Distributed Dynamic Weaving is a Crosscutting Concern

Michihiro Horie Satoshi Morita Shigeru Chiba
Dept. of Mathematical and Computing Sciences
Tokyo Institute of Technology
http://www.csg.is.titech.ac.jp

ABSTRACT

Implementation of distributed dynamic weaving is a crosscutting concern since the implementation is divided into several sub-concerns and some of them are crosscutting concerns. For example, it often includes a monitoring concern, which monitors the progress of the target program running on remote hosts. It must be dynamically woven in the target program in a crosscutting way. Existing dynamic distributed languages do not provide sufficient support for modularly implementing such distributed dynamic weaving. This paper proposes our new language named *DandyJ*, which enables developers to implement distributed dynamic weaving by an aspect. The aspect implementing it is reusable and hence DandyJ allows developers to write an aspect library for weaving a given aspect in distributed environments. We designed DandyJ by integrating a few good ideas borrowed from existing work, such as first-class aspects, remote pointcuts, and atomic weaving. The contribution of this paper is to show a set of language constructs necessary for writing an aspect library for distributed dynamic weaving, which is also a crosscutting concern.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features — Classes and objects

Keywords

Aspect Oriented Programming, Java, AspectJ.

1. INTRODUCTION

Aspect-Oriented Programming (AOP) languages allow programmers to modularize crosscutting concerns by aspects. This idea has been extended to distributed computing and some AOP languages even allow applying a new aspect to a distributed program when it is running. Such languages are called distributed dynamic AOP languages. Weaving an aspect and changing the behavior of a running program is only possible when the program is in a stable state. If a program is distributed, weaving must be more carefully controlled to keep the consistency among the machines where that program is running.

According to our experience, keeping the consistency when dynamically weaving an aspect has not been drawing much attention from researchers. This issue has not been deeply discussed with respect to the design of dynamic AOP languages. Current distributed dynamic AOP languages, therefore, do not provide language constructs that sufficiently support consistent aspect weaving.

This paper reports that the control of distributed dynamic weaving itself is a crosscutting concern, which is another interesting application of AOP. A typical control program depends on the semantics of a target program where a target aspect is woven under that control. It must monitor the progress of the target program to know when it reaches a stable state. Thus, it is often closely related to the target program and the code is tangled with each other. This means that, if the control program is written in a traditional programming language, (at least part of) it is embedded into the target program and hence hard to be reused.

To avoid this problem, this paper proposes using an aspect to implement a control program of distributed dynamic weaving. Since previous distributed dynamic AOP languages do not sufficiently support this, this paper also proposes our new AspectJ-like language named *DandyJ*. It provides a useful set of language constructs for writing an control aspect of distributed dynamic weaving. They are first-class aspects with two-phase weaving, remote pointcut, and one-time aspects. These language constructs are not very new; they have been already proposed in other literature. DandyJ has been developed by a synthesis and re-engineering of ideas borrowed from other related languages. The contribution of this paper is the following:

- Showing a useful set of language constructs that distributed dynamic AOP languages should provide for supporting distributed dynamic weaving, which itself is a crosscutting concern.

In the rest of this paper, Section 2 illustrates issues of distributed dynamic weaving. Section 3 presents DandyJ and Section 4 illustrates sample programs written in DandyJ. Section 5 shows the results of our experiment using DandyJ. Section 6 mentions related work and Section 7 concludes this paper.

2. DISTRIBUTED DYNAMIC WEAVING

Dynamic aspect orientation is a language mechanism for customizing the behavior of a running program at runtime. An aspect is a module implementing one kind of customization. Applying an aspect to its target program is called *weaving*. An aspect can dynamically replace a whole method body with a new one and/or a method call with a call to another method. The substituted methods are described in an aspect and they are called *advice*. The replaced method bodies and method calls are specified by a language construct named *pointcut*, which is also described in an aspect. Here, being dynamic means that an aspect woven at runtime may not be written yet or known when its target program starts running. The code of the aspect must be inserted during runtime into the target program, for example, by code transformation and runtime supports.

Weaving an aspect on multiple hosts in distributed environments is not a simple task. It is more complicated when an aspect is dynamically woven with a running program. Suppose that we are now running a parallel program on multiple hosts to compute a N-body problem. A program on every machine computes the position, velocity, and acceleration of particles allocated to the machine. At every tick, the program exchanges the positions of particles with other machines and computes the new positions *etc.* of the particles allocated to it.

Let us weave a logging aspect with this running N-body program. This logging aspect will send at every tick the positions of particles on every machine to a monitor machine, which then displays the visual image of the current positions of the particles. The visual image will be used by a scientist to watch the computation in progress and see that the program is running as expected. If she is satisfied with the result, we will unweave the logging aspect.

Since the logging aspect is woven on every computing machine, weaving it must be atomic — the aspect code must be woven *"simultaneously and consistently"* on every machine in a coordinated means. Otherwise, the aspect code will start sending data at different ticks since weaving may take different time on each machine due to network latency.

An easiest approach for atomic weaving is to suspend the programs on all the machines at some certain tick, weave the aspect, and restart them. However, which tick do we suspend the program at? How do we do that? We could not know the progress of the running programs or suspend them at a safe point unless we modified the original programs in advance to enable us to do so. Otherwise, before weaving the logging aspect, we must weave another aspect for monitoring the progress and suspend the programs; but that aspect must be also dynamically woven in a distributed environment. We must consider this distributed dynamic weaving as well. This may cause infinite regression.

Another known approach is to split distributed dynamic weaving into two phases: deploy and activate [20]. In this approach, an aspect is not activated immediately after the aspect code is delivered and ready to run. It is first *deployed*; the aspect code is delivered to a remote machine and then embedded into a running program for example, by dynamic code instrumentation. After the deployment finishes, the aspect is activated when it is explicitly *activated*. Thus, by waiting for a while after starting deployment, programmers might be able to guarantee that an aspect is simultaneously activated after it is deployed on all machines. However, a question is *when* we should activate the logging aspect after the deployment of it finishes on all machines. Since network latency differs, a signal of activation will not simultaneously arrive at every machine. We must find a means to activate the aspect on all machines at the same tick. Splitting into deploy and activate may minimize the pause time for atomic weaving but it is not a direct solution of distributed atomic weaving.

Atomic weaving in distributed environments is not independent of the semantics of target applications. An aspect must be woven and activated at the time determined by using knowledge of the target application. For example, to weave our logging aspect atomically, after the aspect is deployed, the N-body program on each machine will check whether or not the aspect is deployed on all the machines. It will check this whenever it blocks by barrier synchronization at the beginning of every tick. When the check result becomes true, the program will successfully activate the aspect on its machine. This will ensure the atomicity.

To do this control, another aspect is needed since the code for controlling the weaving of the logging aspect is obviously distributed and crosscutting over the N-body program. For example, the check code mentioned above must be dynamically embedded into the barrier-synchronization code of the N-body program. Hence, the control of atomic weaving cannot be implemented without dynamic aspect-orientation. A lesson from this discussion is that distributed atomic weaving often needs another aspect for controlling that weaving. However, the support by existing distributed dynamic AOP languages are not sufficient. For example, although the control of atomic weaving is a distributed algorithm, some languages do not provide programming supports for implementing that distributed algorithm. Other languages do not enable an aspect for that control to be an independent aspect. It is merged into the aspect that should be woven under that control. The woven aspect contains the code for monitoring the progress of the weaving on other machines and runs its logging code only after the aspect is woven on all machines. A woven concern (*i.e.* logging) and a concern of atomic weaving of it are not separated but both are implemented in the same aspect. The two concerns are tangling. This fact makes it difficult to reuse (part of) the control code for weaving another kind of aspect. Furthermore, removing the control code after the weaving finishes is also difficult. For example, the check code mentioned above is not necessary any longer after the weaving and hence it should be removed for reducing performance overheads.

3. DANDYJ

We present our distributed dynamic aspect-oriented language named *DandyJ*. This is an extension of Java and it provides sufficient supports for implementing an aspect that controls atomic weaving of another aspect. In DandyJ, an aspect to control weaving can be an independent aspect.

DandyJ provides three language constructs for this purpose. They are first-class aspects with two-phase weaving, remote pointcuts, and one-time aspects. They are not new but there have been no distributed dynamic AOP languages providing all the three constructs as far as we know. We designed DandyJ to illustrate that the combination of these constructs is useful for distributed atomic weaving.

Table 1: The methods available on dynamic aspects

methods	description
void deploy()	deploy a dynamic aspect with being inactive
void undeploy()	remove an aspect
void activate()	activate an aspect
void unactivate()	inactivate an aspect
boolean isDeployed()	returns true if an aspect has been deployed
boolean isActivated()	returns true if an aspect is active

3.1 First-class aspects and two-phase weaving

DandyJ provides *a dynamic aspect*, which is an aspect that can be dynamically woven during runtime. If an aspect declaration has the dynamic modifier, it is a dynamic aspect. A dynamic aspect is a first-class object; it has to be explicitly instantiated by the new operator before being woven but can be assigned to a variable and passed as a method parameter. This enables us to write an aspect that takes another aspect as a parameter and dynamically weaves it while keeping atomicity. Such an aspect can be used as reusable library code.

Any dynamic aspect is implicitly a subtype of DAspect. On a DAspect object, several methods listed in Table 1 is available for dynamic weaving. DandyJ provides two-phase weaving, which is another significant feature of DandyJ since it enables efficient distributed weaving. When an aspect is woven, it first deployed and then activated. If the deploy method is called on an aspect instance, in the current implementation of DandyJ, the runtime system performs code transformation on a running program so that the aspect instance will be ready to run. Since it will take time, the deployment is performed asynchronously; the deploy method starts deployment and returns immediately. After the deployment finishes, the aspect is not active yet. It starts being effective when it is explicitly activated by the activate method, which blocks until the activation finishes (but it is an instant since this method only updates the value of a boolean field). If an aspect is activated before its deployment finishes, it becomes active immediately after the deployment finishes. To know when the deployment finishes, DAspect also provides the isDeployed method. The asynchronous deploy and synchronous activate methods allow us to minimize the duration of the "stop the world" time for dynamic weaving.

These methods enable us to write a method that receives any dynamic aspect and weaves it according to some specific policy. For example,

```
void doDynWeave(DAspect aspect) {
  aspect.deploy();
  do {
    Treed.sleep(1000);
  } while (!aspect.isDeployed())
  aspect.activate();
}
```

This method first deploys a given dynamic aspect and waits for one second. Then it activates the aspect if the deployment has finished.

3.2 Remote pointcut

Although DandyJ supports distributed weaving, an instance of a dynamic aspect is not copied or migrated to other hosts[1] when it is woven. It stays on the machine where it was created by the new operator. The advice of that aspect is basically applied to the program on the same machine where the aspect instance exists.

To replace a method call on a remote machine, DandyJ provides *remote pointcuts* [11]. Remote pointcuts select *join points* (*i.e.* execution events such as method calls) on a remote machine. When a selected join point happens there, the advice body associated with that remote pointcut is invoked on the host where the aspect instance exists. This host may be different from the host where the join point happens. This language construct is useful to implement a control program of distributed weaving, in particular, when the control program adopts a centralized approach. The control program will be implemented by another dynamic aspect on a central host machine and it will use remote pointcuts to monitor the progress of its target programs on remote machines where the target aspect is woven.

For remote pointcut, DandyJ provides two pointcut designators listed in Table 2. The following is sample code of remote pointcut:

```
pointcut sending(String h): execution(void Client.send())
          && hosts("node000", "node001") && hostName(h);
```

This pointcut selects join points that are the execution of the send method in the Client class running on node000 or node001. The name of the host machine where the send method is executed is bound to the parameter h to the sending pointcut. The value of h, which will be either "node000" or "node001", is available in the advice body associated with this pointcut.

Since a remote pointcut selects join points on different machines, when an aspect using a remote pointcut is woven, the runtime system has to perform some preparation on the remote machines. Thus, the deployment of that aspect will take longer time. For example, the hook code for capturing events of method calls will be inserted in the program running on the remote machines. The deployment does not finish until inserting the hook code finishes on all the machines. Note that an aspect instance is not copied or migrated to the remote machine even if a remote pointcut is used.

For convenience, DandyJ also provides a *local* pointcut. A pointcut becomes local if the local modifier is attached at the beginning of the pointcut definition. If an aspect includes a local pointcut, an instance of that aspect is replicated on the remote machines specified by the hosts pointcut designator in the local pointcut when that aspect instance is deployed. Then, when a join point happens (*i.e.* the selected kinds of events happens), the advice body associated with that local pointcut is executed on the replication on the remote machine where that join point happens. Note that replicated copies of an aspect instance are not automatically updated when a field of the aspect instance (or one of the replications) gets a new value. A local pointcut is useful to implement an action that has to be performed on the same machine where a join point happens since a remote pointcut causes an action on a different machine.

3.3 Onetime aspect

DandyJ also provides an *onetime aspect*. It is an aspect automatically undeployed after all its advice bodies are ex-

[1]Currently, DandyJ assumes that there is only a single Java process on every host.

Table 2: Pointcut designators of DandyJ

Pointcut designator	Selected join points
hosts(*hostName1, hostName2 ..*)	selects join points on *hostName1, hostName2, ..*
hostName(*varName*)	binds a variable *varName* to the name of the host machine where the selected join point exists.

ecuted. Each advice body in an onetime aspect is executed only once. An aspect becomes an onetime aspect if the onetime modifier is attached at the beginning of the aspect declaration.

An onetime aspect is useful to implement a program for controlling distributed atomic weaving. Such a program often needs to monitor the progress of the programs that the target aspect is woven with. Monitoring the progress can be implemented, for example, by a remote pointcut but it is not necessary any longer after the target aspect is successfully woven. If that monitoring is implemented by an onetime aspect, it is automatically removed and hence the performance overhead due to the monitoring is eliminated.

3.4 Notes on details of DandyJ

DandyJ is an extension to Java and a program written in DandyJ runs on the Java virtual machine (JVM). Most of the syntax and language constructs are borrowed from AspectJ [7]. When an aspect is dynamically woven at runtime, the bytecode of a running base-program is modified and reloaded to accept the aspect.

The language processor of DandyJ consists of a compiler ddjc and a runtime system. The ddjc compiler is a compiler implemented by extending the abc compiler [2], which is one of the AspectJ compilers [1]. The abc compiler we used is based on the JastAdd compiler [5]. ddjc transforms a source program into Java bytecode.

The runtime system performs bytecode instrumentation for weaving a new aspect at runtime. It uses Javassist [4] for this. When an aspect is deployed, the runtime system generates modified bytecode of the running base-program and substitutes it for the original program by using the HotSwap technology of the java.lang.instrument API.

An instance of the runtime system must be run on every machine where programs will be run. Suppose that on one machine we start running a program that requests to weave some aspect. When an aspect is deployed, the runtime system starts modifying the bytecode on that machine so that the bytecode contains *hook code*, which invokes an advice body of the aspect when a thread of control reaches specified execution points such as method calls. The hook code also checks whether or not the aspect is activated. It invokes an advice body only when the aspect is activated. If a deployed aspect includes remote pointcuts, the runtime system requests other machines to modify the bytecode on the machines. The hook code is inserted so that an advice body will be remotely invoked when a thread of control reaches specified execution points on these machines. If a pointcut is a local one, the hook code directly invokes an advice body on a local replication of the aspect. Like a remote pointcut, the hook code invokes an advice body only when the aspect is activated. To check for being activated, the hook code first inquires the (original) instance of the aspect on a different machine from the machine where the hook code is running. The isDeployed method returns true after the hook code is successfully inserted and ready to run on all machines. The runtime system notifies the original instance of the aspect when the installation of the hook code is finished on every machine.

In DandyJ, the deployment is asynchronous but the activation is synchronous. Thus, if a thread activates an aspect by the activate method, it blocks until the hook code on all machines is activated. It hence may take time. The hook code also synchronously invokes an advice body. When a thread reaches execution points specified by pointcuts, it executes the hook code, which then synchronously invokes an advice body. If the advice body is on a remote machine, the thread will block until the invocation of the advice body finishes.

4. EXAMPLES

This section shows two examples of DandyJ programs where distributed atomic weaving is performed. The control programs of the weaving are implemented as separate aspects, which can be reused for weaving a different aspect.

4.1 Weaving a logging aspect with a n-body problem solver

We first show an aspect for atomically weaving a logging aspect with a solver program of the N-body problem shown in Section 2. This aspect deploys a given logging aspect and then activates it on all the machines simultaneously. To implement this simultaneous activation, this aspect has advice that is invoked when a thread of control of the N-body program reaches barrier synchronization on every host. Recall that the N-body program performs barrier synchronization at every tick for exchanging current positions of particles. The invoked advice checks the status of the weaving and activates the logging aspect if possible. To implement this checking, we need a remote pointcut and thus the program for controlling the weaving of the logging aspect must be another aspect.

The aspect for weaving a logging aspect is shown in Figure 1. Since it is an abstract aspect, it is supposed to be inherited from by a concrete aspect written by the users. This concrete aspect must define the sync pointcut so that it will select execution points where barrier synchronization is performed in the N-body program. This design makes the aspect in Figure 1 reusable, without modifying the code at all, for other aspects and target programs other than the N-body problem. We assume that barrier synchronization is implemented by a method call. The definition of the sync pointcut will be something like this:

```
pointcut sync(): call(void Barrier.doSync())
```

A main method first instantiates the concrete aspect and weaves it. The constructor receives a logging aspect (*i.e.* target aspect) and the names of the remote machines. It starts the deployment of the logging aspect. Weaving this

```
public abstract aspect BarrierSync {
  private String[] hostNames;
  private DAspect dAspect;   // the target aspect
  private Map<String, Boolean> map =
                         new Hashtable<String, Boolean>();

  public BarrierSync(DAspect dAspect, String[] hostNames) {
    this.dAspect = dAspect;
    this.hostNames = hostNames;
    for (String host : hostNames)
      map.put(host, false);
    dAspect.deploy();
  }

  // selects the execution points of barrier synchronization
  abstract pointcut sync();

  before(String host): sync() && hostName(host) {
    map.put(host, true);
  }

  after(String host): sync() && hostName(host)
                           && if(dAspect.isDeployed()) {
    synchronized(this) {
      if (!map.containsValue(false)) {
        dAspect.activate();
        this.undeploy();
      }
      map.put(host, false);
    }
  }
}
```

Figure 1: An aspect for weaving a logging aspect

concrete aspect does not have to be atomic. The hook code installed on each remote machine starts invoking an advice body randomly.

This aspect has before advice and after advice. Both use remote pointcuts. They are invoked before/after barrier synchronization specified by the sync pointcut if the logging aspect is already deployed. The before advice records on which machine this aspect is deployed. It may start being invoked by remote machines at different tick. Then, if the logging aspect is already deployed, the after advice checks that all the machines notifies that the deployment finishes before the last barrier synchronization. If the deployment has not finished, the activation is postponed until the next tick. Otherwise, the advice activates the logging aspect on all machines. Note that only the first invocation of the advice activates the logging aspect. The first invocation is determined by using the hash table map. A synchronized statement is necessary for avoiding race conditions. The call to the activate method blocks until the aspect is activated on all machines. After the aspect activates the logging aspect, it is undeployed to avoid performance penalties due to the before and after advice.

The language constructs of DandyJ are suitable for a centralized algorithm. Although there might be a more efficient distributed algorithm for distributed atomic weaving, we designed DandyJ so that programmers can easily write a (usually centralized) control program for weaving.

4.2 Weaving an encryption aspect with a messaging service application

We next weave an encryption aspect with a messaging service application [19]. Suppose that this messaging service application exchanges a plain text between a client and a server. For some security reason, we dynamically weave

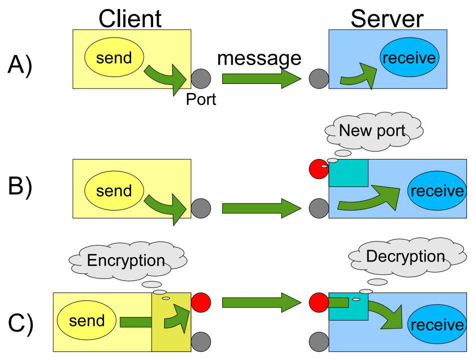

Figure 2: Coordinated activation of encryption and decryption aspects

aspects that implement encryption and decryption of exchanged texts. An encryption aspect is woven on a client machine and a decryption aspect is woven on a server machine.

These aspects must be woven simultaneously as well. Otherwise, a text might be encrypted at the client side but not decrypted at the server side. Thus, the decryption aspect at the server side must be woven first and then the encryption aspect at the client server must be woven. To avoid the "stop the world" approach, in which the application is suspended during weaving, encrypted messages must be sent to a different port and the server program must be able to temporarily receive both plain text and encrypted text (Figure 2).

To guarantee this consistency, weaving encryption/decryption aspects must be controlled by another aspect. Figure 3 shows the skeleton of an messaging service application. Figure 4 shows an aspect that controls the weaving of the encryption/decryption aspects. To be reusable, it is an abstract aspect; to use it, we have to write a concrete aspect inheriting that aspect. The concrete aspect defines two pointcuts: client and server. These pointcuts select the execution points where a text is sent or received, respectively. We assume that they are implemented by a single method call. For example, their definitions will be:

```
pointcut client(): call(void Client.send(String))
pointcut server(): call(void Server.receive(String))
```

A main method instantiates this aspect, which receives four aspects: two for managing a network port and two for client-side and server-side encryption/decryption (Figure 5 and 6). When this aspect is instantiated, it deploys all the four aspects. Then, after the deployment of the server-side aspects finishes, this aspect activates them just before the server sends a new message. The server starts receiving both plain and encrypted messages. On the other hand, this aspect activates the client-side aspects after the encryption aspect at the server side is activated. This ensures atomic weaving. Since this aspect is an one-time aspect, it is automatically unwoven after the encryption/decryption aspects are activated. The four aspects in Figure 5 and 6 use local pointcuts. Since the advice associated with a local pointcut is locally executed, it efficiently works without communicating with a remote machine.

```
public class Client {
  ...
  private void send(msg) {
    ... sendMessage(msg); ...
  }
}

public class Server { ..
  private void run() {
    while (true) {
      ... accept(); ...
    }
  }
  private void accept() {
    Socket socket = getSocket();
    ... receive(input); ...
  }
  ...
}
```

Figure 3: A messaging service application

```
public abstract onetime dynamic aspect ServerClient {
  private DAspect changePort, openPort, client, server;

  public ServerClient(DAspect changePort, DAspect openPort,
                      DAspect client, DAspect server) {
    this.changePort = changePort;
    this.openPort = openPort;
    this.client = client;
    this.server = server;
    openPort.deploy();
    changePort.deploy();
    client.deploy();
    server.deploy();
  }

  abstract pointcut client();
  abstract pointcut server();

  before(): server() &&
            if(server.isDeployed() && openPort.isDeployed()) {
    openPort.activate();
    server.activate();
  }

  before(): client() &&
            if(server.isActivated() && client.isDeployed()
               && changePort.isDeployed()) {
    changePort.activate();
    client.activate();
  }
}
```

Figure 4: An aspect for weaving encryption/decryption aspects

```
public dynamic aspect ChangePort {
  local void around():
        execution (void Client.connect())
        && within(Client) && hosts("client.machine.org") {
    // change the port for sending messages
  }
}

public onetime dynamic aspect OpenPort {
  local java.net.Socket around():
        call(java.net.Socket ServerSocket.accept())
        && within(Server) && hosts("server.machine.org") {
    // create a new thread that opens a new port and waits messages
  }
}
```

Figure 5: Aspects for managing a network port

```
public dynamic aspect Encrypter {
  local void around(String s):
        call(void Client.send(String))
        && within(Client) && hosts("client.machine.org") && args(s) {
    // encrypt the message
    proceed(s);
  }
}

public dynamic aspect Decrypter {
  local void around(String s):
        call(void Server.receive(String))
        && within(Server) && hosts("server.machine.org") && args(s) {
    if (socket.getLocalPort() == /* the new port number */) {
      // decrypt the message
      proceed(s);
    }
    else
      proceed(s);
  }
}
```

Figure 6: Encryption and decryption aspects

5. EXPERIMENT

Atomicity of distributed weaving is a crucial issue in large distributed environments. To see this is a real problem and our solution resolves this, we performed a simple experiment using encryption/decryption aspects for a message service application mentioned in Section 4.2.

For this experiment, we used the InTrigger grid computing platform [6], which is a collection of computer clusters distributed in Japan. We chose two machines in two clusters 700 km away from each other. The network latency was 27 msec. between the two machines. The client machine in Chiba had a 2.13GHz Core2Duo processor and 4 GB memory. The server machine in Hakodate had dual 2.33 GHz Xeon E5410 processors and 16 GB memory. We used Java 1.6.0 on both machines.

We tested four scenarios:

A. deploy and activate aspects without any consideration of atomicity,

B. deploy and activate aspects by our DandyJ aspect in Figure 4,

C. the same as B but the aspect is not one-time (hence the aspect is not unwoven), and

D. stop the application, weave aspects, and then restart the application (naive approach).

In every scenario, the client machine connects to the server machine and sends a message every millisecond. The client repeats this 1000 times. We measured the elapsed time since the client opens connection till the connection is closed. The aspects are woven at around the 100th message sent. The aspect that controls the weaving of encryption/decryption aspects was run on another machine in the same cluster where the client machine in Chiba was running.

Figure 7 and Table. 3 present the results of our experiment. In every scenario, a spike due to weaving is observed in Figure 7. The response time of Scenario A is the best but 100 messages were not successfully encrypted/decrypted since atomicity was not considered. In the other scenarios, no such failures were observed. Among the scenarios considering atomicity, DandyJ with a onetime aspect (Scenario B) showed the best performance. Scenario C was the worst

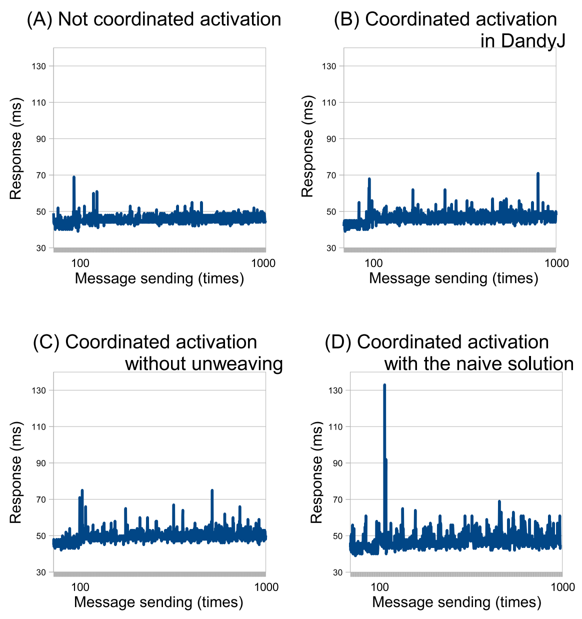

Figure 7: The response time of a messaging service application

since the controlling aspect was not unwoven and thus it was a performance penalty after weaving. Scenario D was faster than C but not than B. Since the application program was suspended while atomic weaving, the highest peak (133 msec.) was observed around 100th message and this fact slowed down the average performance.

The experiment showed that DandyJ allows us to implement a complex but efficient algorithm for distributed atomic weaving by a separated (and hence reusable) aspect. To see that the failure of encryption/decryption in Scenario A is not due to the weaving mechanism of DandyJ, we also performed the experiment with another distributed dynamic AOP language CaesarJ [8]. Since encryption/decryption aspects are statically deployed in advance in CaesarJ, the cost of deployment is zero and only the cost of activation is paid at runtime. Thus the variance of the length of weaving time is small but, however, we observed that one message was still not successfully encrypted/decrypted in the case of Scenario A using CaesarJ.

6. RELATED WORK

Dynamic AOP languages have been actively studied. PROSE [14, 13], Steamloom [3], JAsCo [16], Wool [15], and HotWave [21] are examples. Several dynamic AOP languages even support distributed computing; they are JAC [12], CaesarJ

Table 3: The overview of the response time (msec.)

Scenario	average	max.
A. no consideration of atomicity	45.4	69
B. DandyJ	46.6	71
C. DandyJ without undeployment	49.6	75
D. stop, weave, and restart (naive approach)	47.5	133

[8], DJAsCo [9], ReflexD [18], DyReS [19], a recent version of PROSE [10], and so forth.

However, their supports of distributed weaving is limited against DandyJ, which provides dedicated language constructs. First of all, most of them do not provide a weaving mechanism that consists of two phases: deployment and activation. As we showed, this is significant to implement an efficient weaving algorithm. For example, JAC, which is a Java-based framework for dynamic AOP, does not provide this mechanism. It does not provide remote pointcut, either.

CaesarJ allows a first-class aspect like DandyJ but its weaving mechanism is not a two-phase one. Furthermore, an aspect must be statically deployed in the code on a remote machine. It is not possible to transfer aspect code to a remote machine and dynamically deploy it there. It is only possible to dynamically activate an aspect that is already deployed. Indeed, the operator deploy of CaesarJ performs the activation of the terminology of this paper. Moreover, CaesarJ does not provide remote pointcut.

DJAsCo is a prototype implementation of a distributed AOP language AWED. It is an extension of JAsCo. Although it supports remote pointcut like DandyJ, its weaving mechanism is not a two-phase one. It only allows programmers to specify which aspect is woven first and which is the next. This feature only works for resolving dependency among aspects. ReflexD is another platform for distributed AOP in Java. It is an extension of Reflex [17]. Like DJAsCo, ReflexD supports remote pointcut but not a two-phase weaving mechanism.

DyReS is another Java-based framework for distributed dynamic AOP. Unlike DandyJ, it does not support a two-phase weaving mechanism or remote pointcut. Moreover, since programmers must use XML for describing how to control weaving, the expressive power is weaker than in Java. Although an aspect is a first-class object in XML, being first-class is not equivalent to that an aspect in DandyJ is first-class. In XML, first-class objects are only aspects whereas in DandyJ they are not only aspects but also other Java objects and primitive-type data. Finally, an XML description to control weaving in DyReS cannot be installed on demand during runtime (according to our understanding). It must be statically installed on all machines before a target program starts. Hence it does not enable truly dynamic weaving. In DandyJ, an aspect to control weaving can be also dynamically woven on target machines.

As far as we know, Lasagne [20] is only the framework for distributed dynamic AOP that provides a two-phase weaving mechanism. However, unlike DandyJ, it does not provide programming supports by dedicated language constructs since it is a framework for Java programming. A uniqueness of DandyJ is that DandyJ provides language constructs for programmers to easily implement a reusable aspect to control atomic weaving. Although similar effects might be achievable in Lasagne by using programming idioms and conventions, we believe that distributed atomic weaving should be implemented directly by an aspect since it is a crosscutting concern.

7. CONCLUSION

This paper illustrates that distributed dynamic weaving is a crosscutting concern, which should be implemented by another aspect. Then it proposes our new language *DandyJ*. It provides a useful set of language constructs: first-class aspects with two-phase weaving, remote pointcuts, and one-time aspects. Although these language constructs are not very new, DandyJ is the first language providing all the three constructs as far as we know. In DandyJ, a control program for atomically performing distributed dynamic weaving can be implemented as a reusable aspect. At the lexical level, it is not tightly coupled with its target aspect that will be dynamically woven under the control of that aspect. It is neither coupled with the target base-program where the aspects are woven. Therefore, an aspect for controlling distributed dynamic weaving can be reusable for weaving other target aspects.

8. REFERENCES

[1] The AspectJ Project. http://www.eclipse.org/aspectj/.

[2] Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Jennifer Lhoták, Ondřej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. abc: An extensible AspectJ compiler. In *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*, pages 87–98. ACM, 2005.

[3] Christoph Bockisch, Michael Haupt, Mira Mezini, and Klaus Ostermann. Virtual Machine Support for Dynamic Join Points. In *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development*, pages 83–92. ACM, 2004.

[4] Shigeru Chiba. Load-Time Structural Reflection in Java. In *ECOOP '00: Proceedings of the 14th European Conference on Object-Oriented Programming*, pages 313–336. Springer-Verlag, 2000.

[5] Torbjörn Ekman and Görel Hedin. The Jastadd system — modular extensible compiler construction. *Sci. Comput. Program.*, 69(1-3):14–26, 2007.

[6] InTrigger. Intrigger platform. http://www.intrigger.jp/wiki/index.php/InTrigger.

[7] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An Overview of AspectJ. In *ECOOP '01: Proceedings of the 15th European Conference on Object-Oriented Programming, LNCS2027*, pages 327–353. Springer-Verlag, 2001.

[8] Mira Mezini and Klaus Ostermann. Conquering aspects with Caesar. In *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 90–99. ACM, 2003.

[9] Luis Daniel Benavides Navarro, Mario Südholt, Wim Vanderperren, Bruno De Fraine, and Davy Suvée. Explicitly distributed AOP using AWED. In *AOSD '06: Proceedings of the 5th international conference on Aspect-oriented software development*, pages 51–62. ACM, 2006.

[10] Angela Nicoara, Gustavo Alonso, and Timothy Roscoe. Controlled, Systematic, and Efficient Code Replacement for Running Java Programs. In *Eurosys '08: Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*, pages 233–246. ACM, 2008.

[11] Muga Nishizawa, Shigeru Chiba, and Michiaki Tatsubori. Remote Pointcut: A Language Construct for Distributed AOP. In *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development*, pages 7–15. ACM, 2004.

[12] Renaud Pawlak, Lionel Seinturier, Laurence Duchien, and Gerard Florin. JAC: A Flexible Solution for Aspect-Oriented Programming in Java. In *REFLECTION '01: Proceedings of the Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns*, pages 1–24. Springer-Verlag, 2001.

[13] Andrei Popovici, Gustavo Alonso, and Thomas Gross. Just-In-Time Aspects: Efficient Dynamic Weaving for Java. In *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 100–109. ACM, 2003.

[14] Andrei Popovici, Thomas Gross, and Gustavo Alonso. Dynamic Weaving for Aspect-Oriented Programming. In *AOSD '02: Proceedings of the 1st international conference on Aspect-oriented software development*, pages 141–147. ACM, 2002.

[15] Yoshiki Sato, Shigeru Chiba, and Michiaki Tatsubori. A Selective, Just-in-Time Aspect Weaver. In *GPCE '03: Proceedings of the 2nd international conference on Generative programming and component engineering*, pages 189–208. Springer-Verlag, 2003.

[16] Davy Suvée, Wim Vanderperren, and Viviane Jonckers. JAsCo: an Aspect-Oriented approach tailored for Component Based Software Development. In *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 21–29. ACM, 2003.

[17] Eric Tanter. Aspects of Composition in the Reflex AOP Kernel. In *In: Proceedings of the 5th International Symposium on Software Composition (SC 2006). LNCS*, pages 99–114. Springer, 2006.

[18] Eric Tanter and Rodolfo Toledo. A Versatile Kernel for Distributed AOP. In *In Proceedings of the IFIP International Conference on Distributed Applications and Interoperable Systems (DAIS 2006*, pages 316–331. Springer-Verlag, 2006.

[19] Eddy Truyen, Nico Janssens, Frans Sanen, and Wouter Joosen. Support for Distributed Adaptations in Aspect-Oriented Middleware. In *AOSD '08: Proceedings of the 7th international conference on Aspect-oriented software development*, pages 120–131. ACM, 2008.

[20] Eddy Truyen and Wouter Joosen. Run-Time and Atomic Weaving of Distributed Aspects. In *Transactions on Aspect-Oriented Software Development II*, pages 147–181. Springer, 2006.

[21] Alex Villazón, Walter Binder, Danilo Ansaloni, and Philippe Moret. Advanced Runtime Adaptation for Java. In *GPCE '09: Proceedings of the eighth international conference on Generative programming and component engineering*, pages 85–94. ACM, 2009.