Patterns of Mobile Computation in Java

André R. Du Bois[1], Adenauer Yamin[1], and Mauro Storch[1]

PPGInf, UCPel, Pelotas - RS, Brazil
{dubois, adenauer, mstorch}@ucpel.tche.br

Abstract. Nowadays almost all computing resources are connected, and researchers are trying to take advantage of the computational power available in large scale networks. A promising approach is to use code mobility, or software mobility. A mobile program can initiate its execution in a host and then move itself to another host where it continues its execution. Despite the advantages acquired with mobile computation, this kind of system is still very difficult to program. In [5], three common patterns of mobile computation were identified. In this paper, these patterns are modeled as *template method design patterns* and implemented in Java, a popular object oriented language, extended with primitives for remote execution of computations. All the communication and coordination aspects of a mobile application can be described using the mobility patterns, the programmer has to implement only the application specific parts of the program, i.e., the computations executed locally at every location that is visited by the mobile program. Aspects related to the coordination of programs are inherited from the class that describes the pattern. The mobility patterns hide the coordination structure of the code. To test the mobility patterns developed in this work, a distributed meeting scheduler was implemented that uses the design patterns identified to implement all the code mobility of the system.

1 Introduction

Networks are ubiquitous and the software architecture that runs on top of networks is in constant evolution. Researchers are trying to take advantage of the computational power available in large scale networks. A promising approach is to use code mobility, or software mobility. A mobile code program starts executing in one location and can move to other locations in a network so it can better use the resources available. Mobile programs provide more flexibility to exploit resources than distributed software that is statically distributed.

Besides expressing an algorithm to be executed, a mobile program must also express aspects of *coordination*, i.e., how a program is going to be partitioned between locations, when programs must move, synchronization, etc. Hence, writing mobile software is as difficult, maybe even more difficult, than implementing traditional distributed software. This paper presents the modeling and implementation of design patterns for mobile software that help in the development of mobile code programs. The patterns presented in this paper raise the level

XXXIV Conferencia Latinoamericana de Informática (CLEI 2008), pp. 870-879, 2008.

of abstraction in the development of mobile software. In [5], three common patterns of mobile computation were identified. In this paper, these patterns are modeled as *template method design patterns* and implemented in Java, a popular object oriented language, extended with primitives for remote execution of computations.

All the communication and coordination aspects of the application can be described using the mobility patterns, the programmer has to implement only the application specific parts of the program, i.e., the computations executed locally at every location that is visited by the mobile program. Aspects related to the coordination of programs are inherited from the class that describes the pattern.

This paper is organized as follows: Section 2 background material on mobile code and design patterns are presented. Section presents the coordination patterns for mobile computation. Section 4, uses the mobility patterns to implement a *distributed meeting planner*. Section 5 discusses related work and Section 6 concludes.

2 Background

2.1 Mobile Computation

Mobile computations are logical computing entities that can migrate in a network [14]. Examples include a user program that moves between locations in a network in order to use the processing power available, or a mobile agent, i.e., an application that migrates to act on behalf of its owner.

Mobility related terms can have different meanings depending on the research area. The hardware community usually relates the term *mobility* to the mobility of physical devices like laptops, notebooks and palm computers, while the software community relates it to programs or code that can move between locations. This paper focuses on the latter form of mobility.

In [5], three common patterns of mobile computation that occur in distributed information retrieval (DIR) systems were identified. A DIR application gathers information matching some specified criteria from information sources dispersed in the network. This kind of application has been considered "the killer application" for mobile languages [8]. The patterns identified are: *MMap*: broadcasts a computation to be executed on a set of locations, *MFold*: visits a set of locations executing a computation at each one and combining the results, and *MZipper*: visits back and forth a set of locations trying to find a value that is agreed by all locations.

In [5] the patterns were identified and implemented using and extension of the Haskell functional language for mobile computation [4]. The patterns were implemented as *mobility skeletons*, i.e., higher order functions that encapsulate common patterns of mobile computation. The advantage of the skeletons is that the programmer does not need to worry about the low level details of code mobility, she just have to implement the sequential parts of the algorithm that

are passed as arguments to the skeletons, all the coordination behavior of the application is already implemented in the skeletons.

The objective of this papers is to model the same coordination behaviors in a commercial object oriented language i.e., Java, making it easier for programmers to understand, use, and benefit from code mobility.

2.2 Design Patterns

A design pattern describes a solution to a commonly occurring problem [9]. Design patterns provide tested proven solutions to software development problems. Design patterns can be classified in three main categories: *Creational patterns* that deal with the creation of objects, *Structural patterns* that deal with the relationships between entities, and *Behavioral patterns* that deal with providing flexibility in the behavior of objects.

A *Template Design Pattern* is a behavioral design pattern that describes the skeleton of a solution to a problem, deferring some steps to subclasses. Hence subclasses can implement parts of the algorithm without modifying the structure of the solution. A template design pattern is usually implemented as an abstract class with abstract methods that must be implemented by subclasses. The class also has a *template method* that is not abstract, where the main algorithm is implemented. In this paper template method design patterns are used to model and implement the mobility coordination patterns.

2.3 A simple Mobile Computation Platform for Java

To implement the patterns of mobile computation described in this paper we used a simple platform for mobility implemented in pure Java that uses sockets to move computations and execute programs on remote locations.

The platform is based on three components: **The `Execute` Interface**: it is a Java interface that describes how to execute objects on remote locations. It has only one abstract method `Serializable execute()`, that must be implemented by all objects that will be communicated and run on remote hosts. `RemoteEvaluation` **class**: This class contains two static methods that allow the remote evaluation of objects. The `moveS` method receives two arguments, the name of a remote host and an `Execute` object, and moves the `Execute` object to the remote location were its `execute()` method is called. The result of the call to `execute()` is returned to the original location. The `RemoteEvaluation` class also has a `createA` method that is an asynchronous version of `createS`, meaning that it evaluates an object on a remote location without waiting for the result of the evaluation. **The** `JMServer`: It is a server that must be running on all locations. It receives objects sent by calls to `createS` and `createA` executing them automatically.

The mobility platform used in the experiments is very simple, providing only core functionalities for the implementation of the patterns. These primitives could be easily implemented using any Java extension for mobility, e.g., Voyager [16], Java Go [10].

3 Coordination Patterns for Mobile Computation

3.1 The `Mmap` pattern

A simple pattern of mobile computation is `MMap`, that models the *broadcast* of a computation to a set of locations. `MMap` executes a computation on each host of a list that it receives as an argument and returns a list of results, as can be seen in figure 1. Figure 2 shows the UML class diagram for `MMap` modeled as a *Template Method* design pattern. The abstract class `MMap`, that implements the `Execute` interface, has an attribute `hosts` that contains the list of hosts to where the computation must be sent. The `execute()` method is abstract, hence it should be implemented to describe the computation that must be executed on every host. The template method is `goMMap()` which activates `MMap` by sending the computation to every host.

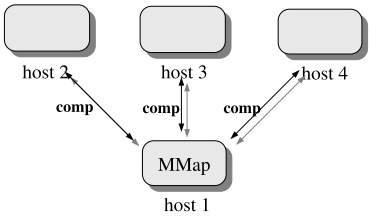

Fig. 1. mmap - Broadcast

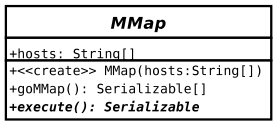

Fig. 2. Class Diagram for MMap

In Figure 3 a simple example using the `MMap` pattern is given. The class `HelloMMap` extends `MMap` and implements `execute()` by simply printing `Hello` on the remote location. As the result returned by the remote computation is of no use, it can be ignored.

```
class HelloMMap extends MMap{

    HelloMMap(String[] hosts){ super(hosts); }

    public String execute(){
        System.out.println("Hello!!");
       return null;
    }

    public static void main(String[] args){
        String [] hosts = (...)

        HelloMMap hello=new HelloMMap(hosts);
        hello.goMMap();
}
}
```

Fig. 3. Using the `Mmap` pattern

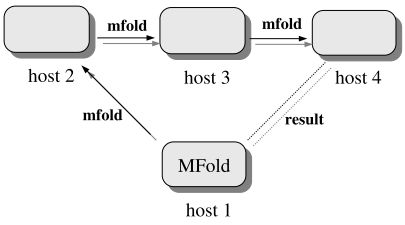

Fig. 4. MFold

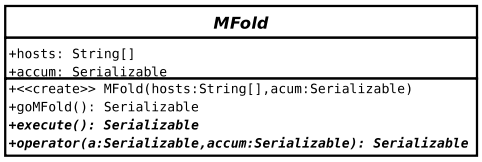

Fig. 5. Class Diagram for MFold

3.2 The `Mfold` pattern

This patterns describes a computation that visits a list of hosts executing a computation at each host and combining the results produced using an operator. When `MFold` reaches the last host it sends the result back to the first host, the one that initiated `MFold` (see Figure 4).

Figure 5 shows the UML class diagram for `Mfold`. As in the definition of `MMap`, `MFold` was implemented as an abstract class, that implements the `Execute` interface, and the method `goMFold` is the template method that launches the mobile program. When a programmer wants to extend the `MFold` class, she must implement two methods, `execute()` that describes what must be executed on each host and `operator()` that is used to combine the results obtained at each host using an *accumulator*. The `MFold` constructor receives as arguments an initial value for the accumulator and a list of hosts to be visited. The program in Figure 6 uses `MFold` to implement a program that visits a set of hosts to collect their names.

3.3 The `MZipper` Pattern

The `MZipper` pattern describes a computation that tries to find a value that is agreed by a set of locations. The value is tested using a predicate at each node, and when the predicate fails the computation returns to the first location to ask for a new value to be tested. Figure 7 presents the behavior of `MZipper`.

Figure 8 shows the UML class Diagram for the `MZipper` implementation. The constructor receives as an argument the list of hosts to be visited. The `goMZipper` is the template method, `execute()` and `predicate()` are the methods to be implemented by subclasses. The `execute()` method is the one to be called on the first location to generate the initial value, and this value is tested at every location using `predicate()`. When `predicate()` fails `MZipper` goes back to the first location to ask for a new value using `execute()`. `MZipper` finishes execution when there is a value agreed by all hosts or when the first host can not generate new values anymore. Figure 9 shows a simple program that tries to find a free time in time tables distributed on a network. The `execute()` method gets a free time in the first location, then this time is tested on all locations using

```
class CollectNames extends MFold{

   CollectNames(String[] hosts, String accum){super(hosts,accum);};
   }

   public Serializable execute(){
      String res="";
      try{
         res=InetAddress.getLocalHost().getHostName();
      }catch(Exception e){}
      return res;
   }

   public Serializable operator(Serializable newName,
                                Serializable accumulator){
      return (accumulator + " " + newName);
   }


   public static void main(String[] args)
                     throws Exception{
      (...)
      CollectNames c=new CollectNames(hosts);

       r=c.goMFold();
       System.out.println(r);
    }
}
```

Fig. 6. Using the `MFold` pattern

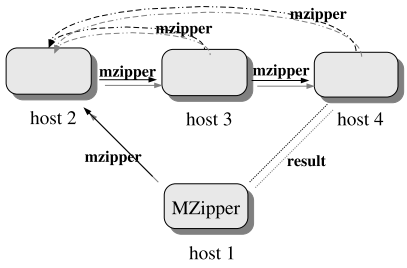

Fig. 7. MZipper

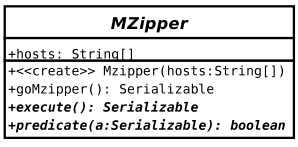

Fig. 8. Class Diagram for MZipper

`predicate()`. If one of the hosts does not agree with the time, `MZipper` moves back to the first location and gets a new free time using `execute()`.

```
class MeetingPlanner extends MZipper{
   (...)
   MeetingPlanner(String []hosts) {super(hosts);}

   public MeetingTime execute(){ return (getFreeTime()); }

   public boolean predicate(Serializable t){
          MeetingTime time = (MeetingTime) t;
          return (checkTimeAtCurrentLocation(time));
   }

   public static void main(String[] args){
          String [] hosts = (...)
          MeetingTime timeForMeeting;
          MeetingPlanner mp =new MeetingPlanner(hosts);

          timeForMeeting=(MeetingTime)mp.goMZipper();
          if(timeForMeeting!=null)
          {
           System.out.println("Free time in all
           time tables:" + timeForMeeting);
          }
   }
}
```

Fig. 9. Using the `MZipper` pattern

4 Case Study: A distributed meeting planner

In this Section we describe the implementation of a mobile application, a distributed meeting planner, that uses mobility coordination patterns to implement all the coordination aspects of the application. In the distributed meeting planner, each user is connected to a workstation and, when one of them wants to arrange a meeting, she launches a mobile program that visits the locations of people involved, trying to find a suitable time. This sort of application has two patterns of mobility. The first is a computation that visits a set of locations in a network performing an action at each location, i.e., visiting locations to find the right time for a meeting. This is exactly the pattern described by `MFold` and `MZipper`. The second is the idea of broadcasting a computation to a list of locations, i.e., sending to all locations the time that was agreed for the meeting, that can be implemented using `MMap`.

In the next two Subsections we describe two different implementations of the meeting planner.

4.1 Meeting Planner using `MFold`

The `MFold` version of the meeting planner uses `MFold` to compute the intersection of the free times of all locations. The meeting planner is implemented by

extending the `MFold` class and implementing the abstract methods `execute()` and `operator()`. The `operator()` method computes the intersection between two lists, one containing the free times of all locations visited so far, and the other being the free times of the current location. The method `execute()` is used to find the free times of the location being visited. When the `goMFold()` method is called, the meeting planner starts visiting all locations getting their free times (`execute()`) and computing the intersection between the free times (`operator()`). When it returns to the first location it displays the times available to the user that chooses one for the meeting. Then `MMap` is used to broadcast a computation that records the new meeting time in the timetables of all locations.

4.2 Meeting Planner using `MZipper`

The `MZipper` implementation of the meeting planner is similar to the one in Figure 9. The `execute()` looks for an empty slot in the time table of the location where it is called, and it is only executed in the first location. After calling `execute()`, `MZipper` visits the other locations testing the free time generated by `execute()`, with the `predicate()`. The `predicate()` compares the current time for the meeting with all the free times of the location being visited. If the time for the meeting is also available in the current location, `MZipper` moves to the next location of the list. If it is not available, it moves back to the first location and asks for a different time using `execute()`. In the end, `MZipper` will return a time that is available at all locations, or `null` meaning that there is no available time for the meeting. If the meeting planner finds a time for the meeting, it is broadcasted to all time tables using `MMap`, as in the `MFold` implementation.

4.3 Comparing the two Implementations of the Meeting Planner

Although the coordination aspects of the application can be described using `MZipper` or `MFold` the pattern of mobility present in each implementation is different. In the implementation using `MZipper`, the computation moves between locations carrying *only one time* for the meeting. Every time a location rejects a time, `MZipper` goes back to the first time table to ask for a different time. In the `MFold` version the computation carries a *list of free times*, and checks this list against the free times of all locations it visits. The implementation using `MFold` seems to be more realistic as the time table that the mobile program carries is small. If the time table is too large to be communicated, e.g. a database, the `MZipper` version would be more appropriate.

5 Related Work

The patterns presented in this paper were first described in [5], were they were implemented as *Mobility Skeletons*, i.e., higher order functions that encapsulate common patterns of mobile computation, using *Mobile Haskell*, an extension of the functional programming language Haskell for mobile computation [4]. Mobile

Haskell is a prototype research language. The contribution of this paper is to give a new design for the skeletons, based on object oriented design patterns, and an implementation using pure Java 1.5, a programming language widely used in the software industry. It is expected that the work presented here may help in the development of new commercial technology for distributed mobile programming. In previous work [7], two of the patterns described here (`MMap` and `MFold`), were described and classified as design patterns [9]. An implementation was given using Voyager [16]. The main problem with this work is that the programmer must understand Voyager in order to use the implemented patterns. Furthermore, our design using *template design patterns* is simpler and reflects the original design in Haskell. No discussion about `MZipper` was given.

High level abstractions that control most aspects of coordination are best developed for parallelism, e.g., in implicit parallel languages such as HPF [13], evaluation strategies [15], and *algorithmic skeletons* [2]. Higher level abstraction are also emerging for distributed programming, e.g., Erlang *Behaviors* are higher order functions that encapsulate patterns of distributed fault tolerant software [6]. There are many mobile languages [3, 11, 16, 10], but they are usually low level languages with few high level abstractions. In [17], a platform for mobile computation is described that is programmed using *Nomadic Pict*, a programming language based on the π-calculus extended with primitives for mobility. Nomadic Pict has two classes of primitives, *low-level* and *high-level* primitives. The low-level primitives describe migration and synchronization of computations. The high-level primitives are implemented using the low-level ones, and provide high-level communication based on names of agents and not on their location. The patterns presented in this paper provide a higher-level of abstraction as each pattern encapsulates several aspects of coordination.

There is also work on design patterns for mobile Agents, e.g., [12], although they focus mainly on modeling mobile systems and not on code reuse and easy of programming.

6 Conclusions and Future Work

This paper presented the design and implementation of high-level coordination patterns for mobile software in Java. The coordination patterns were implemented as *Template Method* design patterns describing common mobility patterns for distributed information retrieval systems [1]. The contributions of the paper are: modeling and implementing the patterns in a commercial object oriented language, giving simple examples that help understanding the behavior of the patterns, and a case study: the implementation of a distributed meeting planner that uses the patterns to express all the coordination behavior of the application. The design patterns presented in this paper can help in the development of mobile programs: if a programmer recognizes one of the patterns in her application, she just has to extend the class that describes the pattern, implementing the sequential parts of the program. Aspects related to coordination, i.e., communication, synchronization and code movement, are inherited from the

superclass. The implementations of the patterns presented in this paper are very simple, as the main objective was to catalogue and describe the patterns identified. More robust coordination patterns could be implemented e.g., by adding extra functionalities to describe what happens when things go wrong, e.g., not being able to reach one of the locations in the list of location to visit. Also, the three mobility patterns implemented encapsulate patterns of mobile computation that usually happen in distributed information retrieval systems. It would be interesting to analyze commercial and research mobile applications trying to identify more mobile coordination abstractions that are general i.e., applicable in many cases, realistic i.e., useful for real applications, and easy to reason about.

The mobility platform used in the experiments and the source code of the patterns can be download from `http://atlas.ucpel.tche.br/~mstorch/`.

References

1. J. Callan. Distributed information retrieval. pages 127–150. Advances in information retrieval in Kluwer Academic Publishers, 2000.
2. M. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. Pitman, 1989.
3. S. Conchon and F. L. Fessant. Jocaml: Mobile agents for Objective-Caml. In *(ASA'99)/(MA'99)*, Palm Springs, CA, USA, 1999.
4. A. R. Du Bois, P. Trinder, and H.-W. Loidl. *m*Haskell: mobile computation in a purely functional language. *JUCS*, 11(7):1234–1254, 2005.
5. A. R. Du Bois, P. Trinder, and H.-W. Loidl. Towards Mobility Skeletons. *Parallel Processing Letters*, 15(3):273–288, 2005.
6. Erlang. WWW page, http://www.erlang.org/, 2006.
7. Z. Field, R. Dewar, P. Trinder, and A. R. D. Bois. Two executable mobility design pattens. In *PLoP 2006 - Pattern Languages of Programs*, Portland, 2006.
8. A. Fuggetta, G. Picco, and G. Vigna. Understanding Code Mobility. *Transactions on Software Engineering*, 24(5):342–361, May 1998.
9. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, 1995.
10. JavaGo. WWW page,http://homepage.mac.com/t.sekiguchi/javago/, 2007.
11. F. C. Knabe. *Language Support for Mobile Agents*. PhD thesis, School of Computer Science, Carnegie Mellon University, 1995.
12. E. F. A. Lima, P. D. Machado, F. R. Sampaio, and J. C. A. Figueiredo. An approach to modelling and applying mobile agent design patterns. *ACM Software Engineering Notes*, 29(4), 2004.
13. D. B. Loveman. High Performance Fortran. *IEEE Parallel & Distributed Technology: Systems & Technology*, 1(1):25–42, 1993.
14. D. Milojićić, F. Douglis, and R. Weeler. *Mobility: Processes, Computers, and Agents*. Addison-Wesley, Reading, MA, USA, 1999.
15. P. W. Trinder, K. Hammond, H.-W. Loidl, and S. L. Peyton Jones. Algorithm + Strategy = Parallelism. *Journal of Functional Programming*, 8(1):23–60, Jan. 1998.
16. Voyager System. http://www.recursionsw.com/voyager.htm, 2006.
17. P. T. Wojciechowski. *Nomadic Pict: Language and Infrastructure Design for Mobile Computation*. PhD thesis, University of Cambridge, 2000.