# FINAL PROJECT:

# Text Stream Protocol

**CE 4390.001**

**Prof. Nhut Nguyen**

**AUTHORS:**

Logan Dennison

Trevor Duncan

Patrick Le

# Project Description

# Project Description

The main goal of this project was to design an application protocol implemented in an application for media consumption. By implementing a controller, renderer and server all three communicate with each other to provide media consumption service to users. The controller requests a list of media files from the server. Then the controller can request for the renderer to render the chosen file. The renderer will then send a request to the server so the server can stream the chosen media file to renderer for rendering. During the streaming session, the controller can request the renderer to pause/resume the streaming. By using mininet we emulate a network implementing those three entities and run our python scripts.

# Challenges

Our biggest challenge was learning an entirely new programming language. All of our coursework preceding this course only taught us using C and C++, we were introduced to basic Object-oriented programming concepts and with that we had to scour the internet and related documentation and all necessary libraries. Another challenge we encountered was utilizing Mininet for our specific project. Mininet has generic documentation and isn’t clear that it isn’t very friendly with Python 3+, and isn’t very descriptive in modifying its topography, but we turned to the internet and discovered a fair set of instructional videos and documentation. The final hurdle we encountered beyond normal debugging was blocking and non-blocking functions. This is only important because our three programs require non-blocking loop functionality at different time, and it took quite a bit of time to find all of the fnctions causing it and circumventing them.

# Lessons Learned

We learned quite lot from this experience. We developed stronger object-oriented programming ability and became rather competent in Python. We learned the ins and out of UDP Sockets and what kind of issues can arise from sending too many messages on one port at a time. Honestly, I wasn’t even aware how important blocking functions were, but after this assignment the knowledge will help us with future debugging and programming missteps. We also learned the constraints on Mininet regarding what version of Python is compatible and how to create custom topography for our custom network set up.

# Techniques Used

* **If/Else switch statements**

Since the Python language doesn’t have switch statements, we implemented the same logic via cascading If/Else statements to parse through the multiple operating modes for each class.

* **UDP Socket Programming**

We decided upon UDP Sockets as there are only 3 hosts communicating and each of the hosts are based of looping structures that busy wait until a valid sequence of packets are received

* **Object-oriented Programming**

We used classes and inheritance native to object-oriented programming because Python is an object-oriented language and it was a natural and cohesive choice

* **File I/O**

In order to retrieve and parse through the selected files and its containing directory we utilized Pythons native file I/O library

# Member Contribution

In the design phase we all talked through what we wanted in a protocol and began to build our format. And we brainstormed the pseudocode collaboratively on a white board for each of the three programs and decided on what protocol and functions we wanted to include. After, we divided up what portions of the project that required research amongst ourselves. I researched custom mininet topography, Trevor researched python syntax, and Patrick worked of File I/O and UDP sockets. We then divided up the translation of our pseudocode into actual code between us, and I developed the topography as well as developed the UDP correspondence in multiple iterative test programs building on functionality. After this was done my computer was the only one to successfully utilize mininet with file access because I ran Mininet on Mint vs the Mininet-VM, so we debugged it on my computer together as a group.

# Appendix

**Video:**

[Click to Launch](https://youtu.be/SyDlzick9Ok) (https://youtu.be/SyDlzick9Ok)

**Program Instructions:**

**Set Up:**

1. Ensure that the accurate **absolute filepath** for the “**Text**” folder, zipped with code, is updated in attributes of the **server.py** file as it is hardcoded in.
2. Launch Mininet using the following custom topography command with the appropriate filepath of the accompanying **customTopo.py** file:

**sudo mn --custom [customTopo.py filepath] --topo=mytopo**

1. Then within Mininet launch the three command lines for each host:

**xterm h1 h2 h3**

1. Launch **server.py** in the **h1 command window** first by navigating to the appropriate folder and using the following command:

**python server.py**

1. Launch **renderer.py** in the **h2 command window** first by navigating to the appropriate folder and using the following command:

**python renderer.py**

1. Launch **controller.py** in the **h3 command window** first by navigating to the appropriate folder and using the following command:

**python controller.py**

**Operation:**

1. Upon launching the controller you will be prompted to select a “.txt” file from a list provided files. (whatever “.txt” files that are present in the “Text” folder will be displayed) Each file will be preceded by a number wrapped in square brackets. (E.g. [1] sample.txt) to select a file simply input the corresponding number + Enter
2. The file should start streaming and displayed on the renderer’s command window. The controller window will now provide four streaming control options accompanied by selection numbers as before (0 – Play, 1 – Pause, 2 – Restart, 3 – Disconnect). Upon selection the stream displayed on the renderer will react accordingly.
3. Once the stream has been completed the text “**The End**” will be displayed on the renderer’s command line. At this point you have the option of either restarting the stream or disconnecting.
4. If Disconnect is chosen all three programs will close and must be relaunched to start a new stream.

**Code:**

**Controller:**

#4390 Semester Project

#Patrick Le, Trevor, Duncan, Logan Dennison

#Controller Code

import socket

import sys

import os

import errno

import time

class Controller:

LOCAL\_IP = "10.0.0.3"

R\_IP = "10.0.0.2"

S\_IP = "10.0.0.1"

C\_SEND = (LOCAL\_IP,2500)

C\_RECV = (LOCAL\_IP,5000)

R = (R\_IP,5000)

S = (S\_IP,5000)

sendSock = socket.socket(socket.AF\_INET,socket.SOCK\_DGRAM)

recvSock = socket.socket(socket.AF\_INET,socket.SOCK\_DGRAM)

COMMANDPROMPT = ("Enter the number corresponding to the desired command:\n[0] Play\n[1] Pause\n[2] Restart\n[3] Disconnect\n")

FILEPROMPT = ("Enter the number corresponding to the desired file:")

type = ""

code = ""

inPayload = ["",""]

outPayload = ""

disconnect = False

streaming = False

def \_\_init\_\_(self):

     self.sendSock.bind(self.C\_SEND)

     self.recvSock.bind(self.C\_RECV)

     self.disconnect = False

def isConnected(self):

     if self.disconnect:

         return False

     else:

         return True

def listenCommand(self):

     #print("Controller: Listening")

     packet = ["","",""]

     try:

         rawPacket, rawIP = self.recvSock.recvfrom(2048)

     except:

         raise

     if rawPacket:

         #print "Controller: Received Message"

         packet = rawPacket.split(" ",2)

         self.type = packet[0] #first is type

         self.code = packet[1] #second is type

         if type == "STREAM" and code == "NULL":

             self.inPayload = packet[2].split(" ",1) #will be split if type and code are stream and NULL

         else:

             self.inPayload[0] = packet[2]

         #print "Controller: MESSAGE: " + self.type + " " + self.code + " " + str(self.inPayload)

def sendCommand(self, HOST, STATUS):

     #print "Controller: Sending"

     localPacket = STATUS + self.outPayload

     if HOST == "S":

         self.sendSock.sendto(localPacket,self.S)

     if HOST == "R":

         self.sendSock.sendto(localPacket,self.R)

     #print "Controller: Message Sent to " + HOST

     #print "Controller: MESSAGE: " + localPacket

def play(self):

     print "Controller: PLAY"

     self.sendCommand("S", "PLAY NULL ")

def pause(self):

     print "Controller: PAUSE"

     self.sendCommand("S", "PAUSE NULL ")

def restart(self):

     print "Controller: RESTART"

     self.outPayload = str(0)

     self.sendCommand("S", "PLAY LINE ")

     self.sendCommand("R", "PLAY LINE ")

def disconnectF(self):

     print "Controller: DISCONNECT"

     self.sendCommand("S", "DISCONNECT NULL ")

     print "\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n"

     print "Controller: Disconnected"

controller = Controller()

print "\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n"

while controller.isConnected():

#create initial frame

controller.sendCommand("S", "CONNECT NULL ")

prompt = ""

controller.recvSock.setblocking(1)

controller.listenCommand()

controller.recvSock.setblocking(0)

files = controller.inPayload[0].split(" ",len(controller.inPayload[0])-1)

print controller.FILEPROMPT

index = 0

for f in files:

     prompt += "[" + str(index) + "]" + " " + f + "\n"

     index += 1

index = int(raw\_input(prompt))

print "\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n"

controller.outPayload = files[index]

controller.sendCommand("R", "STREAM FILE ")

#populate list frame

#wait for selection of file by user

while controller.isConnected():

     try:

         controller.listenCommand()

     except socket.error:

         continue #include try and catch in a loop

     else:

         if controller.code == "FNF":

             break

     controller.recvSock.setblocking(1)

     while controller.code != "READY":

         controller.listenCommand()

     controller.sendCommand("S", "PLAY NULL ")

     controller.recvSock.setblocking(0)

     while controller.isConnected():

         buttonPress = int(raw\_input(controller.COMMANDPROMPT))

         controller.recvSock.setblocking(0)

         print "\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n"

         if buttonPress == 0:

             controller.play()

         elif buttonPress == 1:

             controller.pause()

         elif buttonPress == 2:

             controller.restart()

         elif buttonPress == 3:

             controller.disconnectF()

             controller.disconnect = True

         if(controller.code == "END"):

             controller.disconnect = True

     controller.sendCommand("S", "DISCONNECT NULL ")

     controller.sendCommand("R", "DISCONNECT NULL ")

     controller.sendCommand("S", "DISCONNECT NULL ")

     controller.sendCommand("R", "DISCONNECT NULL ")

     controller.sendCommand("S", "DISCONNECT NULL ")

     controller.sendCommand("R", "DISCONNECT NULL ")

     controller.sendCommand("S", "DISCONNECT NULL ")

     controller.sendCommand("R", "DISCONNECT NULL ")

**Renderer:**

#4390 Semester Project

#Patrick Le, Trevor, Duncan, Logan Dennison

#Renderer Code

import socket

import sys

import os

class Renderer:

C\_IP = "10.0.0.3"

LOCAL\_IP = "10.0.0.2"

S\_IP = "10.0.0.1"

C = (C\_IP,5000) #tuple object

R = (LOCAL\_IP,5000)

S = (S\_IP,5000)

rendererSock = socket.socket(socket.AF\_INET,socket.SOCK\_DGRAM) #internet, udp

outPayload = ""

currentLine = 0

inPayload = []

type = ""

code = ""

switchVar = -1

lineCount = 0

totalLines = 0

disconnect = False

def \_\_init\_\_(self):

     self.rendererSock.bind(self.R)

     self.disconnect = False

def isConnected(self):

     if self.disconnect:

         return False

     else:

         return True

def listenCommand(self):

     #print "Renderer: Listening"

     try:

         rawPacket, rawIP = self.rendererSock.recvfrom(2048)

     except:

         raise

     if rawPacket:

         #print "Renderer: Received Message"

         packet = rawPacket.split(" ",2)

         self.type = packet[0] #first is type

         self.code = packet[1] #second is code

         if type == "STREAM" and code == "NULL":

             self.inPayload = packet[2].split(" ",1) #will be split if type and code are stream and NULL

         else:

             self.inPayload = packet[2]

         #print "Renderer: MESSAGE: " + self.type + " " + self.code + " " + str(self.inPayload)

def sendCommand(self, HOST, STATUS):

     #print "Renderer: Sending"

     localPacket = STATUS + self.outPayload

     if HOST == "S":

         self.rendererSock.sendto(localPacket,self.S) #host and port

     if HOST == "C":

         self.rendererSock.sendto(localPacket,self.C)

     #print "Renderer: Message Sent to " + HOST

     #print "Renderer: MESSAGE: " + localPacket

def case0(self):

     #print "Renderer: Case 0 - FILE"

     self.outPayload = self.inPayload

     self.sendCommand("S", "STREAM FILE ")

     self.outPayload = ""

     self.sendCommand("C", "STREAM ACK ")

     self.switchVar = -1

def case1(self):

     #print "Renderer: Case 1 - READY"

     self.totalLines = int(self.inPayload[:2])

     self.outPayload = ""

     self.sendCommand("C", "STATUS READY ")

     self.switchVar = -1

def case2(self):

     #print "Renderer: Case 2 - FNF"

     self.outPayload = ""

     self.sendCommand("C", "STATUS FNF ")

     self.switchVar = -1

def case3(self):

     #print "Renderer: Case 3 - PLAY"

     tempLine = int(self.inPayload[:2])

     line = self.inPayload[2:]

     if tempLine != self.currentLine :

         self.outPayload = str(self.currentLine)

         self.sendCommand("S", "PLAY LINE ")

     else:

         print(line.rstrip('\n'))

         self.currentLine += 1

     self.switchVar = -1

def case4(self):

     #print "Renderer: Case 4 - RESTART"

     self.currentLine = 0

     print "\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n"

     self.switchVar = -1

def case5(self):

     #print "Renderer: Case 5 - END"

     if self.currentLine == self.totalLines :

         self.outPayload = ""

         print "\n\nThe End"

     else :

         self.outPayload = str(self.currentLine)

         self.sendCommand("S", "PLAY LINE ")

     self.switchVar = -1

def case6(self):

     self.outPayload = ""

     self.sendCommand("C", "DISCONNECT ACK ")

     self.disconnect = True

     print "\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n"

     print "Renderer: Disconnected"

#create Frame

renderer = Renderer()

print "\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n"

while renderer.isConnected():

#print("Renderer Main loop begun")

renderer.listenCommand()

if renderer.type == "STREAM":

     if renderer.code == "FILE":

         renderer.switchVar = 0

     else:

         renderer.switchVar = 3

if renderer.type == "STATUS":

     if renderer.code == "READY":

         renderer.switchVar = 1

     if renderer.code == "FNF":

         renderer.switchVar = 2

     if renderer.code == "END":

         renderer.switchVar = 5

if renderer.type == "PLAY":

     renderer.switchVar = 4

if renderer.type == "DISCONNECT":

     renderer.switchVar = 6

if renderer.switchVar == 0:

     renderer.case0()

elif renderer.switchVar == 1:

     renderer.case1()

elif renderer.switchVar == 2:

     renderer.case2()

elif renderer.switchVar == 3:

     renderer.case3()

elif renderer.switchVar == 4:

     renderer.case4()

elif renderer.switchVar == 5:

     renderer.case5()

elif renderer.switchVar == 6:

     renderer.case6()

**Server:**

#4390 Semester Project

#Patrick Le, Trevor, Duncan, Logan Dennison

#Server Code

import socket

import sys

import os

import time

from os import listdir

from os.path import isfile, join

class Server:

#assign values

LOCAL\_IP = "10.0.0.1" #local

R\_IP = "10.0.0.2"

C\_IP = "10.0.0.3"

C = (C\_IP,5000) #tuple object

R = (R\_IP,5000)

S = (LOCAL\_IP,5000)

serverSock = socket.socket(socket.AF\_INET,socket.SOCK\_DGRAM) #internet, udp

type = ""

code = ""

inPayload = ["",""]

outPayload = ""

selectFile = None

switchVar = 4 #escape

lineCount = 0

totalLines = 0

filePath = "/home/milo/Documents/Final/Text/"

disconnect = False

pause = False

def \_\_init\_\_(self):

     self.serverSock.bind(self.S)

     self.disconnect = False

def isConnected(self):

     if self.disconnect:

         return False

     else:

         return True

def isPaused(self):

     return self.pause

def listenCommand(self):

     print "Server: Listening"

     try:

         rawPacket = self.serverSock.recv(2048)

     except:

         raise

     if rawPacket:

         print "Server: Received Message"

         packet = rawPacket.split(" ",2)

         self.type = packet[0] #first is type

         self.code = packet[1] #second is code

         if type == "STREAM" and code == "NULL":

             self.inPayload = packet[2].split(" ",1) #will be split if type and code are stream and NULL

         else:

             self.inPayload[0] = packet[2]

         if type == "PAUSE" and code == "NULL":

             self.pause = True

         else:

             self.pause = False

         print "Server: MESSAGE: " + self.type + " " + self.code + " " + str(self.inPayload)

def sendCommand(self, HOST, STATUS): #work on

     print "Server: Sending"

     localPacket = STATUS + str(self.outPayload)

     if HOST == "R":

         self.serverSock.sendto(localPacket,self.R) #host and port

     if HOST == "C":

         self.serverSock.sendto(localPacket,self.C)

     print "Server: Message Sent to " + HOST

     print "Server: MESSAGE: " + localPacket

#switch statement

def case0(self): #CONNECT

     print "Server: Case 0 - CONNECT"

     self.outPayload = ""

     arr = os.listdir(self.filePath) #directory of files

     for x in arr: #goes through file

         self.outPayload += " " + x #names of files

     self.outPayload = self.outPayload[1:]

     self.sendCommand("C", "CONNECT ACK ")

     self.type = "PAUSE"

     self.code = "NULL"

def case1(self): #STREAM

     print "Server: Case 1 - STREAM"

     arr = os.listdir(self.filePath) #directory of files

     for x in arr: #goes through file, then opens

         if os.path.exists(self.filePath + self.inPayload[0]): #adding to full file path

             self.selectFile = open(self.filePath + "/" + self.inPayload[0],'r')

     #opened file

     if self.selectFile != None: #FIX

         line = self.selectFile.readline() #first line

         while line:

             self.lineCount += 1 #increment total totalLines

             line = self.selectFile.readline()

         self.outPayload = self.lineCount

         self.sendCommand("R", "STATUS READY ")

     else:

         self.sendCommand("R", "STATUS FNF ")

     self.lineCount = 0

     self.selectFile.seek(0) #restart file pointer

     self.type = "PAUSE"

     self.code = "NULL"

def case2(self): #PLAY/NULL

     print "Server: Case 2 - PLAY"

     line = self.selectFile.readline()

     if line:

         self.outPayload = str(self.lineCount)

         self.outPayload = self.outPayload + " "

         self.outPayload = self.outPayload + line

         self.sendCommand("R","STREAM NULL ")

         self.lineCount += 1

         time.sleep(.5) #delay of 1/10s

     else:

         self.sendCommand("R","STATUS END ")

         self.type = "PAUSE"

         self.code = "NULL"

def case3(self): #PLAY/LINE

     print "Server: Case 3 - PLAY LINE"

     self.selectFile.seek(0)

     lineRequest = int(self.inPayload[0])

     x = 0

     while (x < lineRequest):

         line = self.selectFile.readline() #store current line indexed by # x

         x += 1

     self.lineCount = lineRequest

     self.type = "PLAY"

     self.code = "NULL"

def case4(self): #PAUSE, DEFAULT

     if self.isPaused():

         print "Server: Case 4 - PAUSE"

     self.type = "PAUSE"

     self.code = "NULL"

def case5(self): #DISCONNECT

     print "Server: Case 5 - DISCONNECT"

     self.sendCommand("C","DISCONNECT ACK ")

     self.sendCommand("C","DISCONNECT ACK ")

     self.sendCommand("C","DISCONNECT ACK ")

     self.sendCommand("C","DISCONNECT ACK ")

     self.disconnect = True

     print "\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n"

     print "Server: Disconnected"

server = Server()

server.serverSock.setblocking(0)

print "\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n"

print "Server: Connected"

while server.isConnected():

time.sleep(.5)

print "\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n"

try:

     server.listenCommand()

except socket.error:

     pass

if server.type == "CONNECT":

     server.switchVar = 0

if server.type == "STREAM":

     server.switchVar = 1

if server.type == "PLAY" and server.code == "NULL":

     server.switchVar = 2

if server.type == "PLAY" and server.code == "LINE":

     server.switchVar = 3

if server.type == "PAUSE":

     server.switchVar = 4

if server.type == "DISCONNECT":

     server.switchVar = 5

if server.switchVar == 0:

     server.case0()

elif server.switchVar == 1:

     server.case1()

elif server.switchVar == 2:

     server.case2()

elif server.switchVar == 3:

     server.case3()

elif server.switchVar == 4:

     server.case4()

elif server.switchVar == 5:

     server.case5()

**Protocol:**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Text Stream Protocol  Status of this Memo  This document specifies a text transfer track protocol for the Internet community, and requests discussion and suggestions for improvements. Please refer to the current edition of the “Text Stream Protocol STD 1” for the standardization state and status of this protocol. Distribution of this memo is unlimited.  Abstract    Table of Contents   1. Intro   Text Stream Protocol is a simple text based application protocol to send to hosts in a network. The protocol sends a Message Type and Message Code along with a payload coming from a host. The entire message is plaintext, dynamically-sized, and each field is separated by a ‘ ‘ character.   1. Message Format  |  |  |  | | --- | --- | --- | | **Message Format** | | | | **Message Header** | | **Message Body** | | **Message Type** | **Message Code** | **Payload** |  1. Message Types      |  |  |  |  |  | | --- | --- | --- | --- | --- | | **Protocol Message List** | | | | | | **Message Type** | **Code** | **Payload** | **Direction** | **Notes** | | **CONNECT** | **NULL** |  | **C\* -> S\*** | **Used to request list of files at start** | | **ACK** | **List files** | **S -> C** | **Sends list of files back to controller** | | **STREAM** | **FILE** | **String file** | **C -> R\*** | **Passes selected file to renderer** | | **R -> S** | **Initiates stream of selected file with server** | | **NULL** | **Int lineNum** | **S -> R** | **Transmits a line of the selected file** | | **String line** | | **ACK** |  | **R -> C** | **Tells the controller that the stream request has been passed to server and to await STATUS** | | **STATUS** | **READY** | **Int totalLines** | **S -> R** | **Passes that the file was found and the server is ready to stream** | | **R -> C** | | **FNF** |  | **S -> R** | **Signals that the file was not found** | | **R -> C** | **Signals that the file was not found and to request the list of available files again** | | **END** |  | **S -> R** | **Signals that the server has sent the last line in the file** | | **R -> C** | **Signals that the renderer has received the last line in the file** | | **PLAY** | **NULL** |  | **C -> S** | **Tells the server to start the stream** | | **LINE** | **Int lineNum** | **R -> S** | **Tells the server to play from the line passed as it received the incorrect line** | | **Int lineNum = 0** | **C -> S** | **Tells server to restart stream** | | **ACK** |  | **S -> C** | **Acknowledges the message** | | **S -> R** | | **PAUSE** | **NULL** |  | **C -> S** | **Tells server to pause stream** | | **ACK** |  | **S -> C** | **Acknowledges the message** | | **DISCONNECT** | **NULL** |  | **C -> R** | **Signals the program to close** | | **C -> S** | | **ACK** |  | **R -> C** | **Acknowledges the message** | | **S -> C** | | \*C = Controller, R = Renderer, S = Server | | | | | | | |
|  |  |  |

**Design Documentation:**

We structured our protocol into three fields, Type, Code, and Payload. The Type indicates what stage of the text stream the program is in. Code indicates the specific instructions for the host receiving it to proceed with the current stage of the stream either to enact user input or internal correspondence. And the Payload contains whatever information the receiving client needs to execute properly.

The Controller works on a series of embedded loops that exit upon the disconnect bool being set to true. The first loop starts the by contacting the server and requesting a list of potential text files to stream. And when received, prints the option to the user to make a selection. And once selected the server passes the renderer the selected file so it can tell the server to prepare the selected file to stream. After this the controller passes into the second loop where it listens until it receives either a file not found message in which case it will restart the program or a ready message where it sends a play command to the server to commence the stream. After this is done successfully, the controller enters the final loop where it will continuously offer the user four stream control options: Play, Pause, Restart, or Disconnect. It will send the server the corresponding commands until the user selects disconnect. At that point the controller sends disconnect messages to all hosts to do so before it disconnects and closes itself.

The Renderer contains a very simple single loop that blocks on the listen command until it is given a new command from either the server or the controller. The first message it will handle is the Stream message from the controller that carries the selected file to stream. It passes that to the server and then ACKs the controller. The server will then either send a FNF or a ready message to pass back to the controller. If the server sends a ready message the controller will signal it to commence the stream and the server will send individual lines of text to the renderer until the end. The ready command from the server gives the renderer the total number of lines it will be sending so it can maintain what line it is expecting and if a packet is lost in transit it will tell the server to retransmit from the lost packet. Otherwise it will print the lines sent to it according to the commands the controller sends the server. Upon completion of the text the server will send the renderer the end command and it checks if its line counter is in fact equal to the total number of lines sent at the beginning. It will then listen until the user restarts the text or disconnects. Upon a disconnect command it will exit its loop and close.

The Server is structured similarly to the renderer however it is non blocking so it is continuously looping and checking whether a message was received, unlike the renderer which had to be blocking so it can display the text as it is received. The server is the simplest of all of the programs it is either sending the controller a list of available text files, or sending the renderer a line of text either from its current pointer or a specific line denoted by the order checker of the renderer or responding to the commands send by the user via the controller. Upon completing the file it will send the end command the renderer that it has finished sending the file end command for confirmation. From that point it await for the user to restart the stream or to disconnect, at which point it will exit the look and close.