CS3102: Practical 1 (P2P File Transfer) - Report

Matric Number: 170002815

11th February 2020

1 Overview

The objective of this practical was to implement an efficient and reliable file distribution system using Java network programming techniques. The distribution system was required to have at least linear scalability for sending files up to, and including, 6 nodes at a time. My approach for implementing this specification is discussed in the *Design & Implementation* section of this report, and testing for my implementation is discussed in the *Testing* section of this report.

Beyond designing and developing this system, the practical specification required the gathering of data through experimentation using my system, as well as the analysis of these results - these topics are discussed in Ssection 5. Section 6 explores the theoretical and practical limitations of my design, and section 7 provides a brief overview of the extensions I completed for this practical.

The report is completed by an evaluation and conclusion, which reflect back on the learning objectives of the practical, and also evaluate how successful I have been in relation to the specification.

2 Usage

2.1 With GUI & UFIP

Running my program with all extensions is achieved by completing the following steps.

1. Change directory into the `src` directory alongside this report file.
2. Execute `javac *.java` .
3. Execute `java p2p` .

2.2 No Extensions

Running my program with no extensions is achieved by completing the following steps.

1. Change directory into the `src/noGUI` nested directory inside the `src` directory alongside this report file.
2. Execute `javac *.java` .
3. Execute `java p2p` using either of the following sets of parameters.
    - **Sending File:** `java p2p send <filename> <[optional] parallel-connection-count>`
      `<filename>` should be the relative path to the file to send. `<parallel-connection-count>` is optional, and should be the desired number of nodes receiving files in parallel during testing.
    - **Receiving File:** `java p2p rec <hostname> <[optional] filename>`
      `<hostname>` should be the IP address of the host. `<filename>` is optional, and should be the relative path to the file creation location.

2.2.1 Testing Scripts & Files

My test files can be located within the `src/noGUI` nested directory inside the `src` directory alongside this report file. The shell scripts can be run by using `./<scriptname>` and following the command line prompts, while the Orange workspace used to generate plots and statistics can be opened used Orange3 [1]. Please note that some shell scripts may require sshpass to run properly [2].

3 Design & Implementation

3.1 Interface

When beginning to program my system, I decided to contain both the sending and receiving program code in the same file (`p2p.java`). The reason behind this decision is that I wanted both the sender and receiver to execute the same program whether they are sending or receiving, as I could not think of a real-world peer-to-peer application that has two separate programs for sending and receiving data.

3.2 Protocol Choice

3.2.1 Transport Layer

One of the first decisions I made when designing my system was to choose the transport protocol in which data was to be sent between two peers. The two options I considered were UDP and TCP.

UDP offers efficient data transfer, and if implemented with multicast, my system could have excellent scaling potential. Although UDP adheres to the specification with efficient transfer, it is not reliable, as it does not guarantee all packets are received. This means that I would have to implement a complicated application protocol on top of UDP, or would have to combine UDP with TCP to ensure all data is transferred correctly - this lead me to consider TCP as my chosen transfer protocol.

TCP provides reliable data transfer, and although it is not quite as efficient as UDP, it is still efficient by definition in the specification if implemented correctly. TCP is also very easy to program with when using Java's socket interface, making it the ideal choice for the transport protocol in my opinion.

3.2.2 Application Layer

After deciding on TCP for the transport layer protocol, it was important for me to define an application protocol which sending and receiving processes should use to communicate properly. I decided on the following protocol.

1. Sender opens a TCP server socket on port 3000.
2. Receiver opens a TCP socket using the sender's IP and port 3000, in turn establishing a connection.
3. Sender sends a UTF string containing the file name to the receiver.
4. Sender sends the file size as an int.
5. Sender sends chunks of 16384 (2^{14}) bytes until all bytes from file are sent. The final chunk will only contain the remaining bytes after dividing the file size by (2^{14}).
6. Receiver closes the connection once all bytes are received.

The port number of 3000 used in step 1 and 2 was chosen to avoid common TCP ports (otherwise, it is arbitrary), and the chunk size was chosen to facilitate the implementation of potential extensions, including progress monitoring and failure recovery, while keeping the transfer speeds efficient. The file size being restricted to an int's max value is for simplicity, and is discussed further in the *Limitations* section of this report.

3.3 Concurrency

As discussed in Section 3.2.1 of this report, I decided on TCP as the transport protocol for my system, and although it complies with the basic specification, I wanted my design to scale more effectively than linearly. To achieve this, I decided to use multi-threaded programming to enable the sending peer to copy multiple files in parallel. *Figure 1* contains a thread diagram showing what data is passed between which threads in the sending and receiving processes, and a description for each label is given below.

A represents where the receiving process' main thread connects to the sending process', which opens a socket; **B** represents where the receiving process registers this connection. **C** shows the branching point of the sending process into multiple threads. The sending worker thread then sends the file data to the receiving process, represented by **D** and **E**. Once all data is received and written to disk, the receiving process closes the connection with the sender and stops execution - shown by **F**. The sending worker thread is notified of this disconnection at **G**, leading to the worker thread joining the main sending thread at **H**.

The advantage of running each sending process in this manner is that between **C** and **H**, the main sending thread can listen for more connections and create a new thread for each connection, theoretically sending all data perfectly in parallel, giving my system constant scaling capabilities. Unfortunately, there are only a certain number of threads on each CPU (4 in the case of most St Andrews Computer Science lab machine [3]), and networking hardware can become congested, meaning my system's scalability is less than constant in practice - this can be seen in the *Data Collection & Analysis* section of this report.

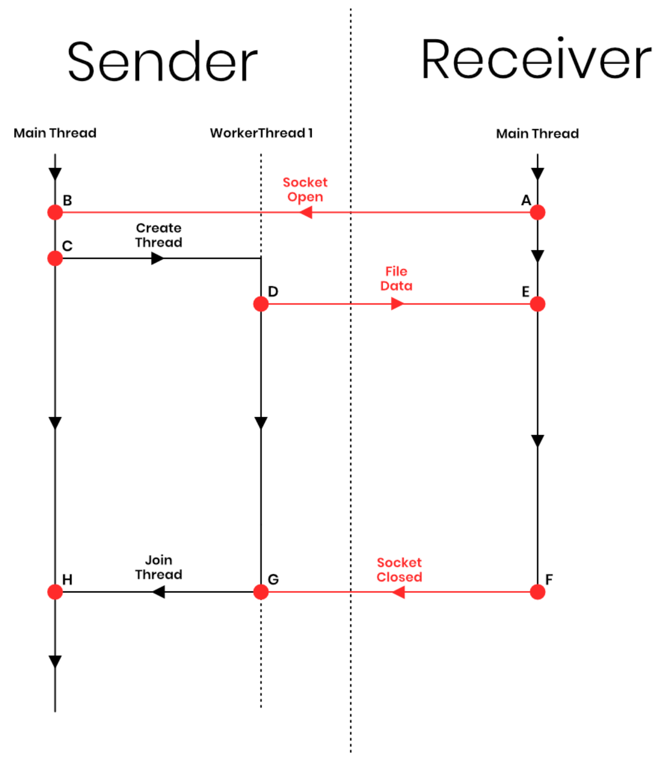

Figure 1: Sender-Receiver Thread Diagram

4 Testing

4.1 Transfer

The most important part of testing my system was to ensure files were sent successfully and in the correct order. To complete this testing, I decided to send increasingly complex and large files using my system.

I started with a simple 13 byte `.txt` file, which was sent and received correctly. The next file I used to test my program was a small `.jpeg` file; although the file sent successfully, my computer would not open the file, as I was sending strings instead of bytes through my TCP connection, making this a successful test in letting me know my system was working incorrectly.

To complete the testing of file transfer, I attempted to transfer all files provided along with the specification. These files included a large (approx. 500MB) `.mp4` file, which I could play back to check the correctness of the file.

4.2 Concurrency

After testing my system's file transfer capabilities, I chose to test whether my programmed concurrency was working as expected. To do this I used Visual Studio Code's debug tool, which displays the number of current running threads; when completing this test, the expected number of threads were shown to be running.

Alongside checking how many threads were running, I also decided to gather some test data to determine whether my multi-threaded programming had increased the scalability of my system. The data collected was similar to that shown in the *Data Collection & Analysis* section of this report.

5 Data Collection & Analysis

Although developing and testing my system was an important part of this practical, the majority of my time spent was on data collection and analysis. This section discusses what techniques and procedures were used during data collection, followed by the graphical representation of my data, and concluded by an analysis of this data.

5.1 Experiment Setup & Procedure

This experiment was split up into two different data collecting sessions; the first session was collecting timing data by using my system, and the second was collecting timing data by using SFTP.

In both data collection sessions, up to 7 lab computers were needed; it was recommended in the specification that the machines used should be the same hardware and operating system wise, and each machine should have no other users except myself. To achieve these requirements, I visited the labs after 11pm on Tuesdays and Sunday afternoons, as there were very few users of the labs at these times - to double check there were no users on the machines I was using, I called the `w` command in the terminal, which lists all current users. To ensure the hardware and operating systems were the same on all machines, I used the *Lab Layout.pdf* file provided alongside the specification, and SSHed only into machines with the same number next to their name (i.e. PC5 or PC3), and used the `-l` appendix on the hostname of each machine when SSHing into the machines.

Each data collection session involved a varying number of nodes (1, 3 and 6), so I decided to collect at least 20 data points for each value to increase the reliability of my experiment. To collect data points, I sent the provided .xml file (approximately 500MB) in parallel to all nodes at once. In order to keep each data point consistent with others, I began sending all files at the same time, which also increased the validity of my experiment - doing so required some extra programming which is discussed below.

5.1.1 My System

To begin sending files in parallel at the same time using my system, I added an optional command line argument and a relating logical path to my Java code. This new path waits for all peers to connect before sending any files. Java is also used to time the file transfer by using a difference between two calls of `System.out.println();`. This code is wrapped in a BASH shell script, which repeatedly runs the program and writes it output to a data file (*experiment-send.sh*). There is also a script to connect the receiving peer to the sender - *experiment-rec.sh*.

5.1.2 SFTP

Although SFTP was suggested as a means of gathering benchmark results for my experiment, I was confident enough in my program to compare it to SCP instead. SCP is commonly regarded as more efficient than SFTP and will therefore challenge my program more effectively in terms of transfer times [4].

To begin sending multiple files at the same time using SCP, I created a BASH script which gathers all peer hostnames using one for loop, and then executes an SCP call to send them the .xml file using another for loop. The BASH command `time` is used to gather timing information and output this to a .csv data file.

5.2 Results

As mentioned in *Section 5.1* of this report, BASH scripts are used to output timings to .csv files. These files can be found in the *src/noGUI/* sub-directory from this report's directory. To represent the data in this report, I used Orange3 [1] to generate scatter plots, remove outliers and plot regression lines.

Figure 2 shows the raw benchmark data, *Figure 3* shows the benchmark data with outliers removed and a regression line added. *Figure 4* shows the raw experiment data gathered using my system, and *Figure 5* shows the experiment data with outliers removed and a regression line added. Please note that the benchmark data is measured in seconds and the experiment data is measured in milliseconds.

The graphs shown below are also included in the *images* directory alongside this report for a clearer view. *Figure 5* also includes 7 nodes, which was completed by accident, however the data was still collected in a proper manner and is therefore included in the graph.

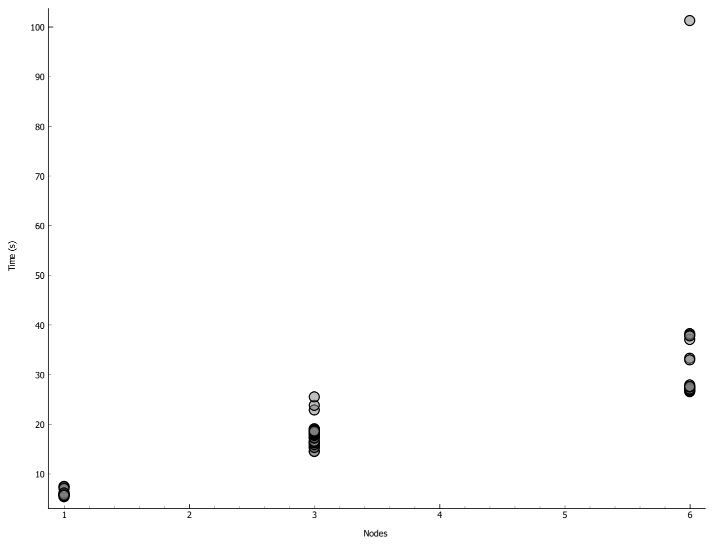

Figure 2: Benchmark Data - Raw

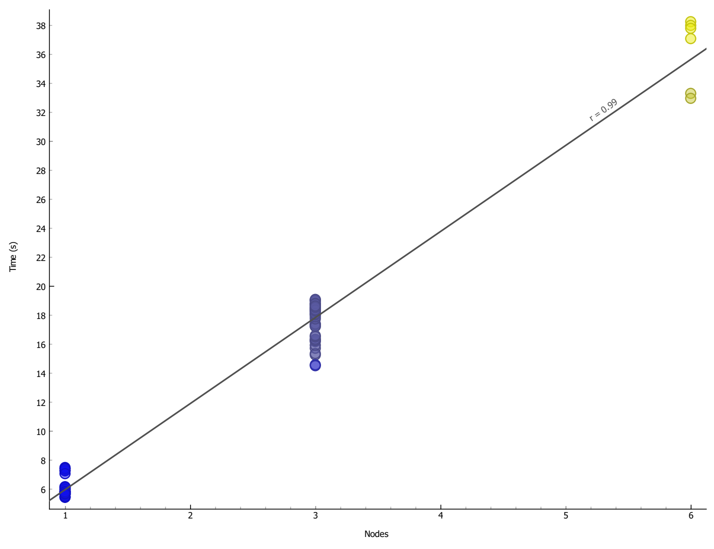

Figure 3: Benchmark Data - No Outliers

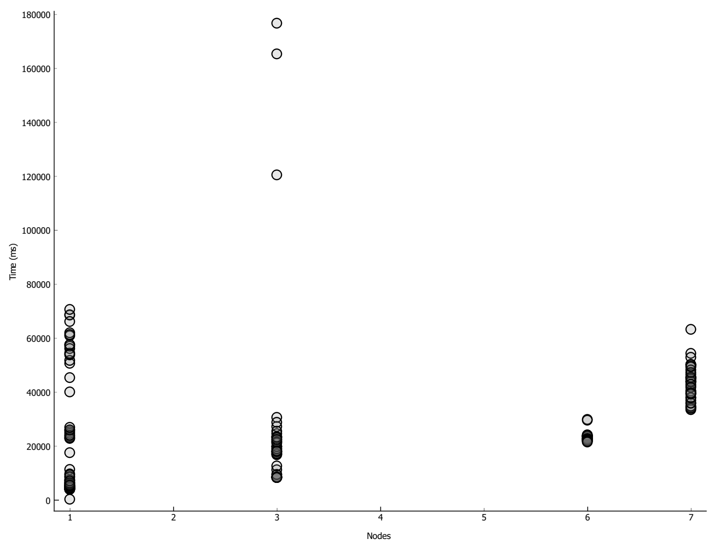

Figure 4: Experiment Data - Raw

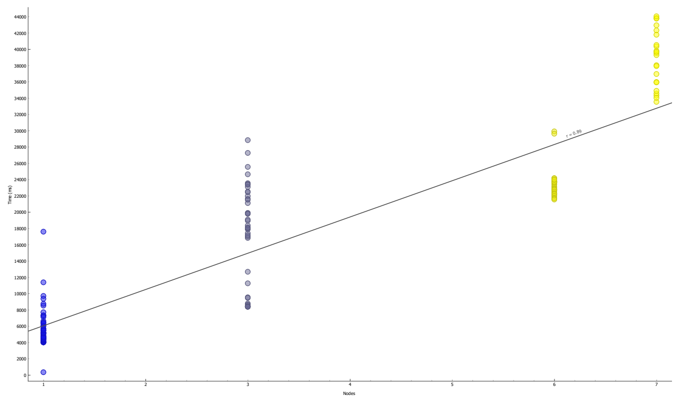

Figure 5: Experiment Data - No Outliers

5.3 Analysis

It can be seen by comparing the enlarged versions of *Figure 3* and *Figure 5* that the benchmark data has a regression coefficient of 0.99, making the scalability of SCP very close to 1, however the experiment data with no outliers has a regression coefficient of 0.89, meaning the scalability of my system is less than linear, and therefore achieves what the practical is requesting.

My system is also seen to be much faster at the upper bounds of the experiment, with the maximum time to send to six nodes (excluding outliers) being roughly 30,000ms, while SCP is close to 38,000ms (38s). Although my system seemed to have better scalability and speed than SCP, my system also proved to have a lot more variance, with the data points being much more spread out than the data gathered by using SCP.

6 Limitations

Although my system is relatively fast at transferring files in comparison to SCP, there are many limitations which affect system's potential, the most prominent of which are discussed below.

6.1 User Friendliness

One application area of my system could be transferring large files between two or more computers within an organisations network, e.g. a developer sending another developer a large `.jar` file - I believe my system has a large limitation of user friendliness in this scenario.

Having to use the command line interface with multiple different input parameters is very unintuitive and time consuming. Another issue of user friendliness is the need to remember a long IP address in order to connect to the sending peer, although this could be solved by using a UDP discovery protocol.

6.2 Security

Any packets sent over a network can be easily detected and read; this fact in combination with none of my files being encrypted before being sent means that any sensitive data contained within any files being sent can be read, leading to a security risk.

A fix to this issue would be to encrypt and decrypt files, and adding a CRC check would prevent any data being tampered with while travelling through the network.

6.3 Hardware

Hardware is usually the largest limitation on any network-oriented system, and my system is both limited by computer hardware and network hardware.

The hardware used by both peers both limits the amount of data sent and retrieved, and the hardware used by the sending peer limits the number of threads which can be used, therefore limiting the scalability of my system.

To decrease network congestion and remove the reliability on threads, I could design my system to use UDP multicast, however this would increase my systems complexity as discussed in the *Design & Implementation* section of this report.

The number of threads which the sender has also limits how many truly parallel connections can be achieved, and this is one of the main reasons why I believe that my system's timing data has a scalability closer to linear than constant.

6.4 File Size

As discussed in *Section 3.2.2* of this report, the file size is limited to the size of an *int* in Java for simplicity. Java specifies that an *int* must be 32 bits in size [5], meaning that my system can send a maximum file size of $2^{31} - 1$ bytes (approx 2.1 GB). Although this is still a relatively large file size, this could be expanded by working with integer overflow in Java.

7 Extensions

Beyond implementing a multi-threaded system design to improve scalability, I decided to approach the issue of user friendliness for my extensions.

7.1 User Friendly IP

As discussed in the *Limitations* section of this report, IP addresses are particularly unfriendly, especially to users who are not used to using the command line or encountering IP addresses.

I decided to approach this problem by first converting all IP addresses users encounter to a group of 4 words, and also displaying the sending peer's IP address clearly in the UI. The effects of these actions are that sending peers will not need to use the command line to find out their IP address, and receiving peers will not need to remember or input a complicated sequence of numbers and dots (an IP address).

To implement this extension, I first created a list of 256 unique words of length 5 characters or less - this list can be seen in the file with filepath `src/assets/UserFriendlyIPWords.csv` from this report file's directory. A class named `UserFriendlyIP` reads this file on instantiation and instantiates two hash maps, which are accessible as attributes. These two hash maps in combination offer an easy and efficient way to convert each section of an IP address into an easy-to-remember word and visa versa. Displaying the User Friendly IP will be discussed in the next subsection.

7.2 GUI

The main time spent on the extensions for this practical was implementing the GUI. Since the networking code was written in Java, I decided to use Java Swing to keep my code base limited to just a few Java files.

The main motivation for creating a GUI was to allow peers to run the same program with no arguments whether they were sending or receiving files. To do this I created 3 different screens (JPanels) for the GUI: one for sending files, one for receiving files, and one for choosing whether to send or receive files - this is also the initial screen.

The sending screen consists of a file selection pane, a send file button, a back button and multiple labels, including an error message and the User Friendly IP discussed in the previous subsection. The file selection pane allows the user to visually choose which file to distribute, and the send button doubles as a cancel button when the file is sending.

The receiving screen consists of a directory selection pane, a receive button, a back button, a progress bar, 4 text inputs, and multiple labels, including an error message. The 4 text inputs allow for receivers to enter the User Friendly IP displayed on the sender's screen, the directory selection pane allows for the receiver to choose the location to where the received file will be saved, and the progress bar displays how much of the file has been received.

Although this extension does not involve much network programming, it has taught me valuable skills in how to implement network programming into user interfaces, especially in relation to working around the blocking calls in order to not halt the UI.

8 Evaluation & Conclusion

Overall I believe I have been successful with this practical - I have implemented the basic specification to a high standard, and have completed both testing and extensions to accompany my work.

Given more time, I would enjoy reducing the limitation put on the file size, as well as implementing a UDP discovery protocol in order to remove any user interaction with IP addresses from the system entirely.

[Page count without extensions: 6]

References

[1] Orange data mining. https://orange.biolab.si/ [Accessed Feb 2020].

[2] sshpass(1) - linux man page.
https://linux.die.net/man/1/sshpass [Accessed Feb 2020].

[3] Hardware class - systems wiki.
https://systems.wiki.cs.st-andrews.ac.uk/index.php/Hardware_Class [Accessed Feb 2020].

[4] Cerberus: Scp or sftp: Which is better?
https://www.cerberusftp.com/comparing-scp-vs-sftp-which-is-better/ [Accessed Feb 2020].

[5] The java tutotials, primitive data types.
https://docs.oracle.com/javase/tutorial/java/nutsandbolts/datatypes.html [Accessed Feb 2020].