User Manual of A Java Distributed Computation Library

Karsten Fritsche[a], James Power[b] & John Waldron[†]

[a] Mathematics, Statistics & Information Systems,
University of Natal, Pietermaritzburg 3209, South Africa.
Email: karsten@alpha.futurenet.co.za

[b] Dept of Computer Science, NUI, Maynooth, Ireland.
Email: James.Power@may.ie Tel: +353-1-7083447

[†] Dept of Computer Science, Trinity College, Dublin 2, Ireland.
Email: jtw_ie@yahoo.co.uk Tel: +353-1-6083685
[†]Corresponding author.

Abstract

This paper describes User Manual of a Java Distributed Computation Library, which provides a simple development platform for developers who wish to quickly implement a distributed computation in the context of an SPMD architecture (Single Program, Multiple Data). The need for this research arose out of the realisation that the currently available distributed computation libraries and systems do not adequately meet certain criteria, such as ease of development, dynamic changes to system behaviour, and easy deployment of distributed software. The proposed solution to this problem was to produce a Java-based distributed computation library which enables developers to use the Java language to quickly and easily implement a distributed computation. The results of experiments conducted using DCL are also presented, as a means of showing that DCL met its design goals.

keywords: Distributed Architecture

1: Introduction

This document describes the functions and usage of the Distributed Computation Library (DCL). DCL is intended for use by developers who wish to implement distributed computations within the context of an SPMD architecture. The software is in the form of a programming library, which can be used to distribute developer-defined tasks across a network for processing by remote clients, gather the results of these tasks together, and allow the results to be handled in a developer-defined manner. This document describes the basic functions and characteristics of DCL. It also provides a brief tutorial on how to develop a simple distributed computation using DCL.

2: Scope of this Manual

The target audience of this manual is any developer who wishes to use DCL in order to develop a distributed computation. It is assumed that the developer will have previous programming experience, especially experience with the Java programming language. The developer should also be familiar with the compiler tools and runtime operation of the Java Development Kit (JDK) version 1.2 (or later). The JDK is freely available from Sun Microsystems Inc., at ¡http://java.sun.com¿. The Java Runtime Environment (JRE) version 1.2 should also be installed on any computer which is expected to run one of the DCL components.

This manual will not cover the use of the Java compiler or compiler tools, basic use of the JDK, or the running of Java applications. It is assumed that the developer will be familiar with these issues. Also, this manual will not cover aspects of basic Java programming, or the Java language in any detail, unless such detail directly impacts any important aspects of the use of DCL.

3: Technical Requirements

As has already been mentioned, DCL requires the JRE version 1.2 in order to run. This applies to both the server-side and client-side components of DCL. The developer must also have access to the JDK version 1.2 (or later), in order to develop distributed computations using DCL.

The DCL library places no specific minimum requirements on the hardware platform. This is due to the fact that DCL is entirely written in Java, and is thus platform independent (as are the distributed computations written with DCL). Rather, hardware requirements are more likely to be dictated by the requirements of a particular computation.

The network on which DCL runs must support the TCP/IP protocols. This is due to limitations of the Java networking API. This restriction is not unreasonable (nor overly restrictive), considering the widespread use of the TCP/IP suite of protocols.

4: Installation

The complete DCL library can be found on the attached disk, under the directory named “DCL”. DCL is broken up into the server-side components (found in subdirectory “server”), and the client-side components (found in subdirectory “client”).

Installation of the client components simply involves copying all the files in the ”client” subdirectory into any desired directory on the local hard drive. The client-side administrator should create a client initialisation file which the client components will use to set up various runtime parameters. The details of how to do this are described in Section 6 below.

Installation of the server components involves the same procedure as the client components: simply copy all the files in the ”server” subdirectory into any desired directory on the local hard drive. The server-side administrator should create a server initialisation file which the server components will use to set up various runtime parameters. The details of how to do this are described in Section 6 below.

5: Developer Classes

The developer need only concern himself with three classes. These are the ParameterGenerator, the ResultHandler, and the Task classes. The developer must extend each of these classes, overwriting certain methods, in order to implement a distributed computation. These three classes provide a simple and complete set of constructs with which the developer can implement a distributed computation. The classes are described below, and follow the logical order of input, process and output.

5.1 ParameterGenerator

The purpose of the ParameterGenerator is to generate successive parameter sets, which are sent to clients for processing by remote Task objects. These parameter sets are in the form of arrays of Objects, whose exact format and composition is application specific and thus defined by the developer. The array may only consist of standard Java Objects and primitive types.

The following code block shows the basic structure of the ParameterGenerator class:

```
public class ParameterGenerator {

     protected LogFile errorLog, systemLog;

     public void init();

     public Object[] getNextParameterSet();

}
```

The most important aspect of this class is the getNextParameterSet() method, which must be overwritten by the developer. This method returns an array of Objects, representing a parameter set. The getNextParameterSet() method will most probably be called several times during the course of a computation, and should generate successive sets of parameters each time it is called. It may thus be useful to introduce state variables into the extended class in order to keep track of how far through the problem space the ParameterGenerator has gone. The format of the array generated by the ParameterGenerator is developer-defined and is specific to a particular computation. If the ParameterGenerator has exhausted the search space of the problem, then it should return "null" as a parameter set. This is used to signal to clients that no more parameter sets are available.

The init() method may be overwritten if any special instantiation needs to take place, such as opening files, setting up state variable, or initialising data structures which will be used within the getNextParamterSet() method. The init() method is called by the server component of DCL as soon as the ParameterGenerator has been loaded into memory and instantiated.

Lastly, the ParameterGenerator provides the developer with an errorLog and a systemLog, which may be used to log errors or system messages.

An example of how to overwrite these methods to implement a simple distributed computation is given in Section 7.

5.2 Task

The Task class represents the executable (or processing) portion of a computation. Since its exact function is application specific, it must be extended by the developer. However, its overall function is to process parameter sets (which are generated by the ParameterGenerator on the server and sent to the client), and produce result sets (which are sent back to the server to be handled by the ResultHandler).

The following code block shows the basic structure of the Task class:

```
public class Task {

    protected LogFile errorLog, systemLog;

    protected Object[] parameterList;
    protected Object[] returnList;

    public void run();

}
```

The important thing to notice is that the only method which must be overwritten by the developer is the run() method. This method is the entry point for the Task's execution. The developer may write additional methods into the extended class, but should always remember that the first method to be called is the run() method. Two Object arrays are provided by the Task base class, namely the paremeterList and the returnList. When the run() method is invoked, the Task can assume that the paramterList contains the parameter set which it is expected to process. This array has exactly the same format as the parameter set generated by the getNextParamterSet() method of the ParameterGenerator on the server. The Task is required to fill the returnList array with the results of this processing. The developer must overwrite the runt() method in order to achieve this. When the Task has completed its processing, it must call the endProcessing() method. This method is extremely important, since it is responsible for notifying the host client that the Task has completed its processing and that the resultList has been filled. The client can then extract the returnList and send it back to the server. If the endProcessing() method is not called, then the Task will simply hang.

The Task class also provides the developer with a systemLog and an errorLog, for logging system and error messages.

An example of how to overwrite these methods to implement a simple distributed computation is given in Section 7.

5.3 ResultHandler

The ResultHandler is responsible for handling result sets that clients send back to the server, and represents the output phase of the computation. The ResultHandler should be extended by the developer to include any application specific functionality with regards to handling results, such as storing results in formatted files. Result sets are passed to the ResultHandler in the form of arrays of Objects, which were generated by a Task residing in a remote client. The exact format and composition of this array of Objects is application specific and defined by the developer.

The following code block shows the basic structure of the Task class:

```
public abstract class ResultHandler {

     protected LogFile resultLog, errorLog, systemLog;

     public void init();

     public abstract boolean handleResults (Object[] results, int ID);

}
```

The most important method for the developer is the handleResults() method. This method takes two parameters: an Object array and an integer. The Object array is the parameter set which the ResultHandler is being asked to handle. The format of this array is exactly the same as was generated by the Task and stored in its resultsList array. The ResultHandler is also provided with the task ID of the result set, which may be useful for administrative purposes. The handleResults() method returns a boolean. This return value is used by the server to determine when the computation is complete. Specifically, if the ResultHandler returns false, it means that the last result set handled did not constitute an exit condition, and thus the computation must continue. If the handleResults() method returns true, then the ServerEngine knows that an exit condition was found, and may begin its shutdown procedure. This mechanism is useful when the system is being used to perform some sort of state space search, where it may encounter the target value or condition in a result set before having exhausted the entire state space, and can thus safely discontinue the rest of the search. In the case where the full state space must be processed, the exit condition can be based on how many parameter sets have been processed out of the total that are known to be available. This technique is illustrated in the tutorial in Section 7.

The init() method of the ResultHandler serves a similar purpose to the init() method of the ParameterGenerator, namely, to allow the developer to instantiate any data structures or variables necessary for the correct functioning of the ResultHandler. The developer may also open additional files for the output of results at this point.

The ResultHandler class provides the developer with a systemLog, an errorLog, and a resultLog. These may be used for logging system message, error messages, or messages relating to the handling of result sets.

6: Initialisation Files

There are two initialisation files used by the system: one on the client-side, and one on the server-side. Both files should be plain text files, and both follow a similar structure. Settings are broken down into broad categories containing sub-settings, in a tree structure. Successive levels are separated by a full stop in the actual file. The categories are described below. A full description of all the options within each category is also given.

6.1 Server-side Initialisation File

The server initialisation file contains three categories. The "server" category describes various parameters necessary for the overall initialisation of the server, including the information necessary to establish the server-side communications mechanisms. The "dev" category describes the developer-defined aspects of the system. Here, the developer must specify the name of the Task, ResultHandler and ParameterGenerator which the ServerEngine should use to perform the computation. Finally, the "log" category describes various options related to how the system logs various types of messages, such as errors or information messages. Log file output can also be formatted with the options in this category.

The specific options within each category, together with default values (if any) are given below:

6.1.1 Category: server

server.port the port on which the server will listen for incoming connections [default=12345]

server.timeout the period (in milliseconds) that the SchedulerThread waits before notifying the ServerEngine [default=60000]

server.backlog the number of simultaneous incoming requests that the server will queue before rejecting further incoming connections [default=5]

6.1.2 Category: dev

dev.task the name of the Task class file which the developer has created for distribution to clients [no default]

dev.task.timePerTask the default amount of time to allocate to the completion of each parameter set [default=10000]

dev.parameterGenerator the name of the ParameterGenerator class file which the developer has created [no default]

dev.resultHandler the name of the ResultHandler class file which the developer has created [no default]

6.1.3 Category: log

log.error.tofile determines whether or not to log errors to a file [default=yes]

log.error.toscreen determines whether or not to log errors to the console [default=yes]

log.error.file specifies the file to which errors are written, if required [default=server_error.log]

log.error.format the format of the log messages, in terms of date and time stamps [default=DT, meaning that both date and time stamps will be attached to all output]

log.error.detail the level of detail to provide [default=0]

log.system.tofile determines whether or not to log systems to a file [default=yes]

log.system.toscreen determines whether or not to log system messages to the console [default=yes]

log.system.file specifies the file to which systems are written, if required [default=server_system.log]

log.system.format the format of the log messages, in terms of date and time stamps [default=DT]

log.system.detail the level of detail to provide [default=0]

log.results.tofile determines whether or not to log results to a file [default=yes]

log.results.toscreen determines whether or not to log results to the console [default=yes]

log.results.file specifies the file to which results are written, if required [default=server_results.log]

log.results.format the format of the log messages, in terms of date and time stamps [default=DT]

log.results.detail the level of detail to provide [default=0]

6.2 Client-side Initialisation File

The client initialisation file follows a similar structure to the server initialisation file, with a few exceptions. The client-side initialisation file only has two categories, namely "server" and "log". The "server" category describes at which address the server resides, and is used by the ClientEngine to establish the communication channel to the server. The "log" category on the client-side performs the same function as on the server-side, including formatting of log output and saving log output to a specified file.

The specific options within each category, together with default values (if any) are given below:

6.2.1 Category: server

server.ip which IP address the server is listening on [no default]

server.port which port the server is listening on [default=12345]

6.2.2 Category: log

In this category, the defaults and descriptions are the same as in the case of the server-side initialisation file. The only differences are in the file name to which log messages are written.

log.error.file the file name to which errors are logged [default=client_error.log]

log.system.file the file name to which errors are logged [default=client_system.log]

7 Tutorial

In this Section, a brief tutorial is presented, which shows how to develop a simple distributed computation which adds up sequences of numbers. Although this example is extremely simplistic, it shows the essential aspects of using DCL to develop distributed computations. The source code and class files for this tutorial can be found on the disk under the directory ”DCL
tutorial”.

Firstly, we examine is the SumParmGen, which generates the parameter sets. The code for this class is given below.

```
1 public class SumParmGen extends ParameterGenerator {
2
3     int i;
4
5     public void init() {
6          i=1;
7     }
8
9
10    public Object[] getNextParameterSet () {
11         if (i>=100) {
12              return null;
13         }
14         else {
15              Object[] output = new Object[2];
16              output[0] = new Integer(i);
17              i += 4;
18              output[1] = new Integer(i);
19              i++;
20              return output;
21         }
22
23    }
24
25 }
```

This code is important in several ways. Firstly, it introduces a state variable on line 3, namely i. On line 5, the init() method is used to initialise i to 1. On line 10, the getNextParameterSet() method is overwritten. This method examines the value of i, and if i is at least 100, then it returns null. This signals that the ParameterGenerator has exhausted the search space of the problem and can generate no more parameter sets. In this case, it means that the computation will sum the number from 1 to 100. If i was less than 100, then there are more parameter sets to be generated. This occurs on lines 14 to 20. The important aspect of these lines is the way in which the Object array ”output” is instantiated (line 15), and then filled with two Integer Objects (lines 16 and 18). These two Integers define a range over which a Task object will be expected to sum.

Secondly, we present the Task code:

```
1 public class SumTask extends Task {
2
3    public void run() {
4         Integer lo,hi;
5         lo = (Integer)parameterList[0];
6         hi = (Integer)parameterList[1];
7         returnList = new Object[1];
8         int temp=0;
9         for (int i=lo.intValue(); i<=hi.intValue(); i++) {
10             temp += i;
11        }
12        returnList[0] = new Integer(temp);
13        endProcessing();
14   }
15
16 }
```

This code shows how the Task class was extended to SumTask, and how the run() method was overwritten. It is important to note how the parameterList array is used to extract the data to be processed, in this case simply two Integer objects (lines 5 and 6). The for loop on line 9 simply sums the integers between the two integers extracted from parameterList, and puts the result into the returnList, which was instantiated on line 7.

Lastly, we examine the ResultHandler which will handle result sets returned to the server from clients. The code for the SumResHandler is given in the following code block:

```
public class SumResHandler extends ResultHandler {

   int sum_so_far;
   int num_workunits;

   public void init() {
        sum_so_far = 0;
        num_workunits = 0;
   }


   public boolean handleResults (Object[] results,int ID) {
        num_workunits++;
        Integer sum_ = (Integer)results[0];
        int sum = sum_.intValue();
        sum_so_far += sum;
        if (num_workunits != 20) {return false; }
        else {
```

```
19              resultLog.write ("Final result: "+sum_so_far,2);
20              return true;
21         }
22
23    }
24
25 }
```

In this class, two state variables are used, namely sum_so_far and num_workunits. The use of state variables has already been described, as has the concept of the init() method on line 6. In the handleResults() method (line 12), the state variables are used to determine whether or not the computation is complete. This is determined by comparing the number of parameter sets returned thus far with the number of parameter sets known to be generated by the SumParmGen class (namely 20 parameter sets). When the 20th parameter set has been processed, then the SumResHandler returns true (line 20). This causes the DCL system to shut down. Otherwise, the SumResHandler simply keeps a running total of the sum found so far. Another important aspect of this class is that it uses the resultLog class to log the final result to the screen or to file, depending on the configuration of the server (line 19).

With these three classes, it is possible to add up all the integers from 1 to 100. The procedure to run the DCL components is simple. First, the three developer defined classes must be copied into the same directory as the DCL server components. The developer must also ensure that the server initialisation file is correctly set up, as described in Section 6. Then, all that remains is for the DCL client components to be started by the client-side administrators (who must configure the corresponding client initialisation file), which will start the process of requesting parameter sets to be processed.

Sample initialisation files for this computation are given below:

Sample server.ini

```
# Sample server init file
server.port=12345
dev.task=SumTask
dev.resultHandler=SumResHandler
dev.parameterGenerator=SumParmGen
```

Sample client.ini

```
# Sample client init file
server.ip=127.0.0.1
server.port=12345
```

If any value is not explicitly set in the initialisation file, then the default value for that option is used. Also, an line which starts with a # symbol is considered a comment, and is ignored.

The command line to start the server is of the form: java ServerEngine ¡initFile¿ where ¡initFile¿ is the initialisation file that the server should use. The command line to start the client is similar: java ClientEngine ¡initFile¿

Using these simple classes as guidelines, a developer can begin building more complex distributed computations.