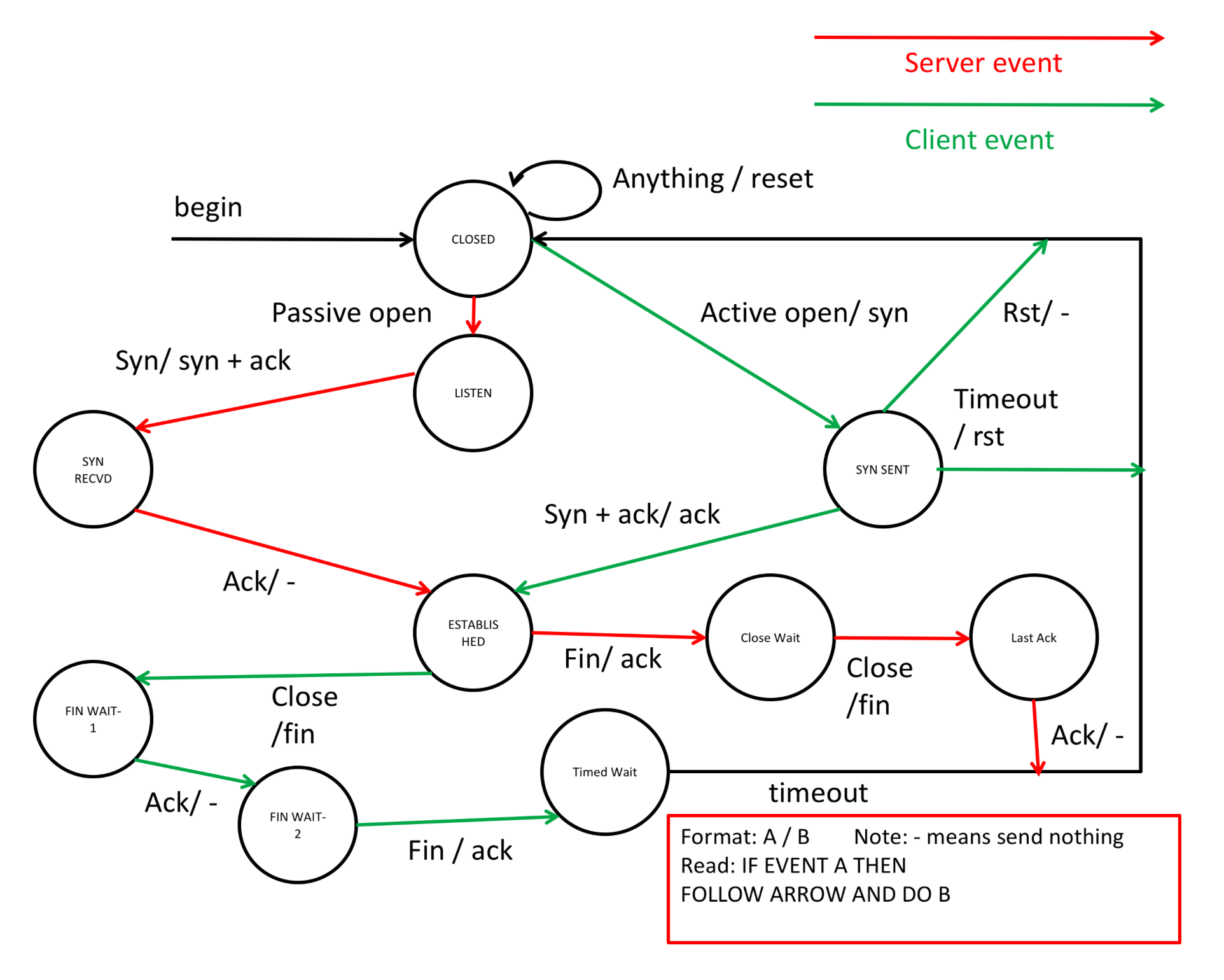
# Computer Networks Assignment Report

## Introduction

I am a programmer working for Electronic Communications LTD and I have been asked to develop a TCP protocol simulator. I have to follow the guidelines of the State Diagram provided below: The

There are two separate python files. One represents the Server script and the other the Client script.

Both of these scripts will run concurrently and communicate using the python Socket library. Both of the scripts will use a State.py file which holds the data structure for the available states.

## State.py

The State.py file contains the following code:

class State:

    CurrentContext = None

    def \_\_init\_\_(self, Context):

        self.CurrentContext = Context

    def trigger(self):

        return True

class StateContext:

    state = None

    CurrentState = None

    availableStates = {}

    def setState(self, newstate):

        try:

            self.CurrentState = self.availableStates[newstate]

            self.state = newstate

            self.CurrentState.trigger()

            return True

        except KeyError: #incorrect state key specified

            return False

    def getStateIndex(self):

        return self.state

This was provided by the company just like the state diagram was.

## 17145545\_Server.py

The Server.py hold all the server side of the state diagram.

The server imports the State.py file and also the socket and time libraries to use.

There are 8 classes in total one being for the transitioning between states and the other the constructor class. These 8 classes are:

1. Transition
2. Closed
3. Listen
4. Syn\_Recvd
5. Established
6. Close\_Wait
7. Last\_Ack
8. TCPServer

### Transition class

class Transition:

    def passive\_open(self):

        print ("Error! Cannot Transition to passive open!")

        return False

    def syn(self):

        print ("Error! Cannot Transition to syn!")

        return False

    def ack(self):

        print ("Error! Cannot Transition to ack!")

        return False

    def close(self):

        print ("Error! Cannot Transition to close!")

        return False

    def fin(self):

        print ("Error! Cannot Transition to fin!")

        return False

The above class is the Transition class of which the server uses to transition through states.

### Closed Class

class Closed(State, Transition):

    def \_\_init\_\_(self, Context):

        State.\_\_init\_\_(self, Context)

    def passive\_open(self):

        print("Transitioning to Listen")

        self.CurrentContext.setState("LISTEN")

    def trigger(self):

        try:

            self.CurrentContext.socket.close()

            self.connection\_address = 0

            print("Closing the connection.")

            return True

        except:

            return False

The Closed state is the initial state the server starts on. This state returns back to itself at any event. If the server is set to passive\_open it sets its state to LISTEN.

The trigger for this state if after the socket is closed and the connection address us set to 0 and if this works it returns True. Anything else it returns False.

### Listen Class

class Listen(State, Transition):

    def \_\_init\_\_(self, Context):

        State.\_\_init\_\_(self, Context)

    def syn(self):

        self.CurrentContext.connection.send("SYN\_ACK".encode())

        message = self.CurrentContext.connection.recv(1024)

        sleep(self.CurrentContext.sleep\_time)

        print("Sending the SYN\_ACK.")

        if message.decode() == "SYN\_ACK":

            print("SYN\_ACK recieved.")

            self.CurrentContext.setState("SYN\_RECVD")

            print("Transitioning to Syn\_recvd")

        else:

            return False

    def trigger(self):

        if self.CurrentContext.listen() is True:

            print("SYN sent to " + str(self.CurrentContext.connection\_address))

            self.CurrentContext.syn()

        else:

            return False

The LISTEN state triggers the syn() function in the class which firstly send the message “SYN\_ACK” across the connection. Then the server is set to a sleep time of 2 bwfore letting the user know that the SYN\_ACK has been sent . Then the message is decoded and checked to see if correct message has been received lets the user know that it was received then proceeds to change the state to “SYN\_RECVD”

### SYN\_RECVD Class

class Syn\_Recvd(State, Transition):

    def \_\_init\_\_(self, Context):

        State.\_\_init\_\_(self, Context)

    def ack(self):

        message = self.CurrentContext.connection.recv(1024)

        sleep(self.CurrentContext.sleep\_time)

        if message.decode() == "SYN\_ACK":

            print (message + " Recieved.")

            self.CurrentContext.setState("ESTABLISHED")

            print("Transitioning to ESTABLISHED")

    def trigger(self):

        return self.CurrentContext.ack()

This is used to set the state to established. It set the state to established after the message is decoded and informed the user to what was received. Then the state is changed to established.

### Established Class

class Established(State, Transition):

    def \_\_init\_\_(self, Context):

        State.\_\_init\_\_(self, Context)

    def fin(self):

        self.CurrentContext.connection.send("ACK".encode())

        message = self.CurrentContext.connection.recv(1024)

        sleep(self.CurrentContext.sleep\_time)

        if message.decode() == "ACK":

            print("Sending the ACK.")

            self.CurrentContext.setState("CLOSE\_WAIT")

            print("Transitioning to Close\_wait.")

    def trigger(self):

        return self.CurrentContext.fin()

The event for Established state is send ack if the event is fin. So, the fin() function sends the encoded message over the socket and then the function checks if the received message is “ACK” then proceeds to set the state to Close wait.

## Close\_Wait Class

class Close\_Wait(State, Transition):

    def \_\_init\_\_(self, Context):

        State.\_\_init\_\_(self, Context)

    def close(self):

        self.CurrentContext.socket.send("FIN".encode())

        message = self.CurrentContext.connection.recv(1024)

        sleep(self.CurrentContext.sleep\_time)

        if message.decode() == "FIN":

            self.CurrentContext.setState("LAST\_ACK")

            print("Transitioning to Last\_Ack")

    def trigger(self):

        return self.CurrentContext.close()

This state is used to send the “FIN” message on the socket connection. And checked to see if the received message decoded is correct which then sets the state to LAST\_ACK.

### Last\_Ack Class

class Last\_Ack(State, Transition):

    def \_\_init\_\_(self, Context):

        State.\_\_init\_\_(self, Context)

    def ack(self):

        self.CurrentContext.setState("CLOSED")

        print("Transitioning to Closed.")

    def trigger(self):

        return self.CurrentContext.ack()

This class is used just to timeout the server and return to the Closed state.

### TCPServer Class

This is the constructor class where everything comes together. This holds the host IP address and the port, it also initiates the states and functions.

class TCPServer(StateContext, Transition):

    def \_\_init\_\_(self):

        self.sleep\_time = 2

        self.host = "127.0.0.1"

        self.port = 1024

        self.connection\_address = 0

        self.socket = None

        self.availableStates["CLOSED"] = Closed(self)

        self.availableStates["LISTEN"] = Listen(self)

        self.availableStates["SYN\_RECVD"] = Syn\_Recvd(self)

        self.availableStates["ESTABLISHED"] = Established(self)

        self.availableStates["CLOSE\_WAIT"] = Close\_Wait(self)

        self.availableStates["LAST\_ACK"] = Last\_Ack(self)

        print("Transitioning to closed!")

        self.setState("CLOSED")

    def passive\_open(self):

        return self.CurrentState.passive\_open()

    def syn(self):

        return self.CurrentState.syn()

    def ack(self):

        return self.CurrentState.ack()

    def close(self):

        return self.CurrentState.close()

    def fin(self):

        return self.CurrentState.fin()

    def listen(self):

        self.socket = socket(AF\_INET, SOCK\_STREAM)

        try:

            self.socket.bind((self.host, self.port))

            self.socket.listen(1)

            self.connection, self.connection\_address = self.socket.accept()

            print ("Connection Address: " + self.connection\_address)

            return True

        except Exception as err:

            print(err)

            exit()

## 17145545\_Client.py

The Client.py file holds all of the client side of the state diagram.

The server imports the State.py file and also the socket and time libraries to use.

This script contains 8 classes like the server file.

1. Transition
2. Closed
3. Syn\_Sent
4. Established
5. Fin\_Wait1
6. Fin\_Wait2
7. Timed\_Wait
8. TCPClient

### Transition

class Transition:

    def active\_open(self):

        print ("Error! Cannot Transition to active open!")

        return False

    def syn(self):

        print ("Error! Cannot Transition to syn!")

        return False

    def ack(self):

        print ("Error! Cannot Transition to ack!")

        return False

    def rst(self):

        print ("Error! Cannot Transition to rst!")

        return False

    def syn\_ack(self):

        print ("Error! Cannot Transition to syn ack!")

        return False

    def close(self):

        print ("Error! Cannot Transition to close!")

        return False

    def timeout(self):

        print ("Error! Cannot Transition to timeout!")

        return False

The above class is the Transition class of which the server uses to transition through states.

### Closed Class

class Closed(State, Transition):

    def \_\_init\_\_(self, Context):

        State.\_\_init\_\_(self, Context)

    def active\_open(self):

        print("Contacting the server...")

        self.CurrentContext.make\_connection()

        self.CurrentContext.socket.send("SYN".encode())

        message = self.CurrentContext.socket.recv(1024)

        sleep(self.CurrentContext.sleep\_time)

        if message.decode() == "SYN":

            self.CurrentContext.setState("SYN\_SENT")

            print("Transitioning to Syn\_Sent")

    def trigger(self):

        try:

            self.CurrentContext.socket.close()

            self.connection\_address = 0

            print("Closing the connection.")

            return True

        except:

            return False

In this state the active\_open function is used to connect with the server that is running. Once a connection is made the client script then sends the message “SYN” encoded using the socket. After checking if the decoded message is correct it then sets the current state to SYN\_SENT.

### Class Syn\_Sent

class Syn\_Sent(State, Transition):

    def \_\_init\_\_(self, Context):

        State.\_\_init\_\_(self, Context)

    def rst(self):

        self.CurrentContext.setState("CLOSED")

        print("Transitioting to closed!")

        return True

    def timeout(self):

        self.CurrentContext.socket.close()

        self.connection\_address = 0

        print("Closing the connection.")

        self.CurrentContext.setState("CLOSED")

        print("Transitioting to closed!")

        return True

    def syn\_ack(self):

        self.CurrentContext.socket.send("ACK".encode())

        message = self.CurrentContext.socket.recv(1024)

        sleep(self.CurrentContext.sleep\_time)

        if message.decode() == "ACK":

            print("Sending the ACK.")

            self.CurrentContext.setState("ESTABLISHED")

            print("Transitioting to Established.")

        else:

            return self.CurrentContext.rst()

        return True

    def trigger(self):

        return self.CurrentContext.syn\_ack()

The Syn\_Sent class has multiple options such as Rst, Timeout and Syn + Ack.

The rst() is used to set the state back to closed.

The timeout() is used to close the connection and set the state to closed.

The syn\_Ack() is used to send the message “ACK” and if the message received is correct then it sets the state to Established.

### Established Class

class Established(State, Transition):

    def \_\_init\_\_(self, Context):

        State.\_\_init\_\_(self, Context)

    def close(self):

        self.CurrentContext.socket.send("FIN".encode())

        message = self.CurrentContext.socket.recv(1024)

        sleep(self.CurrentContext.sleep\_time)

        if message.decode() == "FIN":

            print("Sending FIN.")

            self.CurrentContext.setState("FIN\_WAIT1")

            print("Transitioning to Fin\_Wait1.")

        else:

            return False

        return True

    def trigger(self):

        return self.CurrentContext.close()

This class is used to send the “FIN” message across the socket. Once the message is checked to be correct it then sets the state of the client to FIN\_WAIT1.

### Fin\_Wait1 Class

class Fin\_Wait1(State, Transition):

    def \_\_init\_\_(self, Context):

        State.\_\_init\_\_(self, Context)

    def ack(self):

        self.CurrentContext.setState("FIN\_WAIT2")

        print("Transitioning to Fin\_Wait2.")

    def trigger(self):

        self.ack()

This class is just sued to transfer to the state Fin\_Wait2.

### Fin\_Wait2 Class

class Fin\_Wait2(State, Transition):

    def \_\_init\_\_(self, Context):

        State.\_\_init\_\_(self, Context)

    def fin(self):

        self.CurrentContext.socket.send("ACK".encode())

        message = self.CurrentContext.socket.recv(1024)

        sleep(self.CurrentContext.sleep\_time)

        if message.decode() == "ACK":

            print("Sending the ACK.")

            self.CurrentContext.setState("TIMED\_WAIT")

            print("Transitioning to Timed\_Wait.")

        else:

            return False

        return True

    def trigger(self):

        self.fin()

In this class when the fin() function is called at the trigger is sends the mssage “ACK” through the socket and once this is checked the programme sets its state to TIMED\_WAIT.

### Timed\_Wait Class

class Timed\_Wait(State, Transition):

    def \_\_init\_\_(self, Context):

        State.\_\_init\_\_(self, Context)

    def timeout(self):

        self.CurrentContext.setState("CLOSED")

        print("Transitioning to Closed as a timeout has occured.")

    def trigger(self):

        self.timeout()

In the Timed\_Wait class the timeout function just sets the state to closed to terminate the connection.

### TCPClient Class

This is the constructor class for the client where all the states are constructed with the available functions, so the programme is directed in the correct direction. It is also where the make\_connection function is created.

class TCPClient(StateContext, Transition):

    def \_\_init\_\_(self):

        self.sleep\_time = 2

        self.host = "127.0.0.1"

        self.port = 1024

        self.connection\_address = 0

        self.socket = None

        self.availableStates["CLOSED"] = Closed(self)

        self.availableStates["ESTABLISHED"] = Established(self)

        self.availableStates["SYN\_SENT"] = Syn\_Sent(self)

        self.availableStates["FIN\_WAIT1"] = Fin\_Wait1(self)

        self.availableStates["FIN\_WAIT2"] = Fin\_Wait2(self)

        self.availableStates["TIMED\_WAIT"] = Timed\_Wait(self)

        print("Transitioning to closed!")

        self.setState("CLOSED")

    def active\_open(self):

        return self.CurrentState.active\_open()

    def rst(self):

        return self.CurrentState.rst()

    def timeout(self):

        return self.CurrentState.timeout()

    def syn\_ack(self):

        return self.CurrentState.syn\_ack()

    def close(self):

        return self.CurrentState.close()

    def ack(self):

        return self.CurrentState.ack()

    def syn(self):

        return self.CurrentState.syn()

    def make\_connection(self):

        '''initiates an outbound connection'''

        self.socket = socket(AF\_INET, SOCK\_STREAM)

        try:

            self.socket.connect((self.host, self.port))

            self.connection\_address = self.host

        except Exception as err:

            print(err)

            exit()