CS 4504 Project
Part 1

Brandon Solon - bsolon@students.kennesaw.edu
Dillon Horton - dhorto23@students.kennesaw.edu
Leiko Niwano - lniwano@students.kennesaw.edu
Mark Walker - mwalk229@students.kennesaw.edu
Matthew Krupczak - mkrupcza@students.kennesaw.edu
Sahan Reddy - sreddy13@students.kennesaw.edu
Thomas Roberts - trobe137@students.kennesaw.edu

Abstract

The goal of this project is to deploy a working lab environment that uses the Client/Server paradigm. Part 1 concerns modifying the given code so servers and clients will be able to message each other.

Introduction

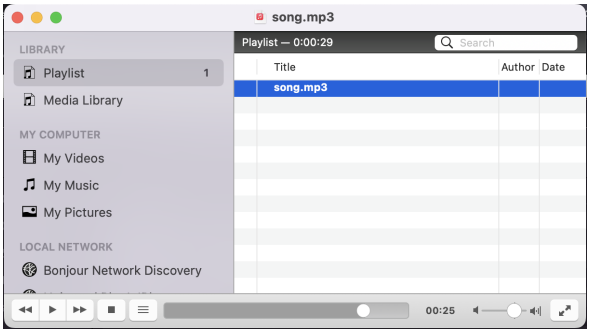
Routing is a ubiquitous problem in distributed computing. For distributed (often heterogeneous) systems to be connected, 1 or more routers must route traffic between disparate hosts, subnets, and networks. The importance of this central role of the router makes it suitable for examination in the context of distributed computing. In this report, these authors describe a set of systems which simulate a basic 1-to-1 client-server protocol over a wireless network, with a multithreaded ServerRouter (hereinafter referred to as the "Router") acting as an intermediary. The Router program is a facsimile of a true IP layer router, with its own lookup table which stores the mapping of each discrete, non-contiguous client-server pair.

Design Approach

Our design approach is centered around simplicity. Our reference implementation provided for the project contained a simple call-response protocol, where every plaintext "call" from a client must be reciprocated with a plaintext "response" from the server, and vice-versa. Our group was tasked with making the system capable of transferring audio and video files as well as text and to benchmark the system under varying conditions.

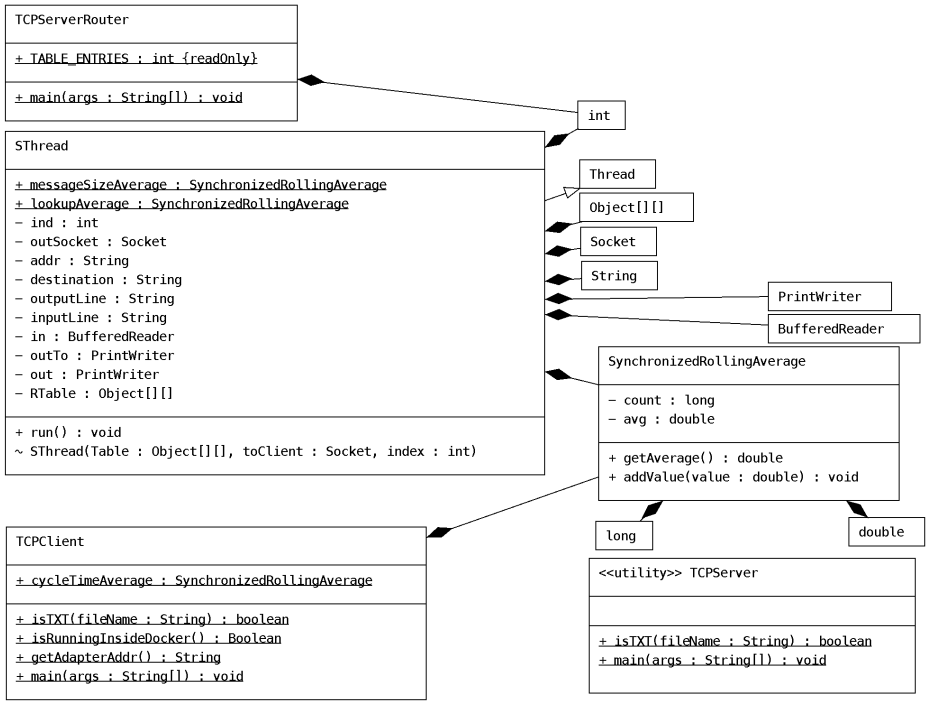
In order to achieve the specifications provided, our group elected to extend the protocol. The first extension to the protocol we implemented was the addition of the datasource's filename to the initial handshake messages. This allowed us to distinguish whether a file was plaintext or binary in order to maintain the behavior of the reference implementation for plaintext. Both the client and server analyze the filename and use its ending (file extension) to select between one of two transmission modes. The first mode, "plaintext", is used if and only if the filename ends in ".txt".

The second extension to the protocol we implemented was a new, secondary transmission mode called "base64payload" mode, which is used if the filename is anything other than ".txt". In this mode, the client encodes the entire datasource file's bytesequence as a String where each character represents 6 bits. Subsequently, the server decodes the base64 String (the payload) and saves it on the server to a file of the same name specified in the handshake. Through running a checksum and audio/video playback on each the source and destination file, we have been able to verify that the payload is delivered completely intact.



For our approach, we modified the given source code for the client-side, the server-side, the router side, and threading code that were written in Java. We made some modifications and improvements where necessary, such as increasing the robustness of the router, adding configurable launch parameters for the client, server and router, and adding systems which collect statistics.

The finished deliverable is provided as 3 JAR files for easy multi-platform distribution.



Implementation Modules

A dedicated host running the ServerRouter program is connected via ethernet to a Wifi Router, which serves as the basis of our testing LAN. The ServerRouter program internally spins up an SThread for each incoming and outgoing connection.

For running each the client and server, we used two commodity x86_64 laptops connected to the same LAN as the dedicated host. Using 2 laptops connected via 5GHz Wifi and the dedicated host connected via ethernet on the same LAN, we executed the TCPClient on one laptop, the ServerRouter on the dedicated host, and the TCPServer on another laptop. The ServerRouter program on the dedicated host calculated statistics such as average message size and average routing table lookup time. The Client calculated a statistic for the average roundtrip, or "Cycle" time. These results were output to stdout on each of their respective runtimes and collected. Each component program was completely reset between tests.

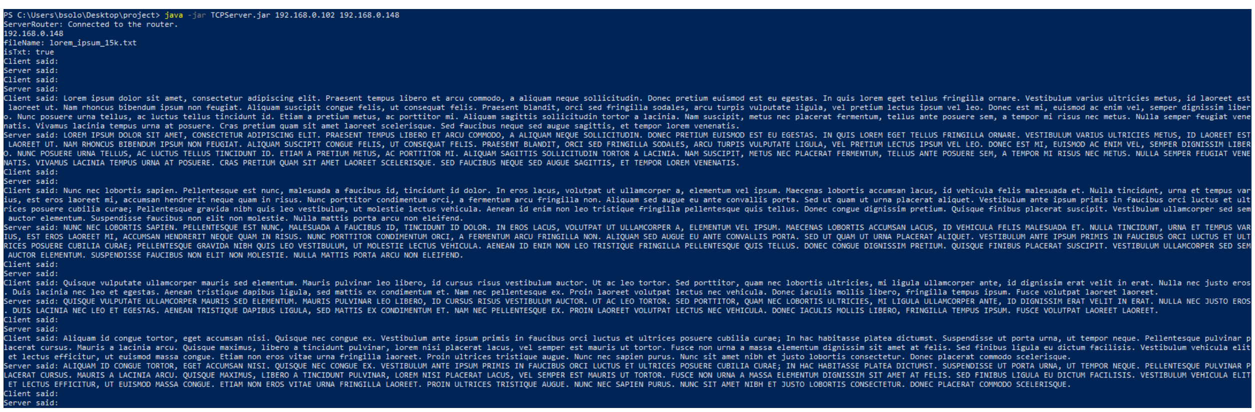
Simulation Methodology

To conduct our testing in Parallel & Distributed Programming, we performed six tests using a variety of file sizes and types. These included four text files with sizes of 11898, 15045, 30101, and 60191 bytes, as well as an audio file of 120330 bytes and a video file of 295982 bytes. Throughout the testing, we focused on four key metrics: Average Message Size, Cycle (roundtrip) Time, Route Table Lookup time, and Number of Messages. To support the tests, we utilized an edge server equipped with 24 threads, 128 gb of memory, and ethernet for the server/router socket. Other computers, which acted as the servers and the clients in the network utilized WIFI in Wilder Communications Building. The TCPClient records System.nanoTime() at the start of its file transfer and once again when it receives a return statement. The reason we measure this round-trip time rather than recording wall clock time both when the client sends and the server receives (directly measuring network transmission time) is because of potential time-syncing errors between the server and the client; NTP (the time-syncing protocol used on Windows, Linux, and macOS) can have errors of 100ms or more. This would cause comparing times recorded between systems to be erroneous. Instead, recording round-trip time keeps the recorded times on the same system, avoiding time-syncing issues; the downside is that it takes into account processing time on the server, though it is still strongly correlated with transmission time.

Running ServerRouter on Linux:

Sending a video file via TCPClient on Linux:

Receiving text file via TCPServer on Windows:



Receiving audio file via TCPServer on Windows:

```
PS C:\Users\bsolo\Desktop\project> java -jar TCPServer.jar 192.168.0.102 192.168.0.148
ServerRouter: Connected to the router.
192.168.0.148
fileName: test-files/video.mp4
isTxt: false
Server said: payload is A-OK kthx
Client said: Bye.
```

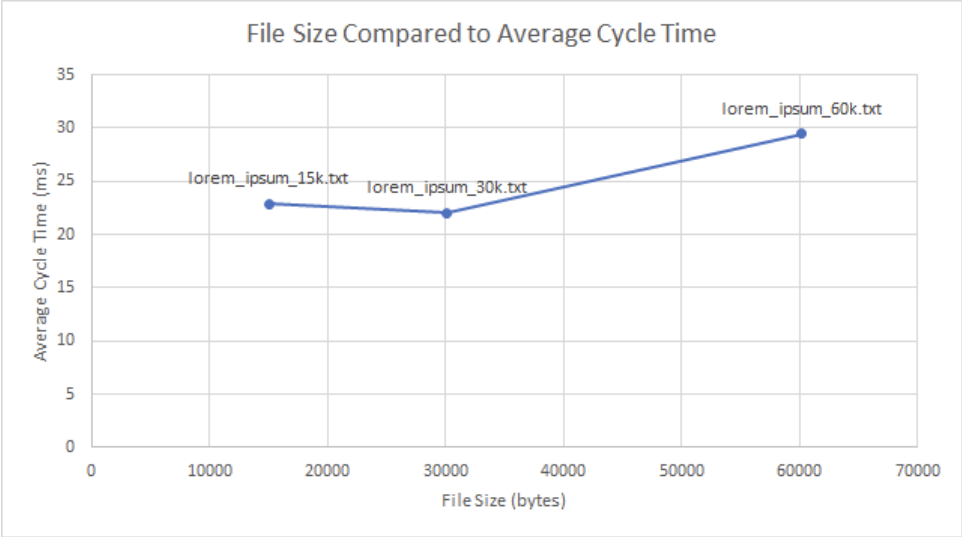
Receiving video file via TCPServer on Windows (and `ls` showing it in the directory):

```
PS C:\Users\bsolo\Downloads\project (3)\project> java -jar TCPServer.jar 192.168.0.102 192.168.0.148
ServerRouter: Connected to the router.
192.168.0.148
fileName: video.mp4
isTxt: false
Server said: payload is A-OK kthx
Client said: Bye.
PS C:\Users\bsolo\Downloads\project (3)\project> ls


    Directory: C:\Users\bsolo\Downloads\project (3)\project


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
d-----         2/13/2023   8:51 PM                build
d-----         2/13/2023  10:21 PM                src
d-----         2/13/2023  10:30 PM                test-files
-a----         2/13/2023   8:51 PM           1228 build-jars.bash
-a----         2/13/2023  10:31 PM           9466 TCPClient.jar
-a----         2/13/2023  10:31 PM           9466 TCPServer.jar
-a----         2/13/2023  10:31 PM           9472 TCPServerRouter.jar
-a----         2/13/2023  10:34 PM         295987 video.mp4
```

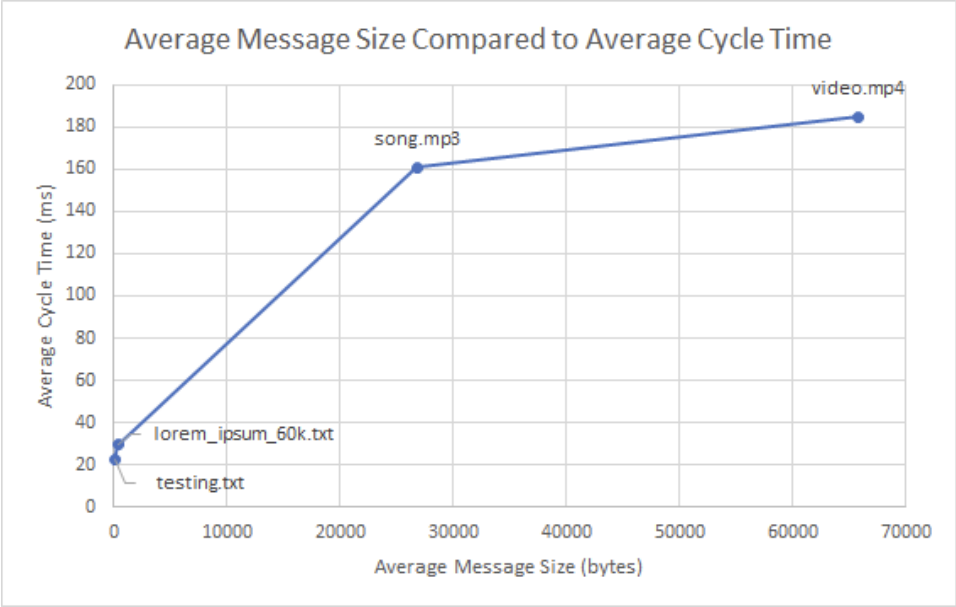
Data Tables

Filename	File Size (bytes)	Average Message Size (bytes)	Average Cycle Time (ms)	Route Table Lookup (ms)	Number of messages	Transmission mode
testing.txt	11898	10.886	22.195	0.8721610	1003	txt
lorem_ipsum_15k.txt	15045	319.543	22.856	0.9507100	48	txt
lorem_ipsum_30k.txt	30101	291.442	22.050	0.9082490	104	txt
lorem_ipsum_60k.txt	60191	310.977	29.456	0.9916125	194	txt
song.mp3	120330	26747.667	160.917	0.9582520	4	Base64 uni-directional
video.mp4	295982	65781.833	184.912	1.0454235	4	Base64 uni-directional

Throughout this data, the route table lookup time stays relatively similar. They are small enough in magnitude to be negligible compared to cycle times (the roundtrip time between sending a message from the client and receiving a response from the server). Cycle times remain at similar values until a certain average message size is exceeded, where it then greatly increases. This could be due to exceeding the maximum size for a single TCP packet; if the underlying socket has to send multiple packets for the message, this would increase latency. The average message size varies based on both the file format and the size; text files remain at a relatively low average message size, since they are sent line-by-line (one message per line), whereas audio and video files have a high average message size, since those files are all sent in one message, with video files tending to be larger than audio files. Note: the number of messages and average message size include header messages (filename, etc).



Comparing cycle time across files transmitted using the same mode and with the same line size shows a negligible difference in average cycle time.



Comparing the average cycle time to the average message size provides a statistic that is mostly unaffected by the differing operation of the txt and base64payload mode, given that the statistic corrects for the message size. It appears plainly that the average cycle time positively correlates with message size. This is likely twofold: first, in text mode, processing time on the server is correlated with the size of the message it needs to be processed. Second, in binary mode, once the message size has exceeded 65535 bytes (the max TCP packet data size), multiple TCP packets will be necessary to transfer the data; the larger the file, the more TCP packets necessary.

Conclusion

In conclusion, we successfully implemented a distributed system that utilized both wired and wireless communication to facilitate communication between the server and the client. This was achieved by compiling, installing, and running the TCP modules that were provided.

Our independent variables for this experiment were the types of files sent, while the control variables were the type of connection and the hardware used to send and receive messages. The unstable connection in Wilder Communications Building gave our data some variance over repeated tests. We concluded that there was not a correlation between cycle times and file size, however there was a correlation between average message size and cycle time. Routing table lookup time had a negligible effect on the overall roundtrip time.

Overall, this project provided valuable hands-on experience in working with distributed systems and networking protocols, and has helped give a deeper understanding of the challenges and opportunities presented by these technologies.

Enclosed testing files:

testing.txt - 12 KB - md5sum: 9181d3f89e4cd9bd745dda41775f5a95
lorem_ipsum_15k.txt - md5sum: be66e1a56e91961c1740b7d41ce764c2
lorem_ipsum_30k.txt - md5sum: 502a5f50fc11f7e6e4da4490816f2cb1
lorem_ipsum_60k.txt - md5sum: 919245f1225ca8646530e5080512d068
song.mp3 - 120 KB - md5sum: 23de70de8289ce7d352ec3870de76f1e
video.mp4 - 296 KB - md5sum: db3e1891edf7c1e7359113d2d7475a15

Appendix (AKA, User Guide)

Introduction

This assignment is intended to replicate a client-server connection. In this project, the client and server will connect to a server-router using a shared TCP port number. The ServerRouter accepts connections on a TCP port 5555 by default. It then passes the routing table and socket data as parameters to a new thread.

The SThread represents a connection between a client and server. As each thread is instantiated, it adds the client IP address and socket data to the routing table. As the thread runs, it will read an IP address sent over the client socket. This IP is the destination address the client is trying to reach. The thread waits 10 seconds so the routing table has time to fill with each client/server information. Each thread has access to the routing table which allows the thread to find an IP address matching the destination IP address. Once a match is found, the thread connects the client to the server. It will continue to hold the connection until the exit statement is called.

The purpose of this assignment is to collect statistics on transferring different files of varying size and type. The TCPClient records the wall-clock time at the start of its file transfer and once again when it receives a return statement. By taking the difference of the two times, it provides the total roundtrip time of the communication. The SThread class contains a static variable which records the average message size and average routing-table lookup time for each experimental trial. These statistics are collected and analyzed in this report to find common trends.

Execution

To start communication, the Router must be running first, and it is important that each client/server pair should be executed in relatively short spans of time (less than 10 seconds). Each program is bundled as a .jar file in the project/ folder of the ZIP file, and all following examples assume that the current working directory is "<path to extracted zip>/project". All three programs need to be run on three separate machines with unique IP addresses; they should all be on the same subnet. These addresses are stored in a 2D array which means that there is a finite number of connections.

First, run the ServerRouter program on one machine. The ServerRouter only has one argument, the port to listen on:

```
java -jar TCPServerRouter.jar <port number>
```

Once TCPServerRouter is running, we can run TCPServer and TCPClient, which should be started within 10 seconds of each other on two other machines.

For TCPServer, the server-router address, destination address, and server-router port number have been parameterized in that order E.g.:

java -jar TCPServer.jar <router IP> <client IP> <router port number>

For TCPClient, the file to send, server-router address, destination address, and port number have been parameterized in that order. The file to send is a mandatory argument that must be entered:

java -jar TCPClient.jar <file to send> <router IP> <server IP> <router port>
For example:
java -jar TCPClient.jar test-files/video.mp4 192.168.0.102 192.168.0.148 5555

For both the server and client class, the defaults for optional arguments are designed for running in a Docker-containerized environment. These would likely be invalid for a multi-computer network.

The protocol used by the server and client differs between text and binary files (checked via file extension). If the file is a standard .txt file, each line will be passed to the server class, changed to uppercase, and returned. If the file is not a .txt file, it will be encoded into a base64 payload-carrying String, transferred, decoded, and saved to a file (of the same name) on the server side. Once successfully decoded, the server will send an acknowledgement of receipt. For both protocols, communication will continue until the client sends the exit statement, "Bye." which allows the client, server, and router-server to break communication and close all sockets.

Modifying Code

The source code for the above JAR executables is provided inside the project/src/ directory in the ZIP file. Although the generated executables are cross-platform, you must use a Unix system capable of running the bash shell to build new executables after changing any code.

First, make any desired edits in projects/src/. Then, simply run the following:

```
cd projects/
./build-jars.bash
```

New JAR executables for each program (TCPServerRouter, TCPServer, and TCPClient) will be created in projects/. Note: users on a Windows system can't take advantage of this build automation script, but can still manually compile and assemble the .jar files if necessary; manifest files will need to be created for each JAR.