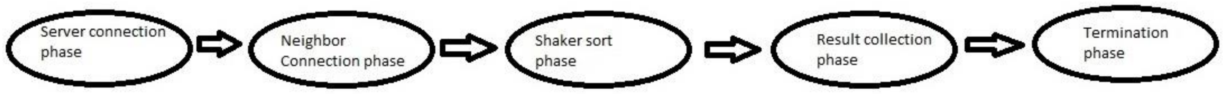
NWEN 303 Yao Hong Chun
Project 2 Report 300232587

Introduction

In this project, distributed computing is demonstrated. The problem is to sort a list of random numbers using the shaker-sort algorithm, but solving it on multiple machines instead of just one. Each machine can only communicate to adjacent neighbors, left or right and no others. Communication of each machine requires precise planning of a protocol to follow.

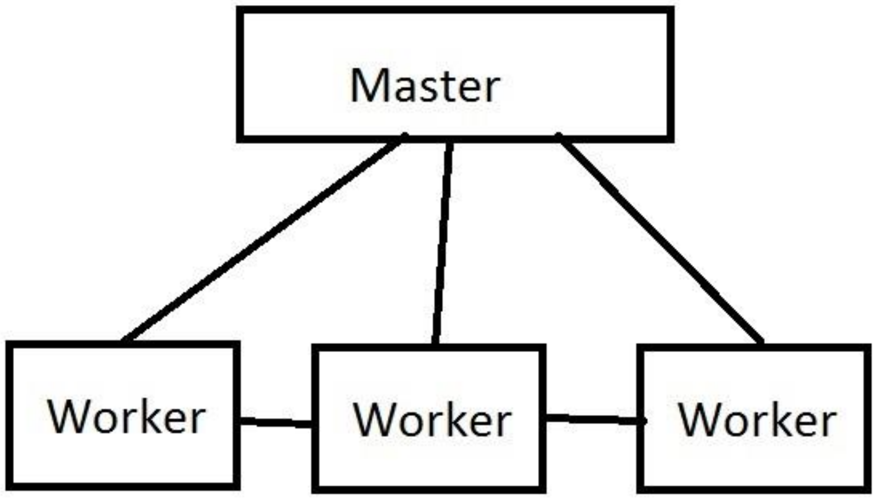
Design and Implementation



The diagram above shows that the program goes through five phases; server connection phase, neighbor connection phase, shaker sort phase, result collection phase and termination phase.

Communication among machines is perhaps the core of this project. The server connection phase commences first.

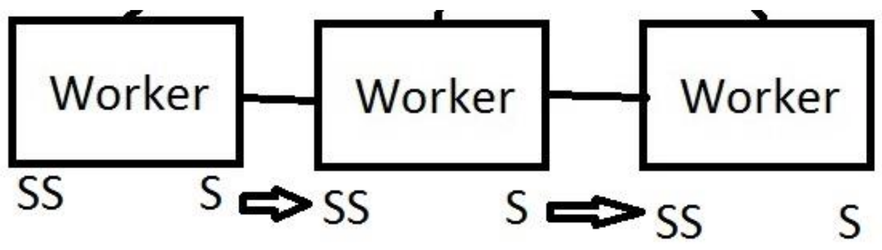
Server Connection phase:



Each box in this diagram represents a machine; the master is a machine where the server is run. Initially, the master will wait until the desired number of workers has connected to the server. Once there are enough workers, the master will generate random numbers and distribute them equally to the workers that are on different or same machines. Each worker will need to know the address of the master server and the port to use, which is 9000.

After the server connection phase, where the workers connect to the server and retrieve required information, the neighbor connection phase commences.

Neighbor Connection phase:



The workers need a way to communicate with each other, but only to adjacent ones. In this diagram, each client has a *server socket*, and a *socket*. The server socket will accept connections coming from the left side. And the socket will connect to the right neighbor. This means that each worker will need to know the address and the port to use for the right neighbor before it can begin. Only the master server knows the address of all workers, so the master will send to each worker the required address of the right neighbor. The worker on the left-most side will not need to start up a server for its left neighbor to connect to because it won't have a left neighbor. Likewise with the right-most worker, it will not need a socket to connect to the right neighbor because it doesn't have a right neighbor.

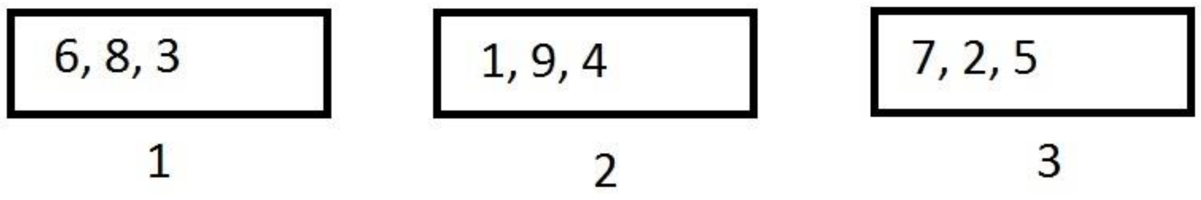
The neighbor connection phase will proceed like this if there are 3 workers:

Workers 2 and 3 will start up their server for their left neighbor to connect, and at the same time, Workers 1 and 2 will connect to its right neighbor using an address and a port.

Once the workers have established neighboring connections the shaker sort algorithm phase commences.

Shaker Sort Phase:

To sort the numbers, each worker needs to send correct numbers its neighbors.



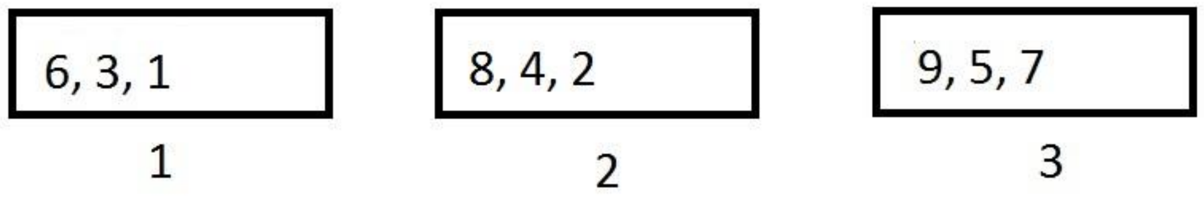
In this diagram, each box represents a worker with their numbers received from the master server. Worker 1 and 2 will call the bubbleUp() method which returns the highest number which would be 8 and 9 respectively and then send the highest number to their right neighbor workers 2 and 3. Worker 3 will call bubbleUp() but won't send it to any workers.

Workers 2 and 3 will then call the bubbleDown() method which returns the lowest number, in this case, it is 1 and 2 respectively and then send it to their left neighbor workers 1 and 2.

Worker 2 and 3 will receive the highest number from their left neighbor and will compare it with the number at the 0th index of the number array. If it is greater than the number at the 0th index then it will replace the number, otherwise it won't do anything with the number that is received.

Workers 1 and 2 will receive the lowest number from their right neighbor and will compare it with the number at the very end of the array. If it is less than the number at the very end of the array then it will replace the number, otherwise it won't do anything with the number that is received.

After one run of the algorithm, the contents of each worker will now look like this:



The algorithm keeps repeating even when it is sorted *locally*, this is because there may be a case where a worker has sorted locally, but not globally. In the diagram above, if worker 1 has sorted its numbers so that it is {1, 3, 6} then it is sorted, but only locally. The number 6 should not be in worker 1, but worker 2 instead.

Each worker will know that its numbers are sorted when there are no swaps performed when calling the bubbleUp() or bubbleDown() methods. If there are no swaps, the worker will notify the master server that its numbers are sorted, the master will know which workers are sorted. If and only if ALL workers have finished sorting, the master will detect this and tell ALL the workers to begin the result collection phase.

Result Collection Phase:

Once every worker has been notified by the master that it's time to stop, the workers will send its results to the master. Not much of a phase.

Termination Phase:

Once every worker has sent their results to the server, it will terminate and once the master receives results from every worker, it will terminate.

Testing

Testing was done using print statements and the useful debugging tool in Eclipse. Numbers inside the array of each worker can be inspected and validated. There weren't many issues that were run into mainly because of precise planning and past networking experience.

Results and performance

The results are always correct, but very poor in terms of performance. There is too much overhead with the use of network communication which causes the operation to slow down significantly. Also, the algorithm was not 'truly concurrent'; it was as if the bubble sort algorithm was performed on one machine but with the added overhead time of the network communications. The performance is better with fewer machines involved in the operation.

How to run the program

There are two runnable jar files, one called Master and the other called Worker. The Master must be run first before the Workers.

To run the Master, open the terminal and type in

```
java –jar Master.jar
```

and press enter. It will ask how many workers to wait for, and then how many numbers in total to sort.

To run the Worker, open the terminal and type in

```
java –jar Worker.jar
```

and press enter. It will ask for the server address of the master, be sure to find out the master's IP address first. The worker must be run multiple times depending on how many workers the server needs to begin operation.

Conclusion

Although this project did not involve any true concurrency to sort numbers quicker, it did prove to be difficult with the need of network communications. If this algorithm was to achieve true concurrency, it wouldn't be that much faster because the numbers that each worker has may need to be in another worker elsewhere.

Appendix

Master.java

```
import java.io.IOException;
import java.net.ServerSocket;
import java.net.Socket;
import java.util.ArrayList;
import java.util.Collections;
import java.util.Scanner;


/**
 * Represents the server.
 * The server will wait for a certain amount of workers to connect first.
 * Then it will distribute the work equally among the workers and other required
information.
 * It will then wait for the workers to sort using the bubble up and down algorithm.
 * @author Yao Hong Chun
 *
 */
public class Master extends Thread {
	private ServerSocket serverSocket;
	private ServerSocket resultSocket;
	private ArrayList<Connection> connections = new ArrayList<Connection>();
	private int currentConnections=0;//number of connections connected to the
master
	private ArrayList<Integer> numbers;
	private int maxConnections;
	private int size;
	boolean finished=false;
	/**
	 * @param maxConnections - Max number of connections to wait for.
	 * @param size - The total number of numbers to distribute.
	 */
	public Master(int maxConnections, int size){
		this.maxConnections=maxConnections;
		this.size = size;
	}
	public void run(){
		try {
			//ACCEPT INCOMING CONNECTIONS
			serverSocket = new ServerSocket(9000);
			System.out.println("Server up and running!");

			while(currentConnections < maxConnections){
				System.out.println("Still need " + (maxConnections-
currentConnections) + " workers to start.");
				Socket client = serverSocket.accept();
				Connection connection = new Connection(client);
				System.out.println("Incoming connection...");

				int request = connection.receiveInteger();
				System.out.println("Received request!");
```

```
                        if(request ==0){
                                System.out.println("A new worker has connected.");
                                currentConnections++;
                                connections.add(connection);
                        }

                        for(Connection c: connections){
                                c.sendInteger(maxConnections-currentConnections);
                        }

                }
                System.out.println("A total of " + currentConnections + " workers
have connected! Now distributing work!");

                //CREATE RANDOM NUMBERS
                numbers = new ArrayList<Integer>();
                for(int i = 1;i<=size;i++){
                        numbers.add(i);
                }
                Collections.shuffle(numbers); //Shuffle the numbers around

                int[] nums = convertIntegers(numbers);
                Worker.printArrayValues(nums);
                //SEND NUMBERS TO CLIENTS
                int from = 0;
                int to = size/currentConnections;
                int i = 1;
                ArrayList<AwaitCompletion> await = new
ArrayList<AwaitCompletion>();
                for(int k =0;k<connections.size();k++){
                        System.out.println(i+":Created numbers to send!
Sending...");

                        if((k+1)<connections.size()){ //send address of right
neighbor
                                String address=connections.get(k+1).getAddress();
                                byte[] data=address.getBytes("UTF-8");
                                connections.get(k).sendInteger(data.length);
                                connections.get(k).sendByteArray(data);
                        }
                        else{
                                connections.get(k).sendInteger(-1);
                        }
                        connections.get(k).sendInteger(i); //send position
                        connections.get(k).sendArray(createSegment(from,to,nums));
                        i++;
                        from = to;
                        to = to+(size/currentConnections);
                        System.out.println("Done!");
                        //Wait for completion
                        AwaitCompletion ac= new
AwaitCompletion(connections.get(k));
                        await.add(ac);
```

```
						ac.start();
					}
					while(!finished){ //Constantly check if the workers are finished
						//If all workers are finished then set finished =true;

						for(Connection c:connections){
							if(!c.isDone()){
								finished = false;
								break;
							}
							else{
								finished = true;

							}
						}
					}
					System.out.println("All finished!! Notifying all workers...");
					for(Connection c: connections){
						c.sendInteger(1);
					}
					for(AwaitCompletion aw: await){
						aw.shutdown();
						aw.interrupt();
					}

					serverSocket.close();
					System.out.println("Retrieving results...");
					int temp = 0;
					resultSocket = new ServerSocket(8999);
					while(temp < maxConnections){
						Socket s = resultSocket.accept();
						Connection c = new Connection(s);
						System.out.println("Results from worker "+(temp+1)+":");
						Worker.printArrayValues(c.receiveArray());
						temp++;
						System.out.println();
					}

					System.out.println("Operation complete");

				} catch (IOException e1) {
					e1.printStackTrace();
					System.exit(-1);
				}
		}

		/**
		 * This class instantiates for every connection to a worker.
		 * It will keep receiving updates from the worker, whether or not the worker
has finished.
		 * This information is required for the server to know whether or not ALL
workers have finished,
		 * and if they have all finished, the server will tell the workers to
terminate.
		 * @author Yao Hong Chun
	 *
	 */
	private class AwaitCompletion extends Thread{
		private Connection connection;
		public AwaitCompletion(Connection c){
			System.out.println("Started awaitcompletion");
			connection = c;
		}

		public void run(){
			try{
			while(!finished){
				int status = connection.receiveInteger();
				if(status > 0){ //if status is over 0 then the worker is 
done. The worker will send it's position if its done, otherwise -position.
					connection.setDone();
					System.out.println("Worker position " + status + " 
has sorted");
				}
				else{
					connection.notDone();
				}
			}
			}
			catch(Exception e){
			}
		}

		public void shutdown(){
			connection=null;
		}

	}
	public int[] createSegment(int from,int to,int[] nums){
		int[] n = new int[to-from];

		int j =0;
		for(int i = from;i<to;i++){
			n[j] = nums[i];
			j++;
		}
		System.out.println("FROM: " + from + " TO: " + to);
		Worker.printArrayValues(n);
		return n;
	}

	/**
	 * Puts all the numbers in an arraylist into an array
	 * @param numbers The arraylist containing all the numbers
	 * @return The array containing the numbers that was in the arraylist
	 */
	public static int[] convertIntegers(ArrayList<Integer> numbers){
		int[] nums = new int [numbers.size()];
		for(int i = 0;i<nums.length; i++){
			nums[i] = numbers.get(i).intValue();
		}
		return nums;
	}
	public static void main(String args[]){
		Scanner userinput = new Scanner(System.in);
		int workers = 0;
		while(workers < 2){
			System.out.println("Enter number of workers:");
			workers = userinput.nextInt();
			if(workers < 2){
				System.out.println("Must be more than one worker.");
			}
		}
		System.out.println("Enter number of numbers:");
		int numbers = userinput.nextInt();
		new Master(workers,numbers).start();
		userinput.close();
//		new Master(4,4000).start();
	}

}
```

Worker.java

```
import java.io.IOException;
import java.net.ServerSocket;
import java.net.Socket;
import java.net.UnknownHostException;
import java.util.Scanner;


/**
 * @author Yao Hong Chun
 *
 */
public class Worker implements Runnable{
	private int position;//position of where the worker is working
	private boolean finished = false;
	private boolean sorted = false;
	//Communication to the server
	private Connection serverConnection;

	//Communication to neighbors
	private ServerSocket serverSocket;
	private Socket rightSocket; //Right neighbor
	private int leftPort=9000;
	private int rightPort=9000;
	private int thisPort = 9000;

	Connection leftConnection;
	Connection rightConnection;

	String rightAddress;
	private int[] numbersToSort;
	int length;//length of array
	String serverIp;
	public Worker(String serverIP){
		serverIp = serverIP;
		initializeNumbers(serverIP);
		initializeNeighborCommunications();

		new Thread(this).start();
	}

	//
	public void run(){
		System.out.println("Started.");
		while(!finished){
			if(rightConnection!=null){
				int highest = bubbleUp(numbersToSort);
				rightConnection.sendInteger(highest);
				int rightLowest = rightConnection.receiveInteger();
				if(rightLowest < numbersToSort[numbersToSort.length-1]){
					numbersToSort[numbersToSort.length-1]=rightLowest;
				}
			}

			if(thisPort!=9001 && leftConnection!=null){
				int lowest = bubbleDown(numbersToSort);
				leftConnection.sendInteger(lowest);
				int leftHighest = leftConnection.receiveInteger();
				if(leftHighest > numbersToSort[0]){
					numbersToSort[0]=leftHighest;
				}
			}
			if(thisPort==9001) {
				bubbleDown(numbersToSort);
			}
			if(rightAddress == null){
				bubbleUp(numbersToSort);
			}
		}
		try {
			Thread.sleep(1000);
		} catch (InterruptedException e) {
			e.printStackTrace();
		}
		//close neighboring connections
		if(rightAddress != null){
			rightConnection.shutdown();
		}
		if(thisPort!=9001)
			leftConnection.shutdown();

		//Send results to server
		try {
				Thread.sleep(position*1000);
				System.out.println("Sending result to server...");
				Socket resultSocket = new Socket(serverIp, 8999);
				Connection c = new Connection(resultSocket);
				c.sendArray(numbersToSort);
				c.shutdown();
			} catch (UnknownHostException e) {
				e.printStackTrace();
			} catch (IOException e) {
				e.printStackTrace();
			} catch (InterruptedException e) {
				e.printStackTrace();
			}

			serverConnection.shutdown();
			System.out.println("All connections closed.");
			System.out.println("Result:");
			//print results
			printArrayValues(numbersToSort);
			System.exit(0);
	}

	/**
	 * This runnable object runs concurrently on a separate thread. It keeps 
updating the server by telling its current
	 * state. Whether it is sorted or not. If the server knows that ALL the 
workers are sorted then the server will tell all workers
	 * to terminate.
	 */
	private Runnable finishSearch = new Runnable(){
		public void run(){
			while(!finished){
				if(sorted){
					System.out.println("Sorted!! Telling server 
now...");
					serverConnection.sendInteger(position); //Tell the 
server that this worker has sorted its array completely
					System.out.println("Waiting for reply...");
				}
				else{
					serverConnection.sendInteger(-(position)); //Tell 
the server that this worker hasn't sorted its array completely yet
				}

				try {
					Thread.sleep(1000);
				} catch (InterruptedException e) {
					e.printStackTrace();
				}
				System.out.println("Resending...");
			}
		}
	};

	/**
```

```
	 * This runnable object waits for the server to reply. If the reply is 1, then 
it is ok to terminate everything.
	 */
	private Runnable receiveReply = new Runnable(){
		public void run(){
			while(!finished){
				int reply = serverConnection.receiveInteger();
				if(reply == 1){ //1 means all workers are sorted
					System.out.println("Server says all other workers 
are done and sorted. Terminating now...");
					finished = true;
				}
			}
		}
	};

	/**
	 * Communicates to the master on port 9000. Then waits until there are 
sufficient workers connected to the server.
	 * Once enough workers have connected the master will send address of right 
neighbor, length of array, numbers to sort and a position.
	 * Otherwise it will terminate.
	 */
	public void initializeNumbers(String serverIP){
		boolean connected = false;
		while(!connected)
			try{
				//contact server on port 9000
				Socket contactServerSocket = new Socket(serverIP,9000);
				serverConnection = new Connection(contactServerSocket);
				serverConnection.sendInteger(0); //Send 0 to server.
				connected = true;

				System.out.println("Waiting until server has enough 
workers"); //Wait until server has enough workers.
				int reply = 1;
				while(reply != 0 ){
					reply = serverConnection.receiveInteger();
					if(reply>0)
						System.out.println(reply + " more workers 
needed!");
				}
				if(reply == 0){//OK to receive numbers
					System.out.println("Enough workers!");
					int strLength=serverConnection.receiveInteger();
					if(strLength != -1){
						byte[] data=new byte[strLength];
						serverConnection.readFully(data);
						rightAddress=new String(data,"UTF-8"); 
//receive address of the right neighbor
					}
					else{
						rightAddress = null; //Then this worker is on 
the right most side
					}
```

```
                                        position = serverConnection.receiveInteger();
//receive position
                                        //                          length
=serverConnection.receiveInteger();//receive length of array
                                        numbersToSort
=serverConnection.receiveArray();//receive array
                                        System.out.println(rightAddress);
                                        System.out.println("Received numbers!");
                                        leftPort = leftPort +position-1;
                                        rightPort = rightPort + position +1;
                                        thisPort = thisPort + position;

                                        System.out.println("Position: "+position);
                                        System.out.println("# Numbers to sort: " +
numbersToSort.length);
                                        System.out.println("This port: " + thisPort);
                                        System.out.println("Left port: " + leftPort);
                                        System.out.println("Right port: " + rightPort);
                                        printArrayValues(numbersToSort);
                                        try {
                                                serverSocket = new ServerSocket(thisPort);
                                        } catch (IOException e) {
                                                e.printStackTrace();
                                        }
                                }
                                else {
                                        System.exit(0);
                                }
                        }
                catch(Exception e){
                        System.out.println("Could not reach the server, please enter a
valid IP Address.");
                        Scanner userinput = new Scanner(System.in);
                        System.out.println("Enter IP Address of server:");
                        serverIP = userinput.next();
                        userinput.close();
                }
        }

        /**
         * Initializes neighbor communications. If the workers port is 9001 it means
the worker is on the left most side of the collection of workers therefore
         * it doesn't have a left neighbor. If the worker did not receive an address
to the right neighbor then the worker is on the rightmost side of the collection
         * of workers therefore it doesn't have a right neighbor. Otherwise the worker
will need to connect using a socket to the right neighbor and
         * wait for a client from the left to connect to its server.
         */
        public void initializeNeighborCommunications(){
                System.out.println("\nInitializing neighbor communications...");
                boolean connected = false;
                if(thisPort!=9001){ //If not the leftmost worker
                        new Thread(grabLeftNeighborServer).start(); //Then get
leftneighbor
```

```
			}
			if(rightAddress!=null){ //if not the rightmost worker
				while(!connected){ //Keep trying until connected to the right 
neighbor.
					try{
						System.out.println("Attempting to connect to right 
neighbor...");
						rightSocket = new Socket(rightAddress,thisPort+1);
						rightConnection = new Connection(rightSocket);
						connected = true;
						System.out.println("Connected to right neighbor.");
					}catch(Exception e){
						try {
							Thread.sleep(1000); //Retry every second
						} catch (InterruptedException e1) {
						}
						System.out.println("Connecting to right neighbor 
failed... retrying...");
					};
				}
			}
			new Thread(finishSearch).start();
			new Thread(receiveReply).start();
		}


		/**
		 * This Runnable object runs on a separate thread concurrently and constantly 
waits for the left neighbor to connect.
		 * If the port is 9001 it means the worker is on the left most side of the 
collection of workers therefore
		 * it doesn't have a left neighbor so it doesn't need to run this thread.
		 */
		Runnable grabLeftNeighborServer = new Runnable(){
			public void run(){
				try {
					serverSocket = new ServerSocket(thisPort);
					System.out.println("Started server.");
				} catch (IOException e) {
					//				e.printStackTrace();
				}
				boolean connected = false;
				while(!connected)
					try {
						if(thisPort != 9001){
							Socket leftNeighbor = serverSocket.accept();
							System.out.println("Left neighbor 
connected");
							leftConnection = new 
Connection(leftNeighbor);
							connected = true;
						}
					} catch (IOException e1) {
						e1.printStackTrace();
					}
			}
		};

		/**
		 * @param numbers - The array to bubble up on
		 * @return the highest number in the array
		 */
		public int bubbleUp(int[] numbers){
			int swaps =0;
			for(int i = 0; i<numbers.length-1;i++){
				if(numbers[i] > numbers[i+1]){
					int temp = numbers[i];
					numbers[i] = numbers[i+1];
					numbers[i+1] = temp;
					swaps++;
				}
			}
			checkIfSorted(swaps);
			return numbers[numbers.length-1];
		}

		/**
		 * @param numbers - The array to bubble down on.
		 * @return the lowest number in the array.
		 */
		public int bubbleDown(int[] numbers){
			int swaps =0;
			for(int i =numbers.length-1; i>1;i--){
				if(numbers[i] < numbers[i-1]){
					int temp = numbers[i];
					numbers[i] = numbers[i-1];
					numbers[i-1] = temp;
					swaps++;
				}
			}
			checkIfSorted(swaps);
			return numbers[0];
		}

		/**
		 * Checks if the array is sorted. There are two conditions:
		 * 1) Can only be sorted if certain connections are not null
		 * 2) swaps == 0
		 * @param swaps - number of swaps, if swaps == 0 that means the numbers are 
sorted.
		 */
		protected void checkIfSorted(int swaps) {
			boolean previous = sorted;
			//Can only be sorted if certain connections are not null and swaps == 0
			try {
				if(leftConnection==null)
					sorted = swaps==0&&rightConnection!=null;// if no swaps 
has been performed then the array is sorted.
				if(leftConnection!=null&&rightConnection!=null){
```

```
                        sorted = swaps==0;
                    }
                    if(rightConnection==null){
                        sorted = swaps==0 && leftConnection!=null;
                    }

                    if(previous && !sorted){ //If it was sorted before and became
unsorted again then tell the server immediately.
                        System.out.println("Not sorted anymore, telling the
server...");
                        Thread.sleep(20); //Prevent it from clogging up the stream
                        serverConnection.sendInteger(-(position)); //Tell the
server that this worker hasn't sorted its array completely yet
                    }
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
        }

        /**
         * prints all elements in the array in a line
         * @param num The array to be printed
         */
        public static void printArrayValues(int[] num){
                for(int i = 0;i<num.length;i++){
                        System.out.print(num[i] + " ");
                }
        }

        public static void main(String args[]){
                Scanner userinput = new Scanner(System.in);
                System.out.println("Enter IP Address of server:");
                String ip = userinput.next();
                System.out.println();
                new Worker(ip);
                userinput.close();
                //              new Worker("192.168.1.50");
        }
}
```

Connection.java

import java.io.DataInputStream;

import java.io.DataOutputStream;

import java.io.IOException;

import java.net.Socket;

import java.util.ArrayList;

```
/**
 * This represents a connection between a client and a server.
 * A worker will connect to the server and both the server and worker will create a connection so
 * that they can communicate back and forth.
 * @author Yao Hong Chun
 *
 */
public class Connection {
	Socket s;
	DataOutputStream out;
	DataInputStream in;
	boolean done=false;
	/**
	 * @param sock - The socket of the connected client or server.
	 */
	public Connection(Socket sock){
		s = sock;
		try {
			out = new DataOutputStream(s.getOutputStream());
			in = new DataInputStream(s.getInputStream());
		} catch (IOException e) {
			e.printStackTrace();
		}
	}

	/**Sends an integer to the server or client
	 * @param i - The integer to send
	 */
	public void sendInteger(int i){
		try {
			out.writeInt(i);
		} catch (IOException e) {
		}
	}

	/** Receives an integer from the client or server
	 * @return The integer that was received
	 */
	public int receiveInteger(){
		try {
			return in.readInt();
		} catch (IOException e) {
		}
		return -1;
	}

	/** Sends an array to the client or server, contents of array must be >= 0
	 * @param array - The array to be sent
 */

public void sendArray(int[] array){

        try {

                for(int i =0;i<array.length;i++){

                        out.writeInt(array[i]); //send values

                }

                out.writeInt(-1); //end

                System.out.println("Array successfully sent.");

        } catch (IOException e) {

                e.printStackTrace();

        }

}


public int[] receiveArray(){

        ArrayList<Integer> nums = new ArrayList<Integer>();

        while(true){//receive number of numbers to sort

                int reply = receiveInteger();

                if(reply!=-1)

                        nums.add(reply);

                else{

                        break;

                }

        }

        int[] numbers = Master.convertIntegers(nums);

        return numbers;
	}


	public void sendByteArray(byte[] b){

		try{

			out.write(b);

		}

		catch(Exception e){}


	}


	public void readFully(byte[] b){

		try{

			in.readFully(b);

		}catch(Exception e){}

	}


	public void shutdown(){

		try {

			out.close();

			in.close();

			s.close();

		} catch (IOException e) {

			e.printStackTrace();

		}

	}
	public void setDone(){

		done=true;

	}

	public void notDone(){

		done=false;

	}


	public boolean isDone(){

		return done;

	}

	public String getAddress(){

		return s.getInetAddress().getHostAddress();

	}


}
```