UNIVERSITY OF VAASA

FACULTY OF TECHNOLOGY

COMMUNICATION AND SYSTEMS ENGINEERING

Zhuang ZhiZhong

Implementation and Performance Evaluation of Algorithms Running on Distributed Systems

Master's thesis for the degree of Master of Science in Communication and Systems Engineering submitted for inspection, Vaasa, 13 September 2016

Supervisor	Timo Mantere
Instructor	Tobias Glocker

ACKNOWLEDGEMENT

This thesis aim to study the different time complexity of three sorting algorithms in Raspberry Pi, personal computer and distributed systems.

First of all, I would like to express my sincere gratitude to my thesis instructor Tobias Glocker for his constant guidance and patient instruction during my thesis research. Moreover, I should present my great appreciations to Timo Mantere and the staffs in the University of Vaasa who has provided me the essential devices for my thesis. At last, I would like to express great thanks to my families and friends who give me encouragement and support.

TABLE OF CONTENTS **PAGE**

ABBREVIATIONS

CPU	Central Processing Unit
GPIO	General-purpose input/output
GPU	Graphical Processing Unit
GUI	Graphical User Interface
HDMI	High Definition Multimedia Interface
IDE	Integrated Development Environment
IP	Internet Protocol
IT	Information Technology
JRE	Java Runtime Environment
LAN	Local Area Network
PC	Personal Computer
RAM	Random-Access Memory
SSH	Secure Shell
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
USB	Universal Serial Bus

University of Vaasa
Faculty of Technology

Author:	ZhiZhong Zhuang
Topic of the Thesis:	Implementation and Performance Evaluation of Algorithms Running on Distributed Systems
Supervisor:	Timo Mantere
Instructor:	Tobias Glocker
Degree:	Master of Science in Technology
Degree Programme:	Degree programme in Communications Systems Engineering
Major:	Telecommunication Engineering
Year of Entering the University:	2014
Year of Completing the Thesis:	2016 **Pages:** 65

ABSTRACT:

With the rapid development of technology, people are fully engrossed by the information age. A single computer does not have enough ability to process the huge information and communication that generated on the Internet every day. However, a distributed system offers quick and precise solutions for a variety of complex problems in different fields. There are several definitions on distributed systems, these definitions can be summarized as a system in which hardware or software components located at networked computers communicate and coordinate their actions only by message passing. In comparison to a personal computer (PC), a distributed system has more resources to increase performance. Because distributed system can separate the task when handling complexity problems or massive data.

This thesis focuses on implementation and performance evaluation of algorithms running on distributed systems. Mainly measures the time complexity of algorithms (bubble sort, quick sort, and heap sort) executed on distributed system and a personal computer. Then takes the comparison between them. Distributed systems consist of two Raspberries Pi and a personal computer. Two raspberries Pi regarded as two clients and the personal computer is a server. Sockets are used for the communication between the clients and the server. The Graphical User Interface (GUI) has been implemented on the server. The server generates the random numbers, selects the sort algorithm, separates the task and sends it to two clients. On the client, the random numbers will be sorted then two clients return the result of the task to the server. The server receives the sorted numbers and displays it. Furthermore, the GUI shows the measured sorting time.

KEYWORDS: Distributed System, Sorting Algorithm, Time Complexity, Random Numbers, GUI, Socket Communication

1. INTRODUCTION

Nowadays, sorting algorithms are regarded as one of the most important areas in computer science. Sorting is normally considered to be the procedure of repositioning a known set of objects in ascending or descending order according to specified key values belong to these objects. Different sorting algorithms use different techniques to sort a set of objects. The sorting algorithm is a fundamental application in computer science and mathematics. Thus, this thesis uses sorting algorithms to test the performance of distributed systems and a single computer. An efficient sorting algorithm not only saves time but also consumes less energy. In order to choose the best sorting algorithm for application, several factors like size, data type and distribution of the elements in a data set need be considered. Generally, the performance of algorithms is measured by the standard Big $O(n)$ notation which is used to describe the complexity of the algorithm.

Over the years, distributed systems become more and more popular which has been implemented widely in the world. Furthermore, a computer has been used to process a complex problem not only email, reading news, shopping or simply accessing information. However, it has to be noticed that the single CPU still has significant limitations of computing ability. The research found that a collection of the microprocessor could produce better performance than a single CPU with a lower cost, which is the original intention of distributed systems. That means the distributed systems that consists a collection of the average computer could have better performance than a single supercomputer. With distributed systems, many difficult and complex tasks can be solved in short time. This is one of the benefits of distributed system. In addition, there is no denying that some application itself is distributed. A case in point is a bank may have many branches, distributed systems are necessary to create for management. Most queries and updates are performed in the local branch. Each branch can exchange information and share the data for cooperation and management. What’s more, there are huge hardware resources wasted in the world, if these resources can be utilized, there will be lower cost and shorter time to solve a complex task. Therefore, it is reasonable to believe that the distributed system will be more and more important in the future, and it is worth to studying it.

On the other hand, currently, electronic devices with embedded systems are widely utilized in a variety of technical fields. Raspberry Pi is a kind of embedded systems. In this thesis, raspberry pi used to act as the client in distributed system. Different from the personal computer, the program executed on embedded system normally has been coded before the equipment is being used.

This thesis mainly focuses on the comparison of three sorting algorithms (bubble sort, quick sort, and heap sort) running on one and two client(s). The sorting algorithms have been evaluated based on the sorting time of the different amount of random numbers.

This thesis consists of five chapters. In the first two chapters, sockets and socket communication, why to choose JAVA and functions in JAVA, the definition of distributed systems, the properties of Raspberry Pi has been introduced. In the third chapter, the sorting algorithms and the software implementation are described. The fourth chapter presents the results of experiments. The fifth chapter mentioned what have introduced in and contained the conclusion and the future work.

2. BACKGROUND INFORMATION

2.1. Socket and Socket Communication

As figure 1 shows that socket is an interface between an application process layer and transport layer which used to describe IP address and port number. IP address identifies a unique computer within the network and port number specifies which application the message will send to. Most of the application will take the initiative to bind a port when it starts otherwise the operating system will assign a port to it automatically. Thus with socket system can build a link between client and server.

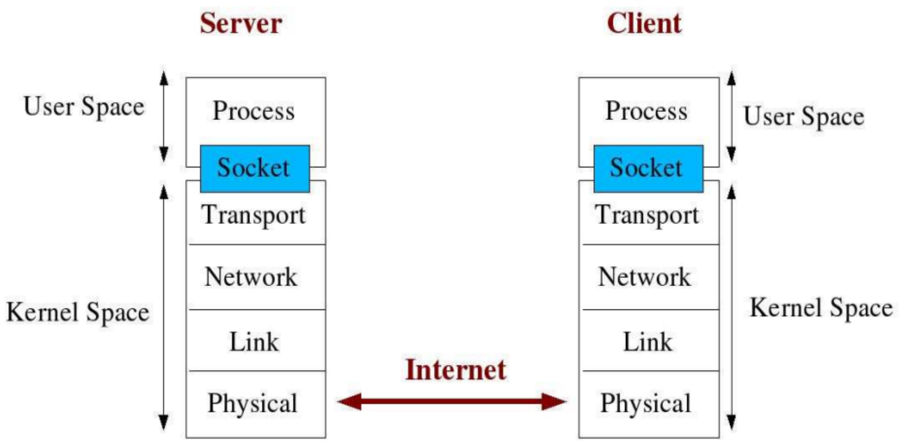


Figure 1. Socket Description (Chebrolu 2015).

There are two types of sockets, stream socket and datagram socket. Stream socket used for connection-oriented TCP service applications, if server output two items into a socket in order, the client will receive them in the same order, the stream socket is secure but inefficient. The second kind is datagram socket, used for connectionless UDP service. It needs the receiver to analysis the integrity, rearrangement or request retransmission but high efficiency. In this thesis, stream socket has been applied in the application of experiment.

A socket enables a Client-Server communication. The socket in a server is passive socket because the server is passively waiting for the request of client. The socket in a client is an active socket because client initiates the communication but the client must know the address and the port of the server.

2.1.1 TCP/IP

TCP/IP represented as Transmission Control Protocol/Internet Protocol. TCP (Transmission Control Protocol) is a connection-oriented transport protocols. TCP can provide reliable transmission by using sequence number and acknowledgement messages. If the data has been lost in transit from source to destination. TCP can retransmit the data until successful delivery has been achieved. Internet protocol is the principal communication protocol in the Internet. IP defines the structure of packet and addressing method. Thus, the data can be encapsulated as packet and then the packet can be delivered to the destination based on the IP address.

TCP/IP is normally considered to be a 4-lager structure: Link layer, network layer, transport layer, and application layer. Therefore, TCP/IP is a protocol suite, which is the combination of different protocols at various layers. TCP/IP defines how electronic devices connect to the Internet and the transmission standard between each device.

2.1.2 Running Process (Application mode) of Socket Communication

As the figure 2 shows, the server initiates a new socket of a certain socket type with socket () method for communication. Then, bind the IP address and port number with bind () method, begin listening with listen () method. The client connects to server’s listening socket. Once connection establishes successfully, the server will create a new socket immediately responsible for communication. Server and client exchange information by read () method and write () method until the client closes the connection.

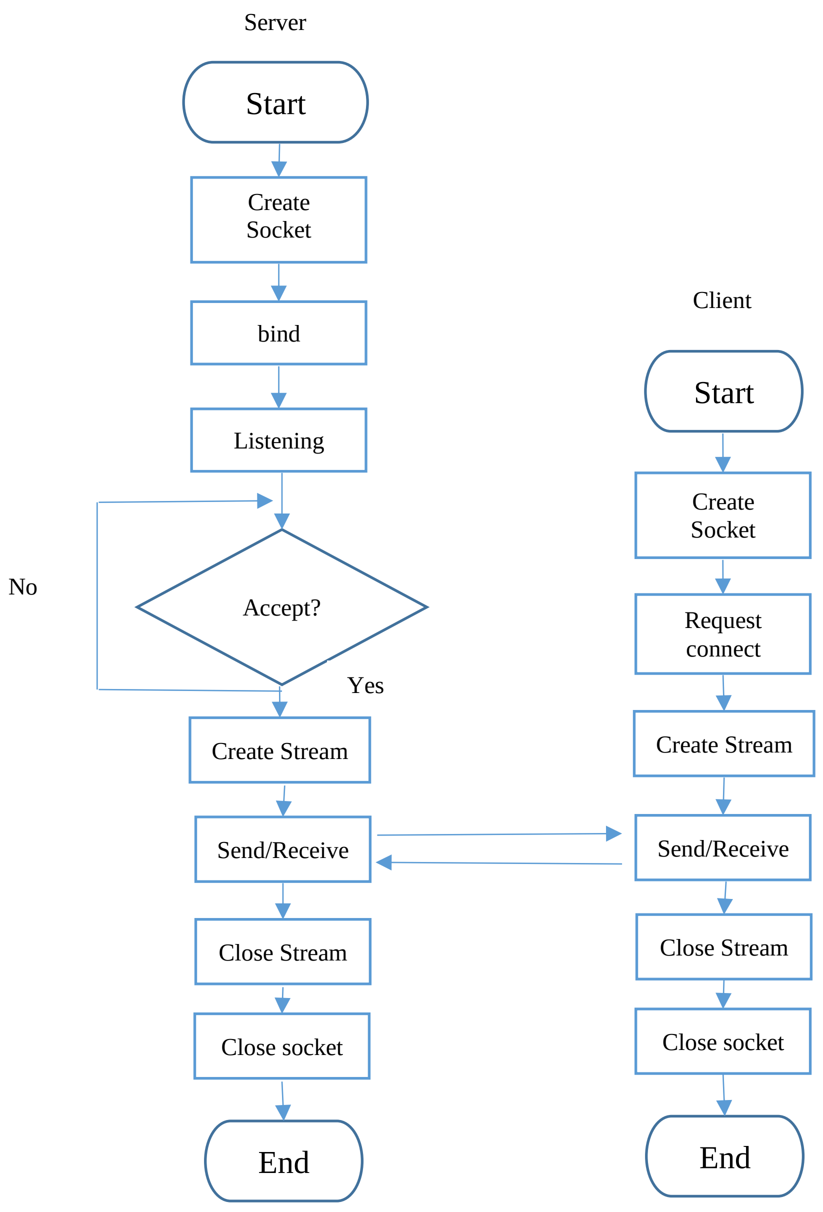


Figure 2. Flowchart of sockets communication.

2.1.3 Sockets Programming in JAVA

JAVA is one of the most popular programming languages in the world. In this thesis, all programs are coded in JAVA. There are some reasons why choosing JAVA. Firstly, there are many excellent IDE, a vast array of 3rd party libraries and the huge amount of documentation available, which provide great convenient to develop a project with JAVA. Secondly, JAVA is a cross-platform language so that JAVA programs can be executed on PC and Raspberry Pi properly. Thirdly, the features of JAVA such as sturdy garbage collection, memory management, and native threads are suitable for the project in this thesis.

In JAVA, the class is different in server and client. In the server, the socket that has been applied in the class named “ServerSocket”. In the client, the socket has been applied for the socket communication.

In the server side, firstly, create ServerSocket object, bind listening port and keep listening to the request from the client by accept () method. Secondly, create an input stream that used to send the message to the client and create an output stream that used to receive the message from the client. After connection established, read the message send from the client by input stream. Send respond message by output stream. Then close the socket after communication is terminated. (Chebrolu, 2006)

In the client side, create socket object, get aim IP address and port number to request connect. After connection established, send the message by output stream and receive the message by input stream. Close socket after communicating is terminated.

ServerSocket (int port), create socket bind to specify port.

Create basic Sever class:

```
package com.zhuang;



public class Server{

    public static void main(String[] args){

     ServerSocket serverSocket = new ServerSocket(8888);

     While(ture)
```

```
   {
       Socket Socket = Server.accept();

       DataOutputStream dos = new DataOutputStream(Socket.getOutputStream());

       DataInputStream dis = new DataInputStream(Socket.getInputStream());

       dos.writeUTF(Message);
       dos.flush();

       String listFin = null;
       listFin = dis.readUTF()

       dos.close();
       dis.close();
       Socket();

   }
  }

}
```

The code above is a basic part of the class on a server. It needs more programming to achieve additional functions.

2.2. Distributed Systems

The technical development of the systems creates more and more difficult computing problems that cannot be solved by one single processor in a short time. On the other hand, the developments in computer science have resulted in the availability of fast and inexpensive processors. Many processors that connected to the network are inactive. If these processors work together in a system over a network, the system may have the great computing power to deal complex tasks. A collection of microprocessors offer a better performance than mainframes, this is the idea of distributed computing announced in the 1970s.

A distributed system is based on the concept of distributed computing, which is an application that executes a collection of protocols to coordinate the actions of multiple processes on a communication network so that all components in the system cooperate to perform a single or small set of related tasks. In this thesis, a simple distributed system model consist of one PC and two Raspberry Pi Boards. The task is an implementation of the sorting algorithms. (M. Thampi, 2009)

With the continuing advances in communication technology, distributed systems will be an important field in computer science. It is a very valuable subject.

2.3. Raspberry Pi

Raspberry Pi board is a complex integrated circuit that integrates the major functional elements include a programmable processor, on-chip memory, accelerating function hardware (e.g. GPU). Both hardware and software are analogy components. As figure 3 shows the Raspberry Pi Model B that used in this thesis. In experimental part, the Raspberry Pi board is used to act as the client in the distributed system.

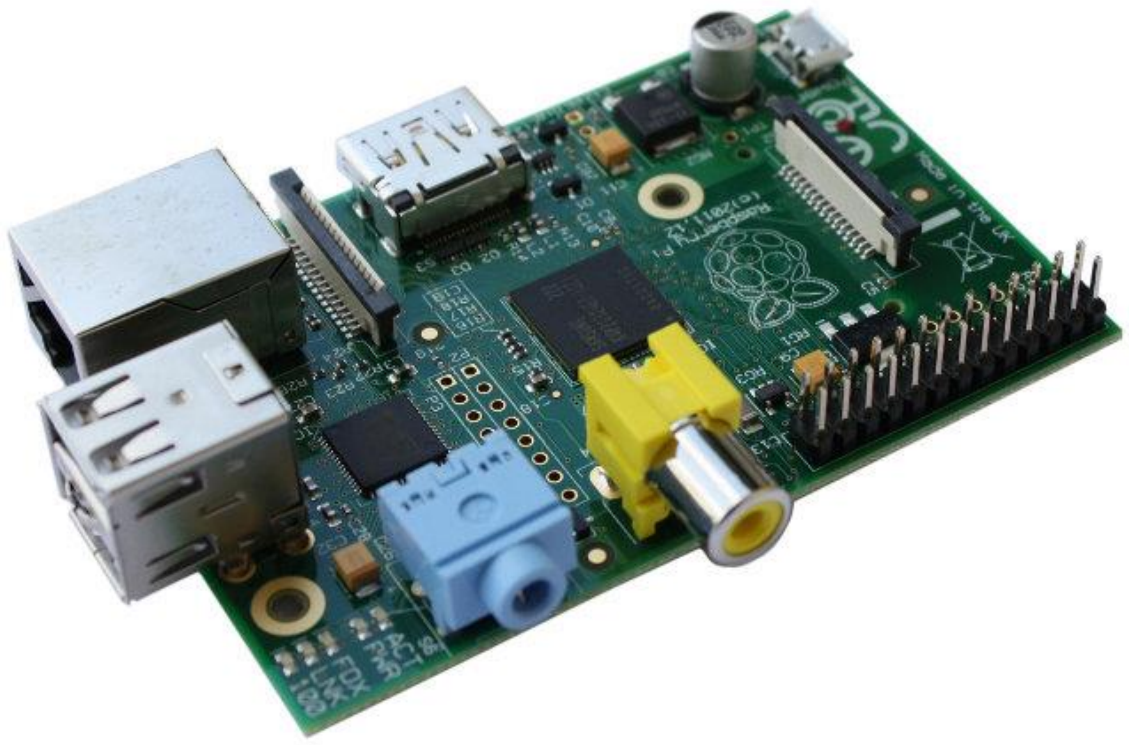
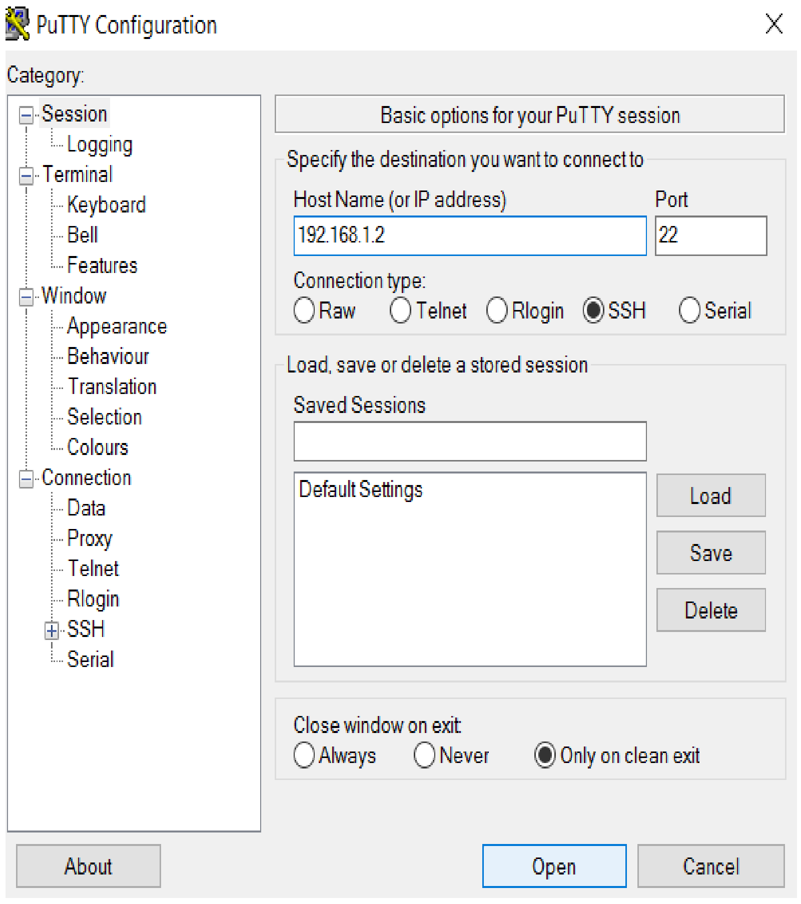


Figure 3. Raspberry Pi Model B (Burkepile 2013).

In order to access Raspberry Pi board, it needs to connect Raspberry Pi to PC via cable physically. Then download an SSH software in order to access Raspberry Pi. In this thesis, an SSH software **putty.exe** is used to connect Raspberry Pi to the computer. Raspberry Pi will start automatic when plugging in the power cable and SD card of it. Finally, open **putty.exe** on PC, type in IP address and click “Open” button, as the figure 4 shows. (Seighman, 2012)



Specify the destination you want to connect to

Host Name (or IP address): 192.168.1.2

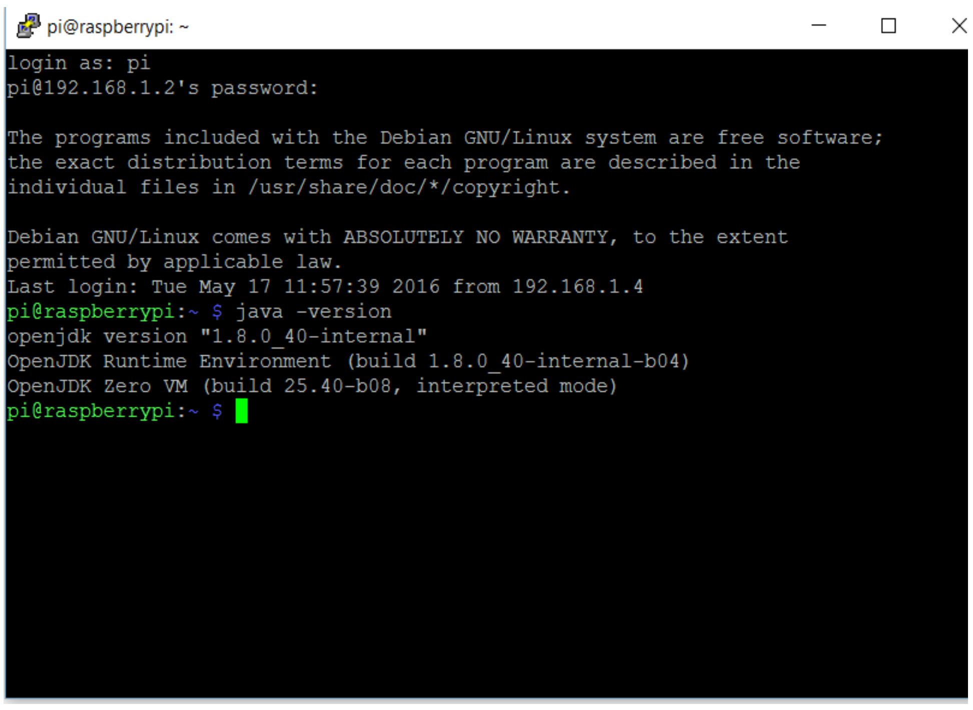
Port: 22

Connection type: Raw Telnet Rlogin [✓ SSH] Serial

Close window on exit: Always Never [✓ Only on clean exit]

Figure 4. Interface of Putty.exe.

Figure 5 shows how to log in and check JRE of Raspberry Pi.

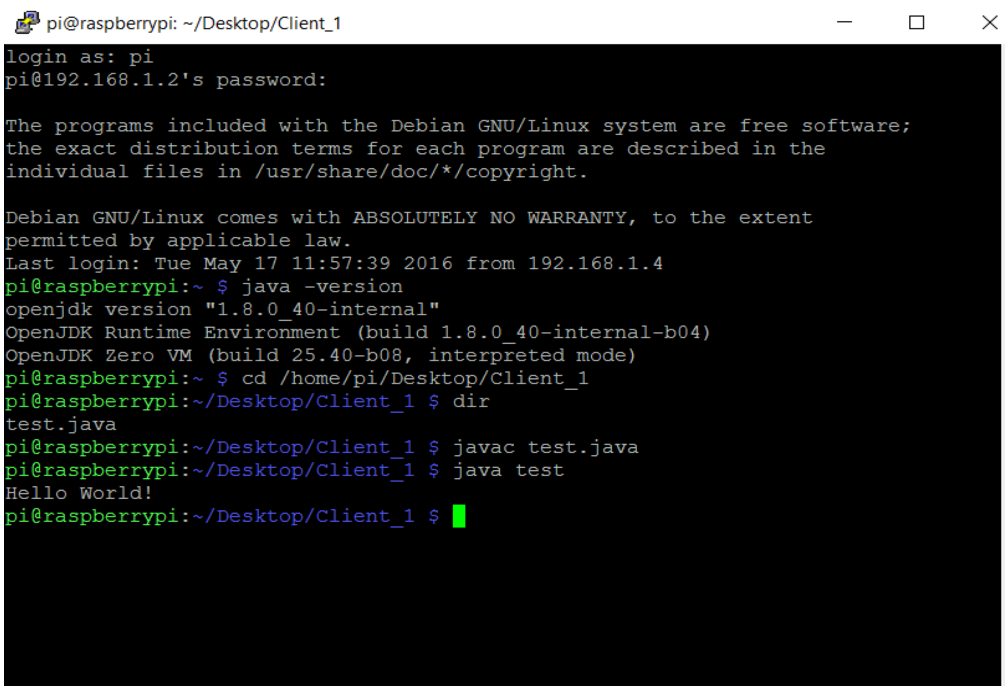


pi@raspberrypi: ~

```
login as: pi
pi@192.168.1.2's password:

The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
Last login: Tue May 17 11:57:39 2016 from 192.168.1.4
pi@raspberrypi:~ $ java -version
openjdk version "1.8.0_40-internal"
OpenJDK Runtime Environment (build 1.8.0_40-internal-b04)
OpenJDK Zero VM (build 25.40-b08, interpreted mode)
pi@raspberrypi:~ $
```

Figure 5. Log in and Execute Command in Putty.exe.

Figure 6 shows the terminal of the Raspberry Pi.



pi@raspberrypi: ~/Desktop/Client_1

```
login as: pi
pi@192.168.1.2's password:

The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
Last login: Tue May 17 11:57:39 2016 from 192.168.1.4
pi@raspberrypi:~ $ java -version
openjdk version "1.8.0_40-internal"
OpenJDK Runtime Environment (build 1.8.0_40-internal-b04)
OpenJDK Zero VM (build 25.40-b08, interpreted mode)
pi@raspberrypi:~ $ cd /home/pi/Desktop/Client_1
pi@raspberrypi:~/Desktop/Client_1 $ dir
test.java
pi@raspberrypi:~/Desktop/Client_1 $ javac test.java
pi@raspberrypi:~/Desktop/Client_1 $ java test
Hello World!
pi@raspberrypi:~/Desktop/Client_1 $
```

Figure 6. Execute a Small Program on Raspberry Pi.

2.4. The Function of Random Number Generation and Time Measurement

The generation of random numbers is important for filling the array that need to be sorted. With random number, there are two methods of random number generation: true randomness and pseudo-randomness. Since this thesis only needs the unpredictability, pseudo-randomness is suitable for the experiment. The random number generation function generates the required number of elements from 10 to 98 randomly. The function code is given:

```
public static String listGenertor(int size)
{
        String list = "";
        int[] intList = new int[size];
        for(int i=0; i<size; i++)
        {
                intList[i] = (int)(Math.random()*89 + 10);
                list = list + intList[i] + " ";
        }

        return list;
}
```

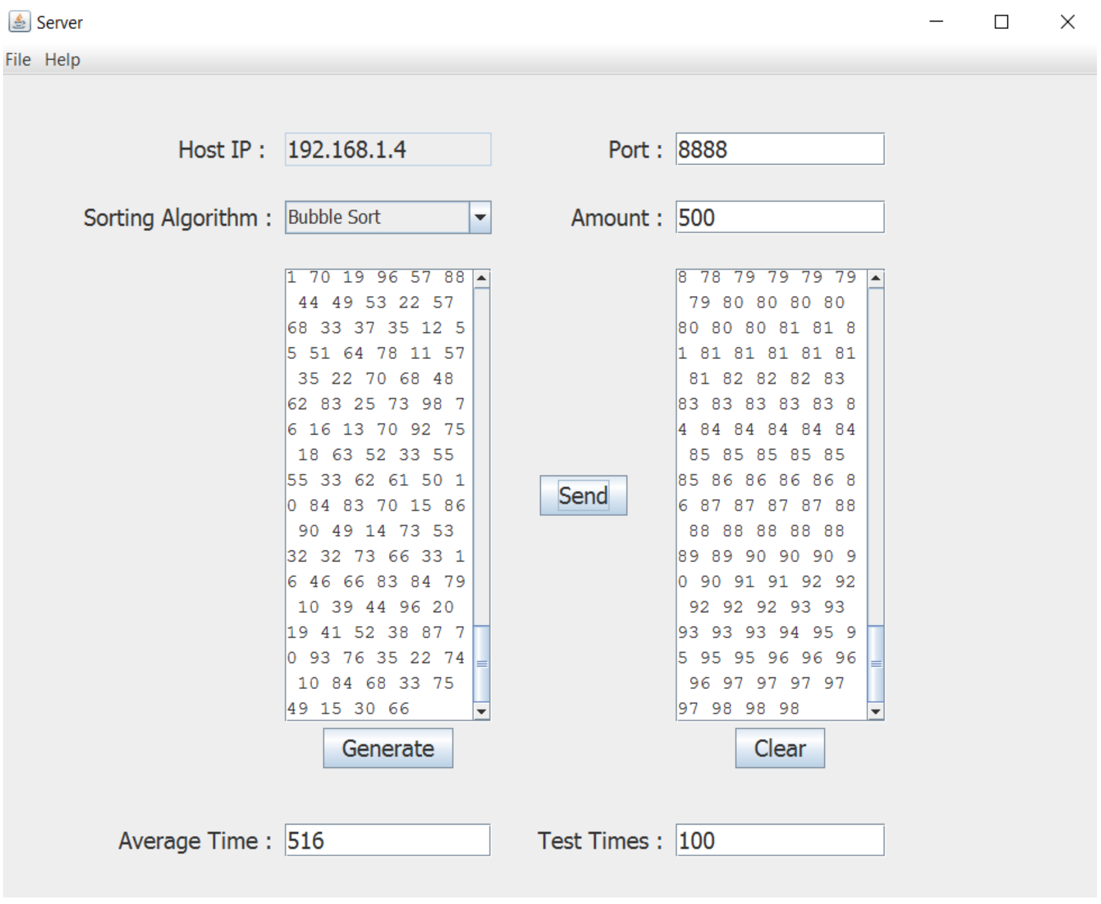
The time measurement in the experiment applies the method in JAVA default library called **java.lang.System.currentTimeMillis()**. This method returns the current time in milliseconds. The following example function shows the usage of this method.

```
public static String timeMeasurement(long time)
{

        Long startTime = System.currentTimeMillis();

        {
             // Operation need be time measurment
        }

        Long endTime = System.currentTimeMillis();
```

```
                Long time = endTime – startTime ;

                return time;
        }
```

2.5. GUI Server

In the experiment, a GUI of the server is developed to set port and the required number of elements. With GUI, the original array and sorted array can be shown on it as well as the sorting time. The GUI is developed by JAVA language with eclipse, which is one of the most popular IDE (Integrated Development Environment) in the world. Figure 7 shows the GUI of the server.



Host IP : 192.168.1.4
Port : 8888
Sorting Algorithm : Bubble Sort
Amount : 500

```
1 70 19 96 57 88
 44 49 53 22 57
68 33 37 35 12 5
5 51 64 78 11 57
 35 22 70 68 48
62 83 25 73 98 7
6 16 13 70 92 75
 18 63 52 33 55
55 33 62 61 50 1
0 84 83 70 15 86
 90 49 14 73 53
32 32 73 66 33 1
6 46 66 83 84 79
 10 39 44 96 20
19 41 52 38 87 7
0 93 76 35 22 74
 10 84 68 33 75
49 15 30 66
```

```
8 78 79 79 79 79
 79 80 80 80 80
80 80 80 81 81 8
1 81 81 81 81 81
 81 82 82 82 83
83 83 83 83 83 8
4 84 84 84 84 84
 85 85 85 85 85
85 86 86 86 86 8
6 87 87 87 87 88
 88 88 88 88 88
89 89 90 90 90 9
0 90 91 91 92 92
 92 92 92 93 93
93 93 93 94 95 9
5 95 95 96 96 96
 96 97 97 97 97
97 98 98 98
```

Average Time : 516
Test Times : 100

Figure 7. GUI of the Server.

3. ALGORITHMS AND SOFTWARE IMPLEMENTATION

In computer science, sorting algorithm is an algorithm that puts the elements of an array in a certain order. The order includes numerical and lexicographical order. The Internet generates inestimable data every day, the most basic and common operation to data is sorting. Therefore, sorting algorithm is very important in computer science and researcher usually use the sorting algorithm to evaluate the performance of a system.

In this thesis, a distributed system is developed to test different sorting algorithms. Each of the sorting algorithms (bubble sort, quick sort, and heap sort) is tested on a distributed system and on a single PC.

So what determines the performance of an algorithm? Typically, the faster program has the fewer operations. The operations include data movement or swaps and comparisons.

1. Time-frequency is the time spent for an algorithm. The number of statement executions in an algorithm is called statement frequency or time-frequency, denoted by *T (n)*. The “n” is the size of the array. When n is changing *T (n)* is also changing. If exists an auxiliary function called f (n) that makes the limit of *T (n)/ f(n)* is equal to non-zero constant when n tends to infinity. Call f (n) is the same order magnitude of *T (n)*. Denoted by *T (n) = O (f (n))*, which is the time complexity. In the algorithm, if the number of statement execution is a constant, then is the time complexity is O (1). There are some common time complexities: O (n), O (n^2), O (n$\log_2 n$) and O (2^n).

2. Space complexity is a measure for an algorithm that needs temporary storage when running.

3. An algorithm can be declared as stable if the relative order of the elements is preserved after sorting.

3.1. Bubble Sort

The bubble sorting algorithm is the one of the most famous sorting algorithm in computer science. The idea of bubble sort is scan elements repeatedly. After the first round, show up the biggest element at the end of the array. (Dalal, 2004)

Bubble sort works in following way:

1. Compare adjacent elements from the first element, if the first one is greater than the second one, swap it (ascending order). The largest element will be the last element after this round.
2. Repeat step 1 for all elements except last one.
3. Repeat step 2 until there are no elements needed to swap.

Figure 8 shows procedures of how bubble sort work in steps.

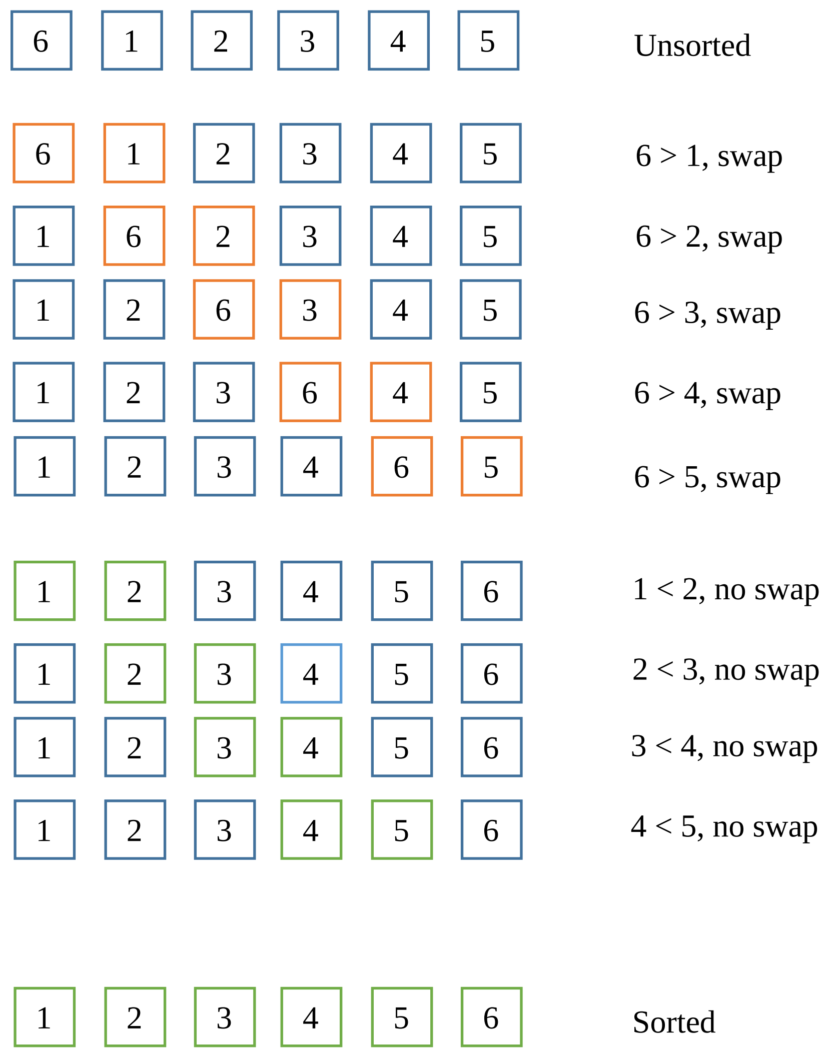


Figure 8. Flow chart of bubble sort.

The code of bubble sorting algorithm used in the experiment is shown below:

```
public static String bubbleSort(int List[])
        {
```

```
        int size = List.length;
        String result_1 = "";
        for(int i = 0; i < size; i++){

            for(int j = 0; j < size - 1; j++){
                int temp;
                if(List[i]>List[j+1]){
                    temp = List[j];
                    List[j] = List[i];
                    List[i] = temp;
                }
            }

        }
        for(int n=0;n<List.length;n++){
            result_1 = result_1 + List[n] + " ";
        }
        return result_1;

    }
```

For the bubble sort, the worst case is if all elements in the array are in descending (ascending) order and if the array must be sorted in ascending (descending) order.

Table 1. Bubble sort in worst case.

Line	Execution Time	Number of Times
From m to length of array	C1	n
From n to length of array	C2	n-1
Comparison	C3	n(n-1)
Exchange	C4	n(n-1)/2

According to table 1, the time consumption is:

$T(n) = C1*n + C2*(n-1) + C3*n*(n-1) + C4*n*(n-1)/2$

$= n^2 (C3 + C4/2) + n*(C1+C2-C3-C4/2)-C2-C3$

$\leq X*n^2$ (when $n \geq 1$, $X = (C3 + C4/2) + (C1+C2-C3-C4/2)-C2-C3)$)

The result is $T(n) = O(n^2)$ in the worst case.

In the best case, the array is already in desired order. In this case, a comparison statement is needed to check if the array is in the desired order or not.

```
public static String bubbleSort(int List[])
    {
            int size = List.length;
            String result_1 = "";
            for(int i = 0; i < size; i++){

                for(int j = 0; j < size - 1; j++)
                {
                    int temp;
                    int swap = 0;
                    if(List[i]>List[j+1]){
                        temp = List[j];
                        List[j] = List[i];
                        List[i] = temp;
                        swap++;
                    }
                }
                if(swap == 0)
                 {
                      break;
                 }


            }
            for(int n=0;n<List.length;n++){
                result_1 = result_1 + List[n] + " ";
            }
            return result_1;

    }
```

According to the program, assume:

Table 2. Bubble sort in the best case.

Line	Execution Time	Number of Times
From m to length of array	C1	n
From n to length of array	C2	n-1
Comparison	C3	n-1
Exchange	C4	0
Check	C5	1

According to the table 2, the time consumption is:

$$T(n) = C1*n + C2*(n-1) + C3*(n-1) + C4*0 + C5$$

$$= n*(C1+C2+C3) - C2-C3+C5$$

$$\leq X*n \text{ (when } n\geq 1,\ X = (C1+C2+C3)-C2-C3+C5)$$

The result is T(n) = O(n) in the best case.

The relative order of each elements is preserved so bubble sort is a stable sorting algorithm.

3.2. Quick Sort

Quick sort is one of efficient sorting algorithms. The idea of the algorithm is first to choose an element from the array as a pivot. Sort the array by comparing every element to the pivot, so all elements smaller than or equal to the pivot come before the pivot. In contrast, all elements that are greater than pivot come after the pivot. Of course, it can be defined that the elements equal to pivot come before or after the pivot. Recursively apply previous steps to the sub-array of elements before the pivot and after the pivot until getting a sorted array. Figure 10 shows how quick sort works. (Iliopoulos, 2013)

Quick sort works in following way:

1. Choose an element in array as a pivot.
2. Create two pointer point to the first element and the last element in array. Move pointer from the first element to pivot, until an element greater than pivot (Ascending order) is found. Move another pointer from the last elements to the pivot until an element smaller than pivot is found and then exchange these two elements.
3. Repeat step 2 until all elements greater than pivot are placed after pivot and all elements smaller than pivot are placed before pivot.
4. Repeat step 1, 2 and 3 to the sub-arrays until all elements sorted.

Figure 9 shows an example of quick sort.

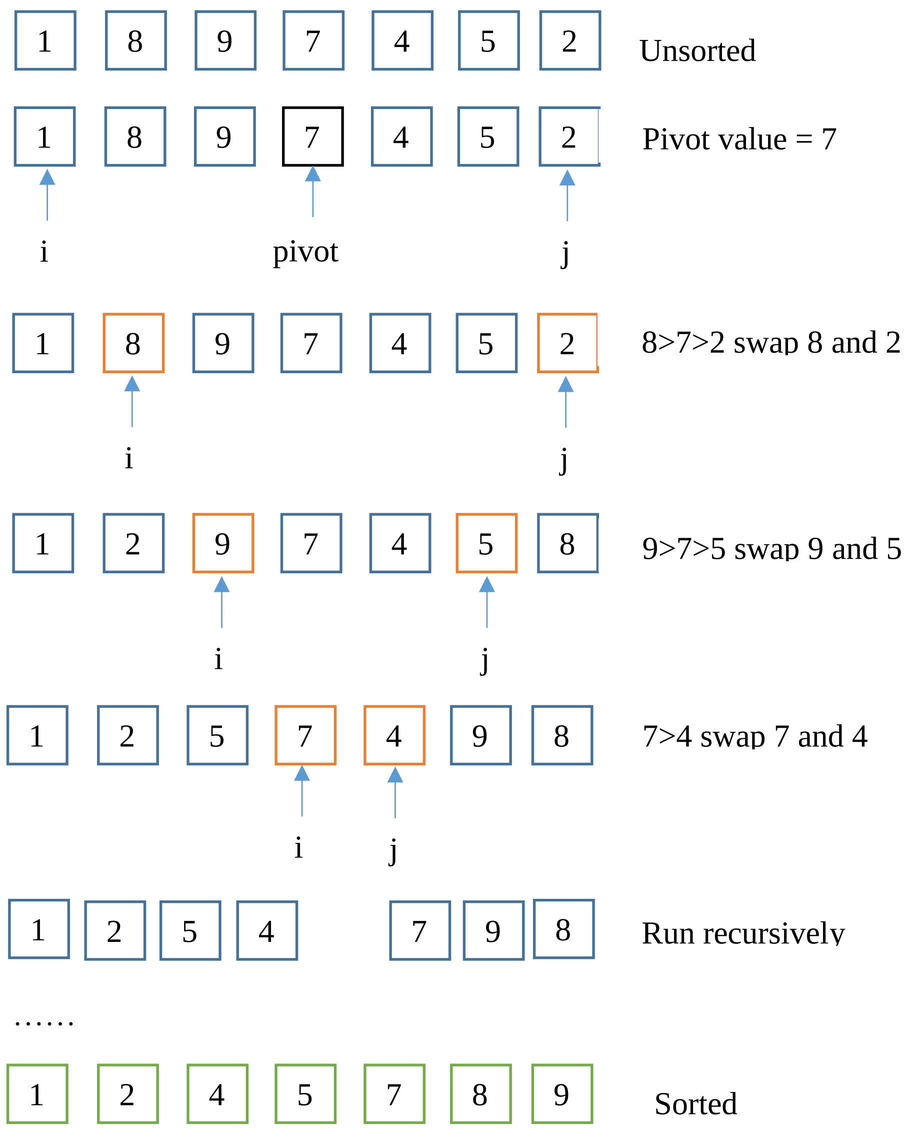


Figure 9. Chart of quick sort.

The code of quick sorting algorithm used in experiment shown below:

```
public class QuickSort{
        public String quickSort(int List[])
        {
                String quickResult = "";
                if(List.length > 0){
                        quickSort(List,0,List.length - 1);
                }

                for(int n=0;n<List.length;n++){
                        quickResult = quickResult + List[n] + " ";
                }

                return quickResult;

        }

        private static void quickSort(int[] List, int low, int high){

                if(low < high){

                        int middle = getMiddle(List, low, high);
                        quickSort(List, low, middle-1);
                        quickSort(List, middle+1, high);
                }

        }

        private static int getMiddle(int[] List, int low, int high){

                int temp = List[low]; // Choose first element as pivot
                while(low<high){

                        // Find a element smaller than pivot then exchange
                        while(low < high && List[high] >= temp){

                                high--;
                        }
                        List[low] = List[high];

                        while(low < high && List[low] <= temp){

                                low++;
                        }
                        List[high] = List[low];

                }
                List[low] = temp;

                return low;
        }
```

```
}
```

According to the program, the time consumption of quick sort includes three part, let T(N) be the time cost to quicksort n elements:

1. Time to sort left partition = T(L).
2. Time to sort right partition = T(R).
3. Time for partitioning at current recursive step = O(n)

In the worst case, the pivot always is the smallest element or greatest element.

T(n) = T(0) + T(n-1) + O(n)

= O(1) + T(n-1) + O(n)

= T(n-1) + O(n)

= T(n-2) + O(n-1) + O(n)

= T(n-3) + O(n-2) + O(n-1) + O(n)

$=\sum_{i=1}^{n} O(\text{n}) = \text{O}\ (n^2)$

In the best situation, the pivot is the middle value of the array:

T(n) = T(n/2) + T(n/2) + O(n)

= 2T(n/2) +O(n)

Let's assume the size of array is **n**, **k** = $\log_2 n$.

First recursion needs **n** loops; Second recursion, needs 2*(**n**/2) loops ….

So $\mathbf{n} + 2*(\mathbf{n}/2) + 4*(\mathbf{n}/4) + \ldots.. + \mathbf{n}*(\mathbf{n}/2) = k*n = n\log_2 n$

Result is $T(n) = O(n\log_2 n)$

Average time complexity of quick sort is close to the best situation so the time complexity is O $(n\log_2 n)$. Moreover, the relative order of equal sort items is not preserved so quick sort is not a stable sort.

3.3. Heap Sort

3.3.1 Tree and Max Heap

Before introducing heap sort, it necessary to understand what is the tree (Binary Tree) and heap. The tree is a collection of nodes. Each node can have a parent node and sub-node. The node that has not a parent node named the root node. A node without children named a leave node. Figure 10 is a model of the tree. A heap is a special binary tree called a complete binary tree.

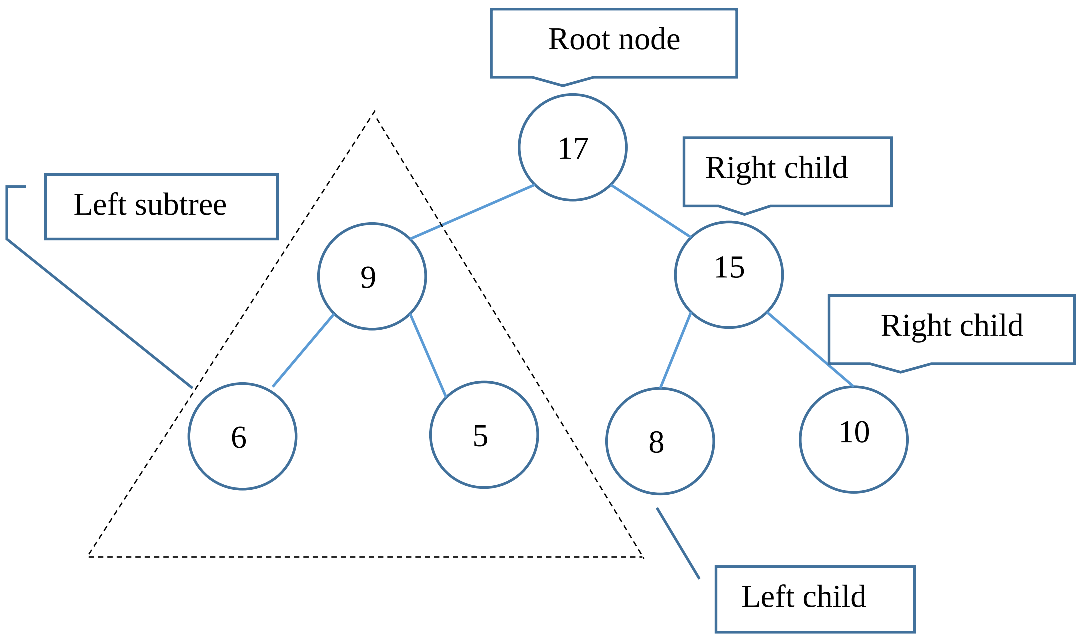


Figure 10. Binary Tree.

Heap always meet some properties: max heap, each node's value is equal to or smaller than the value of their parent. Min heap means each node's value is not less than the value of their parent. Heap sort adapted to sort huge amount of data because heap sort does not require much recursion, this is an advantage of this sorting algorithm. Heap sort needs to apply a kind of data structure called max heap (etc. complete binary tree). In this tree, each layer is filled except for the last layer (see figure 11). (Stassiy, 2014)

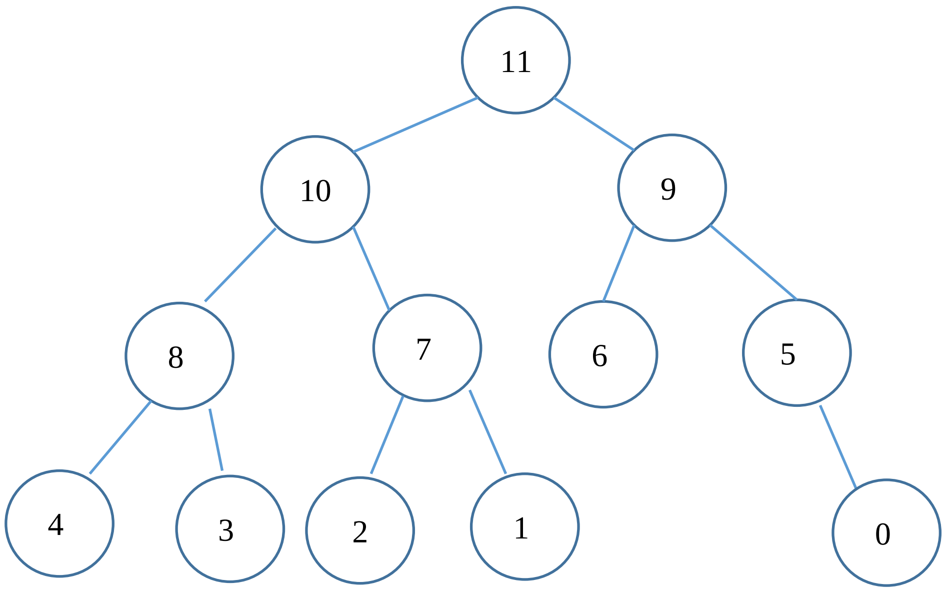


Figure 11. Complete Binary Tree (Max Heap).

Let i be the node number of one node, then we can get its parent node, left node and right node number. Parent node number = i/2, left node number = i*2 and right node number = 2*i + 1.

3.3.2 Heap Sort

Heap sort is based on max heap, max heap is any root node bigger than its child node and every child tree is max heap as well. Therefore, in max heap, the root node is the maximum one. Then exchange root node with the last node in the array. After this, rebuild a new max heap. Repeat this procession until getting a sorted array.

Heap sort works in following way:

1. Build max heap.
2. Exchange the root node and the last node in the tree. Move the last node to the last position of the array.
3. Length of array minus one. Rebuild max heap.
4. Repeat step 2 and 3 until all element sorted.

The figures 12-24 shows an example of heap sort.

16	31	62	90	14	83	11

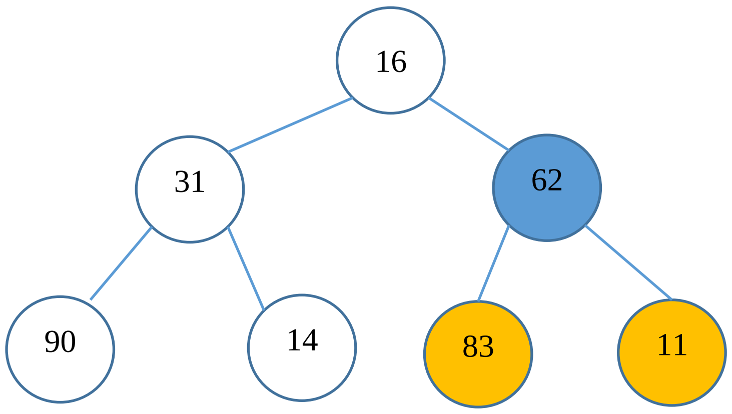


Figure 12. Original Array and Tree.

It needs to get max heap first before sorting. General, start with the rightmost node. Compare this node with its sub-node, if sub-node larger, exchange them until getting the largest node in this position.

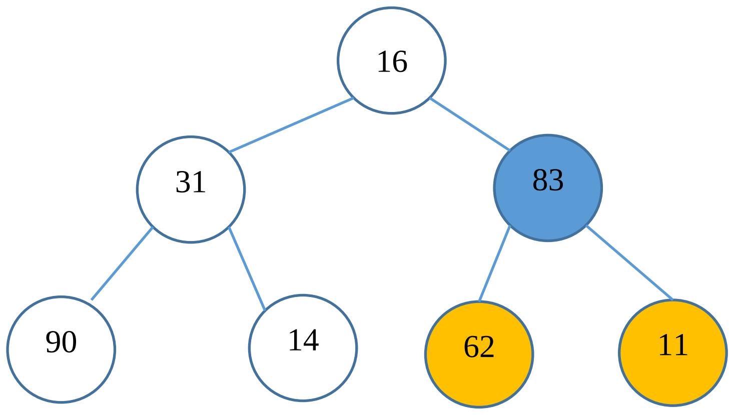


Figure 13. Exchange node [3] and node [7].

Then move to the node before the start node, repeat the operation.

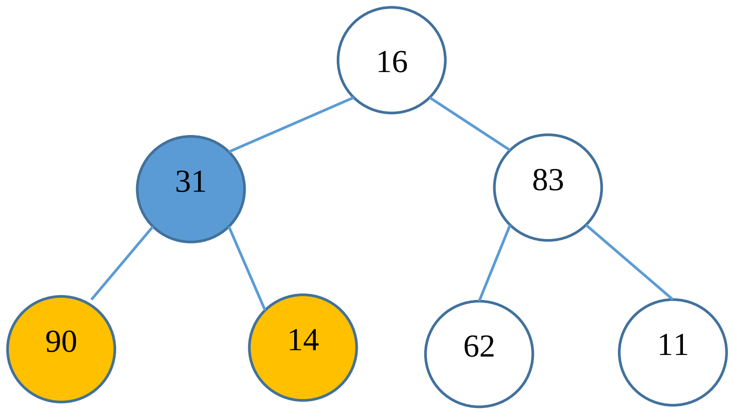


Figure 14. Exchange node [2] and node [4], then more forward node.

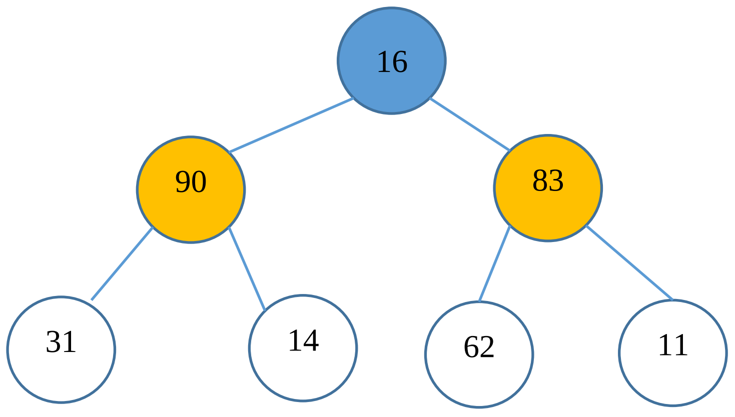


Figure 15. Last step is to compare root node with its sub-node then exchange if necessary.

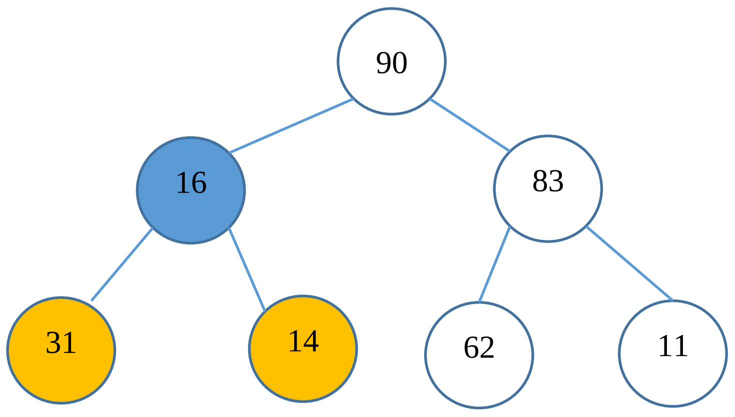


Figure 16. Get the largest node in first node, and then repeat operation in node order.

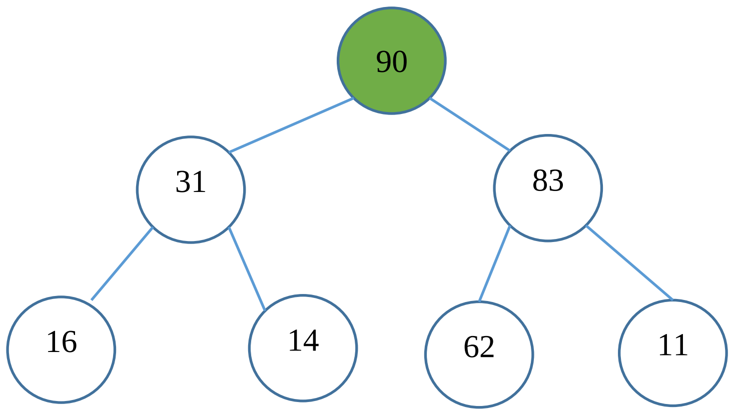


Figure 17. The Max Heap is finished.

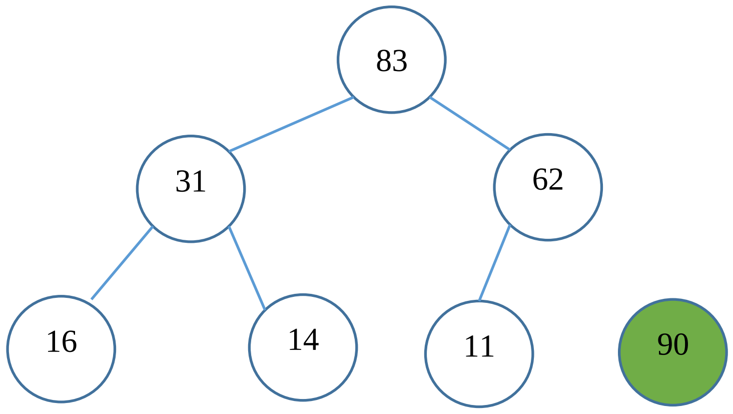


Figure 18. Exchange the first node with the last node then rebuild max heap.

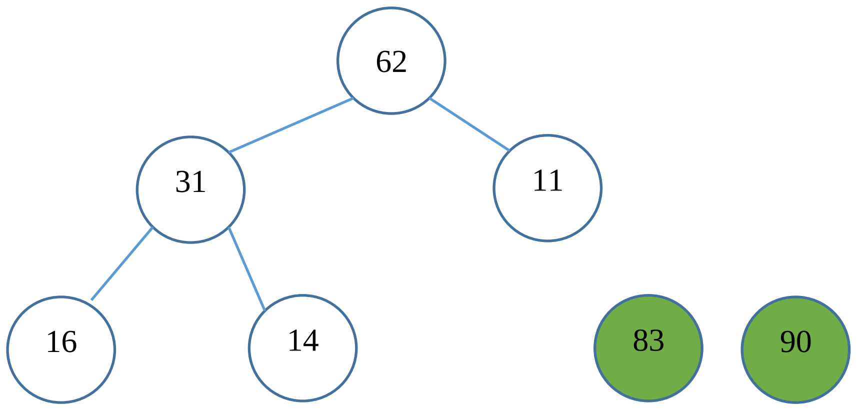


Figure 19. Exchange the first node with the last node then rebuild max heap.

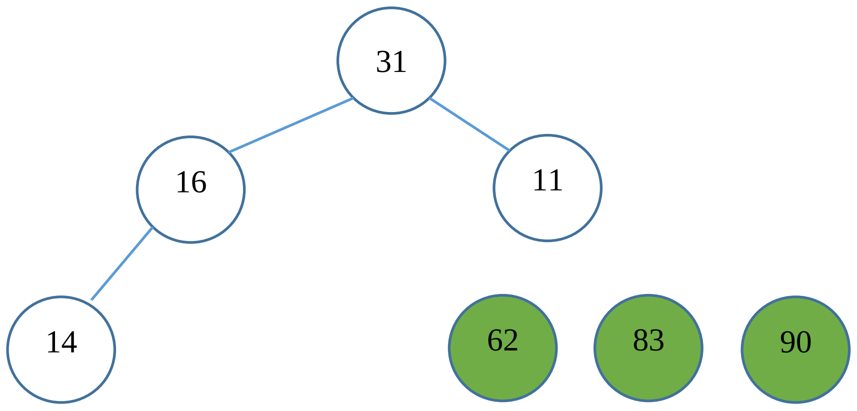


Figure 20. Exchange the first node with the last node then rebuild max heap.

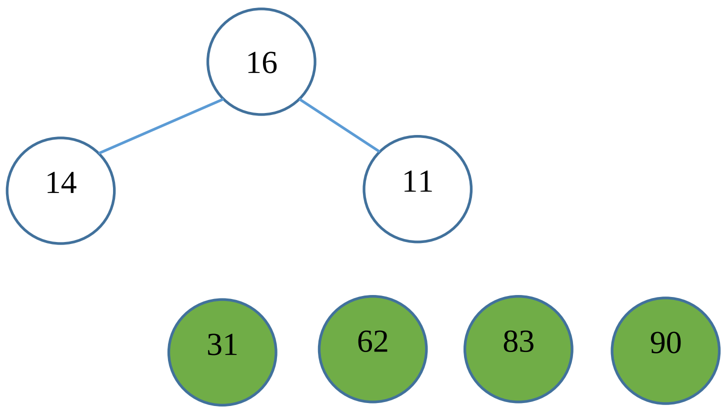


Figure 21. Exchange the first node with the last node then rebuild max heap.

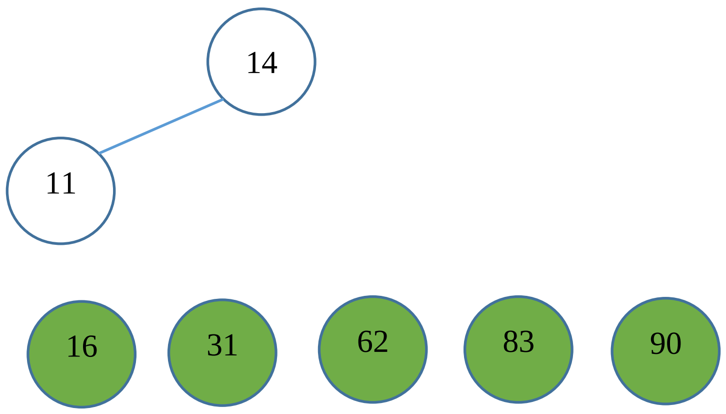


Figure 22. Exchange the first node with the last node then rebuild max heap.

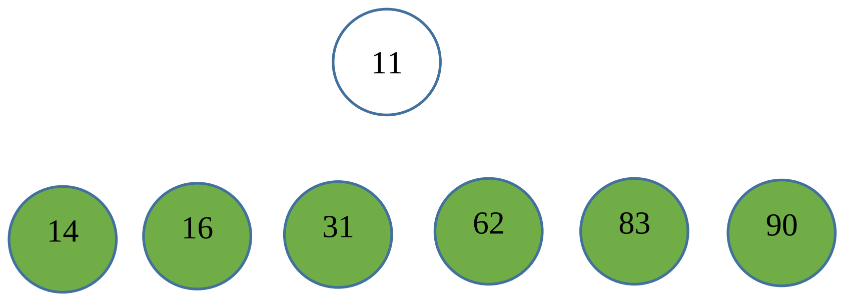


Figure 23. Exchange the first node with the last node then rebuild max heap.

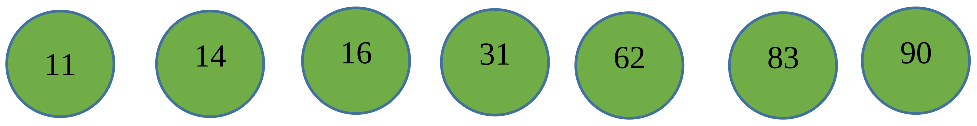


Figure 24. The sorting is completed.

The code of heap sort in Java is shown below:

```
public class HeapSort {


        static int[] List;
        static int n;

        public void HeapSort(int[] List0){

                List = List0;

                buildMaxHeap(List);

                for(int i=n; i > 0; i--){
                        exchange(0,i);
                        n = n-1;
                        maxHeap(List,0);

                }


        }

        private static void buildMaxHeap(int[] List){

                n = List.length -1;
                for(int i = n/2; i >= 0; i--){
                        maxHeap(List, i);
                }
        }

        private static void maxHeap(int[] List,int index){

                int left = index*2;
                int right = index*2 + 1;

                int largest = 0;

                if(left < n && List[left] > List[index]){
                        largest = left;
                }else{largest = index ;}

                if(right <= n && List[right] > List[largest]){
                        largest = right;
                }

                if(largest != index){
                        exchange(index,largest);
                        maxHeap(List,largest);
                }


        }

        public static void exchange(int i,int j){
                int t = List[i];
                List[i] = List[j];
                List[j] = t;
        }

}
```

According to the program, let T(n) be the time to run heap sort on an array of size n.

$$T(n) = T_{buildheap}(n) + \sum_{k=1}^{n-1} T_{heapify}(k) \qquad (1)$$

Since heapify is used in build heap as well:

$$T_{heapify}(n) = O(1) + T_{heapify}(size\ of\ subtree) \qquad (2)$$

Let h be the depth of heap, n be the number of nodes.

$$2^h \le \text{n} \le 2^{h+1} - 1 \qquad (3)$$

If heap is a complete binary tree.

$$\text{n} = 2^{h+1} - 1 \qquad (4)$$

In tree, level 1 has 2^0 = 1 node. Level 2 has 2^1 = 2 nodes. Let l be the number of level count from the bottommost level. There are 2^{h-l} nodes in l level. There are $\sum_{l=0}^{h} 2^{h-l}$ nodes except the nodes in the bottommost level.

Since there is no comparison operation in bottommost level:

$$T_{buildheap}(n) = 2 * \sum_{l=0}^{h} 2^{h-l} \qquad (5)$$

$$= 2^{h+1} * \sum_{l=0}^{h} \frac{1}{2^l} \le 2^{h+1} \qquad (6)$$

Recall equation (4), $T_{buildheap}(n) = T(n) = O(n)$ (7)

For the heapification, each time the root node must be compared level by level, thus the complexity is related to h. From the equation (4):

$$T_{heapify}(n) = O(\log_2(n-1) - 1) = O(\log_2 n) \qquad (8)$$

From equation (1), (2), (7), (8), could be find that:

$$\mathrm{T(n)} = O(n) + O\big((n-1) * \log_2 n\big) \qquad (9)$$

$$= \mathrm{O}(\mathrm{n} * \log_2 n)$$

So the time complexity of heap sort is $\mathrm{n}\log_2 n$. In heap sort, every time, after accessing the largest element in the tree, the max tree must be rebuilt. The relative order of equal sort elements is not preserved and thus the heap sort is an unstable sorting method.

3.4. Software Implementation

To evaluate the performance of sorting algorithms between One-Client-System and distributed systems (two clients) the following setups need to be done. To evaluate the performance of algorithms on One-Client-system, both server and client are running on the same PC. The connection between server and client has been established on the local host. For evaluating the performance of algorithms on distributed systems, the server, as well as two clients, are running on the same PC. The connection between server and client has been established on the local host.

To evaluate the performance of the sorting algorithms between PC and Raspberry Pi the following setups need to be done. To evaluate the performance of the algorithms on Raspberry Pi, the server runs on PC while the client runs on Raspberry Pi. As the figure 25 shows, Raspberry Pi is connected to PC through a switch via Ethernet cable.

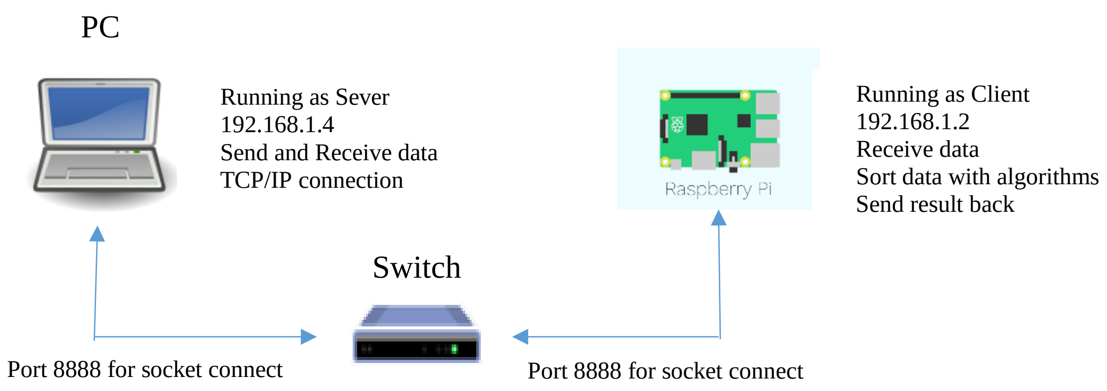


Figure 25. Connection of the devices with one client.

For distributed systems, the server runs on the PC while two clients run on the Raspberry Pi's. Figure 26 shows the connection of the PC with two Raspberry Pi's.

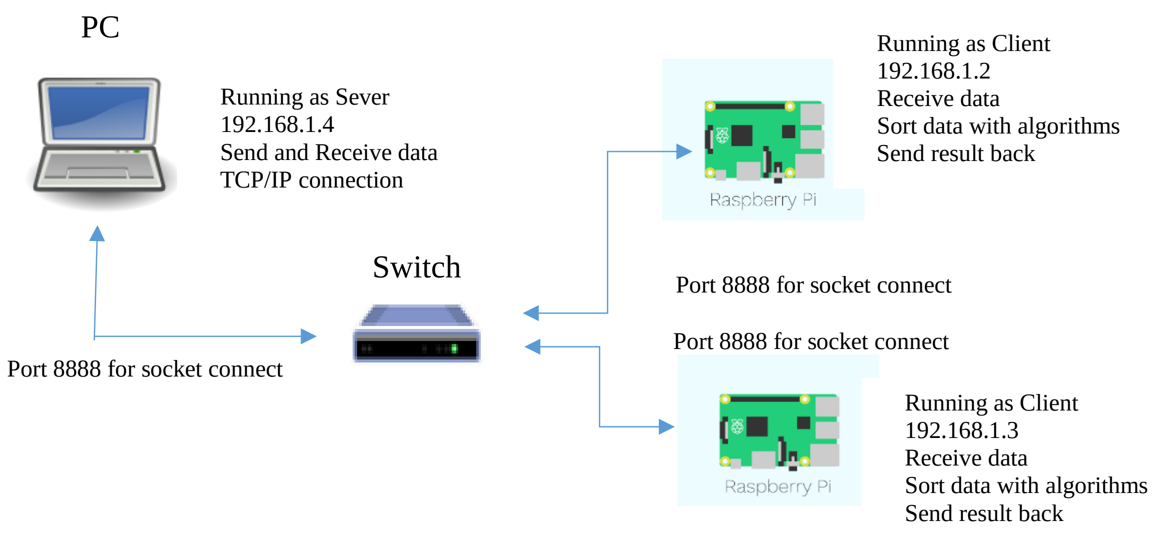


Figure 26. Connection of the devices with two clients.

The program can be executed by setup hardware. Figure 27 shows the flowchart of the program with One-Client-System. There is one server and only one client in this part.

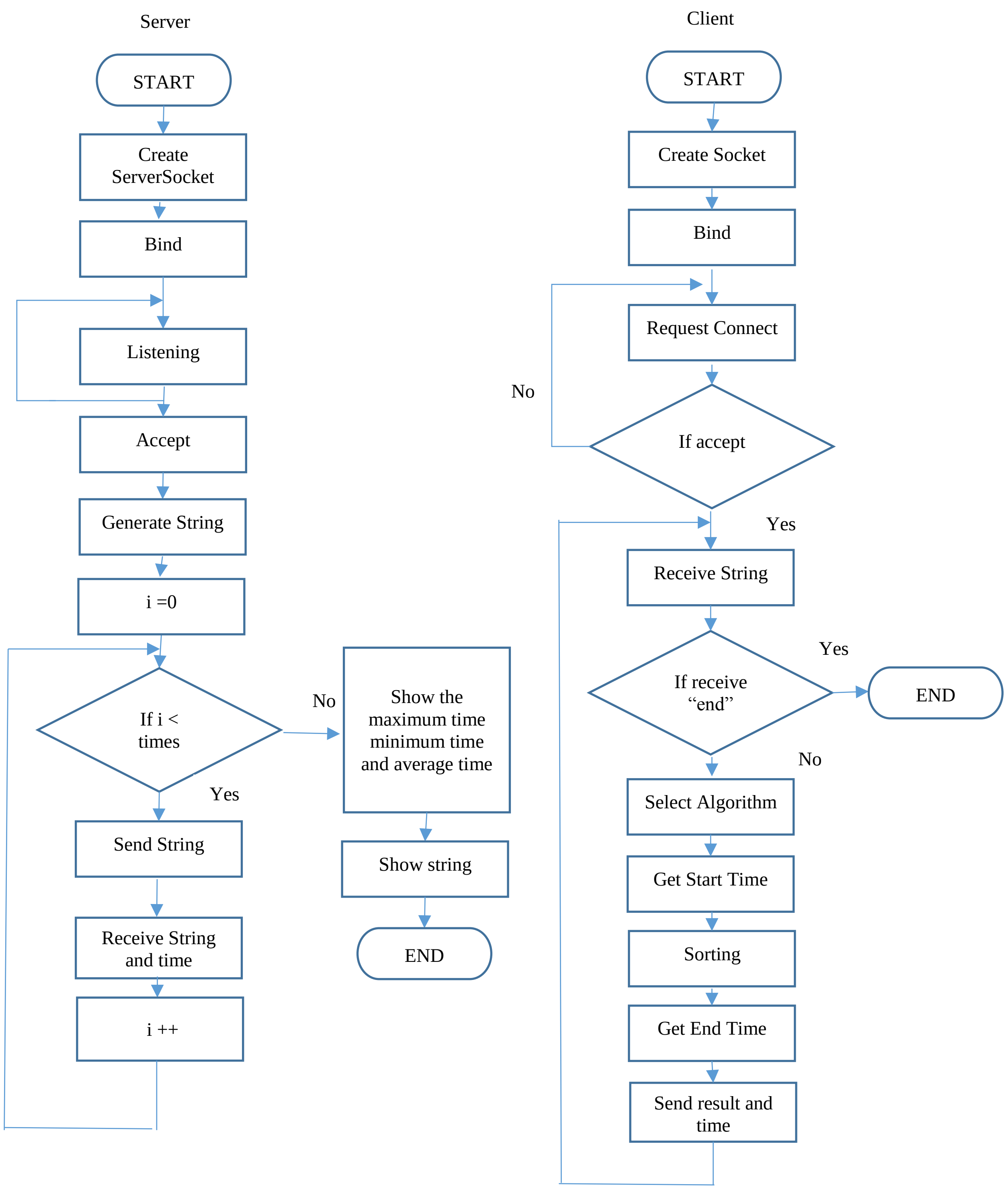


Figure 27. The flowchart with server and one client.

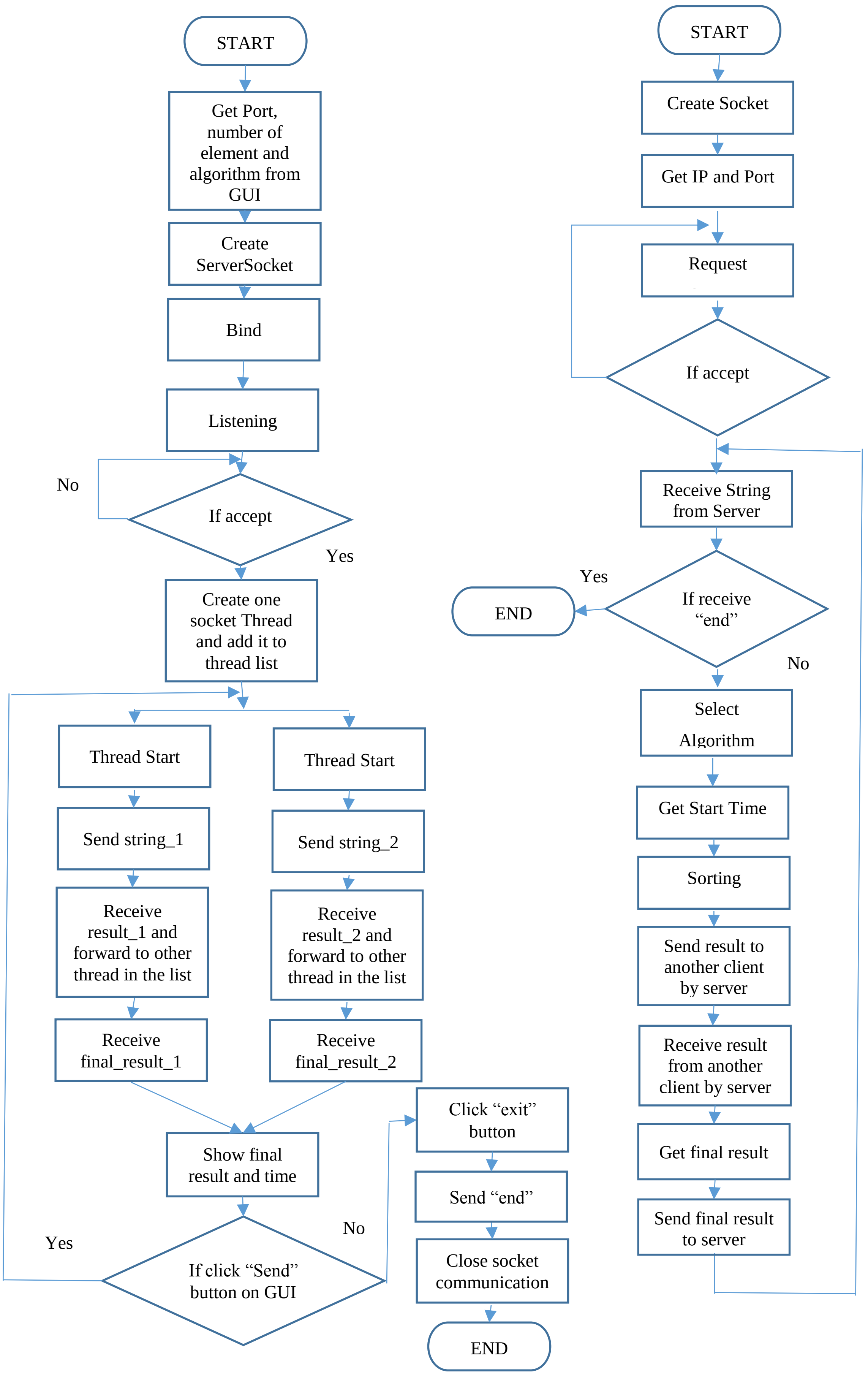

START
Get Port, number of element and algorithm from GUI
Create ServerSocket
Bind
Listening
No
If accept
Yes
Create one socket Thread and add it to thread list
Thread Start
Send string_1
Receive result_1 and forward to other thread in the list
Receive final_result_1
Thread Start
Send string_2
Receive result_2 and forward to other thread in the list
Receive final_result_2
Show final result and time
Yes
If click "Send" button on GUI
No
Click "exit" button
Send "end"
Close socket communication
END
START
Create Socket
Get IP and Port
Request
If accept
Receive String from Server
Yes
END
If receive "end"
No
Select Algorithm
Get Start Time
Sorting
Send result to another client by server
Receive result from another client by server
Get final result
Send final result to server

Figure 28. The flowchart with server and two clients

Figure 28 shows the flowchart of the program running on a distributed system. There is one server and two clients in this part. Each client program runs on one Raspberry Pi.

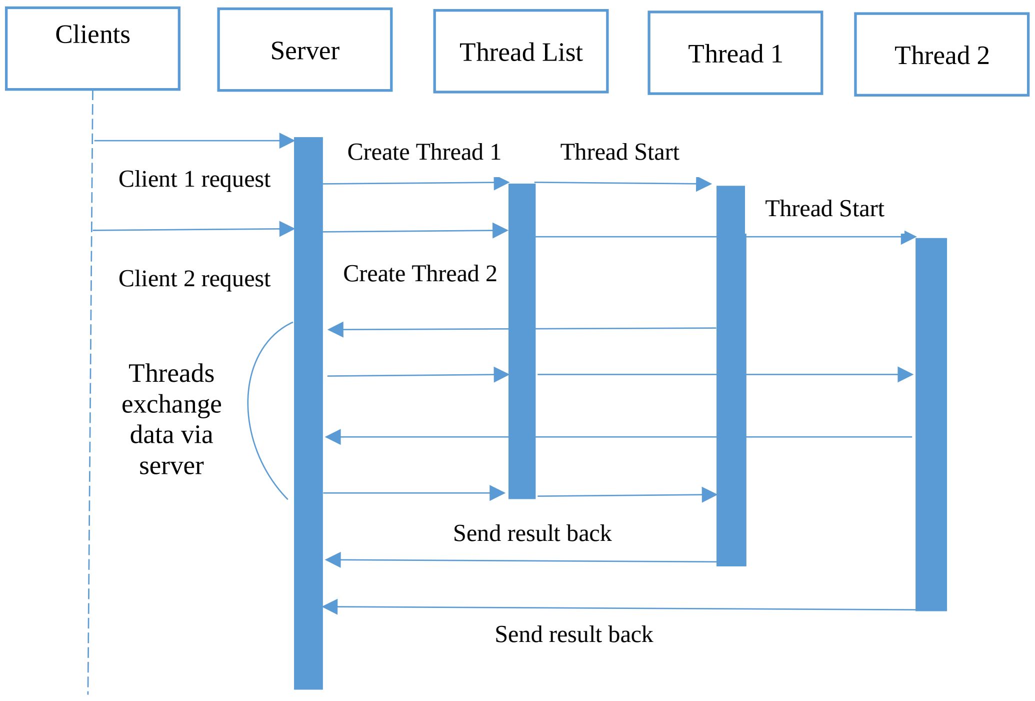


Figure 29. UML timing diagram of programs.

Figure 29 shows interactions between server and the two clients.

4. EXPERIMENTAL PART

In this experimental part, the content is to evaluate and compare the time consumption of different sorting algorithms. The algorithms are evaluated based on the sorting time for different amounts of random numbers that have to be sorted. For each amount of random numbers, each algorithm was executed 100 times and the mean value, the maximum value and the minimum value have been saved to evaluate the algorithms.

4.1. Information of Hardware

As mentioned before, a PC and two Raspberry Pi boards are used to act as distributed systems. The PC consists of an Intel Core I7 processor with 8 GB Random Access Memory (RAM). Raspberry Pi model B used in this experimental part. The parameter and interface of Raspberry Pi model B: it contains a 512 megabyte (MB) of Random-Access Memory (RAM), 700 MHz ARM v6 Broadcomm CPU+GPU. Boots off SD card for filesystem. USB Audio out, LAN, HDMI + Composite video out, GPIO pins, powered off 5V. The switch used in experimental part is HP(j9792a), 12V, 0.17A.

4.2. Experimental Part

There are four main experiments in the experimental part. In the first experiment, the bubble sort, quick sort and heap sort is running on a single client on the PC. In the second experiment, the three sorting algorithms are running on two clients, meaning that the amount of random numbers to be sorted is split. Each client sorts its array of random numbers. When both clients are ready, they exchange the sorted array of random numbers to deliver the complete sorted array of random numbers. In the third experiment, the three sorting algorithms are running on one client on the Raspberry Pi while the server ran on the PC. In the fourth experiment, there are two clients running on two Raspberry Pi while the server ran on the PC.

As mentioned before, the time complexity of three algorithms is bubble sort: O (n^2), quick sort: O $(\mathrm{n}\log_2 n)$, heap sort: O $(\mathrm{n}\log_2 n)$. In theory, bubble sort has the highest sorting time while quick sort and heap sort should have the same performance.

4.2.1 Sorting Algorithms running on PC (one client)

In this experiment, both server and client running on PC. Table 3 shows the maximum, minimum and average sorting time with different amount of numbers. For each amount of random numbers, each algorithm was executed 100 times.

Table 3. Time Consumption of Algorithms with One Client on PC.

One Client on PC / Bubble Sort / Unit: Second						
Amount of random numbers	500	1000	5000	10000	15000	20000
Max	0.00098	0.00243	0.02568	0.09431	0.20901	0.36793
Average	0.00096	0.00221	0.02521	0.09427	0.20894	0.36776
Min	0.00095	0.00209	0.02520	0.09344	0.20862	0.36711

One Client on PC / Quick Sort / Unit: Second						
Amount of random numbers	500	1000	5000	10000	15000	20000
Max	0.00083	0.00198	0.02059	0.07786	0.16992	0.30835
Average	0.00080	0.00185	0.02045	0.07712	0.16927	0.30806

Min	0.00078	0.00167	0.02045	0.07705	0.16854	0.30384

One Client on PC / Heap Sort / Unit: Second						
Amount of random numbers	500	1000	5000	10000	15000	20000
Max	0.00094	0.00197	0.02203	0.07961	0.17837	0.31503
Average	0.00091	0.00196	0.02105	0.07894	0.17694	0.31012
Min	0.00080	0.00189	0.02032	0.07868	0.17427	0.30744

Then compare the time consumption of the three algorithms.

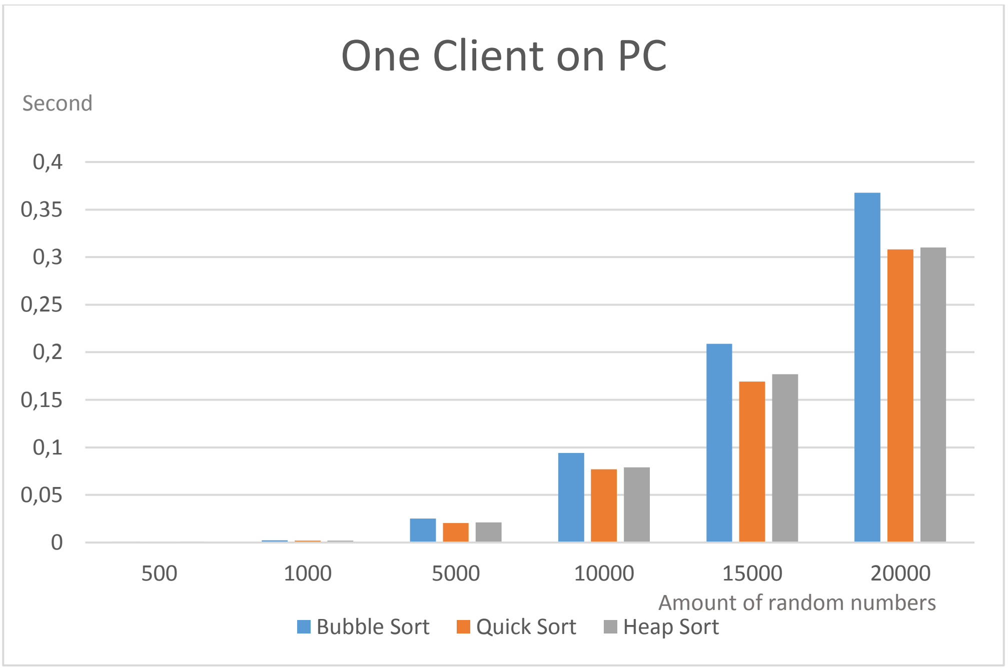


Figure 30. Comparison of the three algorithms executed with one client on PC.

Figure 30 indicate that time consumption of bubble sort increases most with an increased array size. Quick sort and heap sort has almost the same performance with an increased array size, which agrees with the same time complexity of two algorithms.

4.2.2 Sorting Algorithms running on PC (two clients)

In this part, there are two clients. The server divides the original array before sending them to the clients as described in 4.2.

Table 4 shows the average, maximum and minimum sorting times for the different amount of random numbers.

Table 4. Time Consumption of the Algorithms with two Clients on PC.

Two Clients on PC / Bubble Sort / Unit: Second						
Amount of random numbers	500	1000	5000	10000	15000	20000
Max	0.00026	0.00062	0.00931	0.03430	0.00772	0.13476
Average	0.00021	0.00055	0.00778	0.33890	0.07516	0.13379
Min	0.00017	0.00051	0.00883	0.03358	0.07572	0.13294

Two Clients on PC / Quick Sort / Unit: Second						
Amount of random numbers	500	1000	5000	10000	15000	20000
Max	0.00021	0.00051	0.00767	0.02960	0.06680	0.11581
Average	0.00018	0.00047	0.00759	0.02950	0.06619	0.11659

Min	0.00015	0.00041	0.00540	0.02938	0.06576	0.11767

Two Clients on PC / Heap Sort / Unit: Second						
Amount of random numbers	500	1000	5000	10000	15000	20000
Max	0.00018	0.00051	0.00798	0.02981	0.06669	0.11763
Average	0.00017	0.00046	0.00763	0.02966	0.06625	0.11737
Min	0.00015	0.00041	0.00749	0.02945	0.06595	0011708

Figure 31 shows the comparison of three algorithms.

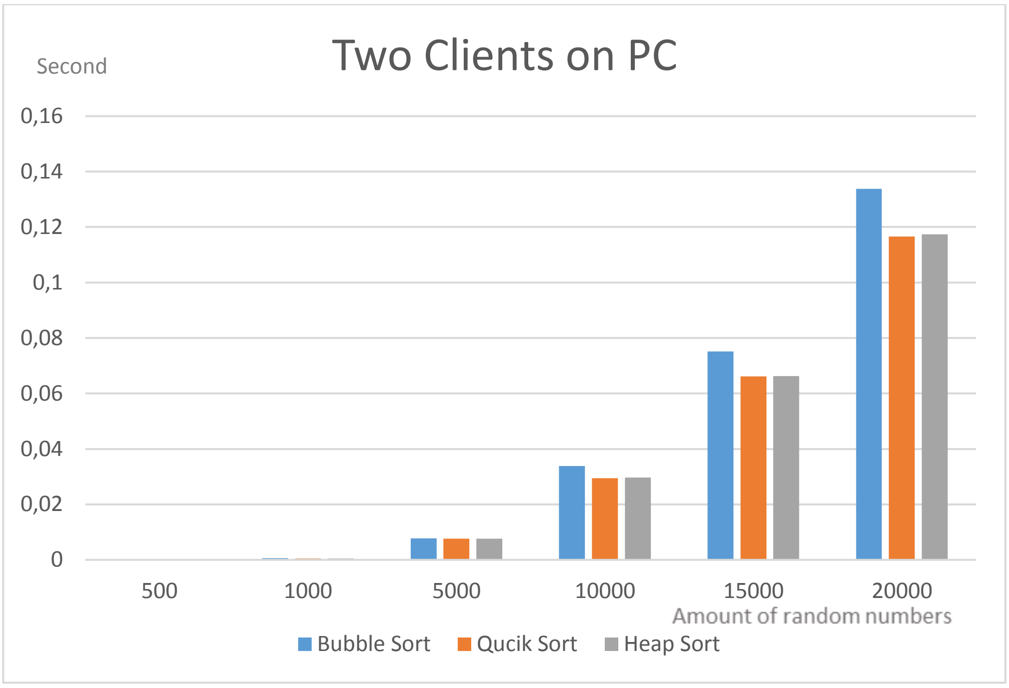


Figure 31. Comparison of three algorithms executed with two clients on PC.

According to figure 32, the results demonstrate that the time consumption of bubble sort increases when the array size is increased. Quick sort and heap sort has almost the same performance with an increased array size.

4.2.3 Sorting Algorithms running on one Raspberry Pi

This experiment is similar to section 4.2.1 except that the client that sorts the random numbers is running on Raspberry Pi. The server is running on the PC and communicates with the Raspberry Pi through sockets. Table 5 shows the maximum, minimum and average sorting times for the different amount of random numbers.

Table 5. Time Consumption of Algorithms with One Client on Raspberry Pi.

One Client on Raspberry Pi / Bubble Sort / Unit: Second						
Amount of random numbers	500	1000	5000	10000	15000	20000

Max	0.42000	1.15400	16.7950	77.9070	121.721	214.813
Average	0.36937	1.01047	15.0556	57.0640	120.557	213.522
Min	0.36000	0.96800	14.4160	55.6190	120.213	212.986

One Client on Raspberry Pi / Quick Sort / Unit: Second						
Amount of random numbers	500	1000	5000	10000	15000	20000
Max	0.28700	0.67000	6.45300	20.4730	39.0560	66.4740
Average	0.27444	0.65695	5.97670	19.2637	38.8660	66.3357
Min	0.26900	0.64800	5.88600	18.9110	38.6600	66.2340

One Client on Raspberry Pi / Heap Sort / Unit: Second						
Amount of random numbers	500	1000	5000	10000	15000	20000
Max	0.39200	0.72100	6.68500	21.3520	42.856	66.8450
Average	0.30070	0.71000	6.58675	21.1230	42.845	66.8520
Min	0.29000	0.70100	6.56000	20.9500	42.781	66.8670

Figure 32 shows the comparison of three algorithms.

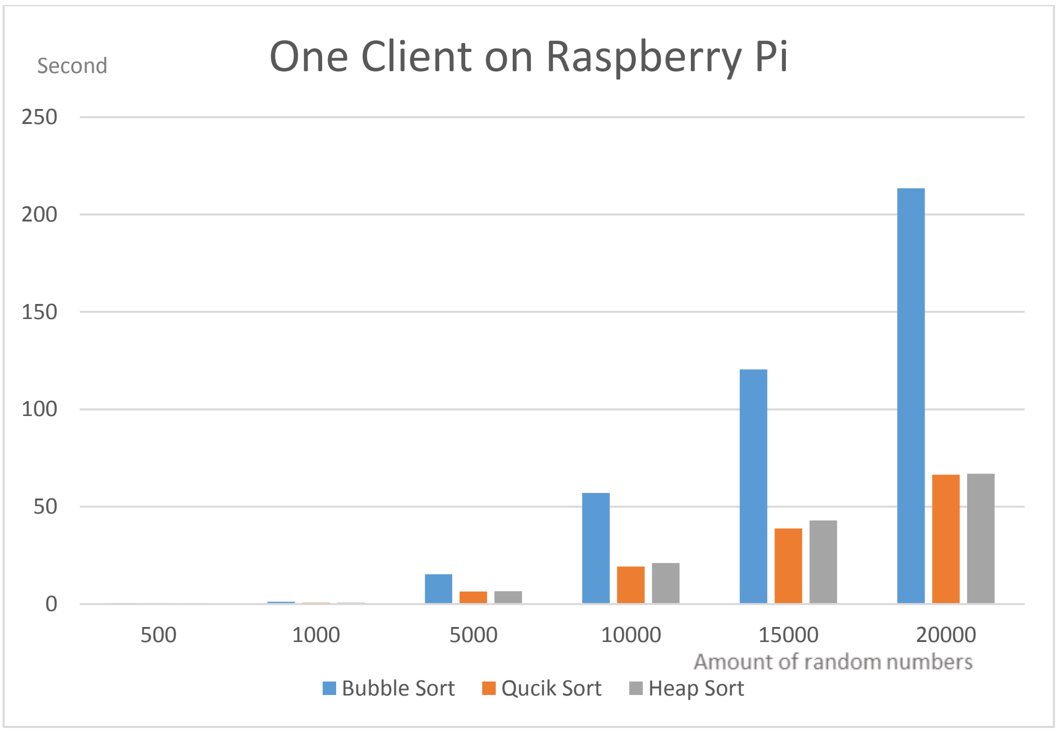


Figure 32. Comparison of three algorithms executed with one client on Pi.

Figure 32 indicates that the performance of algorithms on raspberry pi is close to the algorithms on PC: bubble sort has the highest time consumption, the time consumption of quick and heap sort is much lower. It can be concluded that quick sort is the fastest algorithm.

4.2.4 Sorting Algorithms running on two Raspberry Pi's

This experiment is similar to section 4.2.2 except that two clients are running on two Raspberry Pi's. Table 5 shows the maximum, minimum and average sorting times for the different amount of random numbers.

Table 6. Time Consumption of Algorithms with Two Clients on Pi.

Two Clients on Pi / Bubble Sort / Unit: Second						
Amount of	500	1000	5000	10000	15000	20000

random numbers						
Max	0.40700	0.89700	9.95100	34.6160	68.4990	133.524
Average	0.39160	0.88800	9.50320	32.3420	67.6310	123.431
Min	0.37600	0.88000	9.72100	30.8000	67.1690	116.544

Two Clients on Pi / Quick Sort / Unit: Second						
Amount of random numbers	500	1000	5000	10000	15000	20000
Max	0.36700	0.82800	7.37900	21.394	43.5350	76.3920
Average	0.36600	0.82600	7.34360	21.1660	43.4620	76.3670
Min	0.36400	0.82300	7.30800	21.094	43.404	76.3210

Two Clients on Pi / Heap Sort / Unit: Second						
Amount of random numbers	500	1000	5000	10000	15000	20000
Max	0.40700	0.86200	7.46300	22.9120	46.4550	78.6510
Average	0.39100	0.85500	7.41480	22.4560	45.7610	77.2210
Min	0.38200	0.84800	7.38700	22.2950	45.4020	76.4470

Figure 33 shows the comparison of three algorithms.

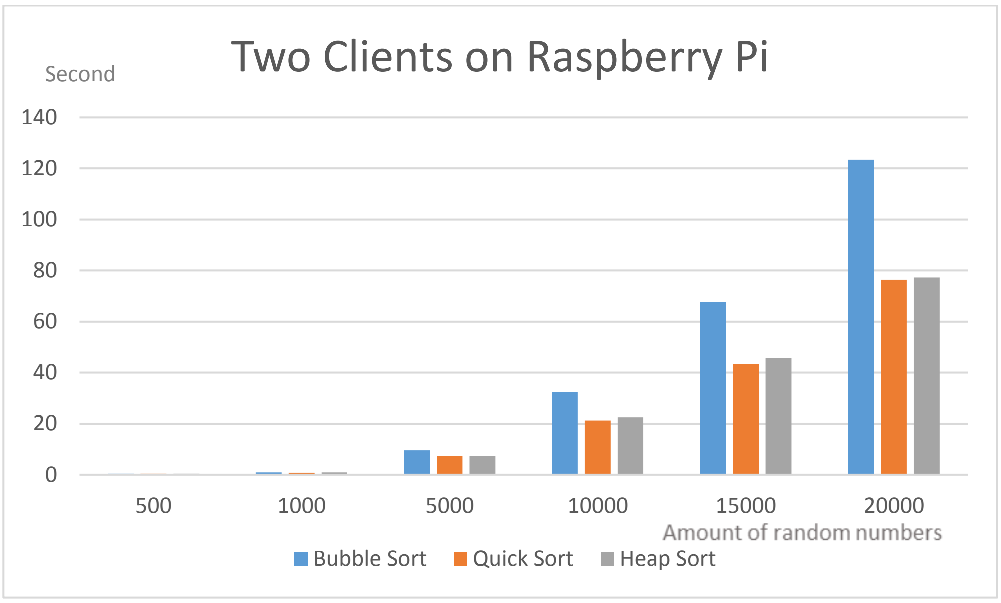


Figure 33. Comparison of three algorithms executed with two clients on Pi.

Figure 33 shows similar results as Figure 33 except the sorting time is much faster.

4.3. Stability performance of Algorithms

An efficient sorting algorithm has a stable performance in a different execution environment. Standard deviation reflects the degree of dispersion of a set of data. Let’s assume a set of data: $X_1, X_2 \dots \dots X_n$. The mean is μ. Standard deviation is σ.

$$\sigma = \sqrt{\frac{1}{n}\sum_{i=1}^{N}(X_i - \mu)^2} \qquad (10)$$

Standard deviation of time consumption can be calculated according to the result of the previous experiments. Table 7 shows the standard deviation of the sorting times of the previous experiments.

Table 7. Standard deviation of Algorithms with One Client on Pi (1000 numbers).

	Bubble Sort	Quick Sort	Heap Sort
Standard deviation	0.061203	0.006502	0.006898
Max	1.154000	0.670000	0.721000
Average	1.010470	0.656950	0.710000
Min	0.968000	0.648000	0.701000

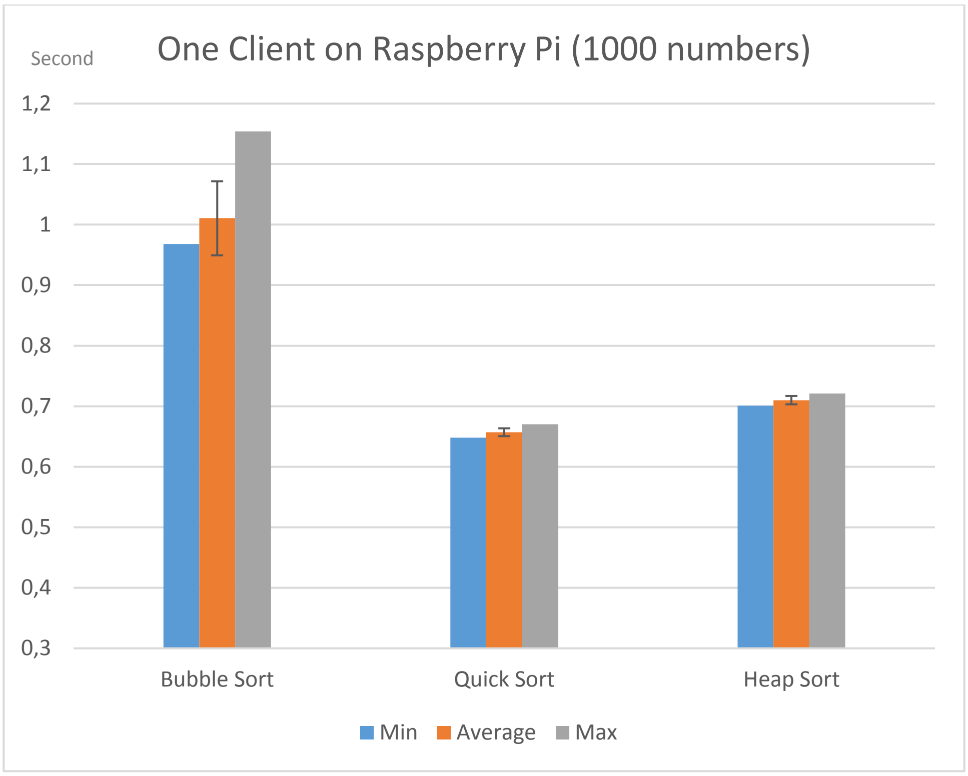


Figure 34. Error bar of the three algorithms executed with one client on Raspberry Pi.

Figure 34 shows the error bar of three algorithms for 1000 random numbers. Which indicates that quick sort and heap sort have more stable performance than bubble sort does.

Table 8 shows the standard deviation of the recorded sorting times for the algorithms running on two Raspberry Pi's.

Table 8. Standard deviation of Algorithms with Two Clients on Pi (1000 numbers).

	Bubble Sort	Quick Sort	Heap Sort
Standard deviation	0.0005657	0.0021210	0.0049500
Max	0.8970000	0.8280000	0.8620000
Average	0.8880000	0.8260000	0.8550000
Min	0.8620000	0.8230000	0.8480000

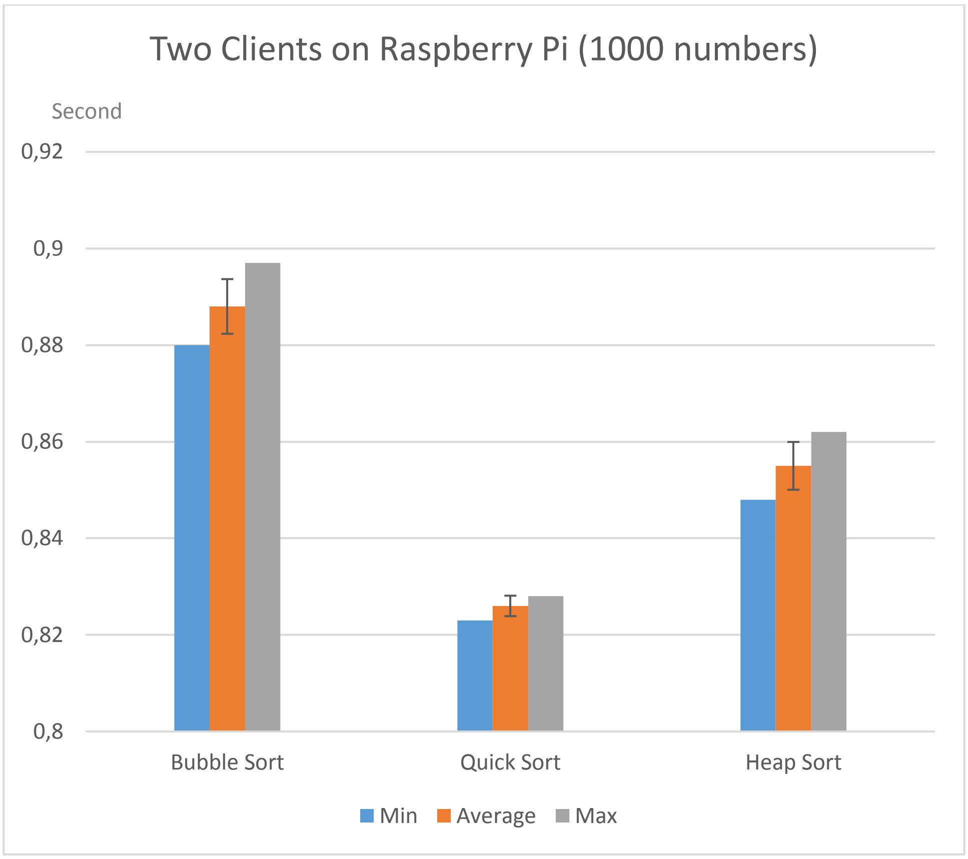


Figure 35. Error bar of the three algorithms executed with two clients on Raspberry Pi.

The result of figure 35 indicates that quick sort is more stable than heap sort and heap sort is more stable than bubble sort.

According to figure 34 and 35, it is obvious that there is a smaller standard deviation when there are two clients running on Raspberry Pi (model of distributed systems) which means algorithms have more a stable performance in distributed systems.

4.4. Result of Experimental Part

The previous experiments show that bubble sort is the most inefficient sort algorithm of these three algorithms. The performance of quick sort and heap sort is quite the same but quick sort is always a little better than heap sort.

The performance of each sorting algorithm executed on PC is much better than execute on Raspberry Pi. Because the processor of PC is more powerful than Raspberry Pi’s.

From the table 3 and 4, the figure 36 shows the comparison of time consumption of the three algorithms with one client and two clients on PC.

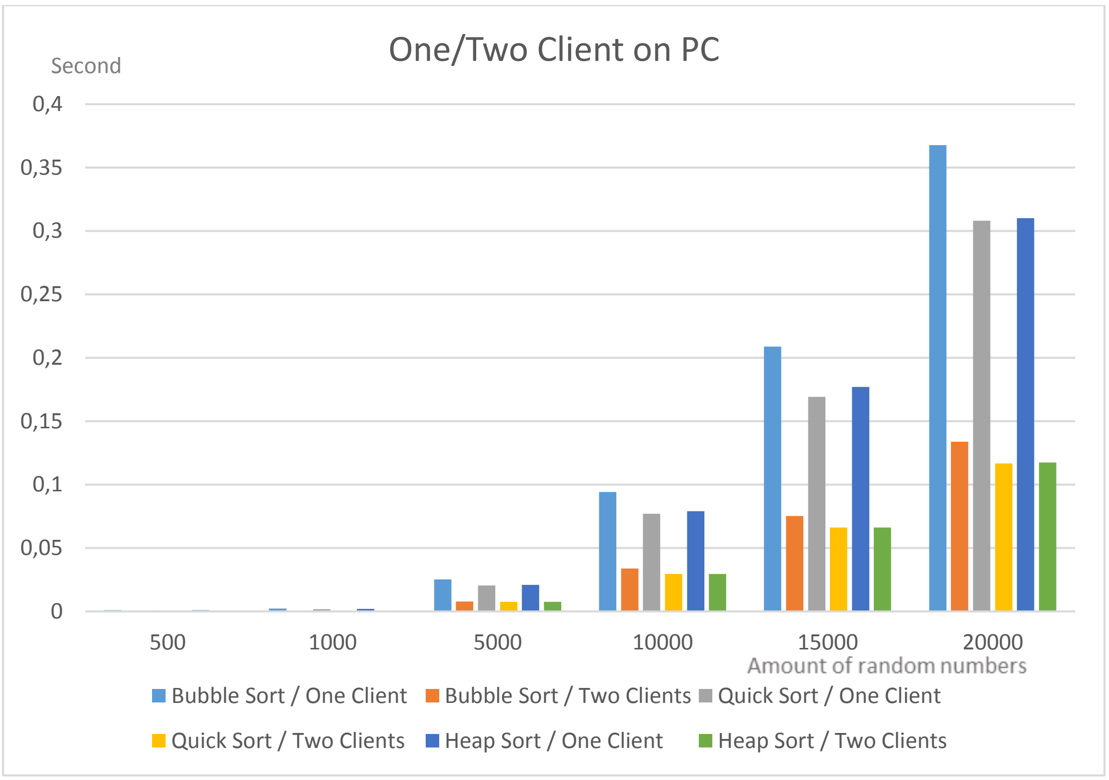


Figure 36. Comparison of the three algorithms on PC.

The result of figure 36 indicates that if there are two clients, the performance of each three algorithms improve a lot in comparison to one client.

From the table 5 and 6, figure 37 shows the comparison of time consumption of the three algorithms with one client and two clients on Raspberry Pi.

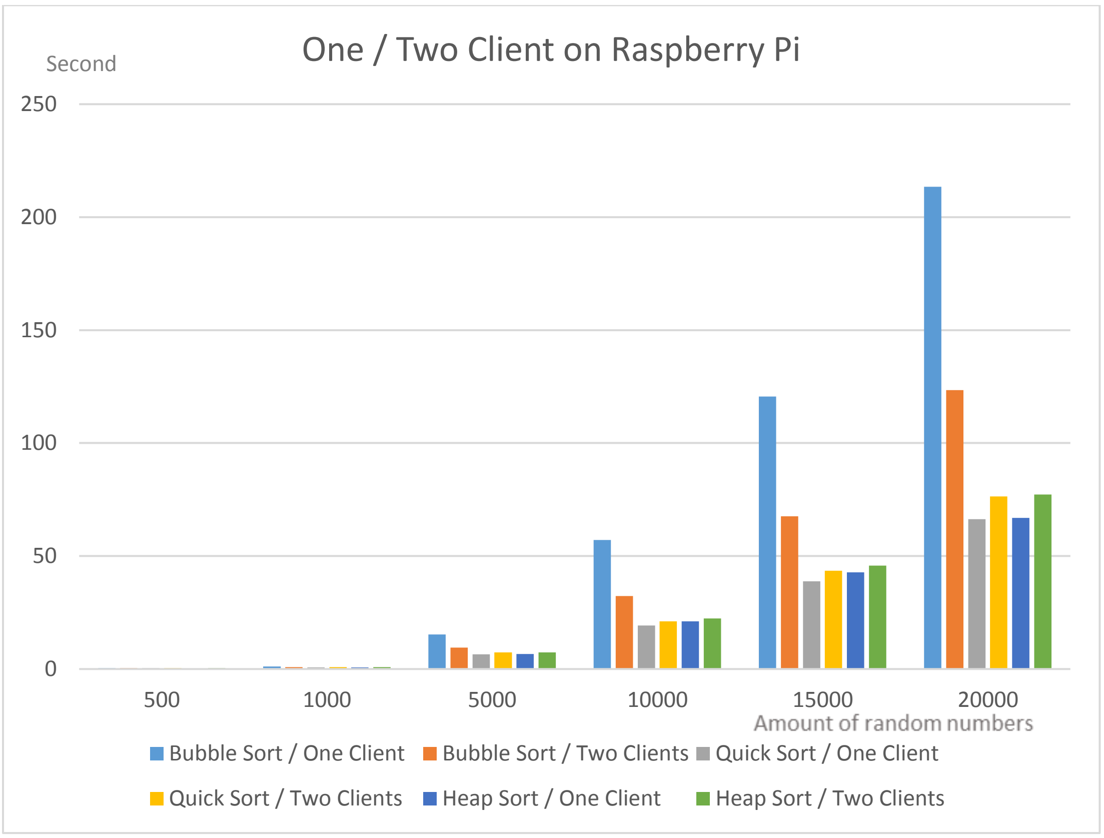


Figure 37. Comparison of the three algorithms on Raspberry Pi.

The results of figure 37 can be concluded that: The performance of bubble sort improve a lot when there are two clients.

The performance of quick Sort and Heap sort is almost the same when there is only one client and there are two clients. Because the time consumption for data transmission even more than the time for saving the sorting times. If there is no time for data transmission between two Raspberry Pi's, the performance of quick sort and heap sort improves lot also there are two clients.

Distributed systems can improve the performance of algorithms by dividing tasks into smaller ones. Nevertheless, there are more time consumption for data transmission at the same time. Therefore, the distributed system is more suitable for a bigger amount of numbers that need to be sorted.

5. CONCLUSION AND FUTURE WORK

In this thesis, the background information includes socket and socket communication, the definition of TCP/IP protocols, sorting algorithm and distributed systems was introduced. The information of Raspberry Pi, time measurement function and the GUI of server were described. Furthermore, the sorting algorithms (bubble sort, quick sort and heap sort) has been explained in detail. For the experimental part, a server software with a GUI has been implemented which generates a certain amount of random numbers and send them to one or two client(s). There the numbers are being sorted. After being sorted the client/clients sends/send the random numbers back to the server and the sorting time is displayed on the GUI. For each amount of random numbers, 100 time measurements were recorded and the mean value was computed. The server program always runs on PC. The algorithms were tested with 500, 1000, 5000, 10000, 15000, 20000 random numbers. In the first experiment, one client ran on the same PC as the server. In second experiment, two clients were running on the same PC as the server. In the third experiment, one client was running on the Raspberry Pi while the server ran on the PC. In the fourth experiment, two clients were running on two Raspberry Pi. It is to mention that when two clients were running, the array with random numbers was split. The first half was sent to the first client; the second half was sent to the second client. When the clients have finished sorting their array part, they exchange it through the server, so that the whole array can be sorted.

The results of the experiments show the following properties:

1. Quick sort has the best performance, heap sort’s performance was close to quick sort and bubble sort was the most inefficient algorithm.
2. The sorting time was much shorter on the PC than on Raspberry Pi. This is also plausible because the CPU of a PC is much more powerful than the CPU of a Raspberry Pi.
3. Distributed systems can reduce the time consumption of solving tasks significantly for bubble sort. However, distributed systems cost more time for

data transmission and collects result from the client. That the reason why there is no remarkable improvement for quick sort and heap sort when the client running on Raspberry Pi. So in practical, the scale of the task, algorithm, additional cost of distributed systems, all above factors need be considered when applying distributed systems.

4. The performance of algorithms is more stable in distributed systems than there is only one client. The reason is the stability of performance can be improved when two clients share one task. This is an advantage of distributed systems.

This thesis contains the performance evaluation of different sorting algorithms implemented and tested for distributed systems. The measurement of the energy consumption as well as further experiments with more clients are left for the future work.

REFERENCES

Burkepile, A. (2013). *Raspberry Pi Airplay Tutorial* [online]. Razeware LLC. Available from the internet:

<URL: http://www.raywenderlich.com/44918/raspberry-pi-airplay-tutorial>

Bryant Geoff. (1996). *Programming TCP/IP with Sockets* [online]. Available from the internet:

<URL:http://www.cs.carleton.edu/faculty/adalal/teaching/f04/117/notes/searchSort.pdf>

Csizmar Dalal Amy. (2004). *Searching and Sorting Algorithms* [online]. Available from the internet:

<URL:http://www.cs.carleton.edu/faculty/adalal/teaching/f04/117/notes/searchSort.pdf >

Chebrolu Kameswari. (2006). *Socket Programming* [online]. Dept. of Electrical Engineering. Available from the internet:

<URL: http://home.iitk.ac.in/~chebrolu/scourse/slides/sockets-tutorial.pdf >

Jorgensen, Beej. (2016). *Beej's guide to Network Programming Using Internet Sockets* [online]. Version 3.0.20. Available from the internet:

<URL: http://beej.us/guide/bgnet/output/print/bgnet_A4.pdf>

Iliopoulos Vasileios. (2013). *The Quicksort algorithm and related topics* [online]. Available from the internet:

<URL: https://arxiv.org/pdf/1503.02504.pdf>

M. Thampi Sabu. (2009). *Introduction to Distributed Systems* [online]. Available from the internet:

<URL: https://arxiv.org/ftp/arxiv/papers/0911/0911.4395.pdf>

Nadiminti Krishna, Marcos Dias de Assunção & Buyya Rajkumar. (2006). *Distributed Systems and Recent Innovations: Challenges and Benefits* [online]. Available from the internet:

<URL: http://www.buyya.com/papers/InfoNet-Article06.pdf>

Seighman Scott. (2012). *Developing with Oracle Java Embedded Technology for the Raspberry Pi* [online]. Available from the internet:

<URL:http://files.meetup.com/1401221/CLE-JUG-Java-Embedded-Raspberry-Pi-v1_0.pdf>

Stassiy Igor. (2014). *Analysis of String Sorting Using Heapsort* [online]. Available from the internet:

<URL: http://arxiv.org/pdf/1408.5422.pdf>