Remote Pointcut
— A Language Construct for Distributed AOP

Muga Nishizawa
Tokyo Institute of Technology
2-12-1 Ohkayama, Meguro-ku,
Tokyo 152-8552, Japan
muga@csg.is.titech.ac.jp

Shigeru Chiba
Tokyo Institute of Technology
2-12-1 Ohkayama, Meguro-ku,
Tokyo 152-8552, Japan
chiba@is.titech.ac.jp

Michiaki Tatsubori
IBM Tokyo Research Lab.
1623-14 Shimotsuruma,
Yamato, Kanagawa 242-8502,
Japan
mich@trl.ibm.com

ABSTRACT

This paper presents our extension to AspectJ for distributed computing. Although AspectJ allows Java developers to modularize a crosscutting concern as an aspect, this paper shows that some crosscutting concerns in distributed computing are not modularized in AspectJ as simple aspects. Rather, aspects modularizing such a concern tend to be in code spread over multiple hosts and explicitly communicated across the network. This paper illustrates this fact with an example of testing a distributed program written in AspectJ with Java RMI. To address this complexity caused by network communication, this paper proposes an extension to AspectJ for distributed computing. The language construct that we call remote pointcut enables developers to write a simple aspect to modularize crosscutting concerns distributed on multiple hosts. This paper presents DJcutter, which is our AspectJ-like language supporting remote pointcuts.

Keywords

Distributed software, AspectJ, Language design

1. INTRODUCTION

Modularizing crosscutting concerns in distributed systems is one of significant demands in software industry. For example, transactions, security, and fault tolerance are typical crosscutting concerns in distributed systems. Current programming systems do not provide mechanisms for modularizing such concerns and thus they are major sources of low readability and maintainability of the software.

Many crosscutting concerns also arise during unit testing of distributed systems. The importance of testing frameworks is becoming widely accepted. In particular, since the XP (Extreme Programming) community [3] began advocating unit tests, unit testing has become a popular practice. Several tools such as JUnit [12] and Cactus [1] have been developed as simple regression test frameworks for automating unit testing.

The code for unit testing includes typical crosscutting concerns that AspectJ [7] can deal with [9]. AspectJ is a widely used language for aspect-oriented programming (AOP) [4, 8, 21, 14] in Java. Unfortunately, if we use AspectJ to modularize testing code for distributed software, the code ("aspect") can be somewhat modular but it often consists of several sub-components distributed on different hosts. They must be manually deployed on each host and the code of these sub-components must include explicit network processing among the sub-components for exchanging data since they cannot have shared variables or fields. These facts complicate the code of the aspect and degrade the benefits of using aspect-oriented programming.

The source of these problems is that the language constructs of AspectJ do not accommodate to distribution or network processing. Combination of AspectJ and an existing framework for distributed software, such as Java RMI (remote method invocation)[19] is not a solution. The existing frameworks extend language constructs for object-orientation, such as method calls, so that they can accommodate distribution. They do not support language constructs for aspect orientation.

To address these problems, this paper proposes a new AspectJ-like language named *DJcutter* . In this language, several language constructs, in particular pointcuts, have been extended for distributed software. The extended pointcuts of DJcutter are called *remote pointcuts.* Although a pointcut in AspectJ identifies execution points on the local host, a remote pointcut can identify them on a remote host. This language construct can simplify the code of a component implementing a crosscutting concern in distributed software. DJcutter also provides another language construct named remote inter-type declaration, which allows developers to declare a new method and field in a class on a remote host. The aspect weaving in DJcutter is performed at load time on each participating host. When a class is loaded from a local file system, it is transformed according to an aspect sent from a remote host. This architecture is useful for distributed unit testing since the users do not have to deploy a woven program to each host whenever they change the description of the aspects.

The rest of this paper is structured as follows. Section 2 illustrates our motivating example. It shows that a program

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

AOSD 04, March 2004, Lancaster UK.

Copyright 2004 ACM 1-58113-842-3/03/0004 ...$5.00.

written in AspectJ with Java RMI often includes complicated network processing in the description of some aspects. Section 3 presents DJcutter and its language constructs such as remote pointcut and remote inter-type declaration. Section 4 shows how the example in Section 2 can be implemented in DJcutter. Section 5 presents the results of our experiment using DJcutter. Section 6 mentions related work. Section 7 is conclusion.

2. COMPLICATIONS OF NETWORK PROCESSING

AspectJ is a useful programming language for developing distributed software. It enables modular implementation even if some crosscutting concerns are included in the implementation. However, the developers of distributed software must consider the deployment of the executable code. Even if some concerns can be implemented as a single component ("aspect") at the code level, it might need to be deployed on different hosts and it would therefore consist of several sub-components or sub-processes running on each host. Since Java (or AspectJ) does not provide variables or fields that can be shared among multiple hosts, the implementation of such a concern would include complicated network processing for exchanging data among the sub-components.

Programming frameworks such as Java RMI do not solve this problem of complication. Although they make details of network processing implicit and transparent from the programmers' viewpoint, the programmers still must consider distribution and they are forced to implement the concern as a collection of several distributed sub-components exchanging data through remote method calls. The programmers cannot implement such a concern as a simple, non-distributed monolithic component without concerns about network processing. This is never desirable with respect to aspect orientation since it means that the programmers must be concerned about distribution when implementing a different concern.

We illustrate this situation with an example of unit testing[1] for distributed software. Distributed test code includes crosscutting concerns but, if they are modularized in AspectJ, the code develops the complexities mentioned above. Writing test code for automating unit tests is an important development process that the XP (Extreme Programming) community recommends. The automation results in cleaner code, encourages refactoring, and makes rapid development possible. Recently, simple regression test frameworks such as JUnit and Cactus have been getting popular for the automated unit testing.

2.1 Unit test for authentication service

As an example, we present test code for a distributed authentication service. The implementation of this service consists of two components: a front-end server AuthServer on a host W and a database server DbServer on another host D. This is a typical architecture for enterprise Web application systems. If a client application needs to register a new user, it remotely calls registerUser() on the front-end server using Java RMI. Then the registerUser() method remotely calls addUser() on the database server, which will actually access the database system to update the user list.

[1]Some might think this example should be called not unit testing but end-to-end testing.

To unit-test the registerUser() method, the test code would first remotely call the registerUser() method and then confirm that the addUser() method is actually executed by the database server. Note that since the test code must confirm that remote method invocation is correctly executed, it must confirm not only that registerUser() on the host W calls addUser() but also that addUser() starts running on the host D after the call.

The test code would be simple and straightforward if the examined program is not distributed. We below show the test code written in AspectJ:

```
 1: aspect AuthServerTest extends TestCase {
 2:   boolean wasAddUserCalled;
 3:   void testRegisterUser() {
 4:     wasAddUserCalled = false;
 5:     String userId = "muga", password = "xxx";
 6:     AuthServer auth = new AuthServer();
 7:     auth.registerUser(userId, password);
 8:     assertTrue(wasAddUserCalled);
 9:   }
10:   before():
11:     execution(void DbServer.addUser(String,
12:                                     String)) {
13:     wasAddUserCalled = true;
14:   }
15: }
```

Although this is not complete code due to the space limitations, the readers would understand the overall structure of the test code. The main part of the test code is testRegisterUser() (lines 3 to 9). It calls the registerUser() method and then confirms the wasAddUserCalled field is true. This field is set to true by the before advice (lines 10 to 14) when the addUser() method is executed.

2.2 Test code in AspectJ

Unfortunately, the test code becomes more complicated if the examined program is distributed. The test code shown below is a distributed version (again, it is not complete code. Access modifiers such as public and constructors are not shown):

```
 1: // on host T
 2: class AuthServerTest extends TestCase {
 3:   boolean wasAddUserCalled;
 4:   void testRegisterUser() {
 5:     Naming.rebind("test", new RecieverImpl());
 6:     wasAddUserCalled = false;
 7:     String userId = "muga", password = "xxx";
 8:     AuthServer auth
 9:       = (AuthServer) Naming.lookup("auth");
10:     auth.registerUser(userId, password);
11:     assertTrue(wasAddUserCalled);
12:   }
13:   class ReceiverImpl
14:       extends UnicastRemoteObject
15:       implements NotificationReceiver {
16:     void confirmCall() {
17:       wasAddUserCalled = true;
18:     }
19:   }
20: }
21:
22: interface NotificationReceiver
23: { // on both hosts
24:   void confirmCall();
25: }
```

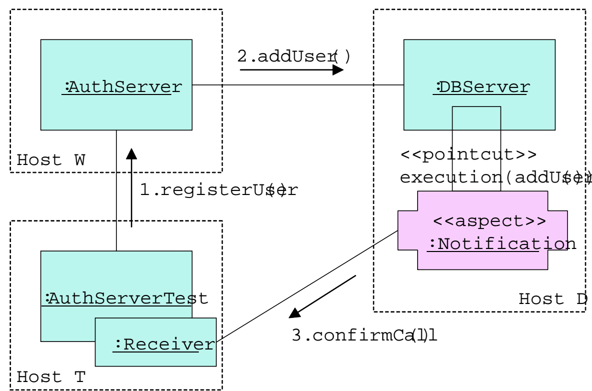

Figure 1: The testing code in AspectJ

```
26:
27: aspect Notification { // on host D
28:   before():
29:     execution(void DbServer.addUser(String,
30:                                     String)) {
31:       NotificationReceiver test
32:         = (NotificationReceiver)
33:           Naming.lookup("test");
34:       test.confirmCall();
35:   }
36: }
```

The test code now consists of three sub-components: AuthServerTest, ReceiverImpl, and Notification (Figure 1). Although the overall structure is the same, the AuthServerTest and ReceiverImpl objects run on a testing host T but the Notification aspect runs on the host D, where the DbServer is running. The host T is different from W or D.

The testRegisterUser() method (lines 4 to 12) on T remotely calls registerUser() on W and then confirms that the wasAddUserCalled field is true. This field is set to true by the confirmCall() method in ReceiverImpl, which is remotely called by the before advice (lines 28 to 35) of Notification running on D. The confirmCall() method cannot be defined in AuthServerTest since AuthServerTest must extend the TestCase class whereas Java RMI requires that remotely-accessible classes extends the UnicastRemoteObject class.[2]

As we can see, even this simple testing concern is implemented by distributed sub-components and hence we had to write complicated network processing code using Java RMI despite that it is not related to the testing concern. In particular, the Notification aspect is used only for notifying confirmCall() on the host T beyond the network that the thread of control on the host D reaches addUser(). The Notification aspect is a sub-component that are necessary only because confirmCall() and addUser() are deployed on different hosts. This means that the component design of the unit testing is influenced by concerns about distributed. Furthermore, this notification code is similar to what the AspectJ compiler produces for implementing the pointcut-advice framework. It should not be hand-coded, but implicit within the language constructs provided by an AOP language.

[2]This is not precisely accurate. Technically, a confirmCall() can be defined in AuthServerTest by using certain programming tricks. However, the test code would be significantly more complicated.

3. DJCUTTER

To address the problems of the previous section, we propose *DJcutter*, which is an extension to the AspectJ language for distributed software. It allows the users to implement a crosscutting concern as an aspect that does not include explicit network processing using Java RMI, even if that concern cuts across multiple components on different hosts.

3.1 Remote pointcut

The most significant difference between AspectJ and DJcutter is that DJcutter provides *remote pointcuts*. A remote pointcut is a function for identifying join points in the execution of a program running on a remote host. In other words, when the thread of control reaches the join points identified by a remote pointcut, the advice body associated with that remote pointcut is executed on a remote host different from the one where those join points occur. Remote pointcuts are analogous to remote method calls, which invoke the execution of a method body on a remote host. Unfortunately, AspectJ does not provide such a pointcut. An advice body in AspectJ is executed on the same host as where the join points identified by a pointcut occur.

Remote pointcuts enable implementing a distributed crosscutting concern as a simple, non-distributed component without concerns about network processing. The following is an aspect written in DJcutter, which conforms to the regular AspectJ syntax:

```
aspect LoggingAspect {
  pointcut setter(int x):
    args(x) && call(void Point.setX(int));
  before(int x): setter(x) {
    System.out.println("set x: " + x);
  }
}
```

This aspect prints a message whenever the setX() method is called on each participating host. The message is printed on a single particular host wherever the setX() method is called.

The setter pointcut in LoggingAspect:

```
call(void Point.setX(int))
```

identifies each join point that is a call to the setX() method in the Point class. Unlike pointcuts in AspectJ, however, this pointcut identifies the join points matching the signature on every host even if the advice body is not deployed on the host.

Aspect server

The body of the advice:

```
System.out.println("set x: " + x);
```

is executed just before each call to setX(), but it is executed on a host different from the host where the caller thread is running. If the thread of control reaches the join point, it implicitly sends a message through the network to an *aspect server* running on a different host[3] so that the aspect server will execute the advice body. The thread of control that sent

[3]Technically, the aspect server might be running on the same host.

the message blocks until the aspect server finishes the execution of the advice body. Since all of the advice bodies are executed by the aspect server on the central host, they can easily exchange values by storing data in the fields defined in the aspect. These fields are locally accessible from the advice bodies. Note that, in AspectJ, the advice body is executed on the same host where the caller thread is running. Thus it may have to explicitly send values through the network to exchange them with other advice bodies executing on other hosts.

Load-time weaving and remote inter-type declaration

DJcutter performs load-time weaving. The normal Java classes on each participating host must be loaded by the class loader provided by DJcutter [10]. This class loader weaves aspects and classes on the fly. The compiled aspects are stored in the aspect server. The parts of the compiled code except for the advice bodies are automatically distributed by the aspect server to each host, so the latest aspects can be woven when the classes are loaded. The users of DJcutter do not have to manually deploy the compiled aspects to every host.

This fact improves the usefulness of the inter-type declaration (formerly called *the introduction*) in DJcutter. An aspect can declare that it will respond to certain methods and field-access requests on behalf of other objects. In DJcutter, these methods and fields can be declared other objects on multiple remote hosts. Since the description of the inter-type declaration is automatically distributed from the aspect server to every host, declaring a method or field to classes on remote hosts is simple. The users only have to install the compiled aspect on the aspect server. Unlike in AspectJ, they do not have to manually deploy the woven aspect and classes to every host. This automatic deployment is useful in the context of distributed unit testing. We will revisit this issue in Section 4.

3.2 Pointcut designators

The pointcut designators provided by the current implementation of DJcutter are listed in Table 1. Most of the pointcut designators are from AspectJ.

A pointcut designator unique to DJcutter is hosts. It identifies the join points in the execution on the designated hosts. Although DJcutter can deal with all the join points on every participating host, this pointcut designator is used to identify the join points on particular hosts.

For example, the users of DJcutter can describe the following pointcut with the hosts pointcut designator:

```
pointcut sample(): call(void Point.setX(int))
                   && hosts(hostId1, hostId2)
```

This pointcut identifies join points that are calls to the setX() method in the Point class on the hosts with the names specified by hostId1 or hostId2. HostId1 and hostId2 are parameters given by the users when the program starts running. These runtime parameters allow the developers to avoid embedding particular host names as constants in the source code so that they can enjoy good flexibility.

DJcutter extends the cflow pointcut designator to handle the control flows of distributed software. cflow identifies join points that occur between the start of the method specified by cflow and the return. It identifies only the join points visited by the thread executing the method specified by cflow. In AspectJ, cflow cannot pick out join points on a remote host since the control-flow data needed to implement cflow is stored in a ThreadLocal variable but the ThreadLocal variable is never passed through a network.

DJcutter provides a custom socket class so that the ThreadLocal variable can be passed through a network. If network communication is performed with this custom socket class [20], then cflow can pick out join points on a remote host. For example, if Java RMI is used for network communication, the following program exports a remote object to make it available to receive incoming calls, using the custom socket class:

```
PointImpl p0 = new PointImpl();
Point p
   = (Point) UnicastRemoteObject.exportObject(
               p0, 40000,
               new DJCClientSocketFactory(),
               new DJCServerSocketFactory());
```

This program exports a PointImpl object, which is accessible from a remote host through the Point interface. The DJCClientSocketFactory and DJCServerSocketFactory classes are the factory classes provided by DJcutter for creating the custom socket. DJcutter also provides a convenient method with which the program shown above can be rewritten as follows:

```
PointImpl p0 = new PointImpl();
Point p
   = (Point) DJcutter.exportObject(p0, 40000);
```

3.3 Access to aspect methods

Although aspects are executed on the aspect server, normal Java classes can remotely call a method declared in the aspects. To make an aspect accessible from remote hosts, the aspect must implement an interface that declares the exported methods.

Suppose that we want to export a displayLog() method to remote hosts. The definition of the aspect should be as follows:

```
interface Logger extends AspectInterface {
  void displayLog(Point p, int x);
}
aspect LoggingAspect implements Logger {
  void displayLog(Point p, int x) {
    System.out.println("set x: " + x);
  }
  ...
}
```

The Logger interface declares the displayLog() method, which is exported to remote hosts. On the remote hosts, normal Java classes can remotely call the displayLog() method as follows:

```
Logger logger
  = (Logger) Aspect.get("LoggingAspect");
logger.displayLog();
```

Aspect is the class provided by DJcutter. The get method in Aspect returns a remote reference to the aspect with the specified name (in this example, LoggingAspect). The type of the remote reference is the interface type implemented by that aspect. If a method is called on the *proxy* object

Table 1: The pointcut designators of DJcutter

designator	join points
within(*TypePattern*)	the join points included in the declaration of the types matching *TypePattern*
target(*Type or Id*)	the join points where the target object is an instance of *Type* or the type of *Id*
args(*Type or Id, ...*)	the join points where the arguments are instances of *Type*s or the types of the *Id*s
call(*Signature*)	the calls to the methods matching *Signature*
execution(*Signature*)	the execution of the methods matching *Signature*
cflow(*Pointcut*)	all join points that occur between the entry and exit of each join point specified by *Pointcut*
hosts(*Host, ...*)	the join points in the execution on *Host*s.

represented by the remote reference, then the corresponding method in the aspect is invoked on the aspect server.

This architecture using the proxy objects is the same as that of Java RMI [6]. The reason why methods in aspects must be called through an interface type is that this architecture enables separate compilation. The developers can compile normal Java classes without aspects, provided that the interface type is available. This is quite helpful in the development of distributed software. Furthermore, this architecture allows the developers to implement components independently of each other. For example, they can start describing a normal class that remotely calls a method in an aspect before they have finished describing the aspect, if the interface declaring the exported method is already available.

3.4 Pointcut parameters

Like AspectJ, DJcutter allows pointcuts to expose the execution context of the join points they identify. For example, the args pointcut designator can expose method parameters and the target pointcut designator can expose the target object. Each part of the exposed context is bound to a pointcut parameter, which is accessible within the body of the advice. For example,

```
pointcut setter(int x):
  call(void Point.move(int,int))
  && args(x, *)
```

This setter pointcut exposes the first int-type parameter to the move method through a pointcut parameter x.

In DJcutter, since remote pointcuts identify join points on remote hosts, the pointcut parameters should refer to data on the remote hosts. By default, they refer to a copy of that data constructed on the aspect server. The runtime system of DJcutter first serializes the data on the remote hosts, transfers it through the network, and constructs a copy from the serialized data. The pointcut parameters available in the advice body refer to that copy.

Pointcut parameters can be specified as remote references instead of local references to the copies. If the configuration file specifies that pointcut parameters of class type C are remote references, then the runtime system of DJcutter dynamically generates a proxy class for C. From the implementation viewpoint, the pointcut parameters are made to refer to instances of that proxy class on the aspect server. If the advice body calls a method on that proxy object, then the method is invoked on the master object on the remote host where the join point occurs. To generate proxy classes, DJcutter uses the replace approach we developed for Addistant [22]. For example, if the remote object associated with a proxy object is a Widget object, then the proxy class is also named Widget. On the aspect server, this proxy class is loaded instead of the original Widget class. The proxy-class generation is performed with our bytecode engineering library Javassist [5].

Remote references are used not only for pointcut parameters but also references to instances of aspects. As shown in Section 3.3, normal Java classes can call methods declared in aspects. The references to the instances of the aspects are also remote references implemented using the same approach as for the pointcut parameters. In addition, the parameters of the methods called on the remote object indicated by a remote reference can be also remote references.

3.5 Reflection by thisJoinPoint

As in AspectJ, DJcutter provides the thisJoinPoint special variable for reflective access to join points. This variable refers to an object representing the context of the current join point or advice. It is available within the body of the advice code.

The thisJoinPoint variable provided by DJcutter has a getHost() method to acquire the name of the host where the identified join point is located. For example, the before advice below records the name of the host that last called a method in the Point class:

```
String lastCallerHost;
before(): call(void Point.*(..)) {
  lastCallerHost
    = thisJoinPoint.getHost();
}
```

3.6 Local aspect

The implementations of crosscutting concerns in distributed systems do not always involve multiple hosts. Such crosscutting concerns can be implemented without remote pointcuts as a simple non-distributed component. If they are implemented with remote pointcuts, the execution performance is rather worse because of the overheads due to network communication to the aspect server. These crosscutting concerns should be implemented with the aspects provided by AspectJ.

DJcutter therefore provides ones similar to the aspects of AspectJ. The developers can specify that copies of an aspect are distributed to each participating host and that body of advice in the aspect is locally executed on the same

host as where the join points exist. These types of aspects, which are called *local aspects*, are equivalent to the aspects available in AspectJ. Since a local aspect is instantiated on each host, the fields declared in the aspect are not shared among the hosts. A value assigned to such a field on one host is not visible on the other hosts. To exchange data among the hosts, the data must be explicitly transferred through the network, for example, by using Java RMI.

4. EXAMPLES

In this section, we show two example programs written in DJcutter to illustrate how remote pointcuts and inter-type declaration can be used for distributed unit testing.

4.1 The use of remote pointcut

The testing code presented in Section 2 was complicated compared to the non-distributed version of the testing code. If we rewrite that testing code in DJcutter, then the resulting code becomes as simple as the non-distributed version:

```
 1: // on host T
 2: aspect AuthServerTest extends TestCase {
 3:   boolean wasAddUserCalled;
 4:   void testRegisterUser() {
 5:     wasAddUserCalled = false;
 6:     String userId = "muga", password = "xxx";
 7:     AuthServer auth
 8:       = (AuthServer) Naming.lookup("auth");
 9:     auth.registerUser(userId, password);
10:     assertTrue(wasAddUserCalled);
11:   }
12:   before(): // remote pointcut
13:     cflow(
14:       call(void AuthServer.registerUser(String,
15:                                         String)))
16:     && execution(void DbServer.addUser(String,
17:                                        String))){
18:     wasAddUserCalled = true;
19:   }
20: }
```

Unlike the code in AspectJ, the testing code in DJcutter is not divided into distributed sub-components (Figure 2). Although the before advice (lines 12 to 19) is executed when the thread of control reaches the addUser() method on the host D, where the DbServer is running, the execution of the before advice is on a different host T, where the testRegisterUser() method is running. Thus the before advice can directly set wasAddUserCalled to true. All the network processing for reporting the execution of the addUser() method to the host T needs not be explicitly described.

Note that the before advice contains the cflow pointcut designator, since DJcutter provides cflow across multiple hosts if the components communicate with the Java RMI. This improves the accuracy of the testing code. The code can examine not only whether or not addUser() is executed, but also whether the caller to addUser() is registerUser().

The testing code in DJcutter has another advantage. Since DJcutter automatically distributes the definitions of the aspects to each participating host and weaves them at load time, the programmers do not have to manually deploy the compiled and woven code to the hosts whenever the definitions of the aspects are changed for different tests.

4.2 The use of remote inter-type declaration

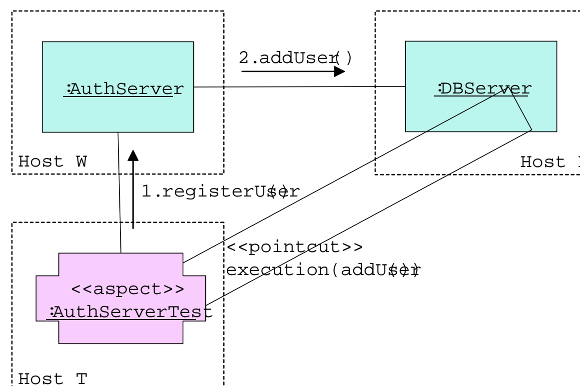

Figure 2: The testing code in DJcutter

Unit testing sometime requires accessor methods for inspecting the internal state of objects. AspectJ can be used to append such accessor methods just for testing if these methods are not defined in the original program. For example, the developer may want to confirm that the data sent by the registerUser() method is actually stored in the database by the addUser() method. To do this, an accessor method containsUser() must be appended to the DbServer class so that the testing code can examine whether the added user entry is contained in the database.

The remote inter-type declaration of DJcutter simplifies such unit testing. If the developers use AspectJ, they have to recompile all the programs and deploy the compiled and woven code to the participating hosts whenever they change the inter-type declaration in the aspect. On the other hand, DJcutter can simplify this deployment. Since DJcutter automatically distributes the new definitions of the aspect to the hosts and weaves it at load time, the new aspect is reflected in the programs if the programs are simply restarted.

The following is the testing code written in DJcutter. It appends containsUser() to the DbServer class (lines 13 to 18). The testRegisterUser() method first confirms that the user muga is not recorded in the database (line 9) and then it calls the registerUser() method (line 10). After that, it confirms that the user muga is recorded in the database (line 11).

```
 1: // on host T
 2: aspect AuthServerTest extends TestCase {
 3:   void testRegisterUser() {
 4:     String userId = "muga", password = "xxx";
 5:     AuthServer auth
 6:       = (AuthServer) Naming.lookup("auth");
 7:     DbServer db
 8:       = (DbServer) Naming.lookup("db");
 9:     assertTrue(!db.containsUser(userId));
10:     auth.registerUser(userId, password);
11:     assertTrue(db.containsUser(userId));
12:   }
13:   boolean DbServer.containsUser(String
14:                                   userId) {
15:     // this method returns true if the user
16:     // entry specified by userId is found
17:     // in the database.
18:   }
19: }
```

5. EXPERIMENT

To examine the execution performance of remote pointcuts, we compared the execution time between DJcutter and AspectJ using Java RMI. For this experiment, we used the testing programs shown in Section 2.2 (AspectJ using Java RMI) and Section 4.1 (DJcutter). These programs examine whether registerUser() in AuthServer remotely calls addUser() in DbServer. We measured the elapsed time of the testRegisterUser() method for each program. The body of the addUser() method was empty. In this experiment, the AuthServer and the AuthServerTest ran on the same host while DbServer ran on another host. The AuthServer host was a Sun Blade 1000[4] and the DbServer was a Sun Fire V480[5]. These hosts were connected through a 100 BaseTX network. We used Sun JDK 1.4.0_01 and AspectJ 1.0.6.

Table 2 lists the results of our measurement. Although the program in DJcutter was slightly faster than in AspectJ, this result does not mean DJcutter is considerably faster than AspectJ using Java RMI. In the program in AspectJ (see Section 2.2), when the body of the before advice is executed, a remote reference test (lines 31 to 33) is obtained for calling confirmCall(). On the other hand, this remote reference is not obtained in DJcutter during the measurement. It is implicitly obtained by the runtime system in advance. Since obtaining this remote reference needs remote access to the registry server, this difference caused about 1 milli-second ahead of DJcutter in the measurement. We confirmed this fact by other experiment.

The programs shown in Section 2.2 and 4.1 do not use pointcut parameters. To evaluate effects by sending pointcut parameters through a network, we also examined the programs in that the before advice (in DJcutter) or the confirmCall() method (in AspectJ) receives one or both of the parameters to the addUser() method (DJcutter). The type of the parameters is the String class. The results of our measurement showed that the performance impacts by pointcut parameters are small.

For fair comparison, we also measured the elapsed time of the program written in DJcutter without cflow since the program in AspectJ did not use cflow. The results were similar to those of the program using cflow since the overhead due to cflow across a network is not significant.

6. RELATED WORK

Soares et al reported that they could use AspectJ for improving the modularity of their program written using Java RMI [18]. Without AspectJ, the program must include the code following the programming conventions required by the Java RMI. AspectJ allows separation of that code from the rest into a distribution aspect. However, the ability of AspectJ is limited with respect to modularization for distributed programs and thus the resulting programs are often complicated and difficult to maintain. To address these complications, we propose remote pointcuts and the inter-type declaration as extended language constructs for distributed aspect-oriented programs.

Although Java RMI is the standard framework, several researchers have been proposing other systems such as cJVM [2], our own Addistant [22] and J-Orchestra [23]. These systems provide a single virtual machine image on several hosts connected through a network. They allow for the distributed execution of a program originally written as a non-distributed one, without code modification for the distribution. An alternative to the approach presented here might be to write a program in AspectJ and run it on these systems, which would appropriately translate local pointcuts into remote pointcuts at the implementation level. We did not take this approach since our target applications are for the unit testing of enterprise server software, and these programs are inherently designed and implemented as distributed software. Therefore, we do not have to translate such software to distributed software by, for example, using Addistant, except for the modules implemented as aspects. If we translate all the modules of such software, the unnecessary indirections due to the proxy objects would cause significant performance penalties, since such software has already included indirections for remote accesses. On the other hand, DJcutter can be regarded as a system that translates only aspects to enable transparent remote accesses. Although Addistant allows the programmers to specify translation only for the classes generated by the AspectJ compiler from the aspects, the programmers must manually describe these specifications. DJcutter provides better syntax so that these specifications can be simple or implicit within the language constructs.

Distribution is a well known crosscutting concern and several systems have been proposed to support such concerns. For example, the D language [11] allows the programmers to separately describe how a parameter is passed to a remote procedure. Such work has explores new crosscutting concerns in distributed programs whereas our work explore general-purpose language constructs for distributed aspect-oriented programs. The goal of our work is to develop language constructs so that programs written in an AspectJ-like language can be simple and easy to maintain.

JAC [15, 13] is a powerful framework for dynamic AOP in Java. Unlike other lanugages such as AspectJ, JAC does not require any langugae extensions to Java. An aspect of JAC is implemented by a set of aspect objects. JAC also supports Java API that easily implements crosscutting concerns in distributed systems such as the codes changing consistency protocol on a set of replications and implementing load-balancing for developers. But, using JAC, even if developers will separate the crosscutting concerns during unit testing, complicated network processing is not necessarily solved. The significant difference JAC and DJcutter is that DJcutter provide the remote pointcut.

DADO (Distributed Adaplets for Distributed Objects) [24] provides a CORBA-like programming model, which comprises several languages, tools, and runtime environment, to support crosscuting concerns in distributed heterogeneous systems. This programming model enables the developers to separate crosscutting implementation that arised in application components on both client and server side such as security, caching. In particular, the DADO programming model has two languages. One of these languages, DADO deployment language, is based on AspectJ and specifies how a QoS feature interacts with an underlying application. However these languages allow modeling the communications between client and server side, don't support remote pointcuts provided by DJcutter.

---

[4] Dual UltraSPARC III 750 MHz with 1 GB of memory and Solaris 8.

[5] UltraSPARC III Cu 900 MHz ×4 with 16 GB of memory and Solaris 9.

Table 2: The elapsed time (msec.) of testRegisterUser()

Pointcut parameters	()	(String)	(String,String)
Java + Java RMI	5.9	5.9	6.0
AspectJ + Java RMI	5.9	6.0	6.0
DJcutter	4.8	4.9	5.0
DJcutter without cflow	4.8	4.9	4.9

7. CONCLUDING REMARKS

This paper presented DJcutter, which provides remote pointcut as a new language construct for distributed AOP. A remote pointcut is a function for identifying join points in the execution of a program running on a remote host. It can simplify the description of aspects with respect to network processing if the aspects implement a crosscutting concern spanning over multiple hosts. To illustrate this situation, this paper used the example of a program written in DJcutter for distributed unit testing. The remote pointcut is a crucial language construct for distributed AOP, corresponding to remote method invocation (RMI) for distributed object-oriented programming.

Although we adopted load-time weaving for DJcutter, runtime weaving is more appropriate if we use DJcutter as a testing framework for distributed software. This allows the developers to change the testing code written as an aspect without restarting the target software they are testing. Extending DJcutter is our future work to enable runtime weaving such as exists PROSE [16] and Wool [17]. Another area of our future work is performance improvement. Although the advice bodies in all aspects are currently executed in the aspect server, this centralized approach might be a performance bottleneck. We will extend DJcutter to allow multiple aspect servers for better performance.

Acknowledgments

We would like to thank the anonymous reviewers. Their suggestions and comments helped us revise this paper. We also thank Shannon Jacobs for his great efforts to fix numerous english problems in this paper.

8. REFERENCES

[1] Apache Software Foundation, Online publishing, URI http://jakarta.apache.org/cactus/. *CACTUS*, 2000.

[2] Y. Aridor, M. Factor, and A. Teperman. cJVM: A single system image of a JVM on a cluster. In *International Conference on Parallel Processing 1999 (ICPP 1999)*, pages 4–11, 1999.

[3] K. Beck. *Extreme Programming Explained: Embrace Change*, chapter 4. Addison-Wesley, 1999.

[4] L. Bergmans and M. Aksits. Composing crosscutting concerns using composition filters. In *CACM*. ACM Press, 2001.

[5] S. Chiba. Load-time structural reflection in java. In *European Conference on Object-Oriented Programming 2000 (ECOOP 2000)*, LNCS 1850, pages 313–336. Springer Verlag, 2000.

[6] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*, chapter 4. Addison-Wesley, 1995.

[7] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of aspectj. In *European Conference on Object-Oriented Programming 2001 (ECOOP 2001)*, LNCS 2072, pages 327–353. Springer, 2001.

[8] G. Kiczales, J. Irwin, J. Lamping, J.-M. Loingtier, C. V. Lopes, C. Maeda, and A. Mendhekar. Aspect-oriented programming. In *European Conference on Object-Oriented Programming 1997 (ECOOP 1997)*, LNCS 1241, pages 220–242. Springer, 1997.

[9] N. Lesiecki. *Test flexibly with AspectJ and mock objects*. IBM developerWorks, Online publishing, URI http://www-106.ibm.com/developerworks/java/library/j-aspectj2/, May 2000.

[10] S. Liang and G. Bracha. Dynamic class loading in the java virtual machine. In *Object-Oriented Programming Systems, Languages, and Applications 1998 (OOPSLA 1998)*, pages 36–44. ACM SIGPLAN Notices, 1998.

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

[12] Object Mentor, Online publishing, URI http://www.junit.org/index.htm. *JUnit.org*, 2001.

[13] ObjectWeb Consortium, Online publishing, URI http://jac.objectweb.org/index.html. *The JAC Project*, 1999.

[14] D. Orleans and K. Lieberherr. Dj: Dynamic adaptive programming in java. In *International Conference on Meta-level Architectures and Separation of Crosscutting Concerns 2001 (Reflection 2001)*, pages 73–80. Springer Verlag, 2001.

[15] P. Pawlak, L. Seinturier, L. Duchien, and G. Florin. Jac: A flexible solution for aspect-oriented programming in java. In *International Conference on Metalevel Architectures and Separation of Crosscutting Concerns 2001 (Reflection 2001)*, LNCS 2192, pages 1–24. Springer, 2001.

[16] A. Popovici, T. Gross, and G. Alonso. Dynamic weaving for aspect-oriented programming. In *Aspect-Oriented Software Development 2002 (AOSD 2002)*, pages 141–147. ACM Press, 2002.

[17] Y. Sato, S. Chiba, and M. Tatsubori. A selective, just-in-time aspect weaver. In *Generative Programming and Component Engineering 2003 (GPCE 2003)*, LNCS 2830, pages 189–208. SV, 2003.

[18] S. Soares, E. Laureano, and P. Borba. Implementing distribution and persistence aspects with aspectj. In *Object-Oriented Programming Systems, Languages, and Applications 2002 (OOPSLA 2002)*, pages 174–190. ACM SIGPLAN Notices, 2002.

[19] Sun Microsystems, Inc, Online publishing, URI http://java.sun.com/products/jdk/rmi/. *Java Remote Method Invocation (RMI)*, 1995.

[20] Sun Microsystems, Inc, Online publishing, URI http://java.sun.com/j2se/1.4.1/docs/guide/rmi/socketfactory/index.html. *Using a Custom RMI Socket Factory*, 1995.

[21] P. Tarr, H. Ossher, W. Harison, and S. M. S. Jr. N degrees of separation: multi-dimensional separation of concerns. In *International Conference on Software Engineering 1999 (ICSE 1999)*, pages 107–119. IEEE Computer Society Press, 1999.

[22] M. Tatsubori, T. Sasaki, S. Chiba, and K. Itano. A bytecode translator for distributed execution of legacy java software. In *European Conference on Object-Oriented Programming 2002 (ECOOP 2002)*, LNCS 2072, pages 236–255. Springer, 2001.

[23] E. Tilevich and Y. Smaragdakis. J-orchestra: Automatic java application partitioning. In *European Conference on Object-Oriented Programming 2002 (ECOOP 2002)*. Springer, 2002.

[24] E. Wohlstadter, S. Jackson, and P. Dvanbu. Dado: Enhancing middleware to support cross-cutting features in distributed, heterogeneous systems. In *International Conference on Software Engineering 2003 (ICSE 2003)*, pages 174–186. IEEE Computer Society Washington, DC, USA, 2003.