First International Conference
Modelling and Development of Intelligent Systems
Sibiu - Romania, 22-25 October, 2009

Parallel-Distributed Programming Tool for the Linda Programming Model

Ernest Scheiber

Abstract

This note presents an implementation of the Piranha model to program parallel - distributed application and a development tool. We use the TSpaces framework from I.B.M. as a middleware for data exchanges. Our implementation does not introduce a new API but requires certain program structure.

1 Introduction

Even if there is a lack of agreements on what exactly grid computing or cloud computing are, the parallel - distributed programming API's or tools are one of the main ingredient to build such computing environment or application.

Linda is not a programming model but a set of operations that can be added to any language. Linda gives a way to communicate and synchronize different processes. We known the following Linda type development frameworks in Java: *TSpaces* from I.B.M., [6] and *JavaSpaces* from SunMicrosystems.

The *Piranha* model is a parallel - distributed computing model based on Linda, [2]. The paradigm behind Piranha is that of dispatcher-worker. The dispatcher coordinates the activities while the workers solve the received tasks. Between them, the communications are programmed using the Linda operations.

Other known parallel - distributed computing models are MPI, [5, 1], map-reduce.

This note presents an implementation of the Piranha model to program parallel - distributed application and a development tool. We use the TSpaces framework, but instead other messaging service may be used. The actual messaging services (*Sun MessageQueue, apache-activemq* or *qpid*) own the required functionality of a Linda framework. The reason of our development activities consists in the lack of free correspondent software. A preliminary version was reported in [3].

An application to be used with the tool is constrained to a specific structure; it is composed from a dispatcher thread and worker threads. Between the dispatcher thread and the worker threads there are asynchronous message changes. These messages are kept by a TSpaces server until they are consumed. The tool allows us to state the network of workstations involved in the computation, to deploy and to launch an application.

2 The Structure of an Application

We suppose that several computers in a network will perform the required computation. The Java application contains:

- A *Console* class - The Console launches to run the *Dispatcher.*

- On any workstation there is a *Service* class, it instantiates and starts the *Worker*s.
- The application associated TSpace server, it keeps the messages until they are consumed.

The structure of an application is outlined in Fig. 1.

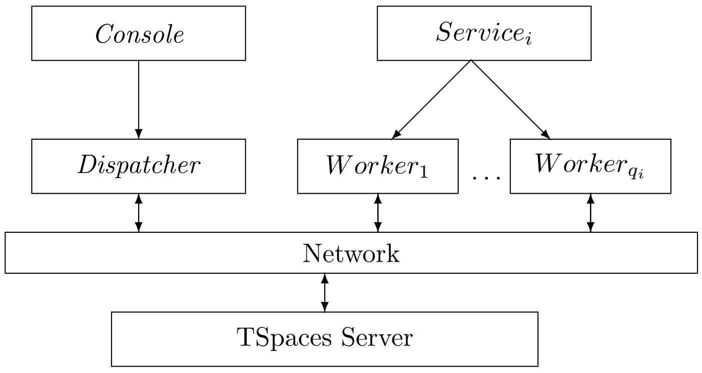

Figure 1: The structure of an application.

The *Dispatcher* and the *Worker* classes are threads. The constructors of the dispatcher and of the worker classes are responsible to establish the connection with the TSpaces server. The `run` methods of these Java threads contain the specific activities to solve the given problem. On a workstation it is possible to start several worker threads. The i index in Fig. 1 denotes a workstation on which q_i worker threads are run. The *Console* and *Service* classes are independent from an application.

Between the dispatcher thread and the worker threads there are asynchronous message exchanges. These messages are kept by a TSpaces server. The data to be exchanged between the dispatcher and the workers are wrapped into objects. To distinguish between different kinds of data, a tag field may be introduced. A message consumer (dispatcher or worker) waits until all the required messages are available. In this way the synchronization problems are solved as well as the coordination of the activities.

3 The anatomy of our implementation

We shall analyze the codes of the above mentioned classes for a very simple application: the dispatcher sends a greeting message to each worker. After receiving the message, a worker responds indicating the name of its computer.

Let us suppose that we use n workstations and that on the $i-$th station there will be launched q_i workers ($\sum_{i=1}^{n} q_i = q$).

Some environment data are fixed in two property files:

propConsole		propService	
Name		Name	
AppName		AppName	
Host		Host	
Port		Port	
TasksNumber	q	TasksNumber	q
		ComputerTaskNumber	q_i
		ComputerFirstTaskIndex	$\sum_{j=1}^{i-1} q_j,\ q_0 = 0$

where *AppName, Host, Port* identifies the name of the application TupleSpace and respectively the host of the *TSpaces* server with the corresponding port number.

As we have mention, the dispatcher is launched by the *Console* class, while the workers are launched by the *Service* class. These two classes are independent of an application.

```
The Console class                                    | The Service class
-----------------------------------------------------+-----------------------------------------------------
package ******;                                      | package ******;
import java.util.Properties;                         | import java.util.Properties;
import java.io.*;                                    | import java.io.*;
                                                     |
public class Console{                                | public class Service{
  Properties p=null;                                 |   Properties p=null;
                                                     |
  public Console(String path2Prop) {                 |   public Service(String path2Prop)
    try{                                             |     try{
                                           // Loads the properties
      FileInputStream fis=new                        |       FileInputStream fis=new
        FileInputStream(path2Prop+"propConsole");    |         FileInputStream(path2Prop+"propConsole");
      p=new Properties();                            |       p=new Properties();
      p.load(fis);                                   |       p.load(fis);
    }                                                |     }
    catch(Exception e){                              |     catch(Exception e){
      System.out.println(e.getMessage());            |       System.out.println(e.getMessage());
      System.exit(1);                                |       System.exit(1);
    }                                                |     }
  }                                                  |   }
                                                     |
  public static void main(String[] args){            |   public static void main(String[] args){
    String fs=System.getProperties().                |     String fs=System.getProperties().
      getProperty("file.separator");                 |       getProperty("file.separator");
    String path2Prop;                                |     String path2Prop;
                        // An argument may contain the path to the property file
    if(args.length>0)                                |     if(args.length>0)
      path2Prop=args[0]+fs;                          |       path2Prop=args[0]+fs;
    else                                             |     else
      path2Prop="";                                  |       path2Prop="";
    Console obj=new Console(path2Prop);              |     Service obj=new Service(path2Prop)
                                            // Set the properties
    String appName=obj.p.getProperty("AppName");     |     String appName=obj.p.getProperty("AppName");
    String host=obj.p.getProperty("Host");           |     String host=obj.p.getProperty("Host");
    String sPort=obj.p.getProperty("Port");          |     String sPort=obj.p.getProperty("Port");
    String sTasks=obj.p.getProperty("TasksNumber");  |     String sTasks=obj.p.getProperty("TasksNumber");
                                                     |     String sIndex=obj.p.getProperty(
                                                     |       "ComputerFirstTaskIndex");
                                                     |     String sComputerTasks=obj.p.getProperty(
                                                     |       "ComputerTaskNumber");
    int port=Integer.parseInt(sPort);                |     int port=Integer.parseInt(sPort);
    int tasks=Integer.parseInt(sTasks);              |     int tasks=Integer.parseInt(sTasks);
                                                     |     int index=Integer.parseInt(sIndex);
                                                     |     int computerTasks=Integer.parseInt(sComputerTasks);
// The dispatcher is launched into execution         | // The required number of workers are
                                                     | // launched into execution
    Dispatcher dispacher=new Dispatcher(appName,     |     Worker[] worker=new Worker[computerTasks];
      host,port,tasks);                              |     for(int i=0;i<computerTasks;i++){
    dispacher.start();                               |       worker[i]=new Worker(appName,host,
                                                     |         port,tasks,index+i);
                                                     |       worker[i].start();
                                                     |     }
  }                                                  |   }
}                                                    | }
```

The wrapper class that contain the data to be exchanged between the dispatcher and workers may be

```
package tshello;
import java.io.Serializable;

public class DataWrapper implements Serializable{
  // tag=0 message sent by the dispatcher
  // tag=1 message sent by a worker
  public int tag;
  public String mesaj;

  public DataWrapper(String mesaj, int tag) {
    this.mesaj=mesaj;
    this.tag=tag;
  }
}
```

The connection to the TupleSpace server is made in the constructors of the *Worker* and *Dispatcher* classes.

The code of the *Worker* class is

```
package tshello;
import com.ibm.tspaces.*;
import java.net.*;
import java.io.*;

public class Worker extends Thread{
  private int id;
        private TupleSpace ts=null;
        private String tsName;
  private int tasks;
  private PrintStream f;

  public Worker(String tsName,String host,int port,int tasks,int id){
    this.tsName=tsName;
    this.id=id;
    this.tasks=tasks;
    ts=startTupleSpace(tsName,host,port);
    try{
      f=new PrintStream("worker"+id+".txt");
    }
    catch(Exception e){
      System.out.println("File error :"+e.getMessage());
    }
  }

  public void run(){
    Tuple tuple=null;
    DataWrapper d=null;
    String source=tsName+id;
    String dest=tsName+" "+id;
    try{
      Tuple template=new Tuple(source,new FieldPS(DataWrapper.class));
      tuple=(Tuple)ts.waitToTake(template);
      d=(DataWrapper)tuple.getField(1).getValue();
      if(d.tag==0){
        String msg="Mesage received by "+id+" from the dispatcher:\n"+d.mesaj;
        System.out.println(msg);
        f.println(msg);
        InetAddress addr=InetAddress.getLocalHost();
        String s=addr.getHostName();
        String mesOut="Hello from "+id+" at "+s;
        d=new DataWrapper(mesOut,1);
        FieldPS ps=new FieldPS(d);
        ts.write(dest,ps);
        f.close();
      }
    }
    catch(Exception e){
      System.out.println("Exception-Worker : "+e.getMessage());
    }
  }

  private TupleSpace startTupleSpace(String tsName, String host,int port){
    TupleSpace ts=null;
    try{
      Tuple active=TupleSpace.status(host,port);
      if((active==null)||(active.getField(0).getValue().equals("NotRunninng"))){
        System.out.println("TupleSpace Server is not available");
        System.exit(1);
      }
      ts=new TupleSpace(tsName,host,port);
    }
    catch(TupleSpaceException e){
      System.out.println("TupleSpaceException "+e.getMessage());
                        System.exit(1);
    }
                return ts;
  }
}
```

Finally, in the *Dispatcher* class, there are send messages to the workers (the *scatter* method) and are waiting to receive the corresponding responses (the *gather* method):

```java
package tshello;
import com.ibm.tspaces.*;
import java.io.*;

public class Dispatcher extends Thread{
  private int tasks;
  private TupleSpace ts=null;
  private String tsName;
  private PrintStream f;

  public Dispatcher(String tsName,String host,int port,int tasks) {
    this.tasks=tasks;
    this.tsName=tsName;
    ts=startTupleSpace(tsName,host,port);
    try{
      f=new PrintStream("dispatcher.txt");
    }
    catch(Exception e){
      System.out.println("File error :"+e.getMessage());
    }
  }

  public void run(){
    try{
      scatter();
      gather();
      ts.cleanup();
    }
    catch(Exception e){
      System.out.println(e.getMessage());
    }
  }

  private void scatter(){
    String mesOut="Hello from the dispatcher !";
    DataWrapper d=new DataWrapper(mesOut,0);
    try{
      FieldPS ps=new FieldPS(d);
      Tuple multi=new Tuple();
      for(int i=0;i<tasks;i++){
        String dest=tsName+i;
        Tuple nextTuple=new Tuple(dest,ps);
        multi.add(new Field(nextTuple));
      }
      TupleID[] ids=ts.multiWrite(multi);
    }
    catch(TupleSpaceException e){
      System.out.println("TupleSpaceException-scatter0 "+e.getMessage());
    }
    System.out.println("Scatter OK");
  }

  private void gather(){
    Tuple tuple,template;
    DataWrapper d=null;
    try{
      for(int i=0;i<tasks;i++){
        String source=tsName+" "+i;
        template=new Tuple(source,new FieldPS(DataWrapper.class));
        tuple=(Tuple)ts.waitToTake(template);
        d=(DataWrapper)tuple.getField(1).getValue();
        if(d.tag==1){
          System.out.println(d.mesaj);
          f.println(d.mesaj);
        }
      }
      f.close();
    }
    catch(Exception e){
      System.out.println("Exception-gather "+e.getMessage());
    }
  }

  private TupleSpace startTupleSpace(String tsName, String host,int port){. . .}
}
```

4 An example

The sixteen grid problem (www.ams.org/ams/16-grid.html). Each of the numbers 1,2,...,16 is used exactly once in the empty cells to form arithmetic expressions connected by symbols for the four basic operations. Each row (column) is an arithmetic expression, read and performed left to right (top to bottom), disregarding the usual order of operations, to yield the result at the right (bottom).

	-		$\times$		$\times$		=-60
$\times$	■	+	■	$\times$	■	$\div$	
	+		+		-		=29
+	■	+	■	-	■	$\times$	
	-		-		-		=-14
+	■	-	■	+	■	-	
	$\times$		-		-		=32
=28		=1		=57		=27	

A solution, presented in [4] is based on a parallelization scheme of the backtracking algorithm. Based on the domain decomposition method, there is a more efficient way to solve this problem. In the set on n order permutations, $\mathcal{P}_n$, considering the lexicographic order the first permutation is $(1, 2, \ldots, n)$ and the last is $(n, n-1, \ldots, 1)$. Relative to this order, it may compute the m-th permutation, $1 \leq m \leq n!$.

The first $q_1 = (n-1)!$ elements of $\mathcal{P}_n$ have 1 on the first position, the next q_1 elements have 2 on the first position and so on. In the group having i on the first position, there are $n-1$ subgroups with $q_2 = (n-2)!$ elements with the second element equals respectively with $1, 2, \ldots, i-1, i+1, \ldots, n$. If the first $s-1$ elements are fixed, $i_1, \ldots, i_{s-1}$ then there exits $n-s+1$ groups with $q_s = (n-s)!$ elements, having on the i-th position an element of $\{1, \ldots, n\} \setminus \{i_1, \ldots, i_{s-1}\}$.

To find the s-th element of the m-th permutation, we compute the group number t_s having the i_s element as constant, $t_s = \lceil \frac{m}{q_s} \rceil$, and finally i_s will be the l_s-th element of the sequence $1, \ldots, n$, after we have deleted $i_1, \ldots, i_{s-1}$, where $l_s \equiv t_s (\text{mod } n-s+1)$, $l_s \in \{1, 2, \ldots, n-s+1\}$.

If there are p workers, then the set of all $n!$ permutations are equally divided into p intervals. Each worker generates sequentially the permutations of its attached interval and verifies the restrictions of the problem.

The unique solution of the sixteen grid problem is

1	-	2	$\times$	4	$\times$	15	=-60
$\times$	■	+	■	$\times$	■	$\div$	
12	+	8	+	14	-	5	=29
+	■	+	■	-	■	$\times$	
13	-	7	-	9	-	11	=-14
+	■	-	■	+	■	-	
3	$\times$	16	-	10	-	6	=32
= 28		= 1		= 57		= 27	

5 The Development Tool

To run the application a lot of activities are to be performed: to deploy the application, to launch the Service program on each workstation and finally to launch the Console on the host station.

To assist these activities, a development tool was created. The *Service* program becomes a servlet and we use the *apache-tomcat* as a servlet container Web server. The *Console* program which starts the dispatcher makes the requests to the *Service* servlets, too.

To the network of workstations it is associated a second TSpaces server to keep the names of the involved computers - denoted as the tuple space of the network.

The tool contains two parts:

- The *Advertiser* serves to state the network of workstations. Using the *Advertiser* a computer is linked to the network of workstations to perform a parallel-distributed computing.

The graphical interface of the *Advertiser* allows to

– Declare the computer as a member of the network of workstations - i.e. a tuple with the name of the computer is written into the tuple space of the network;

– Remove the computer from the network - i.e. remove the above define tuple from the tuple space of the network;

– Show the list of the computers in the network of workstations.

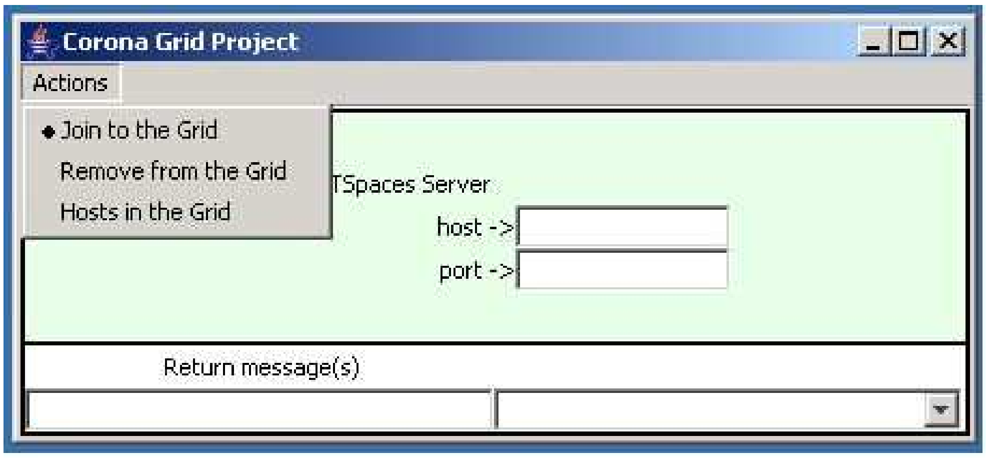

Figure 2: The window of the *Advertiser* tool.

- The *Developer* allows to

  – Generate the folders of the application;

  – Compile and archive the service part of the application;

  – Deploy the service part to the workstations of the network. The deployment is done with the *apache-tomcat-deployer;*

  – Compile the console part of the application.

  – Launch the console to run. In the *Console* class the request are programmed using the *apache commons-httpclient* software;

  – Undeploy the service part of the application.

  These targets are executed through *apache-ant.*

  The *Developer* is installed only on the host workstation.

  A number of parameters are required: The name of the application, The host and the port of the application tuple space, The number of the worker threads, The path to the *apache-ant* and The username and password for the `manager` application of *apache-tomcat.*

  The list of the computers name in the network is required, too. This list may be generated with the *Advertiser,* contained in the *Developer,* too.

The scenario to run a parallel-distributed application using the framework involves:

1. To set up the network of workstations:

   (a) On launches the tuple space associated to the network;

   (b) On each workstation starts the *apache-tomcat* Web server and, with the Advertiser, declares the availability of that computer to join to the network of workstations.

2. Using the Developer, on the host workstation, we can install, deploy, start the tuple space associated to the application and execute the parallel-distributed computation launching the Console.

The current version of the framework may be downloaded from the author's Web page `http://cs.unitbv.ro/site/pagpers/scheiber`. The archive contains several other examples: a numerical integration, quicksort, the queens' problem, the graphic representation of the Mandelbrot set, the dinning philosophers' problem.

Conclusions. An implementation of the Piranha model for a parallel-distributed application is developed. The implementation does not introduce a new API but requires certain program structure. In addition we have created a tool to assist the development of an application that can be ported to any Java enabling platform. As a drawback of our approach is that, if a workstation dies then a running application never finishes. We intend to work on this problem. The security constraints are that of the *apache-tomcat* Web server.

References

[1] R. BISSELING, *Parallel Scientific Computation. A structured approach using BSP and MPI,* Oxford Univ. Press, 2004.

[2] G. A. PAPADOPOULOS, F. ARBAB, Coordination models and languages. CWI Report, SEN-R9834, 1998.

[3] SCHEIBER E., 2007, A TSpaces Based Framework for Parallel - Distributed Applications. *Knowledge Engineering Principles and Techniques* 1 (2007), Cluj University Press, 341-345.

[4] SCHEIBER E., 2009, A Parallelization scheme of Some Algorithms. Knowledge Engineering Principles and Techniques 2 (2009), *Studia Universitas Babeş-Bolyai, Informatica,* 244-248.

[5] M. SNIR, D. OTTO, S. HUSS-LEDERMAN, D. WALKER, J. DONGARRA, *MPI: The Complete Reference.* MIT Press, Cambridge, MA, 1996.

[6] * * * , TSpaces-User's Guide & Programmer's Guide, Distributed with the software, Version 2.1.2, 2000.

Ernest Scheiber
Transilvania University of Braşov
Faculty of Mathematics and Computer Science
str. Iuliu Maniu 50
ROMANIA
E-mail: *scheiber@unitbv.ro*