



NUI MAYNOOTH

Ollscoil na hÉireann Má Nuad

National University of Ireland, Maynooth
MAYNOOTH, CO. KILDARE, IRELAND.

DEPARTMENT OF COMPUTER SCIENCE
TECHNICAL REPORT SERIES

Java Distributed System: Developer Manual

Thomas Keane

NUIM-CS-TR-2003-03

Java Distributed System: Developer Manual

Thomas Keane

(communicated by Tom Naughton)
Department of Computer Science,
National University of Ireland,
Maynooth,
Ireland.

Date: March 2003 (updated May 2004)

Technical Report: NUIM-CS-TR2003-03

Email: tkeane@cs.may.ie

Homepage: <http://www.cs.may.ie/distributed/>

Keywords: distributed computing, programmable, Java, MIMD

Abstract

A distributed Java platform has been designed and built for the simplified implementation of distributed Java applications. Its programmable nature means that code as well as data is distributed over a network. The generality of the system is demonstrated through the emulation of a MIMD (multiple instruction, multiple data) architecture. One of the key features of our system is that it can dynamically alter the size of work units to achieve the optimal processing time per unit. The user of the system is only required to extend 2 Java classes to fully implement a distributed computation. This manual and the supporting webpage give all of the information required to completely design, compile, and test such a computation so that it can be run on the distributed system.

Introduction

This manual is intended as a guide to perspective developers that wish to program a particular problem to run on the Java distributed system at the Department of Computer Science, National University of Ireland Maynooth. It gives details of the general structure that a particular problem should follow in order to be accepted to run on the system. The design of the system can be described as a client server based system. In this model, the distributed application is divided into two parts (see Fig. 1), one part residing on each of the two computers that will be communicating during the distributed computation. The client side of the application resides on the machine that initiates the distributed request. The server side of the application resides on the machine that receives and executes the distributed request. In this model, two different sets of code are produced – one that runs as a client, the other as a server.

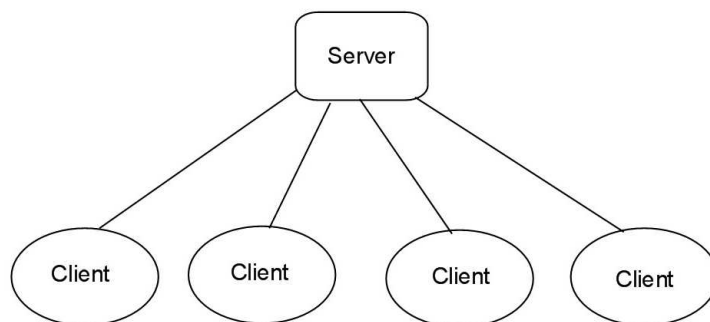


Fig. 1: Overview of Client Server model

To program the system with a given problem, all the developer is required to have is a good understanding of the particular problem they wish to parallelise and a good working knowledge of the Java programming language. To set up a distributed computation, two Java classes must be extended (corresponding to the two sets of code – one that runs on the client and the other that runs on the server). These are the DataManager class and the Algorithm class. Each of these parent classes are part of the system. The developer must overwrite and implement certain methods in order to set up a distributed computation. Once the developer has finished implementing these methods, the problem should be compiled and tested as specified in section 3 of this manual. As an example, there is a complete code listing in appendix A for a sample problem that has been run to completion through the distributed system. There are more complicated examples of problems on the website at the URL above. To facilitate the matching of the computational requirements for particular problems to the donor machines in the system, the user is also asked to specify the minimum memory and CPU requirements for their problem when adding their problem to the system.

1 Problem Suitability

Firstly it should be stated that not all applications are suitable for distributed computing. A potential developer of a distributed computation should think of tasks that take days, weeks, months, or even years to complete on a single machine. It has been widely acknowledged that there are a number of clearly identifiable characteristics that a problem should exhibit in order to be suitable for a distributed computing implementation [1, 2, 3]. The most important characteristic is the ability to split a problem into independent units that can be processed individually. Furthermore, if a distributed system is to operate successfully in an environment as diverse as the Internet where network communication links vary greatly, then it is desirable for the volumes of data being transferred across the network to be kept to a minimum.

1.1 Coarse Grained Parallelism

With the advent of many large scale Internet based supercomputing projects [4, 5, 6, 7], a class of algorithmic parallelism referred to as ‘coarse-grained parallelism’ has emerged as a means of describing the suitability of problems to large scale distributed computing. Coarse grained parallelism refers to the way in which a single large problem can be easily split up into discrete independent sub-blocks that can be processed individually with little or no communication required between the sub-blocks. There a number of areas where this characteristic is clearly evident such as image processing, cryptography code breaking, mathematical applications such as searching for prime numbers, certain bioinformatics applications such as simulating protein folding and drug design [5, 6, 7, 8, 9].

1.2 High Compute-to-Data Ratio

In any large scale distributed computing system, it is expected that donor machines may be located anywhere on the Internet with communication links ranging from the fastest gigabit networks right down to home users with standard 56K phone dial-up connections. This network heterogeneity makes it infeasible for a distributed system to routinely transfer large volumes of data between nodes of the system. Therefore it has been identified that the most suitable problems to distributed computing should exhibit a high ‘compute-to-data’ ratio. This means that a very small amount of data generated from a problem should take a long time to process on a donor machine. A good example of this is in cryptographic or mathematical applications where just two numbers can define an entire search range for a donor machine to search within. Conversely the processing of work units should not generate large volumes of data to be transferred back to a controlling server.

2 Programming the System

Our main goal was to reduce the programming complexity that can be associated with other parallel programming languages, by allowing the developer to program their distributed application at a level that hides all of the lower level details involved in physically distributing out a computation among a heterogeneous set of processors. Therefore to fully define a distributed computation on our system, the developer is only required to extend two Java classes. The `DataManager` class runs on the server while the `Algorithm` class runs on the client. The developer must implement certain methods in order to set up their distributed computation. Extra Java libraries or user-defined classes can also be included with a problem and are made available to the problem through the standard Java library mechanisms (see `import` keyword in Java [10]).

2.1 DataManager Class

The `DataManager` class is a module of the server. The purpose of the `DataManager` is to generate all work units to be sent to the clients, process all results returned by clients, monitor and adjust the granularity of work units, generate status information on the problem to be returned to the remote interface, and terminate the distributed computation. The `DataManager` parent class consists of a number of methods that a developer must implement (corresponding to the functionality outlined in the previous sentence) in order for a computation to be accepted by the system. Each problem that is entered into the system is given its own working directory on the server to use during a computation. Typically this directory is used to create log files, store temporary data, record results, and store data relating to the problem. This directory is preset by the server when a problem is entered into

the system and is available to a `DataManager` through a final variable called `PROBLEMDIRECTORY` (type `java.io.File` – see Java API). When the computation is finished, the contents of this directory are zipped and returned to the user as the results set. It should be noted that each of the `DataManager` methods outlined below should take the minimum amount of complete. Any tasks in the `DataManager` that will take a long while to complete should be implemented using a Java thread.

```
public extendedDataManager() throws Throwable
{
    //initialise DataManager here

    //open problem directory
    File tempFile = new File( PROBLEMDIRECTORY,
"temp" );
}
```

Fig. 2: General format of an extended `DataManager` constructor

The developer must implement a default constructor for their extended `DataManager` class. Any uncaught exceptions or errors will be fed back to the server via the `throws` clause. The server will automatically report these uncaught exceptions back to the user via the remote interface. The sample `DataManager` constructor shown in Fig. 2 also shows a user can create a file in the problem's working directory on the server.

The next method that has to be implemented is the `generateWorkUnit()` method. This method is called by the system every time a client requests a work unit to process. This method should perform whatever calculation is necessary to generate a work unit. The return type of this method is `java.util.Vector`. The `Algorithm` running at the client receives this `Vector` as its work unit to process. Therefore, it is usual for elements of this `Vector` to be explicitly type-casted back to their original types in the `Algorithm` (via Java's safe type casting mechanisms). If at any time in the problem there are no work units currently available, then the `DataManager` should return `null`. This indicates to the server that the problem has no work units to issue at this time. Fig. 3 shows the general format of this method.

```
public Vector generateWorkUnit() throws Throwable
{
    Vector workUnit = new Vector();

    //fill dataUnit here

    return workUnit; //or return null
}
```

Fig. 3: General format of `generateDataUnit()` method

The third method that must be implemented is the `processResults()` method. This method is called every time a client returns a set of results. There are two parameters to this method. The first is the unique work unit ID (type is `java.lang.Long`) that is given to every work unit as it is generated by the `generateDataUnit()` method. The second parameter (type `java.util.Vector`) is the set of results that were returned by the `Algorithm` running on the client. If the distributed computation is finished then this method

should return `true`, otherwise it returns `false`. When this method returns `true`, the server removes the computation from the system and compresses the problem's working directory before it is downloaded by a user. The general format of this method is shown in Fig. 4.

```
public boolean processResults( Long unitID, Vector
results ) throws Throwable
{
    //process results set here

    if( <problem is finished> )
    {
        return true;
    }
    else
    {
        return false;
    }
}
```

Fig. 4: General format of `processResults()` method

There are a number of other methods that must also be implemented by the developer. These are the `adjustGranularity()`, `getStatus()`, and `closeResources()` methods. Our system is expected to operate in an unpredictable environment where network and processing resources are extremely dynamic. Therefore our scheduling algorithm includes a facility where the parallel granularity of problems running in the system be updated continually throughout the lifetime of a computation. The dynamic updates of a problem's granularity is implemented via the `adjustGranularity()` method. This method is called periodically by the server with one parameter corresponding to the percentage (positive or negative) of how much the parallel granularity should be adjusted so that the average processing time will match the optimal processing time.

```
public void adjustGranularity ( int percent ) throws
Throwable
{
    //assuming granularity is controlled by a
    //variable called 'granularity'
    granularity = granularity + ((int) (
granularity * percent ) / 100 );
}
```

Fig. 5: General format of the `adjustGranularity ()` method

This percentage is based on the average processing time of previous result sets computed using the exponentially weighted moving average function. Hence the problem is constantly adjusted to respond to changes in the dynamic heterogeneous network of clients available to the system. The general format of this method is outlined in Fig. 5.

The `getStatus()` method is called every time the remote interface makes a status request to the server about this particular problem. The information returned by this method is of type `java.lang.String` and is expected to contain some useful information about the current state of the computation. The information returned is printed to the screen of the remote interface. Fig. 6 shows the general format of this method.

```

public String getStatus() throws Throwable
{
    String s = //something meaningful about
problem progress
    return s;
}

```

Fig. 6: General format of `getStatus()` method

The `closeResources()` method is called just before the problem is removed from the system. The purpose of this method is to close any resources (e.g. files, directories, input/output streams) that may be open. The user must implement the necessary code to close any resources so that the problem files can be zipped and subsequently deleted from the server's disk. There are no parameters or return values from this method. The general format of this method is shown in Fig. 7.

```

public void closeResources() throws Throwable
{
    //close all resources
}

```

Fig. 7: General format of `closeResources()` method

Each of these methods is an abstract method in the parent `DataManager` class so all of these methods must be implemented before a computation will be accepted by the system.

2.2 Algorithm Class

The `Algorithm` is the piece of code that runs on the client and processes the work units generated by the `generateWorkUnit()` method of the `DataManager`. Only one method must be implemented by the `Algorithm` – the `processUnit()` method. Typically an `Algorithm` can be made up of a number of sub-methods to process different types of work units generated by the `DataManager`. The work unit generated by the `DataManager` is passed as a parameter to the `processUnit()` method and all elements of this `Vector` can be type-casted back to their original types via Java's safe type-casting mechanisms. The results of the work unit are returned from this method as a `Vector` and are subsequently returned to the `processResults()` method of the `DataManager`. The general format of the `processUnit()` method is illustrated in Fig. 8. Any uncaught exceptions or errors will be caught by the `throws` clause of this method and fed back to the server where they are logged in the problem log files that are included with the results set of every computation. If an exception occurs in the same work unit on several different donor machines, then the entire problem is removed from the system.

2.3 Extra User defined Classes and Libraries

When designing our programming API, we recognised that developers may wish to utilise extra third party classes or Java libraries in their distributed application. We have included a facility whereby a user can specify extra libraries (Java JAR files) or class files when they are entering their problem into the system using the remote interface (see section 3.3). To utilise these extra libraries in either the `Algorithm` or `DataManager`, the developer should follow the standard Java mechanisms (using the `import` keyword at the top of each source file [10]).

```

public Vector processUnit( Vector workUnit ) throws
Throwable
{
    //process the work unit here

    //fill a vector with results of computation
    Vector results = new Vector();

    return results;
}

```

Fig. 8: General format of the processUnit method of an Algorithm

5.2.4 Problem Data

We recognised that for certain problems it may be necessary to transfer relatively large amounts of data between the server and client. Therefore we have included a feature in our programming model that makes it possible for large amounts of data to be transferred to the Algorithm running on the client. It should be noted that this feature should only be utilised where there is a high bandwidth network connection between the server and every client. These data files can be specified by the user when they are submitting their computation to the system (via the remote interface). These files are written to the problem's working directory on the server (see section 2.1 - PROBLEMDIRECTORY variable). Whenever a problem data file is required by the Algorithm, the Algorithm should simple attempt to open the file using the PROBLEMDIRECTORY variable set as the parent. The client software will automatically download this file from the server if it has not already been downloaded to the client.

5.4 Compiling, Testing, and Submitting a Problem

5.4.1 Compiling Problem Files

Once a developer has completed the design and implementation of their distributed application, the Algorithm and DataManager files can be compiled using the standard javac command with one minor addition. The file compile.jar should be included in the classpath of the javac command (using the -classpath option). A copy of this file is included in the standard distribution of the system.

5.4.2 Testing a Problem

We found that it may be useful for a developer to have the ability to test their distributed application on a single machine while it is still in development. Although it is possible to install the distributed system on a single machine, we developed a simple Java application the can be used to quickly test a distributed computation on a single machine. This application simulates the calls that are made to the DataManager and Algorithm by the distributed system.

As noted previously, this distributed system is ideally suited to problems that fit into the class of coarse grained parallelism with a high 'compute-to-data' ratio. Hence it is envisaged that each work unit that is sent out should take substantially longer to process than it took to transfer the data over the network. This application can be used by a developer to ascertain an initial estimate of a suitable parallel granularity (amount of processing to be completed by each donor machine per work unit) for their problem. This application is included in the standard distribution of the distributed system and is run from the command line with the following command:

```
java -jar problemTester.jar <Algorithm> <DataManager>
```

Any extra Java libraries that are required by the problem should be included as command line parameters after the Algorithm and DataManager. Any problem data files for the particular problem being tested should be included in the same directory where the problemTester.jar is located.

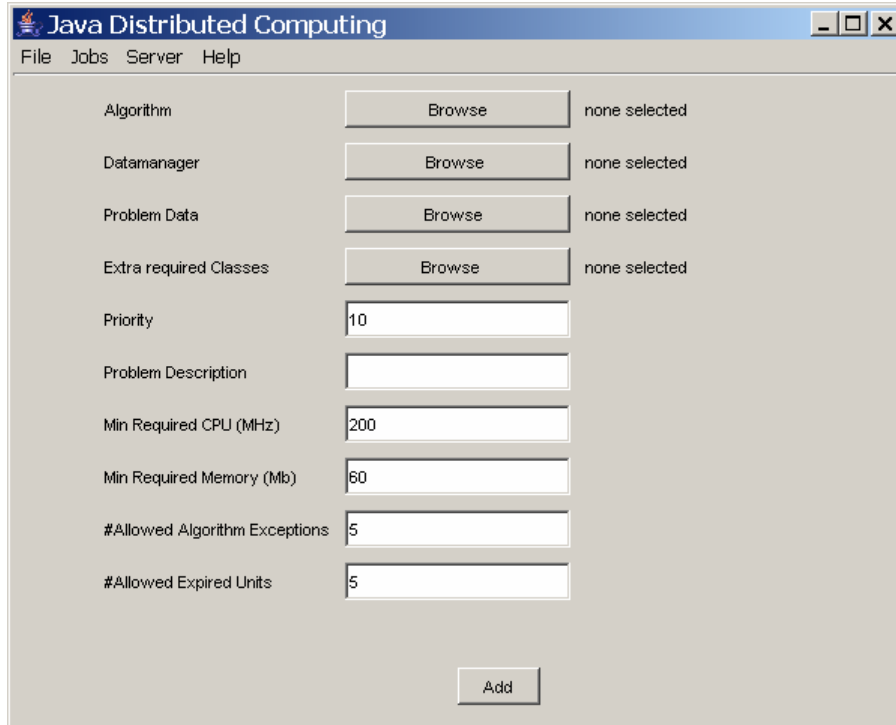


Fig. 9: Screenshot of job add screen on the remote interface

5.4.3 Submitting a Computation

When a distributed computation has been compiled and tested appropriately, the problem can be submitted to the server via the remote interface. To submit a problem to the system, the user is required to be connected to the server in administrator mode. New jobs are added to the system via the jobs-add screen (shown in Fig. 9). The user selects the files for their distributed computation using the various 'Browse' buttons. The user should give each computation a unique name using the 'Problem Description' text box. For a situation where a user has a particularly processor or memory intensive computation, we have provided a facility where the user can specify the minimum processor or memory requirements that any potential donor machine should meet to process work units for the computation. The priority field is used to prioritise computations in the system. So if a user requires the results of their computation very soon, then the priority of the job should be increased from the default value. The last three fields relate to the tolerance of the distributed system to exceptions in the Algorithm and how many times an individual work unit can expire before the system will remove the problem. Most users do not need to concern themselves with these fields and these fields can normally be left to the default values. The computation is submitted to the system when the user presses the 'Add' button. If there is an exception generated while the remote interface adds the computation to the server, then an error message will be displayed. Once a computation has been successfully submitted to the distributed system, the progress of the computation can be monitored in real-time using the remote interface.

References

- [1] Dubey, P., Adams III, G.B., and Flynn, M. (1995) Evaluating Performance Tradeoffs Between Fine-Grained and Coarse-Grained Alternatives, *IEEE Transactions on Parallel and Distributed Systems*, 6(1)
- [2] Coddington, P.D.D. (1993) An Analysis of Distributed Computing Software and Hardware for Applications in Computational Physics, *Proceedings of the Second International Symposium on High Performance Distributed Computing (HPDC-2)*, pp. 179-186, Spokane, WA
- [3] Sperber, M., Klaeren, H., and Thiemann, P. (1997) Distributed Partial Evaluation, *Proceedings of the Second International Symposium on Parallel Symbolic Computation*, pp. 80-87, Hawaii
- [4] Korpela, E., Werthimer, D., Anderson, D., Cobb, J., and Lebofsky, M. (2001) SETI@home-Massively Distributed Computing for SETI, *IEEE: Computer Science and Engineering*, 3 (1), pp. 77-83
- [5] Woltman, G. (1996) Great Internet Mersenne Prime Search
<<http://www.mersenne.org>>
- [6] Larson, S.M., Snow, C.D., Shirts, M.R., and Pande. V.S. (2003) Folding@Home and Genome@Home: Using distributed computing to tackle previously intractable problems in computational biology, to appear in *Computational Genomics*, Richard Grant editor, Horizon Press
- [7] Allen, M. (1999) Do-it-yourself climate prediction, *Nature*, 401, 642
- [8] Krieger, E. and Vriend, G. (2002) Models@Home: distributed computing in bioinformatics using a screensaver based approach, *Bioinformatics*, 18 (2), pp. 315-318
- [9] FightAIDS@Home, accessed March 2004
< <http://fightaidsathome.scripps.edu/index.html>>
- [10] Flanagan, D. (2002) Java in a Nutshell (4th Edition), O'Reilly & Associates, UK, ISBN-0596-0028-31

Appendix A

Simple Sorting Problem

Sorting is one of the most fundamental tasks for computers. The following code is the code for distributing the task of sorting large amounts of numbers. The choice of sorting algorithm is completely up to the developer programming the problem. This is a complete problem and has been tested and ran to completion on the distributed system. Although this problem is a trivial one, it serves as a good example to program a distributed application to run on the system. The full code for the problem follows.

DataManager

```
/*
Thomas Keane, 16:10 03/05/04
A DataManager for a simple sorting problem
This generates groups of random numbers and sends them out to be sorted by the
clients
The results are then wrote to file in the problems working directory

Copyright (C) 2004 Thomas Keane

This program is free software; you can redistribute it and/or
modify it under the terms of the GNU General Public License
as published by the Free Software Foundation; either version 2
of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
GNU General Public License for more details.
*/

import java.io.*;
import java.util.*;
import java.util.logging.*;

public class SortingDataManager extends DataManager
{
    private int size;
    private PrintStream ps;
    private int resultsReceived;
    private int unitsIssued;
    private int totalUnits;
    private Random random;

    private Logger problemLog;

    //init() method - called once by the system
    public SortingDataManager() throws Throwable
    {
        //set up a log file for recording problem progress
        problemLog = Logger.getAnonymousLogger();

        FileHandler phandler = null;
        File logsDir = new File( PROBLEMDIRECTORY, "probLog" );
        logsDir.mkdir();

        phandler = new FileHandler( logsDir.getAbsolutePath() +
"/problem%g.log", 10000000, 5, false );

        //formatter for the logger
        SimpleFormatter simple = new SimpleFormatter();

        phandler.setFormatter( simple );
        problemLog.setUseParentHandlers( false );
    }
}
```

```

        problemLog.addHandler( phandler );

        problemLog.info( "Entering DataManager" );

        //size of each unit
        size = 70000;

        //total number of units to be processed
        totalUnits = 500;

        //number of units issued so far
        unitsIssued = 0;

        //number of results received so far
        resultsReceived = 0;

        //set up a writer to the results file
        File r = new File( PROBLEMDIRECTORY, "results.txt" );
        FileOutputStream fos = new FileOutputStream( r );
        ps = new PrintStream( fos );

        random = new Random();
        problemLog.info( "DataManager constructor complete" );
    }

    public Vector generateWorkUnit() throws Throwable
    {
        //if have issued all data units
        if( unitsIssued == totalUnits )
        {
            problemLog.info( "Finished Issuing units: " + unitsIssued );
            //finished issuing units
            return null;
        }

        //create vector
        Vector unit = new Vector();

        //add the random numbers to the vector
        int[] numbers = new int[ size ];

        for( int i = 0; i < numbers.length; i ++ )
        {
            numbers[ i ] = random.nextInt();
        }

        //add the array to the vector
        unit.add( numbers );

        unitsIssued ++;

        problemLog.info( "Issuing unit: " + unitsIssued );
        return unit;
    }

    public boolean processResults( Long uID, Vector results ) throws Throwable
    {
        //get the results array from the vector
        int[] sorted = (int[]) results.get( 0 );

        problemLog.info( "Processing results set " + resultsReceived );

        //send the results to file
        for( int i = 0; i < sorted.length; i ++ )
        {
            ps.println( sorted[ i ] );
        }
        ps.println( "\n" );
    }

```

```

        resultsReceived ++;

        if( resultsReceived == totalUnits )
        {
            //received all of the results
            problemLog.info( "All results Received - exiting" );
            ps.close();
            return true;
        }
        else
        {
            //more results to come - not finished
            return false;
        }
    }

    //alter the unit size accordingly
    public void adjustGranularity( int percent ) throws Throwable
    {
        problemLog.info( "Adjusting unit size by " + percent + "%" );

        //adjust the size of the work units
        size += ( ( size * percent ) / 100 );

        problemLog.info( "Granularity: " + size );
    }

    //method to close any open streams - in case of exception in datamanager
    public void closeResources() throws Throwable
    {
        try
        {
            ps.close();
        }
        catch( Exception e ){ }

        Handler[] handlers = problemLog.getHandlers();
        for( int i = 0; i < handlers.length; i ++ )
        {
            problemLog.removeHandler( handlers[ i ] );
            handlers[ i ].flush();
            handlers[ i ].close();
        }
        problemLog = null;
    }

    //method to report status of problem via gui interface to system
    public String getStatus() throws Throwable
    {
        return "Number units processed " + resultsReceived + " out of " +
totalUnits;
    }
}

```

Algorithm

```
/*
Thomas Keane, 16:11 03/05/04
Algorithm that accepts an unsorted array and sorts it
the results that are sent back are a sorted array of ints
```

```
Copyright (C) 2004 Thomas Keane
```

```
This program is free software; you can redistribute it and/or
modify it under the terms of the GNU General Public License
as published by the Free Software Foundation; either version 2
of the License, or (at your option) any later version.
```

```
This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
GNU General Public License for more details.
```

```
*/
import java.util.*;
import java.io.*;

public class sortingAlgorithm extends Algorithm
{
    public Vector processUnit( Vector workUnit ) throws Throwable
    {
        //get the unsorted array
        int[] data = (int[]) workUnit.get( 0 );

        //bubble sort the data
        for( int i = 0; i < data.length; i ++ )
        {
            for( int j = 0; j < i; j ++ )
            {
                if( data[ i ] < data [ j ] )
                {
                    int temp = data[ i ];
                    data[ i ] = data[ j ];
                    data[ j ] = temp;
                }
            }
        }

        //put the results into the results vector - client will send this back
        to the server
        Vector results = new Vector();
        results.add( data );

        return results;
    }
}
```