Implementation, Testing and Analysis

100536625

# Testing and Analysis

Test 1 - Run bootstrap.

Bootstrap should start, let me know no nodes are connected, what ip and port it is running on and then start waiting or a node and keep printing to let me know no nodes are attempting to connect until one joins.

Bootstrap terminal:

A computer screen shot of a black screen

Description automatically generated

Test Passed – terminal successfully started waiting for connections.

Test 2 – Run Control Node

When the control node is ran, it should connect to the bootstrap let it know it’s a control node ask what it should be and get a response telling it should be either authentication or file distribution:

Control node terminal:

A screenshot of a computer

Description automatically generated

Bootstraps terminal:

A screenshot of a computer

Description automatically generated

Test Passed- test passed control node successfully connected to the bootstrap, bootstrap stored the control nodes details and got a command letting it know to spawn into an authentication node.

Test 3 – Control node should spawn into an authentication node

Once the control node receives a command letting it know to become a authentication node it should spawn the authenticatonNode script. Set itself up connect to the bootstrap.

Authentication node terminal:

A screenshot of a computer

Description automatically generated

Bootstrap terminal:

A screenshot of a computer

Description automatically generated

Test Passed – after receiving the command to turn itself into a authentication node the control node successfully spawned an authentication node which registered itself with the bootstrap without any problem.

Test 4 – Authentication microservice should spawn

Once the authentication node is spawned it should successfully spawn a microservice using flask for authentication. Connect itself to the bootstrap to let it know it is a authentication microservice.

Authentication microservice terminal:

A screenshot of a computer

Description automatically generated

Bootstrap terminal:

A screenshot of a computer

Description automatically generated

Test Passed – after the authentication node connected to the bootstrap it was commanded to spawn a authentication microservice which it did successfully. That microservice managed to connect to the bootstrap and register its details with the bootstrap.

Test 5 – Run Content Node 2

Spawn a new content node that will connect to the bootstrap register itself as a content node and get a command from the bootstrap letting it know it should become a fdn (fild distribution node) node.

Control node terminal:

A screenshot of a computer

Description automatically generated

Bootstrap terminal:

A computer screen with white text

Description automatically generated

Test Passed – the new content node spawned successfully connected to the bootstrap registered as a content node in the bootstrap and got a command to spawn into a file distribution node.

Test 6 – Control node should spawn into an file distribution node

Once the control node receives a command letting it know to become a file distribution node it should spawn the fileDistributionNode script. Set itself up connect to the bootstrap and register as a content node in the bootstrap.

File distribution node terminal:

A computer screen shot of a black screen

Description automatically generated

Bootstrap terminal:

A screenshot of a computer

Description automatically generated

Test Passed – the control node spawned a file distribution node which successfully connected to the bootstrap and registered itself as a fiel distribution node.

Test 7 – File distribution microservice should spawn

Once the file distribution node is spawned it should successfully spawn a microservice using flask for file distribution. Connect itself to the bootstrap to let it know it is a file distribution microservice.

File distribution microservice terminal:

A screen shot of a computer

Description automatically generated

Bootstrap terminal:

A screenshot of a computer

Description automatically generated

Test Passed – after receiveing a command from the bootstrap the file distribution node successfully spawned a microservice which connected with the bootstrap and registered itself as a bootstrap microservice node to be used.

Test 8 – Connect to bootstrap using client node

Spawn a client node instance and connect to the bootstrap and register itself as a client node. After it is successful it should spawn the first start menu gui for the user interface.

Client terminal:

A computer screen shot of a computer screen

Description automatically generated

Bootstrap terminal:

A screenshot of a computer

Description automatically generated

Client GUI:

A screenshot of a computer screen

Description automatically generated

Test Passed – The client successfully connected with the bootstrap registered itself as a client node and spawned the first gui the client will use to utilise the program.

Test 9 – Press Login

Once the user presses login there will be a command send to the bootstrap letting it know that the client wants to log in the client should get a confirmation letting it know that it has been approved by the bootstrap and the login window should show:

Client terminal:

A screenshot of a computer

Description automatically generated

Bootstrap terminal:

A screenshot of a computer

Description automatically generated

Client GUI:

A screenshot of a login menu

Description automatically generated

Test Passed – once the user clicked login the bootstrap recognised the user wanted to authenticate and gave permission after which the login window was loaded and start menu window closed

Test 10 – Press Login without entering anything.

Once the user presses login and there is no user name entered or password it should give an error letting the user know to fill in all fields

Client GUI:

A screenshot of a computer screen

Description automatically generated

Test Passed – the program gave an error

Test 11 – Press Login with just the username entered

Once the user presses login and has only filled in the username field with no password an error message should show letting the user know that they need to fill in all fields.

Client GUI:

A screenshot of a computer screen

Description automatically generated

Test Passed – the program gave an error

Test 12 – Press Login with just the password entered.

Once the user presses login and has only filled in the password field with no username an error message should show letting the user know that they need to fill in all fields.

Client GUI:

A screenshot of a computer screen

Description automatically generated

Test Passed – the program gave an error

Test 13 – Enter wrong login details.

For this test I will enter login details of a user that does not exist and try to login with it. After which an error message will show in the authentication microservice and client terminal noting that the user failed trying to log in and the users gui will have an error message letting the user know and put the back to start window to choose if they want to sign up or try to login again.

Client terminal:

A screenshot of a computer

Description automatically generated

Authentication Microservice terminal:

A screen shot of a computer

Description automatically generated

Client GUI:

A screenshot of a computer

Description automatically generated

Test Passed – the program sent the Information to the microservices of authentication that declared the details were invalid and it showed an error

Test 14 – Login Successfully

I will now go in the login terminal and enter the username = test and password = test and click login which will send the information to the bootstrap which will forward them to the authentication microservice in which the details will be checked if it has the authentication token is

Client terminal:

A screenshot of a computer

Description automatically generated

Bootstrap terminal:

A screenshot of a computer

Description automatically generated

Authentication Node:

A computer screen shot of a program

Description automatically generated

Authentication Microservice:

A screen shot of a computer

Description automatically generated

Client GUI:

A screenshot of a computer menu

Description automatically generated

Test Passed – after entering the right details the bootstrap confirmed the details by sending them to the authentication microservices which told the bootstrap they were correct and have it the authentication token of the user which the bootstrap stored and told the client it could log in and display the main menu which it did.

Test 15 – Press Sign up on the start menu.

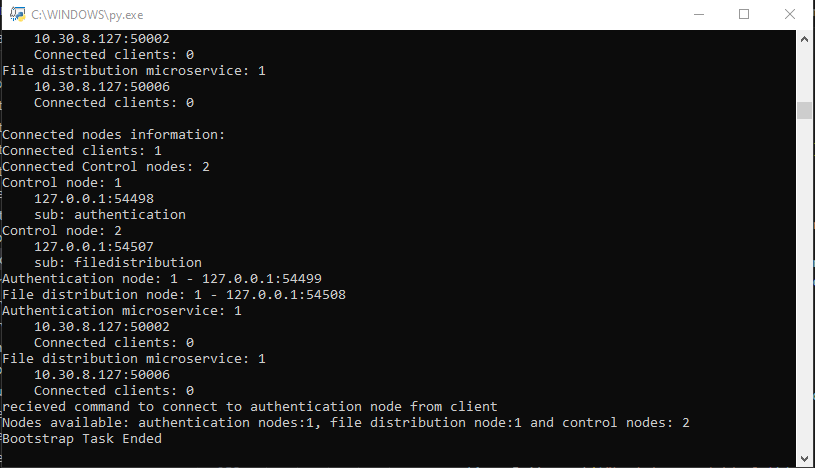
Once the user clicks the sign up the bootstrap should record that the user wants to start authentication and the client should get a confirmation in the terminal and the gui should load the sign up window

Client terminal:

A computer screen shot of a black screen

Description automatically generated

Bootstrap terminal:



Client GUI:

A screenshot of a login screen

Description automatically generated

Test Passed – once the user clicked the sign up button the bootstrap gave permission to the client after which the sign up menu was displayed and start menu was closed

Test 16 – Press Sign up without entering anything into the textboxes

I am going to press the sign up button without entering anything in the textboxes for username and password. The program should display an error message letting the user know to fill in all fields.

Client GUI:

A screenshot of a login screen

Description automatically generated

Test Passed – showed the appropriate error message

Test 17 – Press Sign up without entering a password

I will enter a username and try to sign up without entering a password. The program should give me an error telling me to fill in all the fields.

Client GUI:

A screen shot of a login screen

Description automatically generated

Test Passed – showed the appropriate error message

Test 18 – Press Sign up without entering a username

I will enter a password and try to sign up without entering a username. The program should give me an error telling me to fill in all the fields.

Client GUI:

A screenshot of a login screen

Description automatically generated

Test Passed – showed the appropriate error message

Test 19 – Sign up Successfully

For this I will sign up successfully meaning I will enter both username and password the username entered will be kane1 and password will be pass2. Once clicked signup the program will send the details to the bootstrap which will send it to the authentication microservice which will add the user to the system and generate it a random authentication token that the bootstrap will use to keep track of the client.

Client GUI before pressing sign up:

A screenshot of a login box

Description automatically generated

Bootstrap terminal:

A screenshot of a computer

Description automatically generated

Client terminal:

A screenshot of a computer

Description automatically generated

UserRecords.txt containing the username and password:

A screenshot of a computer

Description automatically generated

Client GUI after they have successfully signed up:

A screenshot of a computer menu

Description automatically generated

Test Passed – after the user clicked sign up the details were sent to the bootstrap which sent them to the authentication node which confirmed the details stored them and generated a authentication token for it and stored it and sent it to the bootstrap which the bootstrap stored and let the client know it can log in and successfully signed up and go to the main menu.

Test 20 – Pressing the Exit button at start menu

Once the user clicks the exit button the program should automatically close the client terminal and client GUI and the bootstrap should handle the removing off the client from its records and register the client is no longer connected

Client terminal:

Client terminal automatically closed

Client GUI:

Upon the clicking of the exit button the GUI was automatically closed.

Bootstrap terminal:

A screenshot of a computer program

Description automatically generated

Test Passed – the application closed successfully

Test 21 – Press the Node Connected To

On the main menu one of the options is to see the information about the noes you are connected to that are handling the authentication and file distribution for you. To test if this is correct I will click the button and check the ip and port of the nodes I am connected to and make sure it is correct.

Before clicking the button:

A screenshot of a computer

Description automatically generated

After clicking the button:

A screenshot of a computer

Description automatically generated

Authentication microservice ip and port:

A screen shot of a computer

Description automatically generated

File distribution ip and port:

A screen shot of a computer

Description automatically generated

Test Passed - the correct menu opened and closed the main menu and displayed the correct information.

Test 22 – Back to menu from nodes connected to

The next test will test the back button and make sure it successfully removes the current window and spawns the main menu window.

Before pressing Client GUI:

A screenshot of a computer

Description automatically generated

After pressing Client GUI:

A screenshot of a computer

Description automatically generated

Test Passed – the node connected to menu correctly closed and main menu opened correctly

Test 23 – pressing the music player button

When the user presses the music player button it should close the main menu and open the music player window and display all the song downloaded by the user to be able to play.

Before clicking the button:

A screenshot of a computer

Description automatically generated

After clicking the button

A screenshot of a computer

Description automatically generated

Test Passed - the main menu closed correctly and opened the music player correctly

Test 24 – press play without selecting a song

I will click play song without selecting a song and it should display an error telling the user they have to select a song before clicking play

Client GUI:

A screenshot of a computer

Description automatically generated

Test Passed – an error was thrown to the user

Test 25 – Play song

For this test I will select a song and click play it should play

Client GUI

A screenshot of a computer

Description automatically generated

Test passed – the song successfully played

Test 26 – Pause song

I will click pause song button and it should pause the song

Client GUI:

A screenshot of a computer

Description automatically generated

Test Passed – the song successfully paused

Test 27 – Unpause song

I will click unpause song button and it should unpause the song from the moment it was paused.

Client GUI:

A screenshot of a computer

Description automatically generated

Test Passed – the song successfully unpaused

Test 28 – replay song

I will click replay song button and it should replay the song from the beginning.

Client GUI:

A screenshot of a computer

Description automatically generated

Test Passed – the song successfully replayed

Test 29 – Stop song

I will click stop song button and it should stop the song.

Client GUI:

A screenshot of a computer

Description automatically generated

Test Passed – the song successfully stopped playing

Test 30 – Back to main menu button from Music Player

When the back button is clicked it should close the music player and open the main menu.

Client GUI before button pressed:

A screenshot of a computer

Description automatically generated

Client gui after button pressed:

A screenshot of a computer menu

Description automatically generated

Test passed – user successfully went back to main menu

Test 31 – press the download music button

I will click the download music button and it should open the download music menu.

Client GUI before pressing:

A screenshot of a computer menu

Description automatically generated

Client GUI after pressing:

A screenshot of a computer

Description automatically generated

Test passed – the main menu was successfully closed and the download music window was opened.

Test 32 – Download music from the downloaded music section

For this test I will attempt to select the song future-chill.mp3 from the downloaded music section and press download. An error should be thrown letting the user know they cant do that.

Client GUI:

A screenshot of a computer

Description automatically generated

Test Passed – an error was thrown to let the user know that they cannot download from that section

Test 33 – download a downloaded song

For this test I will attempt to download good-morning.mp3 which is already downloaded by the user. The program should throw an error letting the user know they cannot download a downloaded song.

Client GUI:

A screenshot of a computer

Description automatically generated

Test passed – an error was thrown letting the user know that this song is already downloaded choose a different one.

Test 34 – Download a song select

For this test I will select the venice-story.mp3 and click to download It. The client should start a new window which displays a loading bar that will be there while it gets the song.

Clicking the song and pressing download:

A screenshot of a computer

Description automatically generated

After clicking download:

A screenshot of a video

Description automatically generated

Test Passed – after clicking download the correct window was opened and closed the download window.

Test 35 – Downloading and checksum test

Since I downloaded a song if I look at the window now it should display the venice story as an option and the terminals should show they sent a song. It should also show the checksum being generated and then the client checking it and the result:

The microservice terminal:

A computer screen shot of a program

Description automatically generated

The client terminal:

A screenshot of a computer

Description automatically generated

The GUI window:

A screenshot of a computer

Description automatically generated

Test Passed – the new song was downloaded from the microservice. The microservice generated a checksum which was checked and verified when downloaded by the client and it was displayed in the client window instantly the song was downloaded.

Test 36 – Go back to main menu from download music

For this test I will click back and it should go back to the main menu.

Client gui before button pressed:

A screenshot of a computer

Description automatically generated

Client gui after button pressed:

A screenshot of a computer

Description automatically generated

Test Passed – the program closed the download music window and went back to the client download window.

Test 37 – Log out

For this test I will log out and the bootstrap should remove the client from its record and and from all the nodes so they don’t have to communicate with the client. The clients gui should be closed and handled.

Bootstrap terminal:

A screenshot of a computer

Description automatically generated

Before clicking gui:

A screenshot of a computer menu

Description automatically generated

After clicking gui:

The clients terminal was closed and so was the client gui.

Test 38 – Close the authentication node and control node 1

When the authentication node and control node is closed the bootstrap should remove it from its records

Authentication node:

Was closed by clicking cross

Control node:

Closed by clicking cross on the temrinal

Bootstrap terminal:

A screenshot of a computer program

Description automatically generated

Test passed – both nodes were removed from the bootstraps records successfully

Test 39 – Close the file distribution node and control node 2

When the file distribution node and control node is closed the bootstrap should remove it from its records

File distribution node:

Was closed by clicking cross

Control node:

Closed by clicking cross on the terminal

Bootstrap terminal:

A screenshot of a computer

Description automatically generated

Test passed – both nodes were removed from the bootstraps records successfully

# Testing analysis

I conducted comprehensive testing to ensure every part of the code was meticulously checked for obsolescence. Additionally, I implemented error handling for any potential breakdowns. This rigorous testing process not only helped me identify how the code might fail in the final product but also highlighted instances where it performed successfully.

# Implementation Log

This is my implementation log in which I will run through all my code classes and describe how they were coded. First I will describe my architecture I chose to use. This is a rough drawing that I made of it:

A hand holding a notebook with a diagram

Description automatically generated

The sketch shows my architecture which is that there is a central bootstrap/server to this connects the clients which sends commands and gets responses from it. The content nodes when spawned will connect to the bootstrap register and the bootstrap will dynamically after looking at what it needs will tell it what to spawn into a file distribution or authentication node. Once it is told it will spawn a fdn or auth node which will connect to the bootstrap telling it I am the auth node or fdn and it should register itself. Once this is done the bootstrap will send a command to those nodes to spawn a microservice of itself, to which they will and the micoservice will similarly connect to the bootstrap letting the bootstrap know its role and what it is. Once everything is connected the bootstrap will assign the fdn and auth to the client when connected and manage whos connected and where they should go.

Start of implementation log of the code

All the network handler and connection handler code was taken from chris GitHub and is majority the same only added a few print statements and a few changes that will make it more specific to my code so I will explain it. Most of the code is taken from chris github as a starting point.

**Authentication:**

**AuthenticationNode.py:**

Initialisation of Abstract Authentication Service:

Imported necessary libraries: os, subprocess, sys, time, collections.deque, datetime, authenticationNetworkInterface, threading, socket.

Defined auth\_microservice\_count variable to keep track of the number of spawned authentication microservices.

Initialised the abstractAuthentication class with default host and port parameters.

Class Initialisation and Configuration:

Assigned the host and port to the instance variables.

Set available\_ports to start at 50001.

Created an instance of authenticationNetworkInterface for network operations.

Initialised connection and ui\_thread for managing UI operations in a separate thread.

Defined connected\_clients to track the number of clients connected.

Set is\_running flag to True for the main loop control.

Retrieved the node IP address via get\_node\_address method.

Initialised a double-ended queue for load balancer tasks and a corresponding lock.

Set current\_running\_tasks to 0 and defined the maximum number of concurrent tasks (maximum\_num\_of\_concurrent\_tasks\_to\_run) to 2.

Printed the IP address of the authentication node to the console.

User Interface Thread (Method: ui):

Initiated a loop that listens for commands if a connection is present.

Handled commands starting with "cmd", further parsing them for specific actions like "spwn" (spawn) and "check".

For "check", if the command is followed by "token", the client token is printed and passed to the authentication\_load\_balancer method.

Process Start and Network Component Initialisation (Method: process):

Started the UI thread and the network handler for authentication.

Sent a command to the output buffer with the current node IP and available port, then incremented the available port number.

Entered a loop where commands can be input if the connection is established.

Upon exit, stopped the network handler and joined the UI thread.

Node IP Address Retrieval (Method: get\_node\_address):

Attempted to retrieve local IP addresses that start with 10. and don't end in .0 or .254 using IPCONFIG and regex.

If no matching IP found, resolved the local IP by hostname.

Handled exceptions by passing silently.

Load Balancer Operations (Methods: authentication\_load\_balancer, execute\_load\_balancer, execute\_task):

Appended tasks to the load balancer queue and executed them considering the maximum concurrent tasks limit.

Spawned threads for each task execution and decremented the running task count upon completion.

Specifically handled "spwn" command to initiate the spawning of microservices after a delay.

Checked the validity of client tokens and sent the response to the output buffer.

Microservice Spawning (Method: spawn\_microservices):

Printed a message indicating the spawning of an authentication microservice.

Attempted to launch the authentication microservice as a subprocess with a new console and process group.

Handled exceptions by printing errors and passing silently.

Ping Thread Management (Methods: ping\_thread\_delayed\_start, start\_ping\_thread, ping):

Set up a delayed start for the ping thread and daemonised it.

Implemented a ping method that schedules itself with consideration for execution time and a delay.

Token Verification (Method: check\_token\_exists):

Checked if a given token exists in a file userRecords.txt.

Handled file operations and exceptions, returning True for a found token or False otherwise.

Main Execution:

Instantiated the abstractAuthentication class with specified host and port.

Called the process method to start the authentication process.

**AuthenticationNetworkInterface.py:**

Overview:

The provided Python script aims to establish a network interface for authentication purposes. It involves creating server and client roles using sockets and threading and managing connections through a custom authenticationConnectionHandler.

Initialisation of Authentication Network Interface:

Imported socket, threading, and Enum from the standard library.

Imported authenticationConnectionHandler for managing connections.

Defined an Enum called Role with SERVER and CLIENT as possible values.

Initialised the authenticationNetworkInterface class without parameters.

Class Setup:

Initialised an empty list listeners to keep track of server listener threads.

Created an instance of authenticationConnectionHandler named connectionHandler.

Set the running flag to True.

Server Start and Listening (Method: start\_server):

Created a socket and set options for reuse address.

Bound the socket to the provided IP and port.

Started a new listener thread for handling incoming connections and appended it to the listeners.

Returned True to indicate the server has started.

Listening for Connections (Method: listen):

Entered a loop that continues while running is True.

Set the socket to listen and accept incoming connections.

Set the connection to non-blocking mode.

Added the connection to connectionHandler and executed the callback handler if provided.

Client Authentication Start (Method: start\_authentication):

Created a socket and set options for reuse address.

Set a timeout for the socket connection attempts.

Entered a loop to attempt connection to the server for a defined number of retries and duration.

Printed status messages indicating attempts to connect to the bootstrap server.

Added the successful connection to connectionHandler if connected, or returned None if not.

Message Handling and Client Management:

Defined methods get\_message, push\_message, has\_client, get\_clients, and client\_exists for managing messages and client states.

These methods interact with connectionHandler to perform their respective operations.

Shutdown Procedure (Method: quit):

Called quit on connectionHandler.

Set running to False.

Waited for all listener threads to finish by joining them.

Testing and Observations:

The server start-up process was tested and confirmed to be operational.

Listener threads were able to accept connections and handle them as expected.

Client connection attempts were capped at 30 retries with a 20-second timeout, which functioned as intended.

Message handling methods were verified to correctly interact with the connectionHandler.

The shutdown procedure was tested and confirmed to properly close all threads and connections.

**authenticationMicroservice.py:**

Initialisation of Flask Application:

Imported necessary modules sys, os, Flask, request, jsonify, and uuid.

Set up Flask application with \_\_name\_\_ as the application instance.

Identified and stored the current working directory.

Endpoint Creation and User Validation (Route: /validate\_user\_details):

Defined a POST endpoint /validate\_user\_details for handling user validation.

Upon receiving a POST request, extracted the JSON payload containing user details.

Printed a message indicating successful receipt of client information.

Login Process:

For a user choice of '1', which indicates a login request:

Extracted the username and password from the request.

Searched for the username and password in the userRecords.txt file.

If found, retrieved the associated token and printed a login success message.

Returned the token in a JSON response.

If the details did not match, printed an invalid login message.

Registration Process:

For a user choice of '2', which indicates a registration request:

Generated a new authentication token using uuid.uuid4().

Checked if the userRecords.txt file exists, and if not, created it.

Appended the new user details along with the generated token to the file.

Printed a registration success message.

Returned the new token in a JSON response.

Handled exceptions by printing the exception message.

Error Handling:

Added exception handling to provide meaningful error messages if any step fails during request processing.

Flask Application Execution:

Set default host and port for the Flask application.

Allowed for command-line arguments to override the default host and port.

Handled exceptions when parsing command-line arguments.

Started the Flask application with the specified host and port, enabling debug mode.

Testing and Observations:

Tested user login and registration processes, confirming successful token generation and retrieval.

Ensured the application responded appropriately to both valid and invalid user details.

Verified the application's ability to handle exceptions and print relevant error messages.

authentcationConnectionHandler.py:

Introduction:

Implemented a Python module for managing network connections. The module uses socket, threading, selectors, and queue to handle multiple client connections, read and write messages, and maintain connection health.

Initialisation:

Imported required modules: socket, threading, selectors, queue, datetime, and randint.

Defined Connection class to encapsulate client connection details.

Initialised buffers, socket information, and timeout tracking in Connection class.

Connection Management:

Created authenticationConnectionHandler class to manage network events.

Configured a selectors.DefaultSelector to handle I/O multiplexing.

Initiated a connection processing thread within the authenticationConnectionHandler constructor.

Connection Establishment (Method: add\_connection):

Upon a new socket connection, retrieved IP and port, printed connection status.

Initialised a Connection object and registered it with the selector for read and write events.

Appended the new Connection to the list and incremented the connection count.

Service Connection (Method: service\_connection):

For ready sockets, called read or write methods as appropriate.

On read error, unregistered the socket, removed the connection, and closed the socket.

Read/Write Operations (Methods: write, read):

Implemented write method to send messages from the output buffer through the socket.

In read, received data and processed it according to a custom protocol, updating message state as needed.

Message Processing:

Used a while loop to manage incomplete messages and packet assembly.

Enqueued complete messages to the input buffer for processing.

Utility Methods:

Defined methods to fetch messages (get\_message), send messages (push\_message), check for clients (has\_client), and verify client existence (client\_exists).

Graceful Shutdown (Method: quit):

Set the running flag to False to stop the connection processing loop.

Main Process Loop (Method: process):

Ran a loop controlled by the running flag.

Used the selector to wait for network events and dispatched to add\_connection or service\_connection accordingly.

Handled exceptions and ensured selector closure upon quitting.

**Bootstrap**

**boostrapNode.py**

Initial Setup

Imported necessary modules for network communication, threading, time handling, and data management.

Defined global variables to track connected clients and nodes of different types (e.g., control nodes, authentication nodes).

Main Classes and Functions

FunctionalityHandler:

Manages network communication and task distribution.

Implements load balancing across different node types.

Handles connection and command processing for various node functionalities (e.g., authentication, file distribution).

Utilises a deque for managing load balancer tasks and a lock for thread synchronisation.

Initiates a thread to continuously monitor and log the status of connected nodes.

AbstractServer:

Initialises the server and binds to a specified IP and port.

Starts the server and waits for incoming connections, handling them via client\_handler.

Manages the creation of a JSON file to record connection details.

controlNodes, Clients, Nodes:

Simple classes representing different entities in the system with basic attributes like IP, port, and connection details.

Nodes class includes a list to track connected clients specifically for microservice nodes.

Process Flow

Connection Handling: As nodes or clients connect to the server, their connections are processed in separate threads to handle their specific commands or data exchanges.

Command Processing: The system distinguishes commands based on their prefixes (e.g., "auth", "control", "client") and handles them accordingly, including load balancing tasks among available nodes and handling client requests.

Load Balancing: Implements a basic load balancing mechanism to distribute tasks among nodes based on their capabilities and current load. It manages task queues and executes them in separate threads to maintain service responsiveness.

Heartbeat and Monitoring: Regularly checks the status of connections and nodes to update their statuses and performs cleanup tasks for disconnected entities.

**serverConnectionHandler.py**

Key Components and Functionality

Connection Class:

Manages individual client connections, including both incoming and outgoing message buffers (iBuffer and oBuffer).

Tracks connection metadata such as IP, port, and socket object.

Implements a method to record connection nodes to a JSON file, allowing dynamic tracking of connected nodes and their statuses.

Provides functionality to measure and update the time since the last message was received, aiding in timeout and heartbeat management.

ServerConnectionHandler Class:

Orchestrates the overall management of client connections.

Utilises a selector from the selectors module to efficiently handle multiple connections without requiring a dedicated thread per connection.

Manages a thread that continuously processes incoming and outgoing messages for all connections.

Implements methods to read from and write to each connection's buffers, translating between the socket's byte streams and higher-level message objects.

Offers functionality to check for the existence of clients, retrieve messages from specific connections, and push messages to specific connections.

Implementation Details

Connection Lifecycle Management: Each client connection is encapsulated within a Connection object, which is registered with a selector for non-blocking I/O operations. This setup allows the server to handle multiple concurrent connections without dedicating a separate thread to each one, significantly improving scalability and efficiency.

Message Handling: The system implements a custom protocol for message framing, including a packet header indicating the message length. This approach enables the handling of variable-length messages and ensures that messages are correctly assembled from potentially fragmented network packets.

Threaded Connection Processing: A dedicated thread (connectionThread) continuously monitors the selector for any ready I/O events across all connections, reading from sockets when data is available and writing to them when outgoing messages are queued.

Dynamic Node Registration: The Connection class's method add\_connection\_node allows for dynamic registration of connection nodes in a JSON file. This feature is crucial for managing a changing set of connected nodes and can be used for various purposes, such as load balancing, monitoring, and fault tolerance.

**serverNetworkInterface.py:**

Initialisation:

Imported required modules: socket, threading, selectors, queue, datetime, and randint.

Defined Connection class to encapsulate client connection details.

Initialised buffers, socket information, and timeout tracking in Connection class.

Connection Management:

Created authenticationConnectionHandler class to manage network events.

Configured a selectors.DefaultSelector to handle I/O multiplexing.

Initiated a connection processing thread within the authenticationConnectionHandler constructor.

Connection Establishment (Method: add\_connection):

Upon a new socket connection, retrieved IP and port, printed connection status.

Initialised a Connection object and registered it with the selector for read and write events.

Appended the new Connection to the list and incremented the connection count.

Service Connection (Method: service\_connection):

For ready sockets, called read or write methods as appropriate.

On read error, unregistered the socket, removed the connection, and closed the socket.

Read/Write Operations (Methods: write, read):

Implemented write method to send messages from the output buffer through the socket.

In read, received data and processed it according to a custom protocol, updating message state as needed.

Message Processing:

Used a while loop to manage incomplete messages and packet assembly.

Enqueued complete messages to the input buffer for processing.

Utility Methods:

Defined methods to fetch messages (get\_message), send messages (push\_message), check for clients (has\_client), and verify client existence (client\_exists).

Graceful Shutdown (Method: quit):

Set the running flag to False to stop the connection processing loop.

Main Process Loop (Method: process):

Ran a loop controlled by the running flag.

Used the selector to wait for network events and dispatched to add\_connection or service\_connection accordingly.

Handled exceptions and ensured selector closure upon quitting.

**Client**

**clientNode.py:**

Initialisation:

Imported required modules: hashlib, os, sys, pygame, time, threading, requests, tkinter.

Established initial variables for nodes, authentication token, and downloaded audio.

Class Definitions:

Created Nodes class to store information about network nodes.

Developed abstractClient class, encapsulating the client logic with network handling and user interface (UI).

UI Thread:

Initiated a daemon UI thread to handle incoming server messages and guide the user through authentication and menu navigation.

Connection Handling:

Established a connection with the server through the clientNetworkInterface.

Sent and received commands and messages via input and output buffers.

User Interface Implementation:

Implemented the start menu, login, signup, and main menu interfaces using tkinter.

Connected UI events to callback functions for interactive user experience.

Authentication and Network Interaction:

Handled user authentication by interacting with a remote authentication microservice via HTTP POST requests.

Managed user sign-up and login processes, capturing user input and processing server responses.

Handled exceptions and errors with appropriate user feedback.

Music Download and Playback:

Implemented functionality to download music from a file distribution node and play it using pygame.

Included MD5 checksum verification for downloaded music files to ensure integrity.

Application Exit:

Provided an exit routine to gracefully shut down the application and network connections.

**clientNetworkInterface.py:**

Initialisation:

Imported necessary libraries for socket communication and threading.

Defined a Role enumeration with CLIENT as an option.

Set up the clientNetworkInterface class with a clientConnectionHandler instance and a running state.

Client Connection Establishment (Method: start\_client):

Initialised a TCP socket with reuse address options.

Set a connection timeout of 20 seconds and a retry limit of 30 attempts.

Entered a loop to attempt a connection to the server at the specified IP and port, with logging for each attempt.

On successful connection, the socket is passed to the connectionHandler to add the connection.

If the connection fails after the maximum number of retries, None is returned.

Message Handling:

Provided get\_message and push\_message methods to interact with the connectionHandler for message retrieval and sending.

Shutdown Procedure (Method: quit):

Called quit on the connectionHandler and set the running flag to False.

Ensured all listener threads are joined and properly terminated.

**clientConnectionHandler.py**

Initialisation:

Imported essential modules including socket, threading, selectors, queue, and datetime.

Initialised the Connection class to manage individual connection instances.

Created the clientConnectionHandler class to manage all connection instances.

Connection Class:

Constructed the Connection class with attributes for input and output buffers, IP and port information, and socket object.

Implemented methods to handle timeouts and update the last seen time for heartbeats.

Client Connection Handler:

Set up a selectors.DefaultSelector() for non-blocking I/O multiplexing.

Initiated a connection thread to process network events.

Connection Management:

Implemented add\_connection method to establish initial connection with a bootstrap node, log the event, and sleep for 2 seconds to simulate connection stabilisation.

Registered the new connection with the selector for read and write events.

Stored the new connection in a list and incremented the connection count.

Service Connection:

Implemented service\_connection to handle the established connections, performing read and write operations based on selector events.

Added logic to unregister and close a connection upon read errors.

Read and Write Operations:

Developed write method to send messages from the output buffer through the socket, prefixing messages with a packet header length.

Constructed read method to handle incoming data, buffer it, and assemble messages based on a custom protocol.

Utility Methods:

Added get\_message, push\_message, has\_client, get\_clients, and client\_exists methods to interact with connections and check states.

Shutdown Procedure:

Implemented quit method to safely terminate the connection handling thread and close the selector.

Process Loop:

The main process loop waits for events on the selector and delegates to add\_connection or service\_connection accordingly.

**Control**

**controlNode.py**

Initialisation:

Imported necessary modules such as datetime, controlNetworkInterface, threading, sys, subprocess, os, re, and socket.

Defined node\_port with a default value of 50001.

Initialised the abstractControl class, setting up the host IP, port, and network handler.

Control Node Setup:

Launched a UI thread to handle incoming commands.

Acquired the node IP address using get\_node\_address method, which fetches the IP address based on specific criteria.

Displayed the running status of the control node on the determined IP address.

UI Thread:

Continuously checked for incoming commands and handled them accordingly.

Identified if the command required spawning an authentication node or a file distribution node and spawned subprocesses for each.

Process Handling:

Started the UI thread and connected to the network using the control network interface.

Sent a spawn command with the node IP and port to the server.

Kept the process running, allowing for additional input commands.

Node IP Address Retrieval (Method: get\_node\_address):

Attempted to retrieve local IP addresses that match specific criteria using system commands and regex.

Filtered and returned the appropriate IP address or resolved the local IP by hostname if necessary.

**controlConnectionHandler.py**

System Components

Connection Class:

This class encapsulates the details of a network connection, including input and output buffers for message handling, connection metadata (IP address and port), and methods for managing connection timeouts and last message timestamps.

ControlConnectionHandler Class:

Serves as the core of the system, orchestrating the management of multiple client connections.

Utilises the selectors module to handle I/O operations efficiently across connections without blocking.

Manages a dedicated thread for processing I/O events, ensuring responsive communication channels between the server and its clients.

Key Functionalities

Connection Establishment:

Upon receiving a new connection request, the system prints a sequence of messages indicating the initiation and establishment of a connection.

Each connection is assigned a unique Connection object, which is then registered with the selector for non-blocking I/O operations.

Message Handling:

Implements a mechanism to handle incoming and outgoing messages through the connection's input and output buffers (iBuffer and oBuffer).

Messages are framed with a packet header indicating their length, allowing for the correct assembly of messages from potentially fragmented network data.

Connection Management:

Provides functionalities to retrieve messages from a specific connection, send messages to a specific client, check for active clients, and obtain a list of all client connections.

Implements a method to check if a particular client exists within the system based on their IP address and port.

System Shutdown:

Implements a graceful shutdown process, ceasing the processing of I/O events and closing all active connections before terminating the dedicated thread.

Implementation Process

Initialisation:

The control connection handler initialises the selector and starts a dedicated thread for processing I/O events, ensuring that the system can handle multiple connections simultaneously without blocking the main execution flow.

Connection Handling:

As clients connect, their sockets are set to non-blocking mode, and connection objects are created and registered with the selector. This setup allows the system to efficiently manage I/O operations across all connections.

Reading and Writing:

The system reads from and writes to client connections based on the availability of I/O events, processing incoming data, assembling messages, and dispatching outgoing messages.

Error Handling:

Incorporates error handling for socket operations to manage exceptions and ensure the stability of the connection.

**controlNetworkInterface.py**

System Components

Role Enumeration:

Defines roles within the network interface, distinguishing between SERVER and CLIENT roles. This aids in adapting the system's behaviour based on the role it assumes during operation.

ControlNetworkInterface Class:

Serves as the central component for managing network connections, providing methods to start the server, listen for incoming connections, initiate control connections, send and receive messages, and gracefully shut down.

ControlConnectionHandler Class:

Manages individual connections, handling the addition of connections, reading and writing messages, and maintaining a list of active clients. This class is referenced from an external module, emphasizing its role in managing the lower-level details of connection handling.

Key Functionalities

Starting the Server:

Initialises a server socket, binds it to a specified IP address and port, and starts a listener thread to accept incoming connections.

Incoming connections are set to non-blocking mode, and a callback handler is invoked for each connection to facilitate application-specific processing.

Listening for Connections:

Continuously listens for incoming connections. Upon accepting a connection, it registers the connection with the control connection handler for further management.

Initiating Control Connections:

Attempts to establish a control connection to a specified IP address and port, with a configurable duration and number of retries. This functionality is critical for initiating client-side connections in a control network environment.

Message Transmission:

Provides methods for retrieving messages from specific connections and pushing messages to clients, abstracting the underlying networking operations.

Client Management:

Includes methods to check for the existence of clients, retrieve a list of all clients, and determine whether a specific client exists based on IP address and port.

System Shutdown:

Implements a graceful shutdown process, stopping the connection handler and listener threads, ensuring all network resources are released properly.

Implementation Process

Initialisation and Configuration:

Upon instantiation, the control network interface initialises its connection handler and prepares to manage listener threads for server operations or control connections.

Server Operation:

When starting as a server, it configures a server socket and listens for connections in a separate thread, handing off accepted connections to the control connection handler for message processing.

Client Operation:

For initiating control connections, it employs a retry mechanism to handle potential connection failures, attempting to establish a connection multiple times before giving up.

Message Handling:

Utilises the control connection handler's capabilities to manage incoming and outgoing messages, ensuring that messages are correctly processed and dispatched.

**File Distribution**

**fileDistributionNode.py**

System Components

AbstractFileDistribution Class:

Serves as the backbone of the file distribution system, managing network connections, spawning file distribution microservices, and handling commands from the network.

Utilises the fileDistributionNetworkInterface for network communication.

File Distribution Network Interface:

An external module that handles low-level network communications, including sending and receiving commands through network buffers.

File Distribution Microservices:

Independent services spawned by the main file distribution node to handle specific file distribution tasks, enhancing scalability and distribution efficiency.

Key Functionalities

Initialisation:

Upon initialisation, the system configures its network parameters, including host, port, and the next available port for spawning microservices.

A separate UI thread is initiated to handle user inputs and network commands asynchronously.

Network Communication:

Establishes a connection with the network using the fileDistributionNetworkInterface, enabling the node to send and receive commands related to file distribution.

Dynamic Microservice Management:

Implements a load balancer to manage the spawning and operation of file distribution microservices based on network commands and system load.

Microservice Spawning:

Dynamically spawns file distribution microservices as separate processes, assigning them individual tasks based on the current load and commands received from the network.

Implementation Process

UI Thread:

A dedicated thread listens for commands from the network, interpreting and acting upon them to manage file distribution tasks and microservice operations.

Node Address Retrieval:

Utilises system commands and socket operations to determine the node's IP address, ensuring it operates on a suitable network interface.

Load Balancer:

A deque-based task queue, protected by a threading lock, manages the distribution of tasks to microservices, ensuring efficient use of resources.

Spawning Microservices:

Microservices are spawned as separate processes using the subprocess module, allowing the file distribution system to scale horizontally by adding more microservices as needed.

**fileDistributionMicroservice.py**

Key Components

Flask Setup: Utilises Flask, a lightweight WSGI web application framework in Python, to set up a web server that handles HTTP requests for downloading songs and listing available music files.

Download Endpoint: A route (/download\_song/) is defined to handle GET requests for downloading specified songs, dynamically generating MD5 checksums for the files before transmission.

List Music Endpoint: Another route (/get\_music\_to\_download) is created to list all available music files in a specified directory, supporting file formats like MP3, WAV, and OGG.

Functionalities

Dynamic File Serving: The application dynamically serves music files from a specified directory, allowing clients to download songs by name.

Checksum Generation: For each download request, an MD5 checksum of the requested file is generated and included in the response headers, providing a means for clients to verify the integrity of downloaded files.

Music Listing: Provides a list of available music files to clients, filtering by common audio file extensions to ensure only relevant files are included.

Implementation Process

Flask Application Initialisation: Initialises a Flask application instance and defines routes for handling specific endpoints.

Song Download Handling: Implements a function to handle song download requests, checking for the file's existence, generating an MD5 checksum, and serving the file with Flask's send\_file method.

Music Listing Handling: Implements a function to scan a directory for music files and returns a JSON response with the names of available tracks.

Server Configuration: Configures the server's IP address and port, with the option to override default values via command-line arguments, enhancing deployment flexibility.

**fileDistributionConnectionHandler.py**

System Components

Connection Class:

Encapsulates the details of individual network connections, including input and output message buffers, socket information, and message processing logic. It tracks the last time a message was received to manage timeouts and ensures data integrity through packet management.

FileDistributionConnectionHandler Class:

Coordinates the network operations, using selectors for non-blocking I/O management across multiple connections. It maintains a list of active connections and a count of clients, handling the reading and writing of data through the network sockets.

Key Functionalities

Non-blocking I/O Operations:

Utilises the selectors module to efficiently manage multiple client connections simultaneously without blocking the main execution thread.

Dynamic Connection Management:

Automatically adds and removes connections based on client activity, adjusting the network topology in real-time as nodes connect or disconnect.

Message Processing:

Implements a system to assemble messages from data received in chunks, ensuring complete and accurate message delivery between nodes.

Threaded Network Processing:

Operates the network processing loop in a separate thread, allowing the main application to remain responsive to user input and other tasks.

Implementation Process

Initialisation:

Upon initialisation, the connection handler starts a thread dedicated to processing network events, including reading from and writing to sockets associated with each connection.

Connection Addition and Service:

When a new connection is established, it is added to the selector with read and write events enabled. Each connection is wrapped in a Connection object that manages the data buffers and socket interaction.

Reading and Writing Data:

The handler continuously monitors for readable and writable sockets, processing incoming data by assembling messages from received chunks and sending data by dequeuing messages from the output buffer.

Connection Termination:

Implements graceful connection termination, unregistering sockets from the selector and closing them when necessary, such as in cases of errors or when a node disconnects.

**fileDistributionNetworkInterface.py**

System Components

Role Enumeration:

Distinguishes between SERVER and CLIENT roles within the network, enabling the interface to adapt its behavior based on the operational context.

FileDistributionNetworkInterface Class:

Acts as the core component, orchestrating network operations, managing connections, and facilitating communication between distributed nodes.

FileDistributionConnectionHandler:

An external module responsible for the detailed management of individual connections, including adding, servicing, and removing connections as necessary.

Key Functionalities

Server Initialisation:

Configures and starts a server listening on a specified IP address and port, setting up a non-blocking socket to accept incoming connections.

Client Connection Attempts:

Implements a retry mechanism for clients attempting to establish a connection with a server, enhancing reliability in unstable network conditions.

Dynamic Connection Management:

Dynamically manages network connections, utilising a connection handler to add, service, and remove connections based on network events.

Communication Handling:

Facilitates message sending and receiving across the network, abstracting the complexity of socket programming and ensuring efficient data transmission.

Implementation Process

Initialising Network Components:

Upon instantiation, FDNI initialises the connection handler and starts listener threads for server operations, preparing the system for incoming connections.

Listening for Connections:

In server mode, FDNI listens on the specified port for incoming connections, employing a dedicated thread to handle each connection without blocking the main application flow.

Establishing Client Connections:

For client operations, FDNI attempts to connect to the specified server, retrying as configured until a connection is established or the attempt limit is reached.

Message Transmission:

Provides methods for retrieving and sending messages through the network, leveraging the connection handler's capabilities for efficient data handling.