100505349 Distributed Systems Usage Document

*100505349*

Contents

[Setup 2](#_Toc92455825)

[Commands & Usage 3](#_Toc92455826)

[Nodes and Distribution 4](#_Toc92455827)

[Modules 4](#_Toc92455828)

[Nodes 9](#_Toc92455829)

[Load Balancing 10](#_Toc92455830)

[External Libraries & Acknowledgments 10](#_Toc92455831)

# Setup

To begin using the program, there are some measures that must be taken to permit the system to work as expected. First and foremost, there are two documents that must be edited and coordinated, the \_ConnectionList.txt and the \_Logins.txt files. The connections list file must contain the list of all IPs that the server will run upon, delimited by a single “|” character, as shown below.

The login file must then be appended with similar data, storing the usernames and passwords of all valid logins to be processed by the device. It is assumed that this document will contain no duplicate usernames, and will be formatted in a system of “*Username*|*Password*”, followed by a new line, as shown below.

As previously mentioned, these files are expected to contain the same data for all instances of the system, and any modifications not mirrored on separate devices may cause unexpected issues to arise. This restriction is also true for any code files, which naturally should not be edited in any form. Outside of this, the contents “Music” folder and “cache” folder can be modified, notably, when attempting to test the distribution of files, different WAV files can be added to the “Music” as appropriate.

After performing this preliminary setup, the system can now be started, this can be done by starting the “Node.py” file, after which the user is prompted to enter the name of a node to start. The only valid inputs for this are “Control” and “Client”. The “Control” node should be started first, and only started on a device that is listed in the \_ConnectionList.txt file – as many control nodes can be started as desired, though the system requires at least one node and only permits one control node per device. A Client node can be started on any device on the same local network as the other control nodes and is not restricted to the IPs referenced in the \_ConnectionList.txt file.

When run for the first time, there is a small chance the script may instantly close, this is due to circumstances outside the control of the program, in these rare cases the program may simply be started again to successfully run. An additional article of note is that the program has three restrictions in place – the system must have access to the ports 50001 through 50010, the system must have an active internet connection (Though this is not used for any functions aside from returning the IP the system should listen upon on the local device) and the system must be running on the windows 10 operating system.

Finally, due to restrictions set in place by the university, there is a chance that a node running on port 50011 or higher can produce anomalous results. As the system functions by checking the next available port after 50001, it is possible to reach this port number by having too many instances of the node file running simultaneously. This issue is not apparent in systems without the 50001-50010 port restrictions in place, but nonetheless a warning is displayed when a node reaches the 50011th port.

# Commands & Usage

When at least one control node exists on an IP specified in the \_ConnectionList.txt file, and at least one client has started, the client may input a number of commands into the system to achieve different results. A list of valid commands is provided below, with any user specified inputs being defined in brackets.

**Authentication Commands**

LOGIN|(Username)|(Password)

This command allows a client to submit a username and password to be processed by an authentication node, which will provide the user with the ability to access other commands if the username and password specified are valid. This command must be used before all other commands can be processed.

**Music Download Commands**

!PLAY:(Music name)

This command, when provided with a valid music file name, will start the download of a music file from a file distribution node, allowing a client node to download music data across the network. This system has been tested with up to 15 minutes of music without any faults occurring, though this relies on the strength of the local network.

PLAYLIST

This command returns a list of all available files on the network and is handled by the control node. Due to the inner workings of the network, the playlist will sometimes take time to update as new nodes connect, especially if there are multiple IPs in the \_ConnectionList.txt file without active control nodes.

**Echo Commands**

ECHO|(String to echo)

This command returns a string back to the user with the same data as they initially input and is handled entirely by the echo node.

ECHODUMP

This command, once again handled by the echo node, displays the values of all echo requests that have been processed on the node in question. This data is node specific, meaning it only shares the echo data provided by clients connected to this specific instance of the echo node.

**Dictionary Commands**

DICT|(String to define)

This command returns the definition of a value as stored in the internal dictionary of the handling dictionary node. Much like the echo node, the contents of this dictionary are node specific.

DICTADD|(Dictionary entry name)|(Dictionary entry description)

This command allows the input of a dictionary entry into a dictionary node, and much like the previous commands is node specific. Dictionary entries added can have their descriptions returned by the “DICT” command.

**Playback Commands**

VOLUME|(Integer volume between 0 and 100)

This command, processed by the client itself, sets the playback volume to a percentage value. This operation can be completed while no music is playing and like other playback commands requires no authentication to be used.

PAUSE

This command, once again processed by the client node, stops the playback of a song temporarily.

RESUME

This command resumes the playback of a song that has been stopped using the “PAUSE” command.

STOP

This command stops the playback of the song and discards the song data, meaning the song cannot be resumed and must be redownloaded if required.

These commands can only be requested by the client, as the server nodes have no functionality permitting user input (though notably other commands do exist that are sent between nodes automatically, as can be seen in the command lines of server nodes). As a result of this system, the client is forbidden from entering any command using the characters “@”, “#” or “\*” in submitted queries.

# Nodes and Distribution

From this point forward, this document refers to the inner workings of the system – the information provided above is sufficient to utilise the program to its full potential. This specific segment will discuss the nodes that make up the system and how they communicate both to the client and between one another.

## Modules

However, before discussing the nodes themselves, it is important to note the approach of the system towards functionality. The system at large uses an object orientated approach to node handling. Each node has a dictionary element that contains the different “modules” applicable to this node; These modules each consist of a class with their own methods and variables that make up the functionality of the node – with a dictionary element within the class storing a number of access codes and their tied methods.

When any other node contacts a system, it does so by providing an access code as the first parameter – for example “ECHO|” – the node then polls its list of modules and passes the request onto the applicable method, returning the result to the socket that requested the command. These modules also have access to two external return methods; “#” which tells the socket outgoing response compiler to return no response and “-1|#” which bypasses the outgoing response entirely. The former of the two is used in functions where the sender is known but no response is expected for the request, whereas the latter is used for instances where the sender is an unknown node and therefore messages cannot be routed towards them.

Authentication Module

This module stores the locally stored username and password data and provides an “AUTHGRANT” key when provided with a valid username and password combination. This module also handles the following commands: “LOGIN”

Control Data Module

This module stores data solely applicable to the control node, including coordinated data between nodes such as the songs across the system. This module handles the following commands: “\*GETMUSIC”, “\*COLLATEMUSIC”, “PLAYLIST”.

Dictionary Module

This module handles the dictionary functions, used for demonstrating the capabilities of the system. It stores the local dictionary data for the applicable node and uses the “DICT” and “DICTADD” commands to report and modify this data.

Echo Module

Similar to the last module, the echo module handles the data relevant to the echo functionality, allowing for automatic returns of a supplied argument via the “ECHO” function as well as return of all echoed data using the “ECHODUMP” command.

File Send Module

This module allows for the reading of music into a hexadecimal stream, to be sent to the client and interpreted using the “Music” module. This module initially only has access to the “\*ROUTEMUSIC” command, but upon initialisation appends the “!PLAY:(Song name)” commands for each available WAV file on the node.

Heartbeat Module

This module, which is used by every node in the system, allows for temporary connection to a specified IP and port address, allowing for verification of the existence of a node at a specified location. This is primarily used in the initial setup of a node, in which it attempts to discover the first available port on its own device by sending ping requests to each port after 50000 in sequence, though there are frequent uses of heartbeat functions for many other reasons. This module has no internal commands and is instead called directly where appropriate.

Load Balancer Module

This module handles the balancing of clients between members of the system, including that of its spawned nodes as well as its own client load. Taking in frequent updates of a nodes “children” it dictates when a new node is needed on the IP (in the case of a node reaching its limit) as well as when newly connected clients need redirection to another node. In addition to this, it stores the number of clients that should be categorised as the maximum amount for both its child nodes as well as itself and is contacted when another node requests a client redirect. This module handles the “\*LOADUPDATE”, “\*CANACCEPTLOAD” and “\*REDRES” commands.

Load Reporter Module

This module handles the reporting and storing of connected clients to a node and works closely with the load balancing node on a network to ensure that it has the optimum number of clients at all times. When receiving a new connection or losing a connection, this module sends a “\*LOADUPDATE” request to the load balancer, updating its dataset of the connected clients on a node. This node also makes use of the “\*GETLOAD” command, allowing the reporting of its own load where applicable.

Music Player Module

This module, applicable only to the client itself, functions differently from most other nodes – as it does not accept commands from external sources but instead handles commands from its master client node. When called internally, this module takes an decoded hex stream representing music data sent by the file distribution module and makes a temporary file in the “cache” folder, to be read by the audio player library before being automatically deleted. This module handles the “VOLUME”, “RESUME”, “PAUSE” and “STOP” commands for the client and makes use of threading to allow music playback without interrupting the services of a module.

Spawner Module

This module allows for the spawning of subprocesses of other nodes on a local machine, permitting a master node to spawn up another node where applicable. This module has no attached commands yet has one of the most important functions in the system – allowing the distribution of functions. This module does not handle the logic behind node spawning itself, instead relying on a calling object to process the data in its stead.

Client Input Module

This module, despite not having its own dedicated file, has the important function of allowing the constant input of client data into a system, while also managing the cases in which a client may not input data. Using its own thread, the class records the input of data, stores inputs that have been made, and generates a flag to the system to allow the appending of data into the outgoing stream when appropriate. This module also handles the storages of authentication data on the client-side, appends said authentication data to outgoing messages to allow detection of an authenticated user and finally checks for invalid characters such as “@”, “\*” or “#”. This module does not have any attached commands due to its nature.

Control Node Networking Module

While arguably not a node due to not being within the module set, nor having its own class, this module plays the important role of handling control node communication simultaneously with the other functions a control node must handle. This threaded function constantly heartbeats the control node port of all IPs listed in the \_ConnectionList.txt file, checking for active connections and registering itself on the remote node when appropriate. In addition, this node reports the available WAV files on the device to a contacted node, allowing the use of the “PLAYLIST” command. This module requires its own threading due to the blocking nature of a heartbeat command, and its necessary high timeout value – manging control node distribution in this way permits the rest of the system to continue functioning while control node contacting is handled. As with similar modules, this module has no attached commands.

Thread Handling Module

Despite its outdated naming convention, the thread handling module organises the writing and reading upon connections to external nodes, as well as the accepting of new connections where appropriate. Due to the importance of this function, this module is one of few that is appended to all nodes on the system. Internally, this module keeps a queue of all incoming and outgoing commands, for which the main node file can pop data from the incoming commands and interpret the data – after which it appends its response to the outgoing commands with the appropriate socket name to route the data towards. This module has no attached commands.

It should also be noted that upon connection to a node all nodes will keep a database of the commands the external node may process, which is shared during the handshaking procedure. When attempting to send a command, most nodes (except in rare cases concerning the control node) will automatically route commands to a known node that can process the request using their own internal routing system, bypassing the control node and therefore reducing the traffic said node receives. This list is kept constantly updated and will remove and append data as connections are lost and made. When a command with no known route is made, the data will be redirected to the connected control node – either providing an “unknown command” response or the spawning of or connection to a node that can handle the request

Additionally, there are three unique symbols that identify special actions that must be taken within the system. Two of these represent commands that cannot be input by a client and are automatically sent when required, even being blocked out by the client command line to reduce the amount of “spam” applied to the console. These are as follows:

* **@** - representing a system command. These commands are outside the scope of any module, meaning all nodes using an unmodified script can process these requests. They often refer to automated system requests, such as handshaking procedures and redirection requests and are in almost all cases blocked out by the client command line. Additionally, clients may not enter the “@” character to avoid attempts to manually operate sensitive system calls.
* \*- representing a server-side command. This symbol handles communications between nodes that are module specific (unlike the prior-mentioned “@” command) but are not usable by the client. This often means the coordination of data or the use of commands that must follow a specific sequence, in which invalid uses may cause issues
* **!**/**:** – representing a play command. This symbol, which is only applicable to one command - !play: , is the only unique symbol that can be used by a client. This symbol designates a dynamically loaded command. As discussed more in the nodes segment, file distribution nodes may have access to different datasets – as such, they will create their own commands in the format “!PLAY:(Song name)”, which, as it is stored as a command, allows music playing to be routed towards the file distribution node that has access to the appropriate song.

These symbols are used in the operation of the following functions:

@REG

Creates a new contact between two nodes on the network – providing the receiving node with sufficient data to register the contacting node in their internal list of known nodes, storing data such as their node type, IP and port as well as a list of commands they can process. This is the first segment of the handshaking procedure between nodes, and the @REG call is always sent by the contacting node and processed by the receiving node, the latter of which will then respond with a @REP call. This command is also automatically called by a newly created node to initiate a connection with the node that spawned it.

@REP

The response to a @REG call – a sender provides its own contact data to the node receiving the @REP call, allowing the registration of the node data within its own known node’s dataset. Unlike @REG, this command does not return any response data, as the network works on the assumption that the data has been received. Due to this “no-response” nature, the @REP call also is used in other areas, particularly the control node networking as a way to register data in a one-way fashion.

@DIR

This command represents a redirection request. This command, which takes parameters including an external IP and port, sends a request to a node to process a command while returning the data towards a new connection. This serves two purposes; Connecting nodes for the first time thereby allowing them to contact each other directly moving forward and preventing the client from having to re-enter a command when they make a request that would start a connection with a new node. Due to the automated nature of the @REG and @REP calls, this command will often occur concurrently with the registration of a node to a previously unknown node.

@DIRNC

This command, which has very few uses in the system, serves the same function as the @DIR command with the caveat that the node receiving the @DIRNC does not attempt to create any contact with the node referenced in the request, preventing its registry. This is used when creating a redirect request for a client to an external file distribution node, a process that would normally create a connection between the client and the external control node, which could cause unforeseen issues – by stopping the control node from contacting this client, the client only makes contact with the file distribution node the external control node either spawns or directs the request to.

@CLOSED

This command is automatically generated internally when the thread handler module detects a closed socket connection – it requests that the node in question purge the data of the closed connection from its known nodes dataset. This allows the node to recognise that this node no longer exists, allowing it to make decisions such as when to spawn new nodes or when to redirect clients without leftover data from dead connections.

@NOSPACE

A variant of the @CLOSED function, sent to a client by a control node when there it has reached the limit of connected clients. This forces the receiving client node to cut the connection with the node in question, meanwhile the control node who had denied the connection attempts to contact its peer control nodes for a node to accept the client connection. If an available node is found, the external control node generates a connection to the client, if not, the original control node initiates a new connection to the former client despite exceeding its maximum client limit.

@FIL

The @FIL command represents the download of a hexadecimal file being completed and decodes the hex stream before passing the stream into the client music player, hence creating a cached file to be played to the client. This command is sent by the file sending module and is only to be processed by clients.

@AUTHGRANT

This command is sent from an authentication module to be processed by a client with the intention of passing the authentication key to the client, thereby allowing the client to process commands that require prior authentication.

@AUTHDENY

Similar to the previous command, the @AUTHDENY command is sent by the authentication node in order to notify a user that they have entered invalid login information – it comes with no authentication key and simply tells the client command line to print an error message referring to the invalid login data.

\*GETMUSIC

Contained in the control data module, this command returns the list of music available to a contacting non-client node, for the purpose of creating a database of all active music on the system as well as the devices they belong to. When received, this command will generate a \*COLLATEMUSIC command to be processed by the node that sent the original \*GETMUSIC request. This command is automatically sent by a control node when it initiates a connection with a previously uncontacted control node.

\*COLLATEMUSIC

This command, with arguments representing the available music files from a sender node, appends the music data received by a node to which a \*GETMUSIC request was sent into the internal music database. In practice, this allows all control nodes to keep an active dataset of all music files available on the system, along with the control node that handles these files. This data can then be used for redirection of clients as well as producing the results of the PLAYLIST command.

\*ROUTEMUSIC

This permits the return of hexadecimal music data to a contacting node, in the form of a @FIL command. This command is automatically generated by a control node when it needs to redirect a client towards an external music file on the network – therefore requiring a route to be made between the client and an external file distribution node.

\*LOADUPDATE

Sent by a load reporting module and interpreted by the load balancer, this command requests the receiving control node with an attached load balancer update its dataset containing the client load on a node, this allows it to determine if new nodes need to be spawned when a new client attempts to utilise a command for which a node already exists. Additionally, it permits previously occupied nodes to report they are now able to accept new connections. This is an automatically generated command that is called whenever the load reporter module discovers a change in its connected clients count.

\*CANACCEPTLOAD

This request is made by a control node and sent to other known control nodes sequentially when it receives a client connection when it has reached its maximum connected clients. It requests a response on if the external node can take the burden of the client connection. If the contacted node can accept the connection, it will respond with a \*REDRES|Y command after which it creates a new connection to the client in question. If the contacted node cannot accept this connection, it will send the \*REDRES|N command to tell the external node it too has reached its limit of clients, allowing it to either contact the next known control node or accept the connection itself.

\*REDRES

A simple command that sends a response designating the response to a \*CANACCEPTLOAD call – it simply returns a Boolean response to tell the node that sent a \*CANACCEPTLOAD request if it can or cannot generate the connection. Additionally, this includes the index of the subject client node as previously supplied in the \*CANACCEPTLOAD call – allowing it to move through the sequence of known nodes in the redirect procedure. If a “Y” response is received the external node has accepted the connection and the receiver will drop the client redirect data from its queue of clients to redirect, in the case of a “N” response, the node will attempt to contact its next known control node – or if it has reached its limit will accept the connection itself.

\*GETLOAD

This command permits the return of the client load experienced by a non-control and non-client node, allowing a direct load request if required. This command is largely depreciated in favour of the \*LOADUPDATE command.

!PLAY:(Song name)

As previously discussed, this request is handled by the file distributor and sends out a @FIL request with the hexadecimal stream of the music to be played. Of note is the fact that each !PLAY command is actually recognised as a different command based on its song name – for example a node with Cascade.wav and Moonsetter.wav will append the commands “!PLAY:Cascade” and “!PLAY:Moonsetter” to itself.

## Nodes

The nodes within the system each operate in a similar basis, using a read in and write out structure, with the difference that each node has its own modules that determine what commands it can process. As these modules have been previously referenced, the following list of nodes on the system will refer to the appended modules for the system, by which this document can be used to reference the applicable functions of the node.

Control

The central node of a system which is always spawned on the 50001 port of a device. Only one control node can exist on a device and the control node known by a client will always be the contacted node if the client has no listed handler for the specified request, after which the control node will attempt to find a node that can be used to handle the command. Control nodes are expected to keep a constant connection to all other active control nodes in the system, as well as nodes it has spawned and clients it is expected to handle.

Modules: Thread Handler, Heartbeat, Node Spawner, Load Balancer, Control Data Handler, Load Reporter.

Client

The only node in the system that has the capacity to handle manual input – it serves as the only way that a user may interact with a system directly and requires an active control node link to process commands. When a control node is known, it may route commands to a known node handler, or in the case of a command with no known handler, routed to the control node directly so it may redirect the node if applicable. A client node will poll all the IPs in the \_ConnectionList.txt file and connect to the first module that it is able to get a successful heartbeat response towards.

Modules: Thread Handler, Heartbeat, Client Input Handler, Music Player.

Echo

A simple node designed to only handle echo module requests.

Modules: Thread Handler, Heartbeat, Load Reporter, Echo Module.

Dictionary

Much like the echo node, this node is designed to purely handle the data relevant to the dictionary module.

Modules: Thread Handler, Heartbeat, Load Reporter, Dictionary Module.

Distributor

The primary node in the music playback system, which handles requests to provide other nodes with hexadecimal music streams to be played back using the music player module. It is the only node that may accept a connection from a client not connected to its master control node.

Modules: Thread Handler, Heartbeat, Load Reporter, File Sender Module.

Authentication

A node with the sole purpose of comparing a login request against the locally cached available login dataset. It is required to allow a client to make more complex requests and is the only node that a client can request the spawning of when they are in a non-authenticated state.

Modules: Thread Handler, Heartbeat, Load Reporter, Authenticator Module.

## Load Balancing

As mentioned in the previous segments, the network has the capacity to balance client loads between its components, as well as redirect clients to external nodes when the control node has no capacity to accept a request. This is conducted using the previously mentioned load balancing and load reporting modules.

Load balancing is conducted primarily in two manners: Functional node replication and client redirection. Each of these are handled by the control node itself, as it is the only node with the ability to handle load balancing other than simply reporting its own load.

In functional node replication, a control node is granted permission to spawn duplicate nodes of a given type when a node reports it has reached the maximum value permitted by the load balancer. This results in the spawning of a new node which the requesting node is directed towards. Each functional node (a non-control and non-client node) has a limit of two clients; however, this may be edited within the scripts as required.

In client redirection, when a control node itself reaches the limit of clients it expects to connect, the control node will attempt to contact external control nodes in order to find a control node able to take the client connection. If no control node on the network is able to accept the client, the node will default to accepting the connection outright – bypassing its own maximum. This prevents there being a limit of clients able to access the network concurrently, at the cost of allowing potential network slowdown. Each control node has a limit of 3 clients per system, though much like the functional load balancing, the internal scripts can be modified to change this maximum as required.

# External Libraries & Acknowledgments

Due to the nature of the software and the focus on distribution rather than music playback, the music module makes use of the audio player library provided by GitHub user mjbrusso (<https://github.com/mjbrusso/AudioPlayer>). The implementation used in this project uses a modified version of the original source code, changing the system to use WAV files over MP3 files, as well as removing the ability to use Linux and Mac devices as was available in the original library – done due to reduce the amount of space used by this module. The thread handler script too makes use of a modified version of a provided script for client server connections provided in this module, used with permission.