**Elton Sia**

**Thilina Ratnayake**

**C8005**

**A3**

**Table of Contents**

Contents

[Summary 3](#_Toc412401694)

[Design Work 3](#_Toc412401695)

[Testing and Results 4](#_Toc412401696)

[Acknowledgements 19](#_Toc412401697)

[Pseudo Code 19](#_Toc412401698)

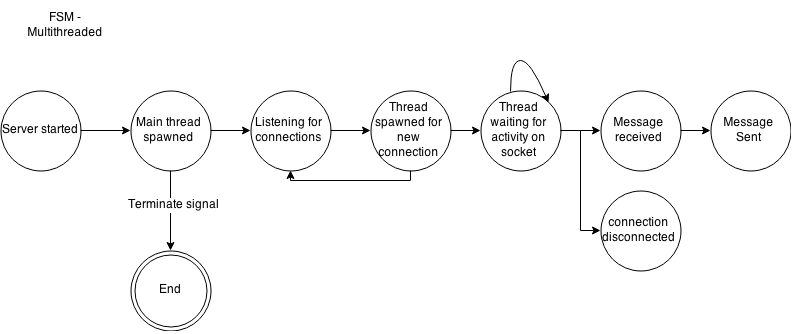
# Summary

In this assignment, we are tasked in creating 3 echo servers. There will be 3 types of servers: multithreaded server, select server, and Epoll server. All the servers do is echo back a string sent by the client. The task at hand is to compare how many connections each server can handle as well as how fast it can deliver back the data the client sent to it until it finally crashes or slows down.

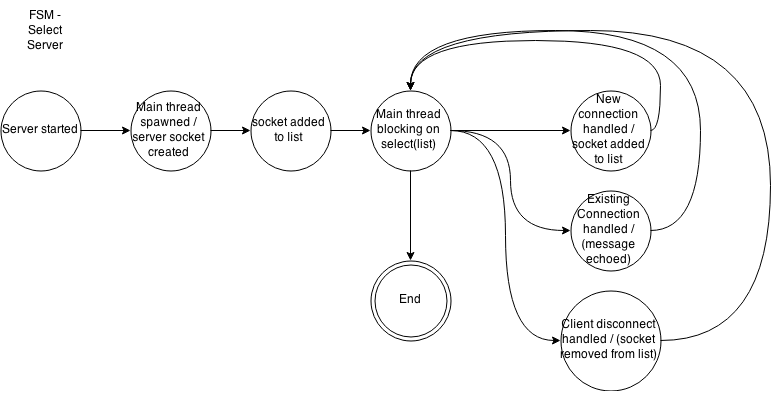
# Design Work

Finite State Machines

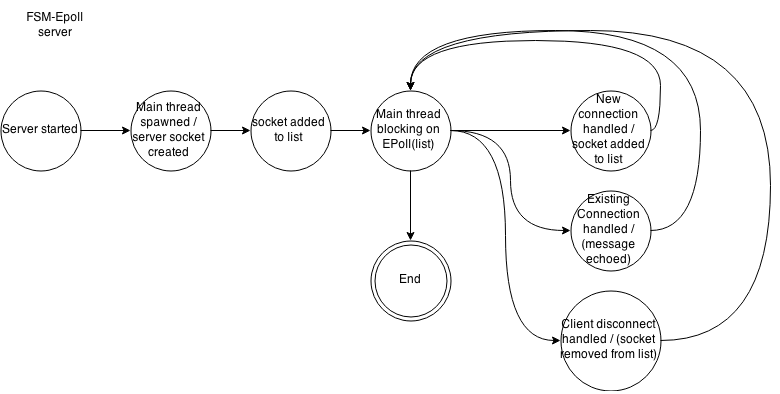
Multithreaded Server



Single threaded select server



Epoll Server



# Testing and Results

Note: These are the tests that we did for this assignment. For the full report of these please refer to the 8005\_A2\_Academic\_Paper.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| # | Name | Resources | Data size | Amount of Messages | Result |
| 1 | Stress test on multithreaded server with data size of 1024, 2 messages per client | EchoServerThreaded.rb | 1024 bytes | 2 messages per client | Pass |
| 2 | Stress test on multithreaded Select server with data size of 1024, 2 messages per client | EchoServerSelect.rb | 1024 bytes | 2 messages per client | Pass |
| 3 | Stress test on Epoll server with data size of 1024, 2 messages per client | EchoServerEpoll.rb | 1024 bytes | 2 messages per client | Pass |
| 4 | Stress test on multithreaded server with data size of 2048, 2 messages per client | EchoServerThreaded.rb | 2048 bytes | 2 messages per client | Pass |
| 5 | Stress test on multithreaded Select server with data size of 2048, 2 messages per client | EchoServerSelect.rb | 2048 bytes | 2 messages per client | Pass |
| 6 | Stress test on Epoll server with data size of 2048, 2 messages per client | EchoServerEpoll.rb | 2048 bytes | 2 messages per client | Pass |
| 7 | Stress test on multithreaded server with data size of 256, 10 messages per client | EchoServerThreaded.rb | 256 bytes | 10 messages per client | Pass |
| 8 | Stress test on single threaded select server with data size of 256, 10 messages per client | EchoServerSelectSingle.rb | 256 bytes | 10 messages per client | Pass |
| 9 | Stress test on epoll server with data size of 256, 10 messages per client | EchoServerEpoll.rb | 256 bytes | 10 messages per client | Pass |
| 10 | Stress test on multithreaded server with data size of 512, 10 messages per client | EchoServerThreaded.rb | 512 bytes | 10 messages per client | Pass |
| 11 | Stress test on single threaded select server with data size of 512, 10 messages per client | EchoServerSelectSingle.rb | 512 bytes | 10 messages per client | Pass |
| 12 | Stress test on epoll server with data size of 512, 10 messages per client | EchoServerEpoll.rb | 512 bytes | 10 messages per client | Pass |

# Acknowledgements

We gratefully acknowledge Andrew Burian, fellow set-mate at BCIT BTech for providing us a distributed client network to facilitate testing.

# Pseudo Code

**Client.rb**

Open a file to log what we want

User input for how many client we are connecting

User input for how many messages per client we will be sending

User input for which port we are connecting to

While $i < $client

{

Increment $i

Create threads

{

Open a socket to connect to

Start timer

Create a loop depending on the user input in how many times the message will be sent

Send message

Read the message from server

End the loop

End the timer

Calculate the time from start to end

Put the threads to sleep so that it does not disconnect

}

}

Join the threads

**EchoServerEpoll.rb**

Open a file to log what we want

Module EchoServer

{

Function for connecting

{

Grab the port and IP address of the client that is connecting

Print the IP and the port the client is using to connect to the server

Create a counter

Print how many clients are currently connected

}

Function for sending the data back

{

Send the data back

}

Function for disconnecting

{

Print that client disconnected

Decrement the counter

}

}

EM.epoll (tell the event machine to use epoll if the machine has epoll capabilities)

Set the descriptor table size to 100,000 so that we can have more than 100,000 connections (if possible)

EM.run (run the event machine)

{

Start the event machine

}

**EchoServerThreaded.rb**

Open a file so that we can log what we want in there

Create server on port specified

While we are accepting connections

{

Create a new thread for each accepted connection

{

Grab the port and host IP address of the client

Show the host IP address and the port the client is using to connect to the server

Create a counter and print it to see how many clients are connected so far

Loop do

{

Set the data that will be read once the client sends to this server

Write the data back

Flush the connection

}

Rescue EOFError (if the client hits ctrl + c or if the client closes)

{

Close the connection

Decrement the counter and show how many clients left are connected

}

}

**EchoServerSelect.rb**

Open a file so that we can log what we want

Function for connecting

{

Grab the host IP address and the port

Show the host IP address and the port that the client used to connect to the server

Create a counter and increment it as long as clients are connecting

Show the amount of clients that are currently connected

}

Function for disconnecting

{

Grab the host IP address and the port

Show the host IP address and the port that the client used to connect to the server

Decrement the counter when clients start disconnecting

Show the amount of clients that are currently connected

}

Create the socket

Create an array

While true

{

Choose select IO on the sockets

For each connection

{

If there is someone trying to connect

{

Accept the client

Push the client into the array

}

Else

{

If socket receives an eof (ctrl + c from client side)

{

Delete clients from the array

}

Else

{

Read the data

Put the data into the socket

Flush the socket

}

}

}