Migrating Existing Applications to a Distributed Environment using Aspect Oriented Programming

Mariano Ceccato and Paolo Tonella
ITC-irst
Centro per la Ricerca Scientifica e Tecnologica
38050 Povo (Trento), Italy
{ceccato, tonella}@itc.it

Abstract

Aspect Oriented Programming (AOP) is a new programming paradigm that offers a novel modularization unit for the crosscutting concerns. Functionalities originally spread across several modules and tangled with each other can be factored out into a single, separate unit, called an aspect.

The source code fragments introduced to port an existing application to a distributed environment (such as Java RMI) are typically scattered and tangled, thus representing an ideal candidate for the usage of aspects. In this paper, we propose a distribution framework based on AOP and we describe the steps necessary to migrate an existing program to it. In our solution, the original application remains oblivious of the distribution concern and all required aspects are generated automatically. The approach was validated on a case study.

1 Introduction

Making an application run in a distributed environment involves several small modifications that are typically spread all over the code. For example, adoption of the RMI (Remote Method Invocation) distribution framework for the Java programming language affects the way objects are created and accessed, and the way methods are called. In fact, in RMI objects accessible from other hosts must be registered in the RMI registry upon creation. Correspondingly, when they are accessed from another host, a remote reference to them must be obtained from the RMI registry. Then, remote method invocation can occur, provided that the exceptions possibly due to the remote nature of the invocation are properly handled.

A consequence of the code changes sketched above is that the original software is adversely affected by the new distribution functionality. In fact, the size and complexity of the original software increase. Moreover, code fragments related to a same functionality (i.e., distribution) are inserted at different places. Maintaining a traceability map to them is a non trivial task. From the point of view of the code understandability and maintainability, a negative impact can be also hypothesized. In fact, the core functions must be understood and maintained together with the distribution code fragments.

Aspect Oriented Programming (AOP) aims at providing a modularization unit for the functionalities that cannot be easily localized with the chosen programming style (e.g., in a single class, with Object-Oriented Programming, OOP). Code fragments devoted to a same function, but scattered across several different modules, become an aspect in AOP and can be implemented as a single, separate module, an aspect.

Distribution is a typical candidate for an AOP implementation [1, 4, 8, 9]. Distribution code can be assigned to an aspect, thus taking it apart from the original application.

In this paper, we consider the problem of migrating an existing application to a distribution environment by means of AOP. The solution described in this paper has several interesting properties, such as the obliviousness of the original code with respect to the distribution functionality and the configurability of the network topology. Migration is achieved by means of a static code analyzer and generator, which automatically produces all the aspects necessary to implement the distribution concern.

The proposed approach has been applied to a medium size application. The clear separation of the distribution from the other functionalities simplified the successive evolution of this software, aimed at parallelizing the core processing algorithm.

After reviewing RMI and its adoption in traditional OOP (Section 2), our AOP distribution framework is presented (Section 3). The source code analyzer and generator developed to support the migration process is described in Section 4. Our experience with the case study is reported in Section 5, followed by a discussion of the related works (Section 6) and by our conclusions (Section 7).

2 Distributed programming with RMI

A distributed program is a program that uses resources (e.g., memory, storage, processors, etc.) located at different hosts. There are many reasons why an application needs be distributed, such us balancing the computing load across the hosts, in order to increase performance, or bringing the computation near the required resources, in order to minimize the network throughput and delay. Among the middleware libraries available to support distributed programming in Java, we will focus on RMI.

2.1 RMI support

The standard Java library supports the development of distributed programs by providing a standard Remote Method Invocation (RMI) mechanism. RMI is a distributed programming framework that enables the invocation of methods on objects resident on different virtual machines, possibly running on different hosts.

RMI provides a middleware layer that takes care of handling communication and synchronization details, such as the network protocol, object serialization and deserialization, concurrency. These facilities make the use of remote objects very simple, because these can be accessed and invoked in a way very similar to the local invocations.

RMI uses object serialization to marshal the dynamic structure of the objects into byte streams that can be sent over a serial network link. In this way, all the parameters involved in a method invocation can be transported. On the other side, the received data stream is used to restore the serialized objects and to execute the code of the remotely called methods. Such a serialization mechanism is completely transparent to the programmer, who must only ensure the "serializability" of each parameter (i.e., implementation of *Serializable* interface).

Each remote object can be distinguished from the others by name. In fact, a remote object is remotely invokable only if it has been exported in the RMI naming registry and if it has been bound to a name. The responsibility of the naming registry is to respond to lookup requests, by returning a reference to the object bound to a given name.

RMI distinguishes two roles: server and client. The *server* provides remotely invokable methods, whereas the *client* invokes them (of course, hosts may play both roles at the same time, for different objects). On the server side, not all the methods in a remote object are exposed for remote invocation, because some of them are private or have local purpose. The remote visibility of each object involved in RMI is declared by a particular interface, the *remote interface*, that contains the specification of all the methods for which the object can receive invocations from distant peers.

The client role consists of performing a remote invocation on a remote object. For that, the client needs to know how to reach the remote object and which methods can be invoked on it. The client code inquires the naming service to obtain a reference to the required remote object. In order to know which methods can be remotely invoked, the client uses their specification in the remote interface.

2.2 Adding RMI support

Although RMI invocations are made in the same way as normal invocations, programmers who use RMI must be aware of the involved roles, because of collateral issues, such as type casting and exceptions handling. In fact, before a remote object can be used, the name to which it is bound must be known. Once a remote reference to the required object has been obtained, a particular exception handling protocol has to be respected in order to perform the remote invocation, because of the problems that can occur in the invocation, such as network failure, server down, virtual machine or library release mismatch, etc. The RMI framework manages all these problems, hiding the details under a generic *RemoteException*.

When the remote method invocation capability is added to an existing application, two different steps must be taken. The first concerns the production of new interfaces, while the second is related to some code transformations.

The first step consists of deciding which services to include in the remote interface. This special interface extends *Remote* and must be implemented by the class of the object being remotized. Each of the remote methods declared in this interface must add *RemoteException* to the *throws* clause (see Figure 1, interface `AR` and class `A`). Moreover, the class being remotized must extend *UnicastRemoteObject*.

Once created, a remote object instance must be exported in order to be visible from the other hosts. This entails the registration of the object in the *RMI registry*, obtained by calling the method *rebind* of the static class *Naming* (see Figure 1, body of method `server` inside class `Server`). When the object becomes unavailable on a given host, it must be un-registered from the *RMI registry*.

On the other side, a client can obtain a reference to a remote object by means of the *lookup* facility offered by the class *Naming* of the RMI framework. Object identification is by name, with names corresponding to the identifiers used at the server side. An explicit type cast is necessary to convert the remote reference to the remote interface implemented by the referenced object (see Figure 1, body of method `client` inside class `Client`). While the original

```
class A {
  private int x, y;
  public void f() {...}
  public void g() {...}
}

class Server {
  public static A a1, a2;
  public void server() {
    a1 = new A();
    a2 = new A();
  }
}

class Client {
  public void client() {
    A a1 = Server.a1;
    A a2 = Server.a2;
    a1.f();
    a2.g();
  }
}
```

$\Longrightarrow$

```
interface AR extends Remote {
  void f() throws RemoteException;
  void g() throws RemoteException;
}

class A extends UnicastRemoteObject
        implements AR {
  private int x, y;
  public void f() throws RemoteException {...}
  public void g() throws RemoteException {...}
}

class Server {
  public static A a1, a2;
  public void server() {
    try {
      a1 = new A();
      Naming.rebind("//host1.itc.it/a1", a1);
      a2 = new A();
      Naming.rebind("//host1.itc.it/a2", a2);
    } catch (Exception e) {...}
  }
}

class Client {
  public void client() {
    try {
      AR a1 = (AR)Naming.lookup("//host1.itc.it/a1");
      AR a2 = (AR)Naming.lookup("//host1.itc.it/a2");
      a1.f();
      a2.g();
    } catch (Exception e) {...}
  }
}
```

Figure 1. *Objects* `a1` *and* `a2` *are made remotely accessible.*

client code contains local variables, fields and parameters whose type is the class of the remote objects, the new code must replace such type with the corresponding remote interface (`AR` replaces `A` inside client code, in Figure 1).

Once a remote reference has been correctly obtained, remote methods can be invoked on it inside a try-catch block for the possibly raised exceptions (such as *RemoteException*). The last lines of the code of method `client` in Figure 1 are such an example.

On remote objects, clients can only issue method calls, since field access is not allowed by the RMI framework. Thus, access to public fields must be converted into the invocation of setter/getter methods, before migrating to a distributed version of the code. Moreover, in a remote method invocation, parameters are always passed by value (objects are automatically serialized). When an object needs be passed by reference, a remote reference, obtained by means of the *lookup* function, must be passed instead of the original object. Its actual, dynamic type will be a subclass of *RemoteStub*, automatically generated by RMI.

3 Distribution as an aspect

Aspect Oriented Programming (AOP) [5] is a new programming paradigm, with constructs explicitly devoted to handling crosscutting concerns. In an Object-Oriented system, it often happens that functionalities, such as persistence, exception handling, error management, logging, are scattered across the classes and are highly tangled with the surrounding code portions. Moreover, the available modularization/encapsulation mechanisms fail to factor them out. Aspects have been conceived to address such situations. AOP introduces the notion of *aspect*, as the modularization unit for the crosscutting concerns. Common code that affects distant portions of a system can be located in a single module, an aspect. The aspect compiler takes care of weaving it with the affected code to produce the executable.

Aspects interact with the given code by means of two main mechanisms: pointcuts and introductions. Pointcuts intercept the normal execution flow at specified join points. Aspect code (called *advice*) can be executed either before, after or instead of (around) the intercepted join point. Introductions modify properties of the classes they affect by adding methods or fields and by making them implement interfaces or specialize super-classes previously unrelated with them.

As described in the previous section, introduction of distribution (and in particular of RMI) into an existing application involves a code modification which is spread system-wide and is intertwined with the original code. Thus, it is an ideal candidate to be best expressed by means of aspects. In this section, our approach to distribution as an aspect is presented, with reference to the AOP language AspectJ [6]. In summary, it involves the following elements:

Remote interfaces: automated creation of the interfaces required by RMI for remote objects.

Object factory: transparent creation of remote objects.

Method invocation: transparent invocation of methods on remote objects.

Parameter passing: transparent management of local/remote objects passed as parameters of method calls.

Exception handling: transparent management of exceptions raised by RMI.

3.1 Remote interfaces

```
class A {
  private int x, y;
  public void f() {...}
  public void g() {...}
}

interface A_Remote extends Remote {
  public void f_remote() throws RemoteException;
  public void g_remote() throws RemoteException;
}

privileged aspect A_RMIAspect {
  declare parents: A implements RemoteClass;
  declare parents: A implements A_Remote;
  public void A.f_remote() throws RemoteException {}
  public void A.g_remote() throws RemoteException {}
}
```

Figure 2. *Automatically generated interface for a given class* A *and related RMI aspect.*

To make a class accessible from remote hosts, an interface declaring all its public methods as remotely accessible is automatically generated. The implementation of the methods in the remote interface is obtained by means of aspect introductions.

In order to introduce distribution without modifying the original code, a solution slightly different from that shown in Figure 1 is adopted. Instead of adding a *throws* clause to each public method of the original class (as in Figure 1), fresh method names are generated, by concatenating the original names with `"_remote"` (see Figure 2, middle). These new methods include the necessary *throws* clause and are declared in the interface being generated (*A_Remote*, extending *Remote*).

The aspect *A_RMIAspect* (see Figure 2, bottom) declares that class *A* implements the interface *RemoteClass*. This requires no new method introduction and will be used to distinguish normal from remote classes upon object creation and method invocation (see pointcuts described below). Moreover, the aspect *A_RMIAspect* declares that class *A* implements the automatically generated interface *A_Remote*. This requires the introduction of methods *f_remote, g_remote* into class *A*. Their body is empty, since their execution is always intercepted by an aspect (described below) that redirects the control flow to the original methods. It should be noted that the aspect *A_RMIAspect*, as well as the interface *A_Remote*, are generated automatically after a simple static analysis of the code of class *A*. Differently from the transformed code in Figure 1, the AOP solution leaves the original code unchanged.

3.2 Object factory

```
public aspect ConstructorInvocation {
  static ObjectFactory objectFactory = ObjectFactory.getInstance();
  pointcut constructorInvocation(): call(RemoteClass+.new(..));

  Object around(): constructorInvocation() {
     ... // uses reflection to get className, signature, args
     return objectFactory.createObject(className, signature, args);
  }
}

class ObjectFactory extends UnicastRemoteObject
                    implements ObjectFactory_Interface {
  private static ClassFactory factoryInstance;
  protected ObjectFactory() throws RemoteException {
    ...
    factoryInstance = this;
    Naming.rebind("//"+ thisHostName + "/ObjectFactory", this);
  }
  public static ObjectFactory getInstance()
                        { ...return factoryInstance; }
  public Remote createObject(String className, Class[] signature,
                             Object[] args) {
    String host = hostAssignment(className);
    if (host.equals(thisHostName)) {
      return createLocalObject(className, signature, args);
    } else {
      ObjectFactory_Remote factory = (ObjectFactory_Remote)
        Naming.lookup("//" + host + "/ObjectFactory");
      return factory.createObject(className, signature, args);
    }
  }
  private synchronized Remote createLocalObject(String className,
                                Class[] signature, Object[] args){
    Constructor constructor = getConstructor(className, signature);
    Remote newInstance = (Remote) constructor.newInstance(args);
    if (!(newInstance instanceof UnicastRemoteObject))
       exportObject(newInstance);
    String reference = "//" + thisHostName + "/Instance" + count++;
    Naming.rebind(reference, newInstance);
    return newInstance;
  }
}
```

Figure 3. *Aspect* ConstructorInvocation *and class* ObjectFactory.

When objects of remote classes are created, it is necessary to insert them into the RMI registry, to make them accessible from other hosts. Once again, an AOP solution avoids changing the original code (differently from Figure 1). It can be obtained by means of the powerful features of the pointcuts.

Object creation is intercepted by a pointcut in a general purpose aspect (*ConstructorInvocation*). The associated advice manages the transparent creation of remote objects. Its (simplified) code is provided in Figure 3. This aspect has a static field (*objectFactory*) referencing a singleton instance of the class *ObjectFactory*, which has the responsibility of creating and registering remote objects.

The pointcut *constructorInvocation* intercepts all calls to constructors of classes implementing the interface *RemoteClass* (syntax: *RemoteClass+.new(..)*). Classes that can be assigned to different hosts (remote classes) are modified by the aspect shown in Figure 2, that makes them implement this interface, so that any invocation of their constructors is intercepted by this pointcut.

The code executed instead of the original constructor is specified in an around advice associated with the pointcut *constructorInvocation* (see Figure 3, top). It delegates object creation to the factory referenced by *objectFactory*.

Figure 3 (middle/bottom) shows the (simplified) code of the class *ObjectFactory*. The constructor of this class is not public and the unique instance of this class can be obtained by means of the static method *getInstance* (singleton design pattern [3]). Its protected constructor inserts such an instance into the RMI registry, so that it can be accessed remotely. A singleton instance of *ObjectFactory* is created and run on every host (virtual machine) in the given network topology.

Method *createObject* is responsible for the creation of an object of a given class (parameter *className*) and for its insertion into the RMI registry for remote access. We assume that *ObjectFactory* knows the topology of the distributed application, that is, the name of the host on which each remote class resides. This information can be provided externally, in a configuration file. If the class of the object being created is assigned to the current host, the object is created locally by calling *createLocalObject*. When the given class is assigned to a host other than the current one, object creation is delegated to the object factory running on such a host. A reference to the remote factory is obtained by means of the *lookup* RMI facility. The remote reference, of type *Remote*, produced by the remote factory is then returned.

Local object creation is managed by method *createLocalObject*. It exploits Java reflection to obtain a *Constructor* object, by which a new instance of the given class (parameter *className*) can be generated. Such a new instance is ensured to implement the interface *Remote* (see remote interface implementation in Figure 2). However, it may be not always possible to make the given class extend *UnicastRemoteObject*, because the class may belong to a hierarchy that cannot be modified by an aspect introduction, to preserve the correctness of the program. In such cases, the method *exportObject*, inherited from *UnicastRemoteObject*, can be used to wrap seamlessly the new instance into an object of proper type. Then, the newly created instance is registered into the RMI registry and is bound to an automatically created identifier (*"Instance"* suffixed by an incremented integer).

3.3 Method invocation

```
public aspect MethodInvocation {
  static ObjectFactory objectFactory = ObjectFactory.getInstance();
  pointcut methodInvocation(RemoteClass obj):
     execution(* RemoteClass+.*(..)) && target(obj);
  pointcut remoteMethodExecution(Object obj):
     execution(* RemoteClass+.*_remote(..)) && target(obj);

  Object around(RemoteClass obj): methodInvocation(obj) {
    // uses reflection to get className, signature, args
    String newMethodName = signature.getName().concat("_remote");
    Method method = objectFactory.getMethod(className,
                                    newMethodName, signature);
    return method.invoke(obj, args);
  }

  Object around(Object obj): remoteMethodExecution(obj){
    // uses reflection to get className, methodName, signature, args
    String origMethodName = methodName.substring(0,
                              methodName.indexOf("_remote"));
    Method method = objectFactory.getMethod(className,
                                    origMethodName, signature);
    return method.invoke(obj, args);
  }
}
```

Figure 4. *Aspect* MethodInvocation.

Method invocations on remotized objects are intercepted by the aspect *MethodInvocation* (shown in Figure 4) and replaced by a remote invocation, respecting the RMI protocol. On the remote host, the remote peer of this aspect intercepts all the executions of remote methods (suffixed by `"_remote"`) and delegates the computation to the original methods.

The pointcut *methodInvocation* intercepts the invocation of any method, with any signature and any return type, of all classes implementing the interface *RemoteClass* (syntax: * *RemoteClass+.*(..)*). In this way, the original code remains unchanged, but its execution is intercepted by the aspect *MethodInvocation*, that takes care of redirecting the call to a remote object, each time a *RemoteClass* is involved (all classes that can be moved to a different host implement this interface, thanks to the aspect shown in Figure 2). The parameter *obj* of this pointcut is bound to the target of the invocation.

The around advice executed when the pointcut *method-*

Invocation is triggered (see Figure 4) creates a new method name by concatenating the original name with `"_remote"`. The resulting name is declared in the remote interface of the given class (see above), so that its invocation is handled by RMI as a remote invocation (last line of code of the advice).

On the remote host, the pointcut *remoteMethodExecution* is triggered each time a remote method is executed. In fact, this pointcut constrains the method name to end with `"_remote"` (see Figure 4, top).

The around advice executed when a remote method has been called redirects the execution to the original method. This is achieved by restoring the original method name (see Figure 4, bottom), removing the suffix `"_remote"`, and invoking it.

3.4 Parameter passing

```
class ObjectFactory extends UnicastRemoteObject
                    implements ObjectFactory_Interface {
  ...
  private Map localMap; // RemoteStub -> Object
  public Object wrap(RemoteStub remoteStub) {
    // uses reflection to get className from remoteStub
    Class[] signature = {RemoteStub.class};
    Object[] args = {remoteStub};
    Constructor constructor = getConstructor(className, signature);
    return constructor.newInstance(args);
  }
  public Remote unwrap(RemoteStub remoteStub){
    return (Remote)localmap.get(remoteStub);
  }
}

privileged aspect A_RMIAspect {
  ...
  private RemoteStub A.remoteStub = null;
  public A.new(RemoteStub remoteStub) {
    this.remoteStub = remoteStub;
  }
  public RemoteStub A.getRemoteStub() {
    return remoteStub;
  }
}
```

Figure 5. *Local and remote parameters.*

The parameter passing mechanism implemented by RMI is by-value for simple types and by-serialization or by-remote-reference for objects. For simple types, passing the parameters by value clearly preserves the behavior of the given application. For objects, serialization may lead to incorrect behaviors, for example, when classes do not support serialization or when serialization is not appropriate, since the object parameter is expected to be shared. In these cases, parameter passing must be by remote-reference. Conservatively, we apply this option to all objects passed as invocation parameters.

When an object parameter is passed as a remote reference to a remote method or is returned from a remote method, it cannot be used directly in the original code, because of a type mismatch. In fact, its dynamic type is an automatically generated subclass of *RemoteStub* instead of the original class. Similarly to the original class, this subclass of *RemoteStub* implements the remote interface declared for the remotized class. However, usage of such an interace would require changing the original code, which we want to leave untouched. Our solution to this problem consists of creating a fake object of the original class, which contains an additional field, named *remoteStub*, storing the remote reference. Each time a remote reference parameter or return value is received, it is wrapped into a fake object of the original class. The aspect *A_RMIAspect* (see Figure 5, bottom) adds the field *remoteStub*, of type *RemoteStub*, to the original class. A new constructor, also introduced by this aspect, assigns the remote reference to such a field.

Object parameters are wrapped inside the advice that surrounds the execution of the called method, when these objects are resident on a host different from that of the target of the invocation. Similarly, the advice that surrounds the method call wraps the returned object (if any). The *wrap* operation is implemented by the class *ObjectFactory* (see Figure 5, top). It inserts the remote reference associated with the given object parameter into the *remoteStub* field, by calling the newly introduced constructor with the *RemoteStub* as parameter. Any successive call on this object can be redirected to its field *remoteStub* by the *MethodInvocation* aspect (see Figure 6), by means of the following statement, added to the code in Figure 4 (advice around the pointcut *methodInvocation*):

```
Remote realObj = obj.getRemoteStub();
...
return method.invoke(realObj, args);
```

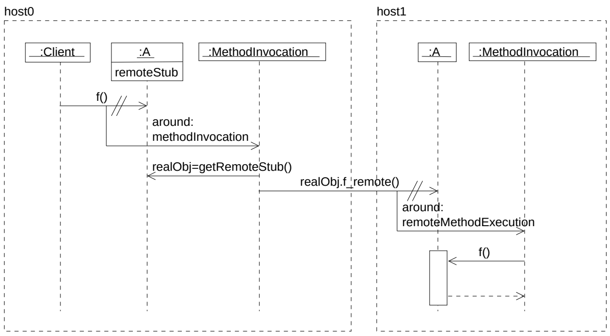

Figure 6. *Interactions during method invocations.*

It should be noted that a particular case where wrapping is required is the creation of a new object by a remote object factory. With reference to the *createObject* method in Figure 3, when the factory is remote (RMI lookup is used for the factory), the object being returned must be wrapped, to be consistent with the type declared in the original code:

```
return factory.wrap(
   factory.createObject(className, signature, args));
```

When a remote reference is passed as a parameter to a host that is the same where the remote object is resident, an *unwrap* operation is to be performed. Such an operation (see Figure 5) exploits a private field of class *ObjectFactory*, called *localMap*, which stores the correspondence between remote references generated by RMI and local objects allocated by the *ObjectFactory*. Unwrapping a given *RemoteStub* accounts just for retrieving the associated local object from *localMap*. In this way, the actual, local object is used in the original code.

Management of the local map requires that before returning from *createLocalObject* (see Figure 3), the following statements be executed:

```
RemoteStub remoteStub = Naming.lookup(reference);
localMap.put(remoteStub, newInstance);
```

3.5 Exception handling

Try-catch blocks associated with RMI exceptions have been omitted for clarity in Figures 3, 4. However, they can be easily added to the aspect code, without any need to change the original code. In fact, all invocations of remote methods are inside aspects (Figure 4) and all invocations of constructors are inside class *ObjectFactory* (Figure 3). Once they are surrounded by proper try-catch blocks, they can handle locally the RMI exceptions possibly raised, leaving the original code unaffected.

3.6 Discussion

The approach described above to migrate an existing single-processor application to a distributed environment enjoys several interesting properties.

The refactored code is completely *oblivious* of the distribution concern, since the existing code remains unchanged and only new code (for the aspects and infrastructure classes) is added. This means that it continues to be possible to run exactly the original application in a single processor. When the application is distributed across multiple hosts, it is sufficient to compile it with the distribution aspects.

The actual topology of the network of hosts on which the application is deployed is completely configurable without any change in the distribution code. In fact, the assignment of classes to hosts is specified in a configuration file that is accessed by each object factory. This gives a great flexibility in trying and changing the distribution of the objects over the network.

The complete obliviousness of the given application with respect to the distribution concern is expected to have remarkably positive effects on the understandability and evolvability of the software.

The code pertaining to the distribution concern is not spread in each involved class, being located in a single place. The program comprehension effort required to understand such a concern is correspondingly simplified, since this concern can be dealt with separately from the others. Benefits are apparent also for the core functionalitites of the given application, since understanding all the other features is not complicated by the presence of the code related to distribution. Moreover, most of the code of the proposed distribution framework (class *ObjectFactory*, aspect *MethodInvocation*, etc.) is fixed and can be included in a library. Only small code fragments are generated specifically for the application being migrated (interface *A_Remote*, aspect *A_RMIAspect*).

Evolution is also expected to be affected positively, since the application does not depend on its aspects. When management of the distribution concern changes, the original application (oblivious of distribution) is unaffected and the change remains local to the distribution aspects and classes.

4 Tool support

The classes and some of the aspects in the proposed distribution framework do not depend on the particular application being migrated, thus they are provided as a library. Production of application-specific aspects and remote interfaces is obtained by means of a code generation tool we developed.

4.1 Code Generation Tool

Before generating application-specific code, the topology of the distributed system has to be specified. This clarifies which classes need a remote interface, in order for their methods to be invokable from code running on distant hosts.

For each class to be remotized, all its super-classes must be made remote as well, because inherited methods have to be also changed into remote methods.

The tool for the automatic generation of the remotization code consists of a program written in TXL [2] for the AspectJ programming language. We extended the Java grammar distributed with TXL to cover the additional constructs introduced by AspectJ.

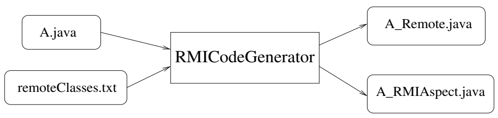

Figure 7. *Code generation tool.*

The tool is run on each class to be remotized, taking in input the source code of the given class and the list of the remote classes. Such a list can be produced automatically, once the topology of the classes (i.e., their distribution among the hosts) has been decided. In fact, a class is classified as a remote class if it is referenced inside code assigned to a different host. This can be determined by a static code analysis (yet to be implemented).

The list of the remote classes is required because the code generation program modifies return and parameter types in remote methods (i.e., methods declared in the remote interfaces), in all the cases where such types correspond to remote classes, to be handled using remote references.

Each remote interface automatically generated by our tool contains the definitions of the remote version of all the public class methods, with altered return type and signature (if necessary) and with a new *RemoteException* in the *throws* clause. The new method names are built by appending the suffix *"_remote"* to them.

Each automatically generated RMI aspect makes the associated class implement its remote interface, by introducing all required remote methods and by declaring the implementation of the interface. It also adds some methods and a field to realize the wrapping mechanism. Eventually, the aspect makes the class implement the *RemoteClass* tagging interface, to let the *MethodInvocation* aspect identify it as a remote class and properly detect invocation pointcuts on it.

5 Case study

The proposed approach has been applied to the Java program FreeTTS (http://freetts.sourceforge.net), a speech synthesis system written in the Java programming language. FreeTTS was developed by Sun Microsystems, based upon Flite, a small, fast, run-time speech synthesis engine, which in turn is based upon University of Edinburgh's Festival Speech Synthesis Sytem and Carnegie Mellon University's FestVox project.

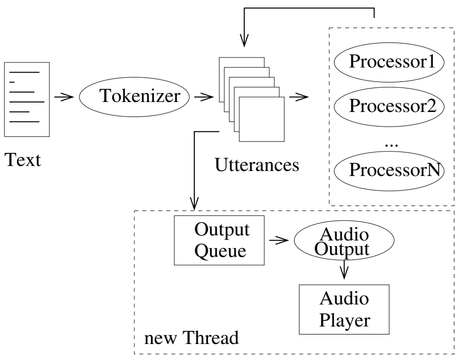

Figure 8. *Architecture of* FreeTTS.

FreeTTS is a medium size application, consisting of around 31k LOC (Lines Of Code) and 173 classes. Its high level architecture is depicted in Figure 8. The text to be spoken is divided into utterances by the function *tokenize*. Utterances contain information about the phones in the text and the wave forms under construction. Such data are incrementally modified by a set of utterance processors applied sequentially (see pseudocode in Figure 9). In the end, the resulting utterances are sent to an output queue. A separate thread dequeues utterances ready for audio output and sends them to the audio player.

```
main
for each u: Utterance
    for each p: UtteranceProcessor
        runProcessor(p, u)
    end for
end for
```

Figure 9. *Pseudocode for utterance processing.*

Utterance processing and audio output are realized by classes which implement the *UtteranceProcessor* interface. Each of them operate on the *features* and *relations* hold inside objects of class *Utterance*. Examples of features associated to each utterance are the *Linear Predictive Coding (LPC) result* and the *units*, eventually concatenated into the LPC result. Examples of relations are the *segment* (ordered list of the smallest units of speech, typically phonemes), the *syllable* and the *phrase*.

Utterance processors can be sharply divided into two groups: those operating directly on the wave form (i.e., on utterance features such as units and LPC result) and those operating on the phones extracted from the text (e.g., on segments, syllables, phrases). For example, the utterance processor *Segmenter* creates the relations *syllable*, *syllable structure* and *segment*. The processor *Durator* annotates the relation *segment* with a cumulative "end time". The *Intonator* annotates *syllable* with "accent", while *PauseGenerator* adds pause information. Examples of processors working on the wave form are the *ClusterUnitSelector* and the *DiphoneUnitSelector*, that create the *unit* relation. *UnitConcatenator* concatenates the units in the given *Utterance* to the LPC result. *AudioOutput* sends the LPC result to the audio player.

Since phone processors and wave processors require different resources and work on different parts of the utterances, a distributed version of this application may consist of two hosts, devoted respectively to phone processing and wave processing. The latter includes the final audio output, so the host for wave processing must be the one with the actual audio device. Phone processing might require heavy computations, so a properly dimensioned host should be allocated for that.

The classes composing the FreeTTS application have been split into two groups, corresponding to the two hosts

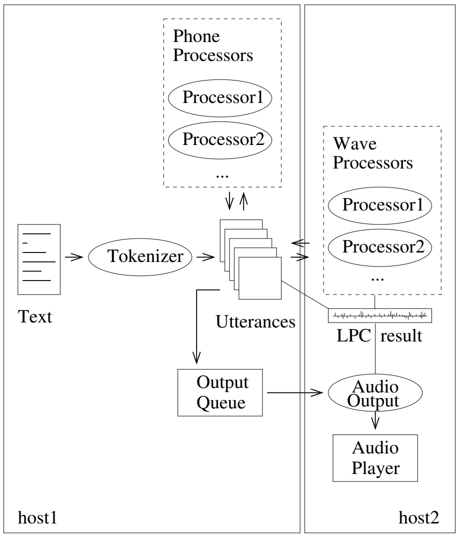

Figure 10. *New architecture of* FreeTTS.

```
main
1  entry := newSyncQueue()
2  out := entry
3  for each p: UtteranceProcessor
4       in := out
5       out := newSyncQueue()
6       runThread(p, in, out)
7  end for
8  for each u: Utterance
9       enqueue(entry, u)
10 end for

runThread(p: UtteranceProcessor, in: Queue, out: Queue)
11 start new thread
12      loop
13           u := dequeue(in)
14           runProcessor(p, u)
15           enqueue(out, u)
16      while (u ≠ null)
17 end start
```

Figure 11. *Parallel utterance processing.*

(see Figure 10). All classes related to utterance creation and phone processing have been assigned to the first host. These include the classes *Utterance* and *OutputQueue*. Utterance processors working on the wave form and strictly related classes are assigned to a second host. They can be easily identified as all the classes belonging to the four subpackages *audio, relp, clunits, diphone*, plus a class from the subpackage *util* (*WaveUtils*).

Once the topology of the distributed application is defined, migration to the new distributed environment accounts just for the automated generation of the remote interfaces and of the aspects required by the remote classes. The original classes remain untouched. Moreover, the general purpose classes (e.g., the *ObjectFactory*) and aspects of the distribution framework are made available in a library, in order for the system to work.

Actual generation of the audio output for a same text in the original and in the distributed environment confirmed the preservation of the speech synthesis functionality after migration. Further advantages (in terms of performances) were obtained in the new, distributed environment by running the different utterance processors in parallel. This required a manual intervention on the code, to make different processors run on separate threads. Figure 11 shows the parallelized pseudocode. Each processor is assigned to a different thread, and communicates with the other processors by means of two synchronized queues, an input and an output queue.

Utterances are enqueued in the entry queue (lines 8-10). The first processor dequeues each utterance from such a queue and enqueues the modified utterance into its output queue, which is the input queue for the second processor. Then, the first processor can dequeue the second utterance and, at the same time, the second processor can work on the first utterance. After an initial transition, all processors will have a non empty input queue, so that all of them can proceed in parallel.

Since phone processing and wave processing are allocated to separate hosts, the parallelism among processors results in the possibility of actually concurrent executions. An initially sequential, mono processor application has been transformed into a distributed, concurrent application. Thanks to the proposed AOP solution, the only necessary code changes are related to parallelization, while distribution was introduced obliviously.

6 Related work

Distribution and concurrency have been considered candidate features for AOP in several works [1, 4, 7, 8, 9, 10].

The authors of [9] report their experience using AspectJ to implement distribution and persistence in a real Web information system. They argue that the AspectJ implementation is superior to the pure Java implementation. Similarly to us, one of their goal was implementing distribution without changing the original code.

Although tailored to a specific application, the solution proposed in [9] has several similarities with ours. Implementation of remote interfaces is declared by an aspect. Remote calls are intercepted and redirected to remote references. In our approach, instead of enumerating explicitly the remote calls to redirect, we identify them by means of reflection. The need of a new static crosscutting mechanism in AspectJ was devised in [9], to support the introduction of *throws* clauses. Our framework would benefit from it as well.

An AOP solution to the synchronization problem in concurrent applications was investigated in [1, 4, 8]. In all these works the synchronization concern is factored out in a separate unit, treated as an orthogonal view on the system's functionalities. Synchronization is one of the aspects mined and refactored in the middleware applications studied in [10].

The application server JAC (http://jac.objectweb.org/) supports distribution following an AOP approach. The original application remains oblivious of the distribution aspect, which is specified in a separate configuration file.

The main difference of our approach with respect to the available solutions is that we focus on the migration problem (instead of the development from scratch) and we insist on a fully automated transition, which leaves the original code oblivious of the distribution concern.

In [7] distribution is added to the AspectJ language itself. The notion of remote pointcut is proposed, aimed at intercepting join points located on multiple hosts.

7 Conclusions and future work

In this work, we have proposed a method for introducing the distribution concern into an existing application using the modularization mechanisms provided by Aspect Oriented Programming.

In traditional code, using the RMI framework to implement a distributed environment requires the insertion of code fragments that are spread in all the participating classes. In each of these classes, the distribution code is tangled and confused with the statements that handle the other application responsibilities. In such a situation, future maintenance activities can be very hard, because the code is difficult to understand and change impact may be difficult to predict.

The main advantage of our proposal is that the evolved application code remains unchanged, being oblivious of the new concern. In fact, the already existing Object Oriented code has no references to the Aspect Oriented code just introduced. This permits adding the new feature without complicating the existing source code.

All the code dealing with this concern is modularized in an isolated and distinguishable part, so any future evolution and maintenance activity can be performed on the concern without affecting the rest of the code. The verification stage has some benefits as well, because we can limit testing to the modified aspects only.

Apart from the current limitations of our tool (remote class list construction), the proposed solution is fully automated. Once the distributed topology of the given application has been decided, the migration step accounts just for running our code generator utility. All necessary aspects and interfaces are added to the existing code, which can be compiled and run in the new environment.

The proposed technique has been validated on a case study. We have applied the distribution concern to a medium size application and we have been able to split and distribute its functionalities among different hosts with a very limited effort, although our initial knowledge of the code was null.

In our future work, we will consider how to support the task of adding other concerns (such as persistence, security, etc.) to existing applications. We will evaluate how much of the involved transformations can be automated and what benefits can be achieved by adopting an AOP approach to the migration problem.

References

[1] T. E. Constantinos A. Constantinides, Atef Bader. An aspect-oriented design framework for concurrent systems. In *Proc, of the 4th International Workshop on Component-Oriented Programming (WCOP'99), in conjunction with the 13th European Conference on Object-Oriented Programming (ECOOP)*, June 1999.

[2] J. Cordy, T. Dean, A. Malton, and K. Schneider. Source transformation in software engineering using the TXL transformation system. *Information and Software Technology*, 44(13):827–837, 2002.

[3] E. Gamma, R. Helm, R.Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object Oriented Software*. Addison-Wesley Publishing Company, Reading, MA, 1995.

[4] D. Holmes, J. Noble, and J. Potter. Aspects of synchronisation. In *Proc. of the 25th Conference on Technology of Object-Oriented Languages and Systems (TOOLS-25)*, pages 2–14, Melbourne, Austrailia, November 1997. IEEE Computer Society.

[5] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect oriented programming. In *Proc. of the 11th European Conference on Object Oriented Programming (ECOOP), vol. 1241 of LNCS*, pages 220–242. Springer-Verlag, 1997.

[6] I. Kiselev. *Aspect-Oriented Programming with AspectJ*. Sams Publishing, Indianapolis, Indiana, USA, 2002.

[7] M. Nishizawa, S. Chiba, and M. Tatsubori. Remote pointcut – a language construct for distributed AOP. In *Proceedings of the International Conference on Aspect-Oriented Software Development (AOSD)*, pages 7–15. ACM Press, March 2004.

[8] R. Ramirez and A. E. Santosa. An aspect-oriented framework for concurrent applications. In *Proc. of the 3rd German Workshop on Aspect Oriented Software Development*, Essen, Germany, March 2003. German Informatics Society.

[9] S. Soares, E. Laureano, and P. Borba. Implementing distribution and persistence aspects with AspectJ. In *Proc. of the 17th Annual ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 174–190, Seattle, Washington, USA, November 2002. ACM press.

[10] C. Zhang and H.-A. Jacobsen. Quantifying aspects in middleware platforms. In *Proc. of the 2nd International Conference on Aspect-Oriented Software Development (AOSD)*, pages 130–139, Boston, Massachusetts, USA, March 2003. ACM press.