Explicit distributed programming using AWED

Luis Daniel Benavides Navarro, Mario Südholt
Obasco project
INRIA-EMN, LINA
École des Mines de Nantes

AOSD-Europe summer school, language course
14 July 2006

Contents

1 Introduction

During this excercise you will work with AWED's implementation which is part of the distribution JAsCoDT (for "JAsCo distributed"). The goal of this exercice is threefold:

1. Develop **three (small-scale) implementations of crosscutting concerns** related to distributed programming and, in particular, the manipulation of replicated caches.
2. Get some knowledge of a corresponding **real-world application**: JBoss Cache, a framework for the implementation of replicated caches.
3. Demonstrate that a solution based on AWED, which provides a DSL for distributed aspects and non-atomic pointcuts, **improves on standard sequential languages**, such as AspectJ, that do not provide such features.

2 Structure of laboratory and environment

This laboratory is divided in **three exercises** requiring the following software to be developed:

1. A simple **chat application**
2. An extended version: a **messenger application**
3. A **refactoring of JBoss Cache**

2.1 Provided Eclipse workspace, projects and run configurations

We provide you with a suitable **eclipse workspace** containing all elements necessary to solve the laboratory. The workspace can be downloaded at the following address (If necessary, it can also be copied from a USB stick.):

`http://bahiavirtual.com/djasco/workspaceUnsolvedLab.zip`

The workspace contains the following **projects**:

1. `01Chat`: the project for the simple chat application
2. `02Messenger`: the project for the messenger application
3. `03jbosschrfc`: the project for JBoss Cache refactoring
4. `99lib`: a project containing some common libraries. These are included the configuration of your workspace.

The workspace contains some ready-to-use **run configurations** that help to run the exercises. These configuration can be found in the tab `Main` in menu `Run->Run...` The included run configurations are:

- `Java Application/01Chat`: standard java run configuration included to test the behavior of the shell that is used in the two first exercises
- `JAsCo Application/01ChatAWED`: JAsCoDT run configuration included to test the behavior of the chat in the first exercise

- `JAsCo Application/02MessengerAWED`: JAsCo run configuration included to test the behavior of the extended version of the messenger
- `Java Application/03JbossCacheStandard`: standard java run configuration included to test the behavior of the standard version of JbossCache
- `JAsCo Application/03JbossCacheAWED`: JAsCo run configuration included to test the behavior of the refactored version of JbossCache

2.2 Setting up and checking the software configuration

Before getting started you should set and checkthe following parameters for the software configuration.

1. Check that the link to the JVM in the projects is ok, in particular, that it points to a **Java virtual machine version 5**
2. Check that the links to the **JAsCoDT libraries** are ok
3. Verify that the following **VM parameters** are set in the **JAsCoDT run configuration** (`Arguments` tab in the `Run->Run...` menu):
   (a) `-Djasco.group.name=ToBeChanged`
   Each student has to use a different group name: you will communicate with all the hosts in the corresponding group, which means that if equal names are used communication with other instances of the JAsCoDT run time that have been created by other students may lead to unpredictable and maybe undetectable errors.
   (b) `-Djasco.distribution.enabled=true`
   To activate the distributed behavior. Should be part of the default configuration.
   (c) `-Dbind.addr=192.168.19.186`
   (Optional) to determine the network IP address to use. You should not have to change this.

Software versions. This laboratory has been developed and tested in the following Eclipse work environment: Eclipse 3.2, JAsCoDT 1.4.8 (includes the AWED implementation), AspectJ 1.3.1. This software should already be installed in the correct version on the machines you are using.

2.3 Potential problems

You may experience, among others, the following problems with the software environment:

1. Sometimes some **Java processes remain alive** (this is a problem of the JGroups API, a framework we use for group communication). Check that there are not old processes inferring with your program (and, if necessary, terminate them).
2. On Linux, if the application is detecting the **loopback interface** (127.0.0.1) instead of your configured network you can edit the /etc/host file in order to put first the line referencing your ip!.
3. Your network interface must support **multicasting**. This should be the default on your Windows (and almost all Linux) configurations.

3 Warming up: a simple chat client

The objective of this part is to present the **JAsCoDT syntax** and familiarize you with the programming environment.
Detailed description:

1. Find, open and study the project `01Chat`. Its sources are in the `src` folder.

2. Open and study the class `test.Messenger`. This class basically implements a shell that supports three commands represents by three keywords. The pair of keywords `active` and `inactive` change the state of the client (active/inactive), the keyword `Exit` exists the client. The client reads input from the keyboard.

3. **Run the client as a normal Java class**: it should allow you to type text. Try out the three commands!

4. Remember that AWED is implemented on top of JAsCoDT, so you will need to **use JAsCoDT syntax** in order to implement your aspects. JAsCoDT introduces **two main abstractions**:

   - **Abstract hooks** group a pointcut definition and an advice definition. Hooks are defined inside aspect files, each of which is defined similar to plain classes.
   - **Connectors** contain an instantiation of an abstract hook, thus mapping the abstract methods of the aspect/hook definition to concrete methods in concrete classes.

   The next steps present a simple hands-on tutorial to become acquainted with the JAsCoDT syntax.

5. **Write an aspect using remote pointcuts to listen to what is written in remote clients** and print it on the screen. To this end, consider two provided files, `test.MessengerAsp.asp` (the aspect definition) and `test.MessengerCon.con`. These files are templates of JAsCoDT aspects. Modify them in order to achieve the distributed behavior: the files contain instructions of how to complete the exercise.

6. In order to run your application you will have to create a **JAsCoDT run configuration** or use one of the provided ones:

   (a) Open the menu ”Run-¿Run...

   (b) Select or add a new JAsCoDT application configuration

   (c) **Configure the run configuration** (concretely the VM parameters in the argument tab) as desribed in section 2.2.

   (d) Using this configuration **run two instances**. **Start chatting**. You should now have developed a working chat application.

7. You can find the unsolved aspect code in figure 1 and the connector code in figure 2

```
1  package test;
2
3  //README: This file is an example of a simple abstract aspect in Jasco.
   //It contains a hook definition. Remember that a hook definition
5  //contains a pointcut and an advice definition.
6
7  class MessengerAsp {
8
9     //This is the hook definition.
      hook MessengerHook {
11
   // README: This is the abstract pointcut, you have many ways to refer to
13 //  an abstract method:
   //
15 //  - MessengerHook(method(..args1)): refers to a method with any number of arguments.
   //  - MessengerHook(method(String s, Map m), method2(..args2)): refers to a method
17 //  with a String and a Map arguments and a method2 with any number of arguments.
   //  The parameters of first method will be bound to variables s and m so they can be used
19 //  in the advice. Note that here you don't need to indicate the return type.
   //
21 //  You can find more information in JAsCo web site.
   //  http://ssel.vub.ac.be/jasco
23
         MessengerHook(method(..args1)) {
25
   // README: This is the actual pointcut definition. the "execution(method)" pointcut
27 //   matches all the executions to the method "method". Remember that AWED's "Host"
   //   pointcut translates to "joinpointhost" in DJAsCo and AWED's "on" translates
29 //   to "executionhost". Here you will be presented with some valid pointcuts
   //   definitions you will have to choose the correct one:
31 //   a) execution(method) && joinpointhost(localhost)
   //   b) execution(method) && joinpointhost("192.34.56.67:5678")
33 //   c) execution(method) && executionhost(jphost)
   //   d) execution(method) && !joinpointhost(localhost)
35            execution(method) && joinpointhost(localhost);
         }
37
   // README: This is an actual advice definition, in this case a before advice definition,
39 //  that simply prints on the screen the value of the first argument of the method call.
   //  it shows how to use the "thisjoinpoint" variable. Remember that you can check
41 //  the methods of "thisjoinpoint" by checking in the API documentation the class
   //  "DistributedJascoMethod". You don't need to change any thing. Why?
43
         before(){
45          System.out.println("MSJ: " + (String) thisJoinPoint.getArgumentsArray()[0]);
         }
47    }
   }
```

Figure 1: Chat aspect: MessengerAsp.asp

```
static connector MessengerCon {

//README: This is an actual connector that basically instantiates a hook relating it
// to an actual method in an actual class. Here is where the actual methods are referenced
// and bound to abstract aspects, in this case the method "proccessMessage(String)" of
// the class "test.Messenger" with return type "void" is bound to method "method"
// in the aspect.You can have multiple hook instantiations in one connector.
// You don't need to modify any thing. Why?

   test.MessengerAsp.MessengerHook hook0 =
        new test.MessengerAsp.MessengerHook(void test.Messenger.proccessMessage(String));

// This is optional, it will help you to decide what advices to execute and the precedence of
// advices. You don't have to modify any thing.
   hook0.before();
}
```

Figure 2: Chat connector: MessengerCon.con

4 A messenger application

The objective of this part is to present a more complex problem involving **remote sequences** by implementing an extension of the previous exercise. Your aspects should now implement a client that only listens to other clients when it is active.
Detailed description:

1. Open and study the project `02Messenger`.

2. This project contains the class `test.Messenger`. As the chat example the messenger is basically a shell that supports three keywords. The pair of keywords `active` and `inactive` change the state of the client, the keyword `Exit` exits the client. The client reads the input from the keyboard.

3. **Complete** the files `test.MessengerAsp` and **test.MessengerCon**. Hint: note that the active/inactive actions are local actions, you are not interested in the remote active/inactive actions. Have a look at the instructions in the files.

4. In order to run your application you will have to create a **JAsCoDT run configuration** or use one of the provided ones:

   (a) Open the menu "Run-¿Run...

   (b) Select or add a new Jasco application configuration

   (c) **Configure the run configuration** (concretely the VM parameters in the argument tab) as desribed in section 2.2.

   (d) Using this configuration **run two instances**. Start messaging.

5. Figure 3 shows the desired unsolved aspect. Figure 4 shows the desired unsolved connector code.

```
package test;

class MessengerAsp {

   hook MessengerHook {

// README: This is an actual pointcut definition. It has multiple methods
//  and define a sequence, the code in the file is valid but it is
//  incomplete. You will have to complete it. Note that it defines a valid
//  sequence, and remote pointcuts are allowed in sequences.

       MessengerHook(startmethod(..args1),stopmethod(..args3)) {
           start>p1;
           p1: execution(startmethod) > p3;
           p3: execution(stopmethod) > p1;
       }

       before p1 (){
           System.out.println("MSJ: " + (String) thisJoinPoint.getArgumentsArray()[0]);
       }
   }
}
```

Figure 3: Messenger aspect: MessengerAsp.asp

```
static connector MessengerCon {


// This is an actual connector that basically instantiates a hook relating it
// to actual methods in an actual class. The connector is wrong you will have
// to correct it in order to make the example work.

   test.MessengerAsp.MessengerHook hook0 =
        new test.MessengerAsp.MessengerHook(void test.Messenger.StatusActive(*),
                                            void test.Messenger.StatusActive(*));
   hook0.before();
}
```

Figure 4: Messenger connector: MessengerCon.con

5 Refactoring JbossCache

The third part of this lab consists in implementing replication using AWED on top of JBoss Cache.

A brief overview of JBoss replicated caching. JBoss Cache is a replicated transactional cache, *i.e.*, the coherence of the data in the caches forming a common cluster is ensured by guarding updates using transactions. A JBoss cache can either be configured to be local, in which case no data is replicated to other caches on other machines, or it can be global, which means that all changes are replicated to all the other caches (on all other machines) that are part of the cluster.

The replication concern in JBoss Cache is crosscutting: it is scattered over large parts of its implementation and tangled with other scattered concerns. Understanding of the existing code as well as modifications of the standard behavior of JBoss Cache are hindered by this crosscutting.

The main class that represents the data structure used to store data is `TreeCache` (that implements a tree :-)) located in the package `org.jboss.cache`. An instance of this class stores the cache contents on a specific machine.

To separate crosscutting concerns from the main code JBoss Cache uses interception filters, an AOP-like mechanism used to redirect a method call to other functionalities: for example, once a `TreeCache` instance receives a call to the method `put` to store a datum in the cache, a `MethodCall` object is created and passed the object through a chain of filters, each of which treats a crosscutting concern, *e.g.*, replication or transactions. Even though the concept is clean in practice the solution still results in tangled code. The filters are located in the package `org.jboss.cache.interceptors`.

The exercise. This exercise consists of the following steps:

1. The environment for the second exercise is found in the project `03jbosschrfc`.
2. This project contains the binary distribution of JBossCache-1.3.0.SP2 with two additional directories `src` and `bin` (the bin directory is not showed in the java view of eclipse). The libraries needed to run the exercise and the examples are already configured. **Use the `src` directory for your classes and aspects.**
3. Under `src` you find a directory `called META-INF`. This directory contains a default configuration file (`replSync-service.xml`) that can be used to test and illustrate the basic behavior of JBoss Cache.
4. Test the default behavior of JbossCache. Using standard Java, execute the class `org.jboss.cache.TreeCacheView2` (Note that you don't have the source code of this class, it is part of the standard JBoss Cache distribution), you can pass the parameter `-Dbind.address` ,*e.g.*, `-Dbind.addres=198.56.45.67`, to the VM in order to use an specific interface for network communication. There is a ready-to-use **run configuration** that can be used for this:
   (a) Select the eclipse menu "run", under this menu select the option "Run...", then select the "Java application" run configuration `03JbossCacheStandard` (you can change the `-Dbind.address` parameter in the arguments tab).
   (b) Run the program twice: this will create two windows with a root node that is displayed like a hierarchy of directories.

(c) Right click on one of the root nodes and select "Add to this node". In the window that appears enter "/a/b/c/d", press "enter". Note how the Information gets replicated to the other window. This is the basic behavior of a replicated cache.

5. Configure the cache to work **locally**. To this end, open the file `META-INF/replSync-service.xml`, and change `<attribute name="CacheMode"> REPL_SYNC </attribute>` by `<attribute name="CacheMode"> LOCAL </attribute>`.

   Test the cache again, no replication this time!.

6. The objective now is to implement the replication using AWED/JAsCoDT. In the class `TreeCache` are two methods that are of interest to us: the first is `TreeCache.start()`, the other is `org.jboss.cache.TreeCache.put(String fqn, Map m)` (This method is the one invoked by the class `TreeCacheview2`). Using aspects and remote pointcuts, you are going to implement a replication mechanism to replicate the data to cache in other hosts once the method `TreeCache.put(String fqn, Map m)` is invoked. Thus when someone adds a node using the graphical interface of `TreeCacheview2`, your aspect(s) should replicate the call to the other caches.

7. The cflow syntax of JAsCoDT is `cflow(method)`; it can be applied only to abstract methods defined in a hook.

   In JAsCoDT you cannot directly define a cflow that is applied to a method in the same class. You therefore have to define an auxiliary wrapper class which defines the argument method.

8. In order to run your application you will have to create a **JAsCoDT run configuration** or use one of the provided ones:

   (a) Open the menu `Run->Run...`

   (b) Select or add a new Jasco application configuration

   (c) **Configure the run configuration** (concretely the VM parameters in the argument tab) as desribed in section 2.2.

   (d) Using this configuration **run two instances**. Start replicating.