriented Load Balancing in Java RMI.** Submitted to *The 2008 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA '08)*

[21] Sun Microsystems, Inc. **Dynamic Proxy Classes**, 2004. 20 Sept 2007. http://java.sun.com/j2se/1.5.0/docs/guide/reflection/proxy.html

[22] D. Suvee, W. Vanderperren, and V. Jonckers. **JAsCo: an Aspect-Oriented approach tailored for Component Based Software Development.** In *Proceedings of the 2nd*

International Conference on Aspect-Oriented Software Development (AOSD), pages 21-29. ACM Press, 2003.