CS3S661 Assignment 2 “TCP Simulator”  
Student Name: Connor Haines

SID: 17025931

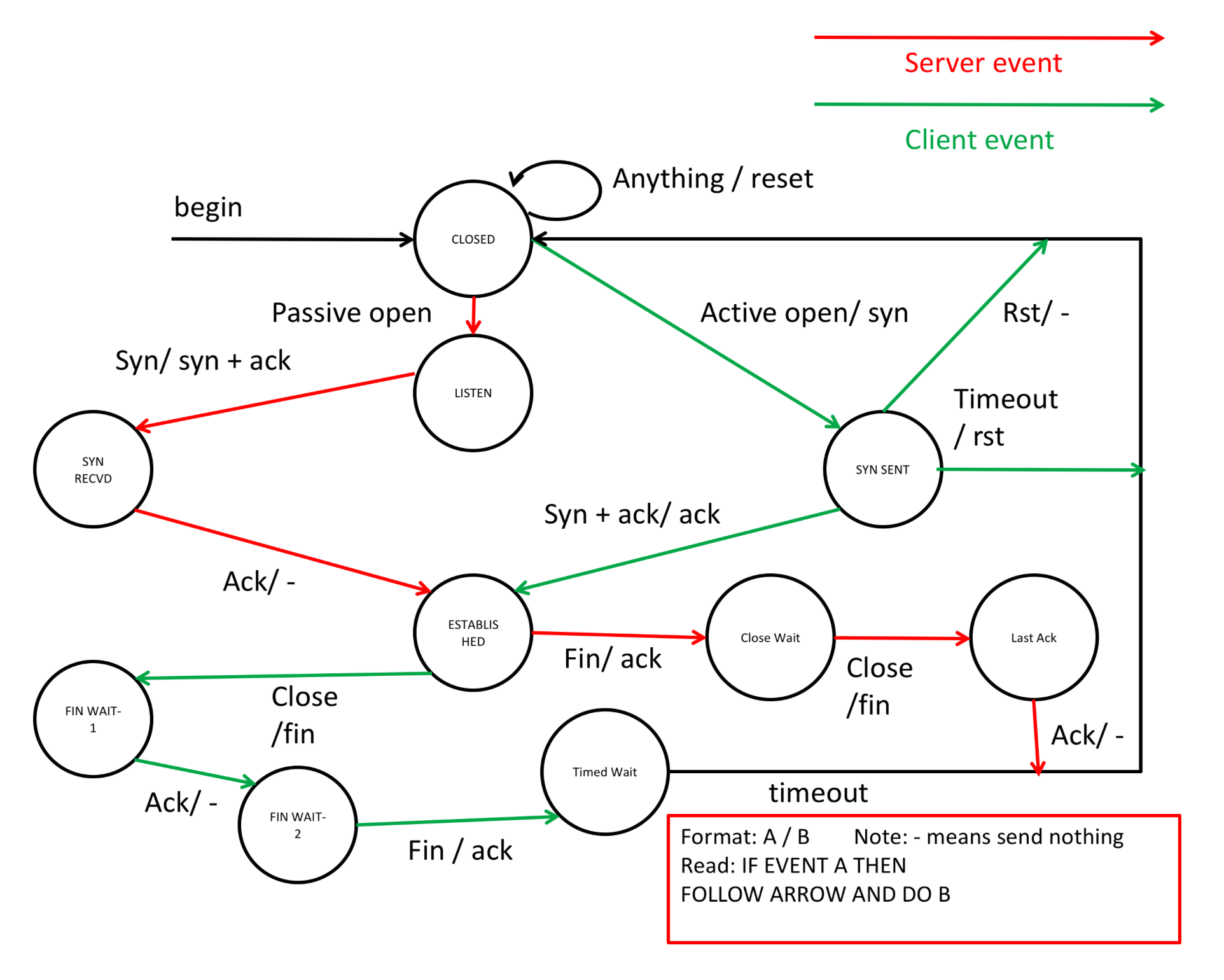
SCHOOL OF COMPUTING AND MATHEMATICS

*An assessment submitted for the degree of   
Bachelor of Science (BSc) in Computer Science.*

April 2020 – University of South Wales

# Assignment Overview

For this assignment the aim was clear, I was challenged to develop and create a TCP protocol simulator for Electronic Communications Ltd. Luckily, all network protocols are examples of state machines and therefore it would be best to use the state pattern for this simulation.  
  
Below is a state-diagram that I was supposed to follow:



# Implementation

## Closed (Server and Client)

A screenshot of a cell phone screen with text

Description automatically generated

The program starts and ends and the closed state for both sides of this program. The only thing that this state does on the server side is close the socket and set the connection\_address to 0. Then, due to the passive open, it automatically transitions into the Listen state. The server always needs to be run first.

A screenshot of a cell phone

Description automatically generated

For the closed state on the clients side, the same can be said. Except, instead of automatically transitioning to synSent it will wait for some user input to decide of the program will end or if it will search for another connection.

## Listen (Server)

This part of the server-side listens for a potential connection and will only move forward if the client is also looking for a connection. It will try to assign a socket to an address using socket.bind and then listen on that socket and wait for a connection. When connected it will wait for a “SYN” flag, if this is not received it will return an error. If it is, it will transition into the Syn\_Recvd state. It finds out what the client sent by using the part of the code outlined in red below:

A screenshot of a cell phone

Description automatically generated

## Syn\_Recvd (Server)

For the Syn\_Recvd state, all this state does is send back a “SYN+ACK” flag and receives another flag back. This is the first time the server sends something and it does so by using: self.Current.Context.connection.send(“your flag”.encode())

Next, it will check that it received the correct flag from the client with the same IF statement as before and if it is “ACK” it will transition to the established state.

A screenshot of a cell phone

Description automatically generated

## Established (Server and Client)

A screenshot of a cell phone

Description automatically generated

For this part of the server-side program, the client and server are connected and are able to send messages via byte streams. All that happens is that within a while loop, the server will send a user input and wait for a response. It will check every response for a flag and if it catches the “FIN” flag, it will transition to the next state.

A screenshot of a cell phone

Description automatically generated

On the side of the client program, much is the same, as it waits for a message from the server and then is able to send a message back. It has its own inbuilt check to see if the user input equals the “FIN” flag, if it does it will transition itself into the next state.

## Close\_Wait and Last\_Ack (Server)

A screenshot of text

Description automatically generatedThese two states are almost exactly the same and are very similar to the Syn\_Recvd state explained above. These two states send flags to the client and receive and decode the flags sent back. If they receive the correct flags they move on to the next state. For Close\_Wait it would transition to Last\_Ack. For Last\_Ack it would transition back to the Close state and then automatically go back into listening

## synSent (Client)

For the synSent state, all this state does is send back a “SYN” flag and receives another flag back. This is the first time the client sends something and it does so by using: self.Current.Context.socket.send(“your flag”.encode())

Next, it will check that it received the correct flag from the server with the same IF statement as before and if it is “SYN+ACK” it will transition to the established state.

**A screenshot of a cell phone

Description automatically generated**

## finWait1 and finWait2 (Client)

A screenshot of text

Description automatically generated

These two states are almost identical. They both wait to receive a flag from the server to be able to transition into the next state. The only real difference is the fact that the finWait1 state sends a flag to the server first and THEN waits for the response. Whereas, the finWait2 state doesn’t send anything

## timedWait (Client)

This state is extremely simple, all it does is send an “ACK” flag to the server to acknowledge the closing and then waits 2 seconds and transitions the state to Closed.

A screenshot of a cell phone

Description automatically generated

# Overall

Overall, I think both the Client and Server sides of the program are fit for purpose and come with almost all of the basic features required. One thing I would add if I had the chance would have been the sequence numbers to allow the user to see where in the sequence or order of things the program was.   
  
The program follows the State Diagram given very closely and even implements a user input for the active open on the client’s side. This was done under the assumption that the user may want to reconnect after disconnecting or they may not and therefore the option was given.   
  
The \_\_main\_\_ section for both the server and the client were remarkably simple and similar and looked like this:

## Server

if \_\_name\_\_ == "\_\_main\_\_":  
 server = TCPserver()  
 server.Listen()

## Client

if \_\_name\_\_ == "\_\_main\_\_":  
 client = TCPclient()  
 client.synSent()