分散アスペクトの一貫性の保持と停止時間の短縮を考慮した動的織り込み手法

Coordinated Distributed Dynamic Weaving with Minimum Pause Time

by

森田 悟史

Satoshi Morita

08M37315

February 5, 2010

A Master's Thesis Submitted to
Department of Mathematical and Computing Sciences
Graduate School of Information Science and Engineering
Tokyo Institute of Technology

In Partial Fulfillment of the Requirements
for the Degree of Master of Science.

Supervisor: Shigeru Chiba

Copyright © 2010 by Satoshi Morita. All Rights Reserverd.

Abstract

分散動的アスペクト指向言語では、アスペクトの一貫性を保つために、織り込みのタイミングを制御することが重要になる。なぜなら、制御をしないと、ネットワークレイテンシやコンピュータのワークロードの違いにより、ホストごとに織り込みが終了するタイミングが違ってしまい、織り込み中にプログラムが予期しない振る舞いをする可能性があるからである。特に、異なるホスト間でアスペクト同士が協調して動く場合、この問題が発生すると考えられる。我々は、実験を通して、グリッドコンピューティングシステムなどの巨大な分散環境では、これが実際に起こる問題であることを確認した。織り込み制御の素朴な方法として、織り込みの間はプログラムを停止させるという方法が考えられるが、これはプログラムのパフォーマンスを低下させるという問題を生み出してしまう。

これらの問題を解決するために、本論文は新しい織り込み機構である、*two phase weaving* を提案する。これは、織り込み (weaving) を *deployment* と *activation* の二つのステップに分割する。deployment はアスペクトを動作可能な状態にする非同期的なオペレータである。そして、activation は deployment が終わったアスペクトを動かし始めるオペレータである。ユーザは、この二つのオペレータを用いると、activation のタイミングだけを制御することで、織り込みの時のプログラムの停止時間を一貫性を保ったまま大きく減らすことができる。また、two phase weaving を用いた織り込み制御プログラムは複雑になってしまうことが多いため、本論文は新たな分散動的アスペクト指向言語である *DnadyJ* を提案する。この言語は、織り込み制御プログラムの実装をサポートするために、*dynamic aspect*, *remote pointcut*, そして *onetime aspect* を提供している。これらの機能を組み合わせて用いることで、シンプルでモジュール化された織り込み制御プログラムを実装することができる。そして最後に、two phase weaving と DandyJ を用いた実験を行い、本機構が有効であることを確認した。

Distributed aspect-oriented programming (AOP) languages allow users to modularize a distributed crosscutting concern and weave it on multiple hosts. Dynamic distributed AOP languages allow the weaving dynamically; it can weave an aspect during runtime even if the aspect had not been written yet when the rest of the program started running on multiple osts. Those languages can be used for altering program behavior without restarting the program.

In distributed environments such as grid computing systems, the users have to carefully control when they start and when they finish weaving an aspect on each host. Without such a control, an aspect may be woven on

only part of the hosts and cause inconsistent effects. To address this issue, this paper proposes two phase weaving and our new dynamic aspect-oriented programming language named *DandyJ*. The two phase weaving allows programmers to minimize the pause time of an application program during coordinated distributed weaving. DandyJ helps programmers implement such coordinated weaving by providing new language constructs.

Acknowledgments

I would like to express my deep gratitude to my supervisor, Shigeru Chiba. He gave me closely teaching with his assiduous guidance and support. Our numerous discussions and his constructive comments have greatly improved my work.

I greatly thank Michihiro Horie. He spent a large amount of time for me and gave me various important advice for my work.

Finally, I greatly thank my colleagues of the Chiba Shigeru Group in Tokyo Institute of Technology.

Contents

1 Introduction **1**
- 1.1 Motivating problem . . . 2
- 1.2 Solution by this thesis . . . 3
- 1.3 Our contributions . . . 4
- 1.4 The structure of this thesis . . . 5

2 Dynamic Distributed Weaving **6**
- 2.1 Diversity of the duration for weaving . . . 6
- 2.2 An experiment to show the need for coordinated weaving . . . 9
- 2.3 Problem of a naive solution . . . 10

3 Two Phase Weaving **13**
- 3.1 Overview . . . 13
- 3.2 Examples of the Coordinated Activation . . . 14
  - 3.2.1 An Encryption Aspect for Messaging Service Applications . . . 14
  - 3.2.2 A Visualization Aspect for N-Body Problem . . . 15
- 3.3 Discussion . . . 19

4 DandyJ **21**
- 4.1 Remote pointcut . . . 22
- 4.2 Dynamic aspect . . . 22
  - 4.2.1 Monitoring aspect . . . 24
- 4.3 Onetime aspect . . . 24
- 4.4 Examples in DandyJ . . . 26
  - 4.4.1 An Encryption Aspect in DandyJ . . . 26
  - 4.4.2 An Visualization Aspect in DandyJ . . . 27
- 4.5 Implementation Notes . . . 27
  - 4.5.1 DandyJ compiler . . . 27

4.5.2 DandyJ runtime system 31

5 Experiment 34

5.1 Overview of this measurement 34

5.2 Measured overhed of dynamic weavings 34

6 Related Work 38

6.1 CaesarJ 38

6.2 AWED 41

6.3 DyReS 41

6.4 JAC 46

6.5 ReflexD 46

7 Conclusion and Future Work 47

7.1 Conclusion 47

7.2 Future Work 48

List of Figures

2.1 A Messaging Service Application 7
2.2 A Messaging Service Application and Encryption/Decryption Aspects 8
2.3 The encryption aspect is woven before the decryption aspect 8
2.4 A grid computing environment InTrigger 11
2.5 A naive solution for the weaving of the messaging service application 12

3.1 Two phase weaving of the encryption and decryption aspects 16
3.2 The process of the coordinated activation of the encryption and decryption aspects 17
3.3 The computation of the N-body problem in distributed environments 17
3.4 The computation loop of the N-body problem 18
3.5 An Visualization of the N-body problem 18
3.6 Activation failure of the visualization aspect 19

4.1 The monitoring aspect for the messaging service application . 28
4.2 The aspect for opening and changing port 29
4.3 The encryption and decryption aspect 29
4.4 The monitoring aspect for the barrier synchronization 30
4.5 The process of the compilation by the ddjc 32

5.1 (A) without the coordination 35
5.2 (B) with the coordination aspects in DandyJ 36
5.3 (C) with the coordination without undeployment 36
5.4 (D) with the naive coordination solution 37

6.1 An Aspect in CaesarJ 39
6.2 An Aspect in CaesarJ 2 40

6.3 An Aspect in DJAsCo 42
6.4 An Aspect in DJAsCo 2 43
6.5 An Aspect for MSA in DyReS 44
6.6 A Script for coordination in DyReS 45
6.7 A program for weaving in DyReS 45

List of Tables

4.1 The pointcut designators provided by DandyJ 22
4.2 The methods available on instances of dynamic aspects 25

5.1 The overview of the response time 37

Chapter 1

Introduction

Modularizing software systems is one of significant demands in software industry. Developers should decompose software into a number of small independent programs so that they can develop high quality software, which is easy to understand, modify, and maintain. To do this, several paradigms have been proposed. One of them is Object-Oriented Programming (OOP). It is a widely used paradigm for developing software systems because the object modeling, which is a central concept of this paradigm, provides a better fit with real domain problems. Therefore, a large number of OOP based systems, languages, and applications have been developed so far.

To separate *crosscutting concerns* in software, Aspect-Oriented Programming (AOP) have been proposed [4, 11, 17, 32]. Crosscutting concerns, such as logging and cut across the boundary between modules, decrease maintainability and understandability of software. Although widely spreading programming language and techniques such as OOP and the design patterns does not elegantly modularize such concerns, AOP allows the users to separate the crosscutting concerns as *aspects*. The users can develop maintainable and understandable systems.

Furthermore, modularizing crosscutting concerns in distributed systems is also one of significant demands in software industry. For example, Web caching, transactions, unit testing of distributed applications, and security have been shown to be subject to serious crosscutting problems. To separate these crosscutting concerns in distributed systems, distributed AOP languages [16, 14, 12] have been proposed. The users can weave aspects on multiple hosts.

On the other hand, to accommodate the increasing need for dynamically adaptive software, dynamic aspect-oriented languages are proposed

[22, 21, 5, 26, 24, 35]. Unlike ordinary AOP languages, Dynamic AOP allows aspects to be changed and code to be rewoven in running systems. Dynamic AOP languages have been used for adding caching at runtime, debugging, and fixing bugs in a running server. Dynamic AOP is often supported in dynamic languages [8], such as Lisp, Smalltalk, Ruby, and so on. This is because, these dynamic languages can easily manipulate the codes at runtime. Although it is more difficult and challenging to offer dynamic AOP for Java-like languages, many Java-based dynamic AOP languages support dynamic weaving by using debugging interface, HotSwap technology, and modified Java Virtual Machine.

Dynamic distributed AOP languages [20, 15, 14, 31] allow aspects to be dynamically woven into distributed systems. They can weave an aspect during runtime even if the aspect had not been written yet when the rest of the program started running on multiple hosts. These languages can be used for altering program behavior without restarting the program. A number of dynamic distributed crosscutting problems have been shown such as caching, unit testing of distributed applications, and message compression and fragmentation functionalities [34].

1.1 Motivating problem

Dynamic weaving in distributed environments is not a simple task. Even if aspect weaving starts simultaneously on all hosts, it does not finish at the same time on all the hosts. The time for completing the weaving varies host by host due to network latency and work load. In other words, there can be notable diversity of the duration for weaving. The users, therefore, have to control when they start and finish weaving an aspect, in particular, if the hosts where the aspect is woven are cooperatively working. Without such a control, unexpected behavior would be seen after the weaving.

For example, suppose we have a messaging service application between a server and clients. Because the message is a simple plain text, the users will try to dynamically add the aspects of encryption and decryption to the messaging service program; the encryption aspect on the client host and the decryption aspect on the server host. One of the aspects will not be woven at a moment because of the diversity of weaving time. Unfortunately, if the encryption aspect on the client host is woven before the decryption aspect on the server host is finished being woven, the encrypted message that is sent from the client will not be correctly decrypted on the server.

The diversity of weaving time is made by the differences on the work

load of each host, the network speed, and so on. There is a geographically separated distributed environment. For example, the Grid computing environment consists of clusters distributed in a large area. Suppose if the application runs in a large scale grid computing environment, the influence of the network latency will be increased and nonnegligible. Hence, unless the users carefully control when aspects start running, the aspect will start on each host at different time. Furthermore, if the users use a Java-based dynamic AOP languages, the weaving will take time. This is because some implementation of Java-based AOP languages need code transformation. The code transformation contain reflective computing and this is relatively slow. Therefore, the aspect weaving will increase the diversity of weaving time especially when the base program is implemented by these languages because of differences on the work load.

To show the need for controlling the dynamic weaving, we carried out an experiment to test whether or not the encryption aspect is consistently woven before the decryption aspect is woven. Our experiment revealed that encrypted messages were sent by the client application before the server application was woven with the decryption aspect in large scale grid computing systems. Thus, those encrypted messages were not decrypted by the server application. This is because of network latency and the difference of time taken for the weaving. A naive approach to solve this problem is to stop the program on all the hosts while weaving an aspect and restart it after the weaving finishes. However, this is not desirable with respect to execution performance.

1.2 Solution by this thesis

To address the problems, this paper proposes a new weaving mechanism called *two phase weaving*. It separates weaving into two steps; *deployment* and *activation*. The deployment makes a given aspect ready to run and it is an asynchronous operation concurrently performed. Thus, the users do not have to wait till the deployment finishes. This reduce the diversity of time taken to make aspect ready to run. The activation starts the aspect running if it has been already deployed. The users have only to control when to activate the aspects for coordinated weaving to minimize the pause time of the applications. The users can exploit the two operations for minimizing the pause time while an aspect is woven in a distributed environment.

Since it is a complex task to write a program that controls distributed weaving by the two phase weaving, this paper also proposes our new dis-

tributed AOP language called *DandyJ*. It is an extension of Java. To support writing a program for controlling distributed weaving, DandyJ provides three features: *dynamic aspect*, *remote pointcut*, and *onetime aspect*. The first two are borrowed from other distributed AOP languages but the last one is unique to DandyJ. These features help the users write a simple and modular control program with a short pause time.

To evaluate the performance overhead of two phase weaving and DandyJ, this paper measured the cost of deployment and activation, in which the two phase weaving and DandyJ showed good performance without causing any inconsistent weaving among hosts. For this experiment, we used an aspect for encrypting a network message sent between a client and a server.

1.3 Our contributions

Our contributions in this paper are the followings.

Presenting significance to keep consistency

We present that, in large distributed environments, controlling when an aspect starts deployment and when it finishes is significant to keep consistency among hosts. To show the need for controlling the dynamic weaving, we carried out an experiment.

Proposing a new weaving mechanism and a new language

We propose a new weaving mechanism called *two phase weaving*, which separates weaving into two step; *deployment* and *activation*. In addition, we design and implement a new dynamic distributed AOP language *DandyJ*, which based on the two phase weaving. We present some example programs in DandyJ.

Evaluating the two phase weaving

We evaluate the two phase weaving by an experiment in a real grid computing environment. For this experiment, we used encrypt and decrypt aspects for a messaging service application.

1.4 The structure of this thesis

From the nest chapter, we presented background, the two phase weaving, and new dynamic distributed AOP language DandyJ. The structure of the rest of this thesis is as follows.

Chapter 2: Dynamic Distributed Weaving

In this chapter, we first illustrates needs of coordination for dynamic weaving. Then, we show the need for controlling the dynamic weaving by an experiment. Finally, we explain a problem of naive solution for the coordination.

Chapter 3: Two Phase Weaving

To address the problems described in the previous chapter, we propose the two phase weaving.

Chapter 4: DandyJ

In this chapter, we first presents new dynamic distributed aspect-oriented language DandyJ, which make it easy to implement a program for the coordination of dynamic weaving. Then, we show sample programs in DandyJ.

Chapter 5: Experiment

To evaluate the performance overhead of DandyJ, we measured the cost of deployment and activation. We use the encryption/decryption aspects for this experiment.

Chapter 6: Related Work

In this chapter, we mention related works.

Chapter 7: Conclusion and Future Work

Finally, we conclude this thesis in this chapter. Moreover, we present future work.

Chapter 2

Dynamic Distributed Weaving

2.1 Diversity of the duration for weaving

Dynamic weaving in distributed environments is not a simple task. The aspects might not be woven on every host at the same time even if the weaving starts synchronously. In other words, there can be notable diversity of the duration for weaving. This diversity of weaving time is problematic when users weave aspects that cooperatively work with each other. If this diversity makes serious inconsistency, the weaving process must be carefully coordinated. For example, suppose we have a messaging service application between a server and clients (Fig. 2.1, Fig. 2.2) [34]. The message is a simple plain text. Therefore, for security reason, the users will try to dynamically add the aspects of encryption and decryption to the messaging service program; the encryption aspect on the client host and the decryption aspect on the server host. The two aspects may not be woven at the same time. One of the aspects will not be woven at a moment because of the diversity of weaving time. Unfortunately, as in Fig. 2.3, if the encryption aspect on the client host is woven before the decryption aspect on the server host is finished being woven, this can introduce a fatal bug. The encrypted message that is sent from the client will not be correctly decrypted on the server.

The diversity of weaving time is made by the differences on the work load of each host, the network speed, and so on. Since target programs are scattered on several hosts, the users have to dynamically weave aspects across networks. Even if users could start the dynamic weaving on every host at the same time, it would be difficult that each weaving process si-

```
1  public class Client {
2    ...
3    private void send (msg) {
4      ..
5      sendMessage (msg);
6      ..
7    }
8    ..
9  }
10
11 public class Server {
12   ..
13   private void run() {
14     while (true) {
15       ..
16       accept ();
17       ..
18   }}
19   private void accept() {
20     Socket socket = getSocket();
21     ..
22     receive (input);
23     ..
24   }
25   ..
26 }}
```

Figure 2.1: A Messaging Service Application

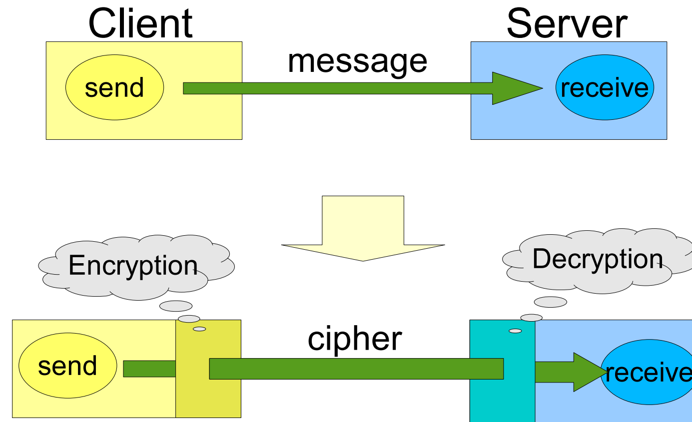


Figure 2.2: A Messaging Service Application and Encryption/Decryption Aspects

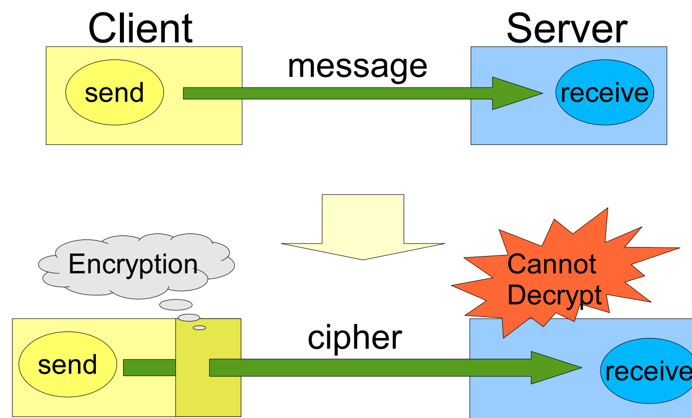


Figure 2.3: The encryption aspect is woven before the decryption aspect

multaneously finishes on every host. Moreover, there is a geographically separated distributed environment. For example, the *Grid computing environment* consists of clusters distributed in a large area. A grid computing system tends to be more loosely coupled, heterogeneous, and geographically dispersed. There is already a case study that uses ReflexD [31] on a grid computing environment [33]. If an application runs in a cluster allocated in one place, the network latency will be low. However, suppose if the application runs in a large scale grid computing environment that is spreads over a country, the influence of the network latency will be increased and nonnegligible. Hence, unless the users carefully control when aspects start running, the aspect will start on each host at different time.

Furthermore, if the users use a Java-based dynamic AOP languages, the weaving will take time. This is because some implementation of Java-based AOP languages need code transformation, for example, by using the HotSwap technology. The language using the HotSwap technology can substitute a modified class in a running application. The weaving is implemented by the code transformation and the HotSwap. The code transformation and the HotSwap contain reflective computing and this is relatively slow. Therefore, the aspect weaving will increase the diversity of weaving time especially when the base program is implemented by these languages because of differences on the work load.

2.2 An experiment to show the need for coordinated weaving

To show the need for controlling the dynamic weaving, we carried out an experiment to test whether or not the encryption aspect is consistently woven before the decryption aspect is woven. In this experiment, the client application tries to connect the server application and then send a message every one millisecond. We carried out this experiment on the InTrigger, which is a distributed platform of information technology research for the Information Explosion Era project [9]. As shown in Fig. 2.4, it is a large grid computing system spread all over Japan. It has over 300 nodes and over 900 CPU cores in 15 sites. We selected the chiba and mirai clusters for this experiment. A node in the chiba cluster sent message to a node in the mirai cluster. The mirai cluster is 700 km far from the chiba cluster. The network latency is 27 ms. We implemented a message service application and encryption/decryption aspects on top of our simple dynamic distributed AOP system using the HotSwap technology, which similar to AOP languages

like DJAsCo [14]. We ran a program on chiba node, which simultaneously requests the weavers on the client node in the chiba cluster and the server node in the mirai cluster to weave the aspects. This program never controls consistency of the weaving.

Our experiment revealed that more than 10 encrypted messages were sent by the client application before the server application was woven with the decryption aspect. Thus, those encrypted messages were not decrypted by the server application. This is because of network latency and the difference of time taken for the code transformation and the HotSwap. Furthermore, as an extreme case, we implemented the message service application and its aspects in CaesarJ [13, 2]. In this experiment, the overhead of the code transformation and the HotSwap were zero because the aspects were compiled, statically linked and loaded on each hosts before the application starts. At the beginning, the aspects do not work. They are enabled after the deploy call. Note that the deploy does not involve code transformation or the HotSwap. Therefore, the diversity of weaving time would be minimum. We wrote a program that connects to the client and server applications to deploy each aspects. We ran the program on chiba node different from the client application node. In this case, only one message was not properly decrypted because of network latency. These results show that the code transformation and the HotSwap take a long time. Furthermore, it is also shown that even if a weaving does not involve code transformation and the HotSwap, the users must give consideration to controlling the weaving.

2.3 Problem of a naive solution

A naive solution is to suspend the programs on all the hosts during the weaving process. However, this approach introduces disruption of the messaging service. In this case, for example, as in Fig. 2.5, the users will weave aspects as follows.

1. The users suspend the client application to suspend sending messages.
2. They wait until the server application finishes receiving messages.
3. They weave aspects on the client and server applications
4. They resume the applications

It is problematic that the application suspend until the four steps above finish. In particular, this is more serious if the code transformation and the HotSwap take a longer time and hence, the total pause time is longer.

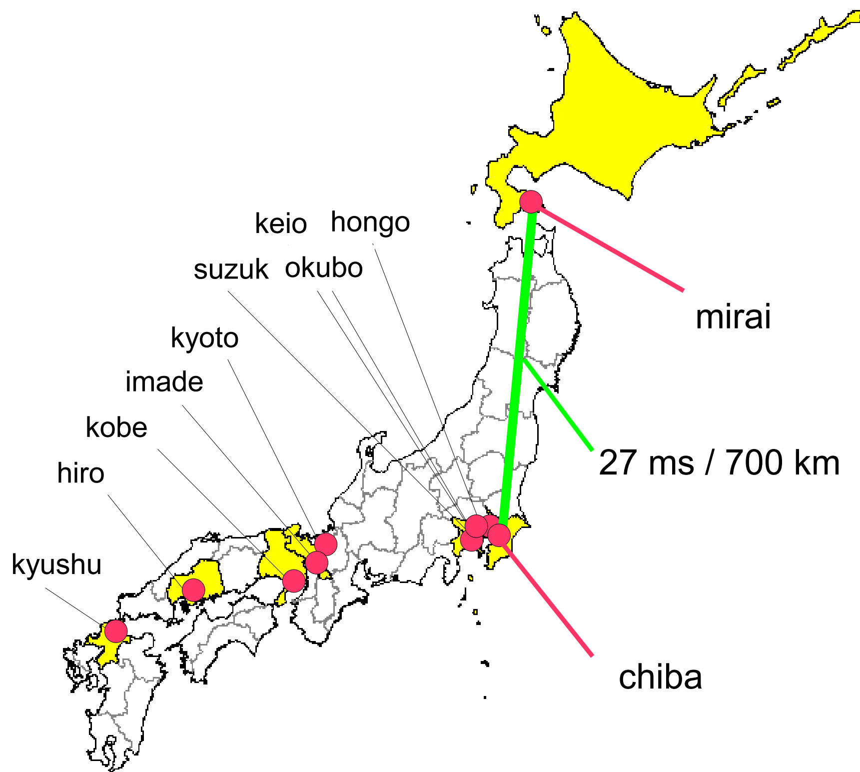


Figure 2.4: A grid computing environment InTrigger

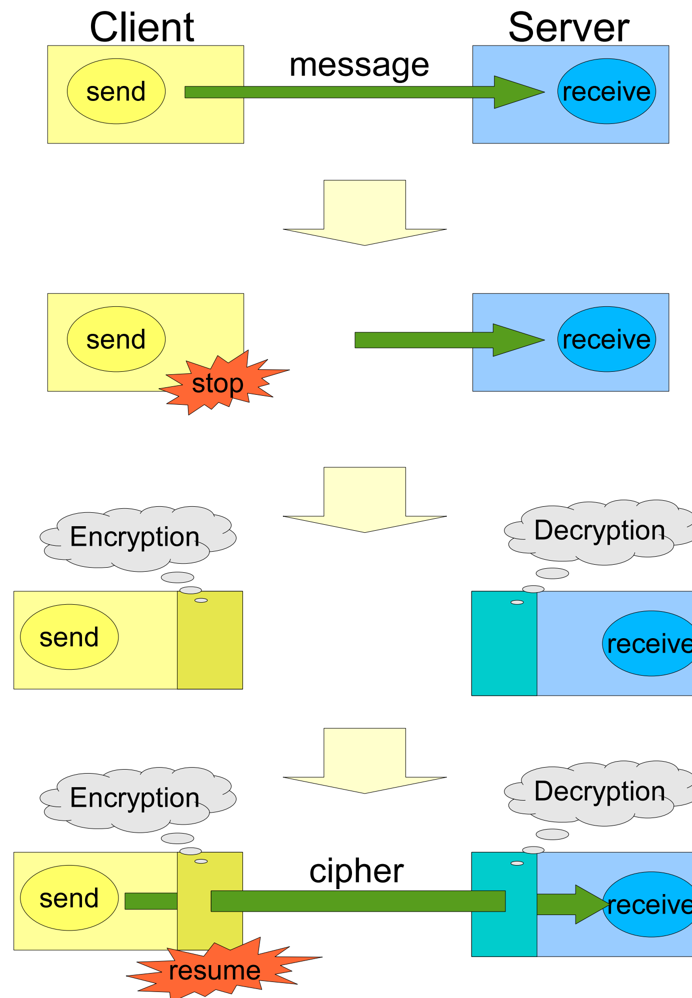


Figure 2.5: A naive solution for the weaving of the messaging service application

Chapter 3

Two Phase Weaving

To address the problem in the previous section, we propose *two phase weaving*. It separates weaving into two steps: *deployment* and *activation*. The deployment performs code transformation and substitutes the bytecode of some classes by the HotSwap. The deployment is executed asynchronously and thus, the users do not have to wait till the deployment finishes. This reduce the diversity of time taken to perform the code transformation and the HotSwap on target applications. The aspects that have been deployed can be activated. Since the code transformation and hotswapping are finished, the activation is instant. The users have only to control when to activate the aspects for coordinated weaving to minimize the pause time of the applications. Thereby, the users can easily write a program that controls weaving.

3.1 Overview

Two phase weaving provides three primitive operations: *deploy*, *isDeployed* and *activate*. The users use the operations to weave aspects coordinately when writing programs. They run the programs on a host different from the weaving targets. The program controls the weaving on remote hosts. Moreover, they can write an aspect that execute the operations to weave another aspect. They can deploy that aspect on remote hosts and activate it for weaving another aspect on the remote hosts.

When the `deploy` is called, a compiled aspect is deployed on its target host and it is embedded in the running application (by code transformation and/or other techniques) to be ready to start running. When users use a Java-based language, this step may take a long time, and the duration of

this step may differ among hosts because of network speed and the workload of each host. The deployment step is performed asynchronously. Therefore, the deploy will not stop the programs. Since it is difficult to know when the deployment is finished, the isDeployed is provided. The isDeployed returns true only if the aspect has been deployed and it is ready to run. The users use this operation to check whether the aspect can be activated. The activate is synchronous but instant. After the aspect is activated, it is executed whenever a thread of control reaches a join point selected by the advice.

By using these operations, the users can write programs that control the activation of aspects in a coordinated way. If aspects must be activated in a appropriate order, the users can sequentially activate the aspects in that order. If aspects must be activated at a fixed moment (for example, before the execution of a method), they can write an advice in another aspect that intercept a method and activate the target aspect.

3.2 Examples of the Coordinated Activation

In this section, we show two examples of two phase weaving. Users weave aspects using two phase weaving to minimize the pause time of programs during the weaving.

3.2.1 An Encryption Aspect for Messaging Service Applications

In Section 2, we presented the example of the messaging service application. When the aspect on the client host is activated earlier, the encrypted message that is sent from the client will not be correctly decrypted on the server. To avoid this problem, these aspects must be activated simultaneously on the server and client host. More concretely, as in Fig. 3.1, the operators are used as the following steps:

1. The users deploy the encryption aspect on the client host and decryption aspect on the server. The applications running on each host do not need to be stopped.

2. They use the isDeployed to check whether these aspects are ready to be activated. The aspects must be activated just after finishing deployment of them on each host.

3. They activate the decryption aspect on the server host, and then, the encryption aspect on the client host.

Additionally, further control will be needed because the aspect on the server must receive both plain and encrypted messages during the weaving process. Unlike the naive solution in Section 2, the client application keeps sending messages during the weaving. One of the solutions is to exchange ports: the users first add a new port, and then, lead encrypted messages to the decryption process through the new port (Fig. 3.2 (B)). They also add a new port and send encrypted messages through it (Fig. 3.2 (C)). Thereby, the encrypted messages from the client host can be surely decrypted on the server.

3.2.2 A Visualization Aspect for N-Body Problem

Another example is computing the N-body problem executed in a distributed environment in parallel as shown in Fig. 3.3 and Fig. 3.4. The data of each satellite such as the position and speed are distributed among hosts and each host computes the data of the satellites allocated to the host. For every tick, the hosts compute the new position and velocity of their satellites, exchange the new data with other hosts, and perform the barrier synchronization.

Suppose that we have now a running program of the N-body problem. Since it is a simple program that does not produce any visual images during computation, we want to write a visualization aspect, which collects the data from every host for every tick and draws an image for illustrating the current positions of the satellites (Fig. 3.5).

When the users try to weave a visualization aspect into the N-body program, the aspect must be activated exactly at the beginning of the same tick after the aspect is deployed on every host. Otherwise, as in Fig. 3.6, the visualization host receiving data from the computing hosts would draw a wrong picture based on the data for different ticks. Therefore, the aspect must be activated synchronously with the barrier synchronization by the N-body program. To accomplish this, the users need two aspects; one is the visualization aspect and the other is a monitoring aspect for coordinated weaving of the former aspect. The users first deploy two aspects. This deployment is executed asynchronously. Then they activate the support aspect immediately after it is deployed. Its advice intercepts the barrier synchronization on every host by the N-body program. Before every synchronization, it reports to a central host if the visualization aspect is already deployed on the host. Then, just after that synchronization, the advice inquires of the central host whether or not the visualization aspect is deployed on every host. If yes, the advice activates the visualization aspect on the host. Note that the activation is a blocking operation. Since every host

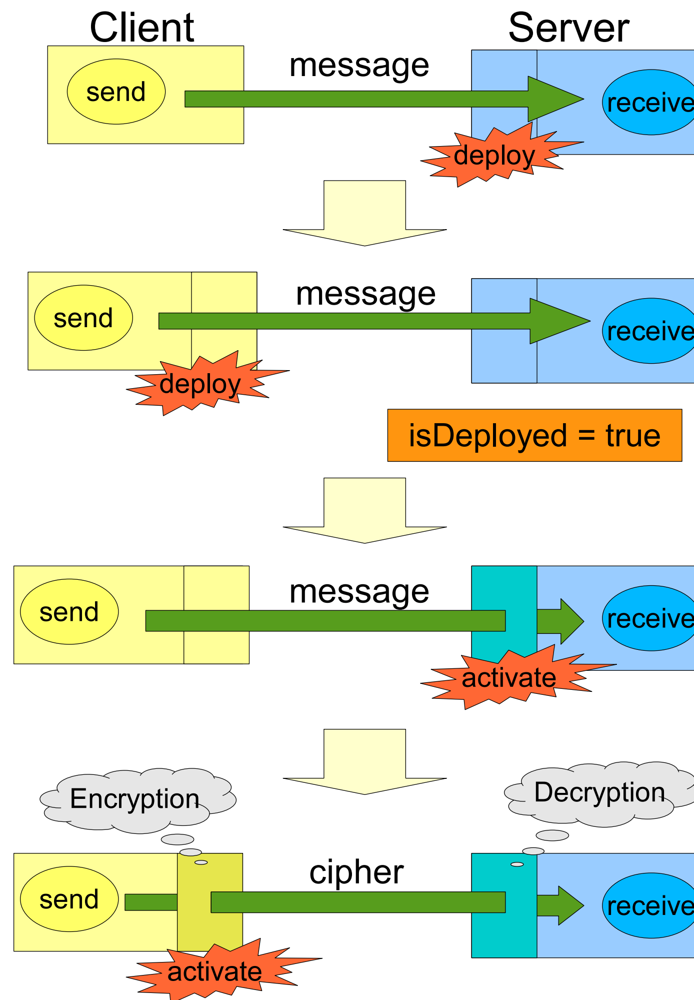


Figure 3.1: Two phase weaving of the encryption and decryption aspects

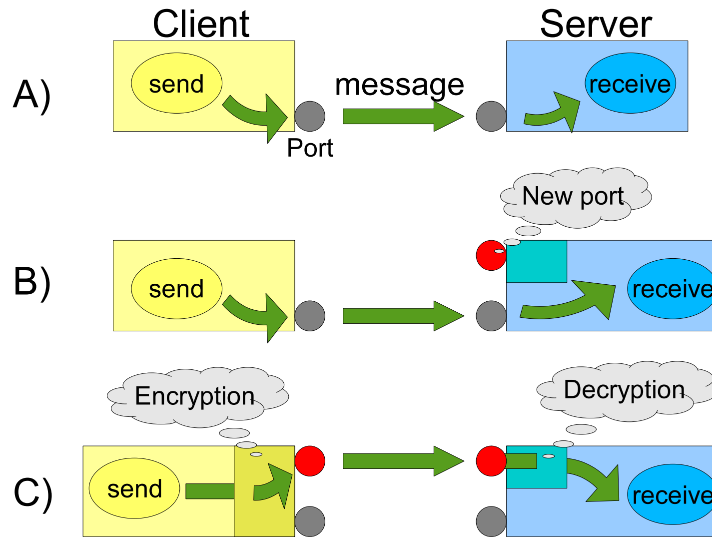


Figure 3.2: The process of the coordinated activation of the encryption and decryption aspects

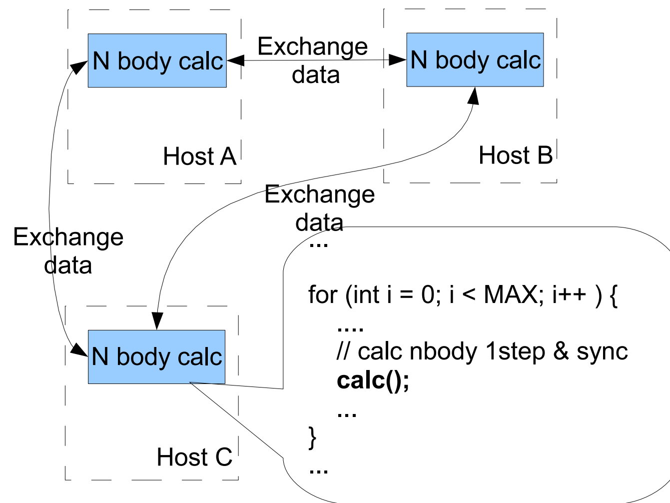


Figure 3.3: The computation of the N-body problem in distributed environments

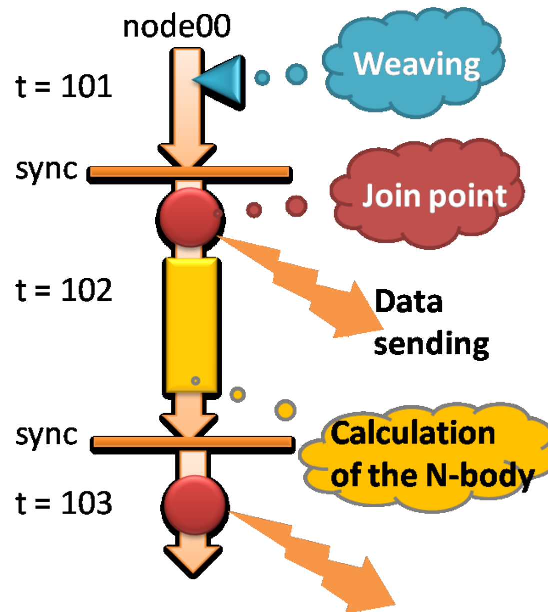


Figure 3.4: The computation loop of the N-body problem

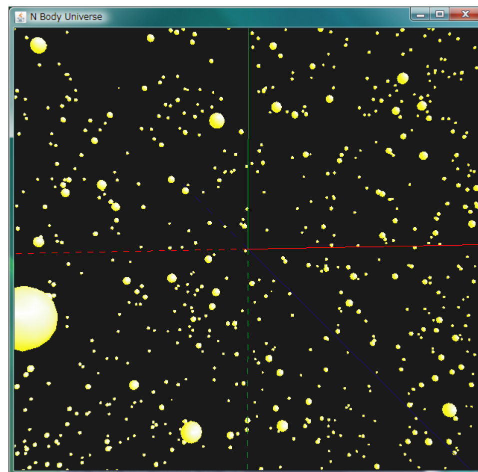


Figure 3.5: An Visualization of the N-body problem

performs barrier synchronization at every tick, the support aspect ensures that the visualization aspect is activated at the same tick.

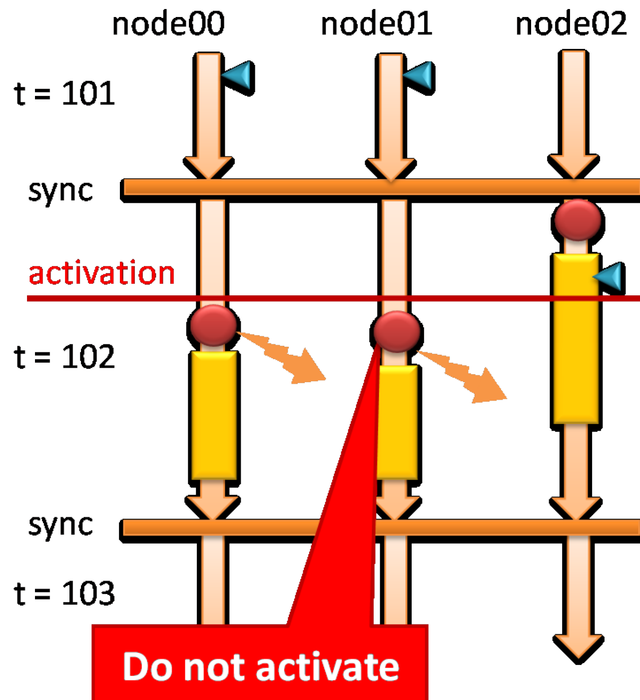


Figure 3.6: Activation failure of the visualization aspect

3.3 Discussion

Users can implement the algorithms that deploy aspects in parallel and coordinate their activation. Because the primitive operations provided by two phase weaving are non-blocking, if they are properly used, the users can minimize the pause time of the applications during the weaving. However, that is not to say that the users can always minimize the pause time. For example, if a program cannot continue to execute without aspects after a certain stage, the program must be suspended until the aspects are woven.

The two phase weaving is effective when the target program works properly either if all the aspects are woven or if no aspects are woven. For exam-

ple, the encryption and decryption aspects shown in Section 2 fit this case. The visualization aspect also fits since it must be effective simultaneously but can be at any time.

On the other hand, the two phase weaving has a limitation. Occasionally, the developers have to implement an aspect to explicitly consider the dynamic weaving. In the case of the encryption and decryption aspects, the aspect on the server host must be able to receive both plain and encrypted messages during the weaving. This is an example of that consideration. On the other hand, in the case of the visualization aspect, because the users activate the aspect along with the barrier synchronization by the application, the aspect does not need to consider dynamic weaving. Instead, the users write a support aspect, which has an advice activating target aspect along with the barrier synchronization. This support aspect modularizes the consideration of dynamic weaving.

Chapter 4

DandyJ

For demonstrating the two phase weaving, we implemented *DandyJ*, which is a dynamically distributed aspect-oriented language providing the primitive operations for the two phase weaving. Aspects woven by two phase weaving are often complicated since a coordination concern and/or a distributed concern tangle with the implementation of the target aspect. DandyJ provides three language supports for decreasing that complication. Two of them have been already proposed in other literature. DandyJ borrows those ideas but one feature is new.

DandyJ provides a first-class aspect [13] to enable writing a *meta* aspect, which coordinates the activation of another aspect. Such a meta aspect is a separate module for coordinated activation and it is reusable for the activation of arbitrary aspects. The first-class aspects, which we call *dynamic aspects*, can be dynamically deployed and then activated during runtime on specified remote hosts. This dynamic weaving accepts an aspect that had not been written yet when the target program started running. The first-class aspects can be also undeployed when they are not necessary any longer. deploy and activate, are executed by calling a special method on the target (first-class) aspect. It is also possible to use isDeployed by calling another special method for inquiring whether or not the first-class aspect has been deployed. In addition, some other backup support operations are available.

DandyJ also provides the remote pointcut [16, 14, 12] for making it easy to activate aspects along with a execution status of remote hosts. Furthermore, it provides the *onetime aspect* for making it easy to write coordinated activation, which is undeployed after all of the advices are executed. This is DandyJ's unique feature.

Table 4.1: The pointcut designators provided by DandyJ

pointcut designator	the selected join points
hosts (*hostName1, hostName2 ..*)	the join points in the execution on *hostName1, hostName2, ..*
hostName (*hostName*)	the join points on the host with the name *hostName*

4.1 Remote pointcut

DandyJ provides the *remote pointcut*, which picks out join points in the execution of a program running on a remote host. The advice body associated with a remote pointcut is executed when a thread of control on a remote host reaches the join points picked out by the remote pointcut.

The pointcut designators provided by the current implementation of DandyJ for the remote pointcut are listed in Table 4.1. For example, the users can describe the following pointcut:

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

This pointcut picks out the execution of send method in the Client class on the hosts node000 and node001, which are specified by the hosts pointcut designator. For the hostName pointcut, the parameter h is bound to the name of the host on which send method is executed. The value of h is available in the advice body.

DandyJ also provides *local advice*. It has the local modifier at the beginning of the advice declaration. If an advice has this modifier, it is locally executed on the same host where the selected join point exists. The local advice is useful, for example, to implement an advice that does not need to perform communication among hosts.

4.2 Dynamic aspect

DandyJ allows programmers to define an aspect that can be woven dynamically. We call such an aspect *a dynamic aspect*. It has the dynamic modifier at the beginning of the aspect declaration. This aspect does not need to be woven statically at compilation time. It must be explicitly instantiated. To

deploy and activate it, programmers have to call a special method on the instance. Table 4.2 lists the special methods available on instances of dynamic aspects. These methods can be available from a class, a static aspect, and a dynamic aspect.

An instance of a dynamic aspect is a first-class object. It can be passed to a method as a parameter. Thus, programmers can write a dynamic aspect that takes another dynamic aspect, deploys it, and activates it on all relevant hosts in a coordinated manner. Such an aspect is reusable for coordinating the activation of other dynamic aspects. The following is a code example:

```
DAspect encrypter = new Encrypter (); /* a dynamic aspect */
encrypter.deploy ();
while (!encrypter.isDeployed ())
  wait(100);
encrypter.activate ();
...
encrypter.undeploy ();
```

In this example, a dynamic aspect Encrypter is instantiated and then deployed by the method deploy(). Note that the type of the variable encrypter referring to the aspect instance is DAspect. It is a super type of all types of dynamic aspects. A variable of this type can hold a reference to an instance of any dynamic aspect. Then this code waits until the aspect encrypter is deployed on all the hosts. To do this, isDeployed () is called. The next is a call to activate(), which activates the deployed aspect encrypter. Finally, when the aspect is not needed any longer, it is undeployed by undeploy(). The code of the aspect is deleted from the Java virtual machines.

Note that an instance of a dynamic aspect is not copied/moved to remote hosts when it is deployed. Thanks to remote pointcuts, an aspect does not need deployment on the hosts where interesting joinpoints exist. Thus, the aspect can be instantiated on any appropriate host. An exception is the case that an aspect have local advice. It is deployed on a specific host although, at the implementation level, the hook code for this aspect may be distributed. If a dynamic aspect contains a remote pointcut, the hook code is woven on every host. Although the advice bodies of that aspect are executed on the same single host, hook code is deployed on all the hosts specified by the pointcuts. The hook code intercepts at the selected join points and remotely invokes the advice bodies associated with those join points. isDeployed returns true only if the aspect and its hook code have been deployed on all the host and they are ready to run.

4.2.1 Monitoring aspect

By using dynamic aspects, the users can easily write activation code, which explicitly control when aspects is activated. A dynamic aspect can be also used for modularly implementing *a monitoring aspect.* A monitoring aspect monitors the progress of the computation on each host. This progress monitoring is a typical crosscutting concern. This monitoring is useful if the time of activation depends on the execution state of the target application program. For example, the activation code for the encryption aspect must monitor the sending process to avoid the activation during a message sending. The activation code for the visualization aspect must monitor the progress to know when each host starts waiting by barrier synchronization and to activate the aspect just after that. It must be deployed and activated before the target aspect such as the encryption (and the visualization) aspect. After the target aspect is activated, the monitoring aspect should be deactivated and undeployed since it is a performance bottleneck and it is not necessary any longer.

The remote pointcut is useful for implementing a monitoring aspect in distributed environment. For example, the aspect should monitor remote hosts to know when they perform message sending (or barrier synchronization). In DandyJ, this monitoring can be simply described by using the remote pointcut. No explicit remote method invocation is necessary. If a remote pointcut selects the execution of a barrier synchronization method, the advice is executed on a central visualization host when one of the computing hosts starts waiting for barrier synchronization.

An aspect for coordinated weaving of another aspect on multiple hosts is normally a dynamic aspect as well. This is because, when we want to dynamically deploy and activate an aspect A, we must first dynamically weave the aspect A_c for coordinated activation of that aspect A. Otherwise, we would have to statically weave A_c in advance but it is usually difficult to expect that we would write A in future and hence to write A_c before.

4.3 Onetime aspect

DandyJ enables to write the aspect that is automatically undeployed after all its advices are executed. The advice in an onetime aspect is executed only once. We call this aspect *an onetime aspect.* It has the `onetime` modifier at the beginning of the dynamic aspect declaration.

An onetime aspect is also useful for implementing a monitoring aspect, which should be unwoven after the activation since it is a performance bot-

Table 4.2: The methods available on instances of dynamic aspects

operation	behavior
void deploy (String... *hostName*)	deploy the dynamic aspect as inactivated state
void undeploy (String... *hostName*)	remove the aspect from the join points
void activate (String... *hostName*)	activate the aspect
void unactivate (String... *hostName*)	inactivate the aspect
boolean isDeployed (String... *hostName*)	returns true if the aspect is deployed
boolean isActivated (String... *hostName*)	returns true if the aspect is activated

tleneck and it is not necessary any longer. If the users use an onetime aspect, they do not need to write explicit undeployment code. The following is a example of an onetime aspect.

```
1 onetime dynamic aspect OnetimeTest {
2   before () : execution (void Test.foo()) {
3     System.out.println("before: foo")
4   }
5   before () : execution (void Test.bar()) {
6     System.out.println("before: bar")
7   }
8 }
```

```
1 class Test {
2   void run() {
3     foo();
4     foo();
5     bar();
6     bar();
7   }
8 }
```

If run() of Test class is called, the advice of the OnetimeTest onetime aspect is also called. Since an advice of an onetime aspect is executed only once, when foo() at line 3 is called, the before advice prints "before: foo". However, at the second call to foo(), this is not printed. The second advice of OnetimeTest is also called when bar() at line 5 is called. The OnetimeTest

onetime aspect is undeployed after the execution of the advice since all the advices of OnetimeTest have been executed.

4.4 Examples in DandyJ

In this section, we show two examples of coordinated activation in DandyJ.

4.4.1 An Encryption Aspect in DandyJ

We below show the example shown in Section 3.2.1 written in DandyJ. The implementations of the aspects are shown in Fig. 4.1, Fig. 4.2 and Fig. 4.3.

In Fig. 4.1, the abstract aspect ServerClient is a monitoring aspect that deploys and activates the several dynamic aspects coordinately that will work on the server and the client hosts. The aspect monitors the progress of the computation on each host. The activation code of the encryption aspect monitors the sending process to avoid the activation during a message sending. This progress monitoring is a crosscutting concern. The aspect modularizes it. After the encryption aspect is activated, the monitoring aspect can be deactivated and undeployed because it is a performance bottleneck and it is not necessary any longer.

These target aspects are given to the constructor in which they are deployed by deploy () to be the inactivated state (line 10-13). ServerClient has two before advices. These advices are associated with the abstract pointcuts that select the join points at which the methods for receiving/sending messages on the server/client hosts are invoked. For example, the before advice associated with the abstract pointcut server will be executed just before the method for receiving messages is invoked. In the before advice, the openPort instance is activated to add a new port, and server to decrypt received messages (line 21-22). As Fig. 4.2, when the advice of openPort is executed after it is activated, a new thread is created to add a new port for receiving encrypted messages and wait these messages. Then, openPort is undeployed because it is a onetime aspect. This is because the port prepared by the aspect is not needed open any longer. In addition, server.isActivated() is turned true because the aspect on the server side is activated. Therefore, the before advice associated with the abstract pointcut client is executed, then the changePort instance is activated to change the port, and client to encrypt sending messages (line 28-29). The port used for sending a message is changed and a message sent by the client host is encrypted. Finally, ServerClient is undeployed since it is also a onetime aspect. The encryption and decryption aspect is implemented as shown in Fig. 4.3.

Note that, this monitoring aspect is defined as an abstract aspect, and it can be reused for coordinated activation of various kinds of aspects in messaging service applications. As an example, the Fragmenter and Reassembler aspects in [34] can be applicable to this monitoring aspect.

4.4.2 An Visualization Aspect in DandyJ

We below write a program in DandyJ for the example in Section 3.2.2. Fig.4.4 presents the monitoring aspect for weaving the visualization aspect. It is an abstract aspect to be reusable for weaving a similar aspect in the N-body program. Since DandyJ provides a remote pointcut, the visualization aspect can be deployed on a host (maybe) where a visual image will be drawn. It collects satellite positions at the end of every tick from computing hosts by a remote pointcut. The monitoring aspect runs on the same host that the visualization aspect does. When it is deployed and activated, it first starts deployment of the visualization aspect, passed to the constructor of the monitoring aspect. The monitoring aspect must maintain on which host the visualization aspect is ready to run but, thanks to a remote pointcut again, this work can be easily implemented. See the before and after advice declarations in Fig.4.4. They are executed whenever the N-body program performs barrier synchronization on each host. Note that the advice is executed multiple times per synchronization if multiple hosts are computing satellite positions. The pointcut if(dAspect.isDeployed()) is evaluated on the host where the advice is executed and it returns true if the visualization aspect has been deployed and the hook code is installed on all the computing hosts. Thus, the visualization aspect is activated by the after advice when it is ready to run on all the computing host.

4.5 Implementation Notes

In this section, we explain the implementation of DandyJ. DandyJ consists of two components: ddjc and a runtime system.

4.5.1 DandyJ compiler

The ddjc is a compiler implemented as an extension of abc [3], which is one of the AspectJ compilers [10, 23], developed with the JastAdd compiler compiler [7, 18]. JastAdd is an open source Java-based compiler compiler system. It is designed to support high-level extensible implementation of

```
1  public abstract onetime dynamic aspect ServerClient {
2    private DAspect changePort, openPort, client, server;
3
4    public ServerClient (DAspect changePort, DAspect openPort,
5                             DAspect client, DAspect server) {
6      this.changePort = changePort;
7      this.openPort = openPort;
8      this.client = client;
9      this.server = server;
10     openPort.deploy ();
11     changePort.deploy ();
12     client.deploy ();
13     server.deploy ();
14   }
15
16   abstract pointcut server ();
17   abstract pointcut client ();
18
19   before () : server () && if (server.isDeployed()
20                         && openPort.isDeployed()) {
21     openPort.activate ();
22     server.activate ();
23   }
24
25   before () : client () && if(server.isActivated()
26                         && client.isDeployed()
27                         && changePort.isDeployed()){
28     changePort.activate ();
29     client.activate ();
30   }
31 }
```

Figure 4.1: The monitoring aspect for the messaging service application

```
1  public onetime dynamic aspect OpenPort {
2    local java.net.Socket around ():
3                          call (java.net.Socket ServerSocket.accept ())
4                          && within (Server) && hosts ("node000") {
5      // create a new thread that opens a new port and waits messages
6    }
7  }
8
9  public dynamic aspect ChangePort {
10   local void around ():
11                         execution (void smessage.Client.connect ())
12                          && within (Client) && hosts ("node001") {
13     // change the port for sending messages
14   }
15 }
```

Figure 4.2: The aspect for opening and changing port

```
1  public dynamic aspect Encrypter {
2    local void around (String s): call (void Client.send (String))
3                                  && within (Client)
4                                  && hosts ("node000") && args (s) {
5      // encrypt the message
6      proceed (s);
7    }
8  }
9
10 public dynamic aspect Decrypter {
11   local void around (String s):
12                                call (void Server.receive (String))
13                                && within (Server)
14                                && hosts ("node001") && args (s) {
15     if (socket.getLocalPort() == /* the new port number */) {
16       // decrypt the message
17       proceed(s);
18     } else {
19       proceed (s);
20     }
21   }
22 }
```

Figure 4.3: The encryption and decryption aspect

```
1  public abstract aspect BarrierSync {
2    private String[] hostNames;
3    private DAspect dAspect; // the target aspect
4    private Map<String, Boolean> map
5                              = new Hashtable<String, Boolean>();
6
7    public BarrierSync (DAspect dAspect, String[] hostNames) {
8      this.dAspect = dAspect;
9      this.hostNames = hostNames;
10     for (String host : hostNames)
11       map.put (host, false);
12     dAspect.deploy ();
13   }
14   /* picks out the execution of the barrier synchronization */
15   abstract pointcut sync ();
16
17   before (String host): sync () && hostName (host) {
18     map.put (host, true);
19   }
20
21   after (String host): sync () && hostName (host)
22                                   && if (dAspect.isDeployed ()) {
23     synchronized (this) {
24       if(!map.containsValue (false)) {
25         dAspect.activate ();
26         this.undeploy ();
27       }
28     }
29     map.put (host, false);
30   }
31 }
```

Figure 4.4: The monitoring aspect for the barrier synchronization

compilers. JastAdd is used for doing computations on an AST. An AST build by a parser can be adjusted by using JastAdd.

The ddjc transforms an AST of dynamic aspects which have dynamic modifier to handle the dynamic weaving as the following steps (Fig. 4.5):

1. ddjc build an AST by using the parser and programs.
2. It collects informations such as advice spec, joinpoints, parameter type, and so on from the AST.
3. It adds some AST nodes to implement some interface in the dynamic aspects to extend DAspect super-type and handle the remote method invocation.
4. It adds special methods and fields such as deploy() and activate() and so on to the AST.
5. It transforms the advices to enable the activate operation.
6. It extra transforms the AST nodes of a onetime aspect and an aspect which has local advices.

4.5.2 DandyJ runtime system

The runtime system can create a new class bytecode that aspect codes are woven into. Substituted bytecodes are created by Javassist [6, 19], which is a class library for modifying Java bytecodes. The runtime system uses the HotSwap technology of the java.lang.instrument API to exchange the bytecode of classes. It has the premain method, which the JVM invokes before the main method of a program. It starts running when a program is started by the ddj, which is the command to start programs.

The runtime system runs on every host where programs are running. When the users deploy a dynamic aspect on a host, the runtime system on the remote host specified by a remote pointcut is requested to deploy the aspect. The advice bodies of that aspect will be executed on the host where the deploy is invoked. We call this host *an aspect server*. If a thread of control on a host reaches the join point selected by a remote pointcut, this host notifies the aspect server of reaching the specified join point. Then, the aspect server executes the advice if the aspect has been already activated. Even though an advice with local modifier is locally executed, the aspect server is inquired of whether or not the aspect should be activated.

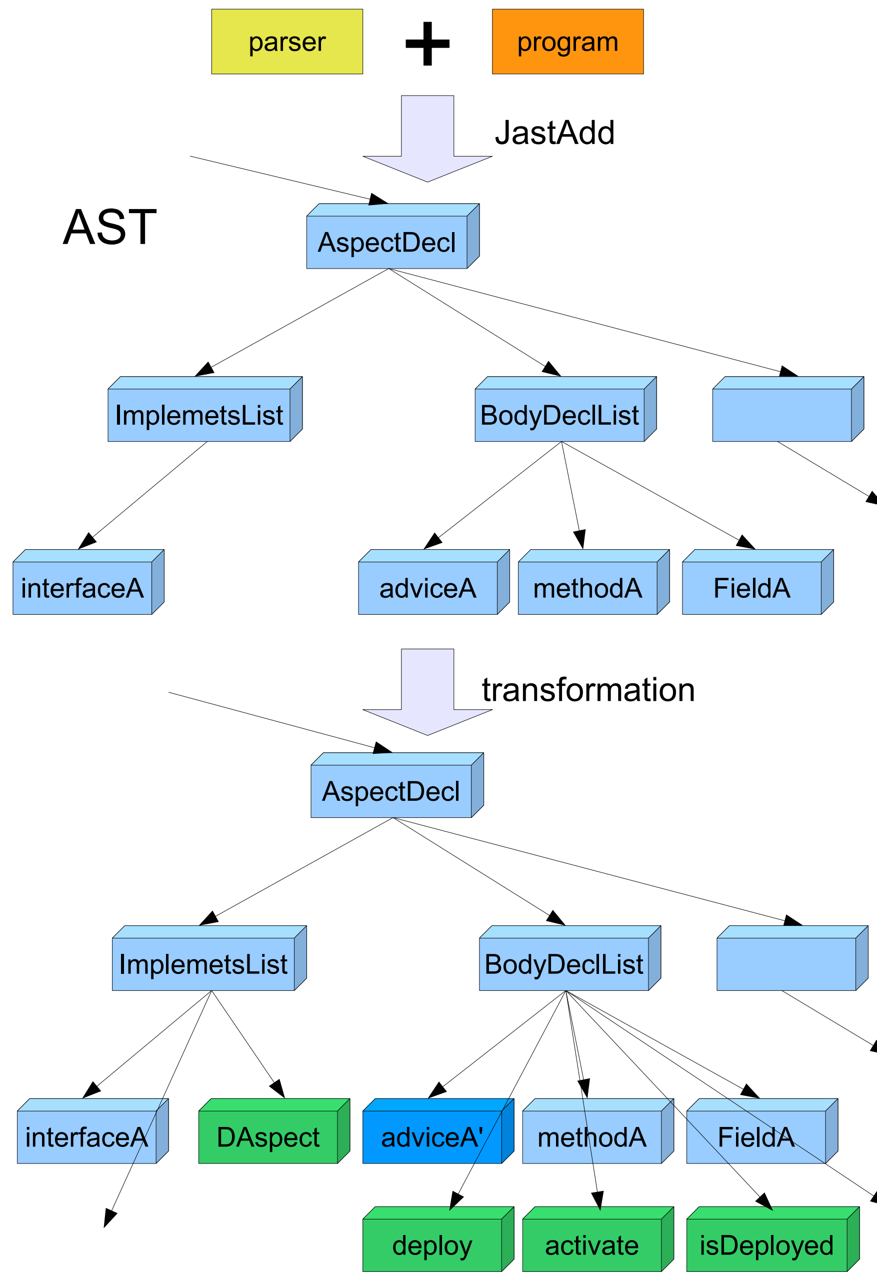


Figure 4.5: The process of the compilation by the ddjc

The isDeployed() returns true only after all the advices of the aspect notify the aspect server. Since the deployed advice notifies once the aspect server of the deployment to be finished, the aspect server can figure out whether the advices are deployed. This operation is necessary because it is difficult to know when the deployment is finished because the deployment is asynchronously performed and there is the restrictions imposed by the HotSwap; the behavior of a method cannot be modified when it is on the call stack.

Chapter 5

Experiment

5.1 Overview of this measurement

To evaluate the performance overhead of DandyJ, we measured the cost of deployment and activation by using the encryption/decryption aspects in Section 4.4.2. We prepared four programs for this measurement:

(A) deploys normally deploys these aspects and activates without the coordination

(B) deploys with the coordination onetime aspects of DandyJ as in Section 4.4.2

(C) deploys with the coordination aspects, however it will not undeployed after the activation

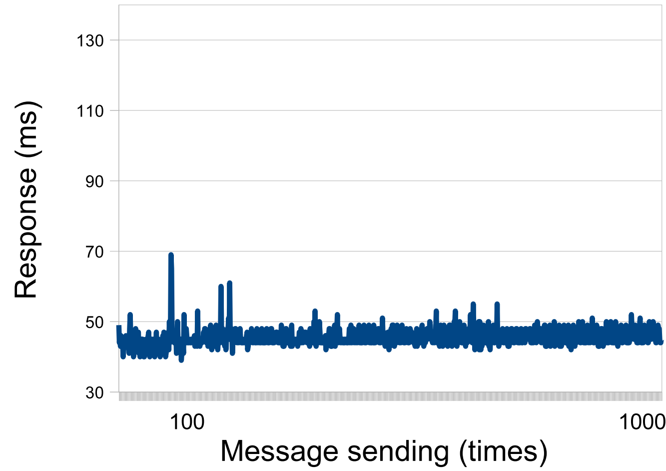
(D) deploys with the naive coordination solution shown in Section 2

The client application tries to connect the server application and then send a message every one millisecond. This trial is repeated 1,000 times. The elapsed time was calculated from when the first connection request is transmitted until when the connection is closed. Our experiment was carried out on the chiba cluster with a 2.13 GHz Core2Duo processor and 4 GB RAM with Java 1.6.0 and the mirai cluster with a 2.33 GHz Xeon processor and 16 GB RAM with Java 1.6.0. There are part of InTrigger [9].

5.2 Measured overhed of dynamic weavings

Fig. 5.1, Fig. 5.2, Fig. 5.3, Fig. 5.4, and Table. 5.1 present overviews of the measured overhead. We wove the aspects around 100th message sending.

Figure 5.1: (A) without the coordination



Therefore, in each graph, a spike is seen around 100th sending. The response time of (A) is faster than the others. However, (A) failed to decrypt the messages 10 times around 100th sending on the server host. This is because the program (A) does not contain the coordination and hence the encryption aspect was activated before the activation of the decryption aspect. On the other hand, the programs (B), (C), and (D) perform the coordination. Therefore, the failure of decryption does not occur in these programs. In the program (D), a higher peak (about 130 ms) was around 100th sending because it stops the application till the deployment finishes. The response time of (C) is slower than the others. This is because the monitoring aspect is a performance bottleneck and is not undeployed after the activation. (B) is our suggestion and faster than (C) and (D). This uses the onetime aspect, which is undeployed after the coordinated activation. Therefore, the overhead of the coordination is almost nothing after the weaving.

Figure 5.2: (B) with the coordination aspects in DandyJ

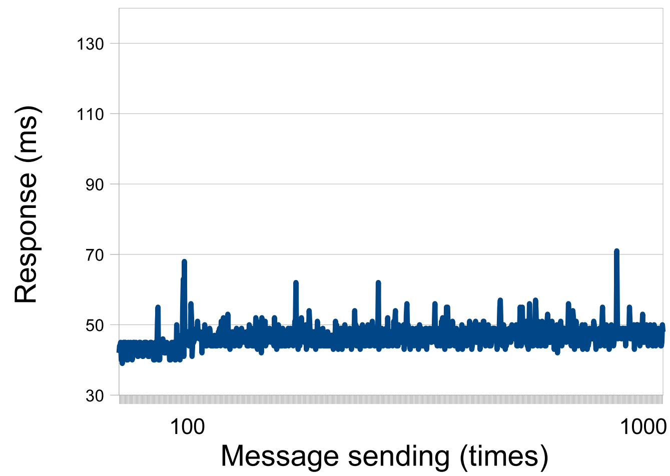


Figure 5.3: (C) with the coordination without undeployment

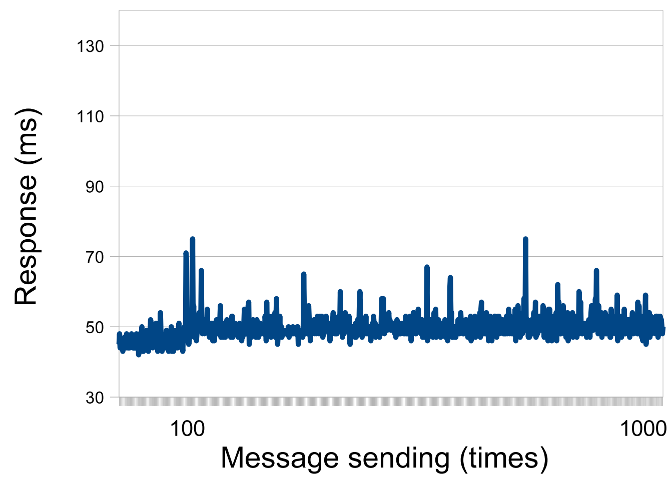


Figure 5.4: (D) with the naive coordination solution

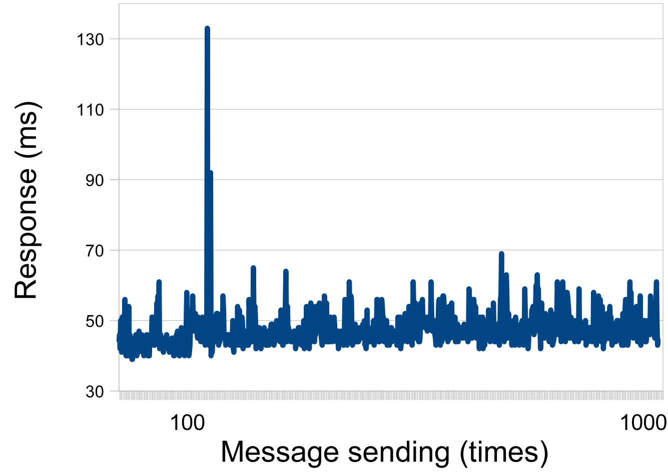


Table 5.1: The overview of the response time

variant	average (ms)	max (ms)
(A) without the coordination	45.4	69
(B) with the coordination aspects in DandyJ	46.6	71
(C) with the coordination without undeployment	49.6	75
(D) with the naive coordination solution	47.5	133

Chapter 6

Related Work

There are many dynamic aspect-oriented languages such as PROSE [22, 21], Steamloom [5], JAsCo [26], Wool [24] and HotWave [35], but these languages do not adequately support distributed computing. On the other hand, the dynamic aspect-oriented languages such as the latest version of PROSE [15], JAC [20], CaesarJ [13, 2], AWED [14], ReflexD [31] and DyReS [34] provide some support for distributed computing. Especially, AWED and ReflexD provide the remote pointcuts.

6.1 CaesarJ

CaesarJ is an aspect-oriented language that can treat an aspect as a first-class object. The users of CaesarJ can specify that an aspect is woven with or without activation. An inactivated aspect can be activated by the deploy. The users can use deploy block to make an aspect active only with that block. In distributed environments, CaesarJ provides an object representing a remote host. The inactivated aspects are activated on the remote host when the users give the inactivated aspects to the remote host object. However, unlike DandyJ, CaesarJ does not adequately support the dynamic weaving. Furthermore, it does not provide the remote pointcut. The users can implement programs such as Fig. 6.1 and Fig. 6.2.

In Fig. 6.1, TestApp class caluculate the Fibonacci number and FibonacciAspect is an aspect for TestApp. FibonacciApsect is deployed by DeploymentAspect using deploy block. In Fig. 6.2, HelloAspect wraps HelloWorld to add a new method.
These programs output follows.

```
m1
```

```
1  public class TestApp {
2
3    public void m1 (int n) { fibstart(n); }
4
5    public void m2 (int n) { fibstart(n); }
6
7    public void fibstart (int n) { fib(n); }
8
9    public int fib (int k) {
10     return (k > 1) ? fib(k − 1) + fib(k − 2) : k;
11   }
12
13   public static void main(String[] args){
14     TestApp ta = new TestApp();
15     System.out.println("m1"); ta.m1(5);
16     System.out.println("m2"); ta.m2(5);
17   }
18 }
19
20 public cclass FibonacciAspect {
21   private int ctr = −1;
22
23   before(): execution (void TestApp.fibstart(int)) { ctr = 0; }
24
25   after(): execution (void TestApp.fibstart(int)) {
26     System.out.println(ctr);
27   }
28
29   before(): execution(int TestApp.fib(int)){ ctr++; }
30 }
31
32 public deployed cclass DeploymentAspect {
33   void around(): call(void TestApp.m2(int)){
34     System.out.println("end");
35     deploy(new FibonacciAspect()){
36       proceed();
37     }
38   }
39 }
```

Figure 6.1: An Aspect in CaesarJ

```
1  public class HelloWorld {
2
3    public void sayHello (String message) {
4      System.out.println(message);
5    }
6    public void sayHello2 (String message) {
7      System.out.println(message);
8    }
9    public static void main (String[] args) {
10     new HelloWorld().sayHello("Hello␣World");
11   }
12 }
13
14 public deployed cclass HelloAspect {
15
16   public cclass WrapHello wraps HelloWorld{
17     public String s = "newField";
18
19     public void newMethod(String msg){
20       System.out.print("newMethod␣");
21       wrappee.sayHello2(s + msg);
22     }
23   }
24   pointcut p(HelloWorld h) :
25            execution(void HelloWorld.sayHello(String))
26            && this(h);
27
28   after(HelloWorld h) : p(h){
29     WrapHello(h).newMethod("␣World!");
30   }
31 }
```

Figure 6.2: An Aspect in CaesarJ 2

```
m2
15

Hello World
newMethod newField World!
```

6.2 AWED

AWED is a distributed aspect-oriented language. AWED provides the remote pointcut. DJAsCo is a implementation of AWED and a dynamic distributed aspect-oriented language extending JAsCo [26] for distributed computing. DJAsCo does not describe joinpoints in an aspect. Instead, the users have to implement a hook and a connector. A hook define crosscutting behavior in an independent way. A connector deploys hooks onto concrete context. An aspect is dynamically woven by the runtime system when the aspect is set in the classpath directory. While the users can control the dependency among advices, adequately controlling the coordinated weaving is difficult because the users cannot write a code for weaving an aspect by using the primitive operations. Even though DJAsCo provides the remote pointcut, the aspects cannot be coordinately deployed or activated by monitoring the remote host. This is because the users cannot write activation code in an advice. The users can implement programs such as Fig. 6.3 and Fig. 6.4.

In Fig. 6.3, LoggingAspect is a simple aspect contains before advice for HelloWorld. In Fig. 6.4, ExcludeStrategy control the dependency between hook1 and hook0. In this case, hook1 is never activated when hook0 is not activated.
These programs output follows.

```
before
Hello World

before
Hello World
after
```

6.3 DyReS

DyReS is a framework implemented on top of the JBoss AOP [1] and the Spring AOP framework [25]. The users can separate the coordination code

```
1  public class HelloWorld {
2    public void sayHello () {
3      System.out.println("Hello␣World");
4    }
5    public static void main (String[] args) {
6      new HelloWorld().sayHello();
7    }
8  }
9
10 class LoggingAspect {
11   hook LoggingHook {
12     LoggingHook(method(..args)) {
13       execution(method);
14     }
15
16     before() { System.out.println("before"); }
17   }
18 }
19
20 connector LoggingConnector {
21   LoggingAspect.LoggingHook hook =
22     new LoggingAspect.LoggingHook(void HelloWorld.sayHello());
23   hook.before();
24 }
```

Figure 6.3: An Aspect in DJAsCo

```
1  class LoggingAspect {
2
3    hook LoggingHook {
4      LoggingHook(method(..args)) {
5        execution(method);
6      }
7      before() { System.out.println("before"); }
8    }
9
10   hook LoggingHook2 {
11     LoggingHook2(method(..args)) {
12       execution(method);
13     }
14     after() { System.out.println("after"); }
15   }
16 }
17
18 static connector LoggingConnector {
19
20   LoggingAspect.LoggingHook hook0 =
21     new LoggingAspect.LoggingHook(void HelloWorld.sayHello());
22   LoggingAspect.LoggingHook2 hook1 =
23     new LoggingAspect.LoggingHook2(void HelloWorld.sayHello());
24
25   hook1.after();
26   hook0.before();
27
28   addCombinationStrategy(new ExcludeStrategy(hook0, hook1));
29 }
30
31 public class ExcludeStrategy implements CombinationStrategy {
32
33   Object hook0;
34   Object hook1;
35
36   public ExcludeStrategy(Object hook0, Object hook1) {
37     this.hook0 = hook0;
38     this.hook1 = hook1;
39   }
40
41   public HookList validateCombinations(HookList hookList) {
42     if(!hookList.contains(hook0))
43       hookList.remove(hook1);
44     return hookList;
45   }
46 }
```

Figure 6.4: An Aspect in DJAsCo 2

into XML and control the activation of an aspect on remote hosts. Unlike DandyJ, the code for monitoring must be included in the program in advance before the program starts running. Furthermore, because scripts are written in XML, the programmers cannot implement coordinated weaving and activation in a reusable manner with high-level abstraction. The readability of the script is often low. The users of DyReS can implement programs such as Fig. 6.5, Fig. 6.6 and Fig. 6.7.

```
1  Aspect classes :
2  public class MessageFragmenter implements MethodInterceptor {
3    protected Thread thread ;
4    public MessageFragmenter () {
5      queue = new LinkedList <MethodInvocation >();
6      thread = new Thread ( this );
7      thread . start ();
8    }
9
10   public Object continueInvocation
11                ( MethodInvocation mi) throws Throwable {
12     queue . add(mi);
13   }
14
15   public synchronized void run () {
16     while ( running ) {
17       if (! queue . isEmpty ( ) ) {
18         MethodInvocation mi = queue.removeFirst();
19         MethodInvocation[] fragments = fragment(mi);
20         for ( int i =0; i< fragments.length; i++)
21           fragments[i ]. proceed ();
22       }
23     }
24   }
25 }
26
27 public class MessageReassembler implements MethodInterceptor {
28     ...
29 }
```

Figure 6.5: An Aspect for MSA in DyReS

In Fig. 6.5, MessageFragmenter and MessageReassembler are aspects for a messaging service application. These aspects are coordinately woven into an application by using the XML script in Fig. 6.6. The program in Fig. 6.7 is a main program for the weaving.

```
1  ./springconig.xml :
2  <!-- declaring application components as beans -->
3      <bean id ="stub" class="Stub" .../>
4      ...
5  <!-- declaring Messagefragmenter as a bean -->
6      <bean id ="fragmenter" class="MessageFragmenter" />
7  <!-- declaring advice binding -->
8      <bean id ="fragmentingBinding"
9           class="org.spring..RegexpMethodPointcutAdvisor">
10          <property name="advice">
11              <ref local="fragmenter" /></ property>
12          <property name="patterns">
13              <value>.send.*</ value></ property>
14     </ bean>
15 <!-- create proxies to apply advice -->
16     <bean class="org.spring..BeanNameAutoProxyCreator">
17          <property name="beanNames">
18              <value>stub</ value></ property>
19          <property name="interceptorNames">
20              <value>fragmentingBinding</ value></ property>
21     </ bean>
```

Figure 6.6: A Script for coordination in DyReS

```
1  Client main program :
2  public static void main(String[] args) {
3    // Loading the client application.
4    // MessageFragmenter gets woven because
5    //it is specified in the springconfig.xml file
6    ApplicationContext ctx =
7     new FileSystemXmlApplicationContext("springconfig.xml");
8    ...
9    // dynamically removing the MessageFragmenter
10   Advised advised = (Advised) ctx.getBean( "stub" );
11   Advisor advisor =
12             ( Advisor ) ctx.getBean ( "fragmentingBinding" );
13   advised.removeAdvisor ( advisor );
14   ...
15 }
```

Figure 6.7: A program for weaving in DyReS

6.4 JAC

JAC is a framework for dynamic aspect-oriented programming in Java. JAC does not require any language extensions to Java. Furthermore, the users of JAC can weave aspects into the programs on remote hosts. However, unlike DandyJ, the users cannot weave aspects without activation. Besides, the users cannot easily monitor the programs running on a remote host because it does not provide the remote pointcuts.

6.5 ReflexD

ReflexD is a platform for distributed aspect-oriented programming in Java. It is an extension of Reflex [30, 27]. ReflexD relies on the notions such as the distributed cut, the distributed action and the distributed binding. ReflexD provides the remote pointcut. However, unlike DandyJ, it does not separate code transformation and activation as primitive operation. Therefore, it is difficult to control the dynamic weaving.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

This paper presents two phase weaving. It separates weaving into two steps: *deployment* and *activation*. Since the deployment is executed asynchronously, the users can minimize the pause time of programs only by controlling the activation for the coordination. This paper also presents DandyJ, which is our new dynamic aspect-oriented programming language that supports two phase weaving. DandyJ provides three features: dynamic aspect, the remote pointcut and onetime aspect. Therefore, DandyJ enables modular and simple implementation of the aspects for two phase weaving.

Finally, this paper presents the applicability of DandyJ by illustrating two examples: a messaging service application and a N-body problem. Additionally, to evaluate the performance overhead in DandyJ, this paper measured the cost of deployment and activation by using a message service application. It shows that two phase weaving reduces the pause time of the coordinated weaving, and also resolves the difficulty in controlling the coordination of aspects.

Our contributions in this paper are the followings:

- Presenting that, in large distributed environments, controlling when an aspect starts deployment and when it finishes is significant to keep consistency among hosts.
- Proposing a new weaving mechanism called *two phase weaving*, which separates weaving into two step. Designing and implementing a dynamic distributed AOP language based on the two phase weaving.

- Evaluating the two phase weaving by an experiment in a real grid computing environment.

7.2 Future Work

Although the states of the aspects are controlled on the aspect server, this centralized approach might be a performance bottleneck. It is future work that we extend DandyJ to make an aspect disconnect the aspect server when checking the aspect state is unnecessary because we can expect it will not change any more.

Besides, the notion of scoping considered this paper is very primitive (on/off). Therefore, we have to consider more structured scoping mechanisms for distributed dynamic aspects [28, 29].

Bibliography

[1] JBoss AOP. `http://www.jboss.org/jbossaop/`.

[2] Ivica Aracic, Vaidas Gasiunas, Mira Mezini, and Klaus Ostermann. An overview of caesarj. 3880:135–173, 2006.

[3] 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, New York, NY, USA, 2005. ACM.

[4] Lodewijk Bergmans and Mehmet Aksit. Composing crosscutting concerns using composition filters. *Commun. ACM*, 44(10):51–57, 2001.

[5] 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, New York, NY, USA, 2004. ACM.

[6] Shigeru Chiba. Load-time structural reflection in java. In *ECOOP '00: Proceedings of the 14th European Conference on Object-Oriented Programming*, pages 313–336, London, UK, 2000. Springer-Verlag.

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

[8] Robert Hirschfeld. Aspects - aspect-oriented programming with squeak. In *NODe '02: Revised Papers from the International Conference NetObjectDays on Objects, Components, Architectures, Services, and Applications for a Networked World*, pages 216–232, London, UK, 2003. Springer-Verlag.

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

[10] 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, London, UK, 2001. Springer-Verlag.

[11] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *ECOOP*, pages 220–242, 1997.

[12] Bert Lagaisse and Wouter Joosen. True and transparent distributed composition of aspect-components. In *Middleware '06: Proceedings of the ACM/IFIP/USENIX 2006 International Conference on Middleware*, pages 42–61, New York, NY, USA, 2006. Springer-Verlag New York, Inc.

[13] 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, New York, NY, USA, 2003. ACM.

[14] 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, New York, NY, USA, 2006. ACM.

[15] 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, New York, NY, USA, 2008. ACM.

[16] 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, New York, NY, USA, 2004. ACM.

[17] Doug Orleans and Karl Lieberherr. Dj: Dynamic adaptive programming in java. In *Reflection 2001: Meta-level Architectures and Separation of Crosscutting Concerns* , Kyoto, Japan, September 2001. Springer Verlag. 8 pages.

[18] JastAdd Home Page. `http://www.jastadd.org/`.

[19] Javassist Home Page. `http://www.csg.is.titech.ac.jp/~chiba/javassist/index.html`.

[20] 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, London, UK, 2001. Springer-Verlag.

[21] 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, New York, NY, USA, 2003. ACM.

[22] 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, New York, NY, USA, 2002. ACM.

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

[24] 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, New York, NY, USA, 2003. Springer-Verlag New York, Inc.

[25] Spring source. `http://www.springsource.org/`.

[26] 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, New York, NY, USA, 2003. ACM.

[27] Éric Tanter. Aspects of composition in the reflex aop kernel. In Welf Löwe and Mario Südholt, editors, *Software Composition*, volume 4089 of *Lecture Notes in Computer Science*, pages 98–113. Springer, 2006.

[28] Éric Tanter. Expressive scoping of dynamically-deployed aspects. In *AOSD '08: Proceedings of the 7th international conference on Aspect-oriented software development*, pages 168–179, New York, NY, USA, 2008. ACM.

[29] Éric Tanter, Johan Fabry, Rémi Douence, Jacques Noyé, and Mario Südholt. Expressive scoping of distributed aspects. In *AOSD '09: Proceedings of the 8th ACM international conference on Aspect-oriented software development*, pages 27–38, New York, NY, USA, 2009. ACM.

[30] Éric Tanter and Jacques Noyé. A versatile kernel for multi-language aop. In Robert Glück and Michael R. Lowry, editors, *GPCE*, volume 3676 of *Lecture Notes in Computer Science*, pages 173–188. Springer, 2005.

[31] Éric Tanter and Rodolfo Toledo. A versatile kernel for distributed AOP. In *Proceedings of the IFIP International Conference on Distributed Applications and Interoperable Systems (DAIS 2006)*, volume 4025 of *Lecture Notes in Computer Science*, pages 316–331, Bologna, Italy, June 2006. Springer-Verlag. Best Paper Award of the three DisCoTec 2006 Conferences.

[32] Peri L. Tarr, Harold Ossher, William H. Harrison, and Stanley M. Sutton Jr. degrees of separation: Multi-dimensional separation of concerns. In *ICSE*, pages 107–119, 1999.

[33] Rodolfo Toledo, Eric Tanter, Jose M. Piquer, Denis Caromel, and Mario Leyton. Using reflexd for a grid solution to the n-queens problem: A case study. In *Proceedings of the 2nd CoreGrid Integration Workshop*, Cracow, Poland, October 2006. springer.

[34] 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, New York, NY, USA, 2008. ACM.

[35] 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, New York, NY, USA, 2009. ACM.