# Distributed DJ

**Project 2 Report**

***Repository***

*github.ugrad.cs.ubc.ca/CPSC416-2018W-T1/P2-DistributedDJ*

**Project members**

Eric Chan (t0c0b)
Leo Chang (o6x9a)
Ryan Koon (z6y9a)
Craig Yu (x2g9)

**Introduction**

DistributedDJ is a distributed web radio that uses a peer-to-peer network where clients provide and stream songs that they locally own. Clients share the metadata of the songs they are willing to stream and are capable of voting amongst themselves to decide on the next song to play. The client with the selected song will become the host and stream the song to all the clients in the network. Playback of music is synchronized like traditional radio. Clients can join and and leave the network. They form a complete graph.

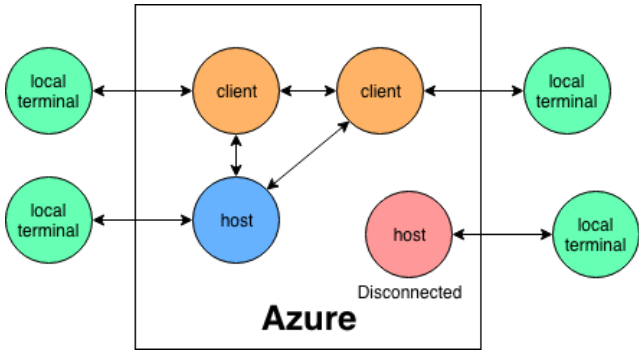
**Nodes Overview**

There are 2 types of nodes in our system: clients and terminals. Clients create a peer-to-peer network and are deployed on Azure. Each terminal is connected to exactly one client. Each terminal upon joining, specifies the songs the user is willing to share and uploads the files to their connected client. After the client joins the system, the terminals can issue votes via its client and play the stream on its machine's audio output hardware (e.g. speakers).

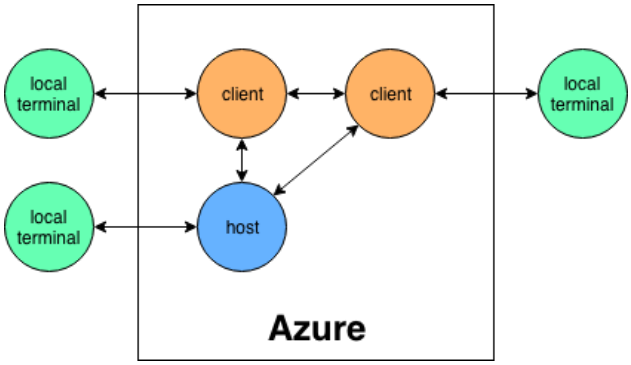
**Client Nodes**

Peer-to-peer clients form a complete graph. Each client has a configuration file that indicates which client to connect to when it initializes. Clients communicate with each other by broadcasting information to all known peers. When songs are uploaded to a client, the song metadata is sent to all peers so that they can update their copy of the list of songs they can vote for. When a client joins the network, its IP address is broadcasted to all the clients in the network. This allows information from peers to be sent to the new client and for the host client to stream its song to the new client. Each client will have a default song on startup which will allow it to stream at least one song in the event that no peers can be connected to. This also allows a client to stand alone and vote among its own song to stream without peers. Votes from each client uses a 2-phase commit protocol instead of directly broadcasting to all its peers.

A client is disconnected if it cannot connect to any of its peers or does not have at least one song cached in memory for streaming. A disconnected client can only vote for its own songs. Therefore, it becomes the host and streams its own songs to terminals connected to it (if any).




Should a host client crash or disconnect from the network, the stream will end and the song will stop playing for all the clients. The remaining clients will determine a new host based on the committed votes.




Clients will stop communicating with any peers that crashes or disconnect from the network unless the disconnected peer rejoins the network.

When streaming a song, the host client uses the copy of the song stored in its memory, which was initially sent over from a connected terminal. Therefore, once the song is uploaded to the client, a song can be streamed even if the file on the terminal machine is moved or deleted.

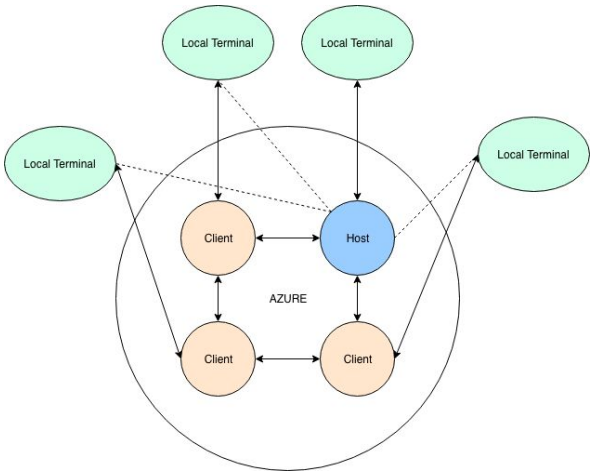
**Terminal**

User interacts through the terminal node on their local machine. The terminal node first connects to a client on Azure to upload all the songs from user's local machine. If the song's name already exist in the network, meaning another user already uploaded the same song, then this song won't be uploaded. The client receives songs and cache them to disk.
A user can interact with a terminal through the following commands:

1. show: requests a list of available songs for streaming
2. start: start playing the stream
3. stop: stop the stream locally
4. [song name]: user can vote by typing the song name when prompt to vote

A terminal can be a *Host Terminal* when the chosen song is uploaded through this terminal. There can be only one *Host Terminal* in the network, it forms a unique pair with *Host Client*.

For example, user A, B, C, D voted for the song VitaminC.mp3 uploaded by user C. User C's terminal becomes the *Host Terminal.* The *Host Terminal* then makes an http call to *Host Client* using the endpoint "/cacheaudio". The *Host Client* will start sending chunks of data to *Host Terminal* while updating a *Cached Chunk* for other terminals to stream from. In this case, Terminal A, B, D will be notified by their corresponding clients to stream from the Host Client(user C's client) using the "/audio" endpoint.




At any point during the song, users can vote for the next song via a terminal. The next song will be selected at the end of the song by tallying latest votes from each client. A user is not required to cast a vote during each song as the client automatically votes for a random song at the start of each stream.
For tied votes, see the **Tie Breaking** Section.
During the playing of a song, if the host terminal experiences a failure, then the *Cached Chunk* won't be updated at *Host Client* resulting the stream pausing for the rest of terminals on the network. This is intended since only songs uploaded by current users can be streamed.

**Voting Overview**

Users can vote for the next song to play out of all available songs in the group of connected clients via their terminal. Clients provide a list of the songs available when the user inputs "show" via the terminal, and the user can vote for any of these songs by inputting the song name. The user can always change their vote after their previous vote has been committed.

The two-phase commit protocol is used to facilitate the voting of songs to stream by replicating votes to all clients in the network. In this protocol, there are two entities, the coordinator, and the worker. Every request to commit by a worker goes through the coordinator to ensure all the workers are ready to commit. Once all workers have responded that they are ready to commit, the coordinator commits and tells all the workers to commit.

We maintain the liveliness of our two-phase commit system by implementing time-outs while waiting for any of the phases. This time-out handling is implemented by signalling via golang's goroutines and channels. When waiting for a response, a goroutine is called which will signal a time-out channel after a reasonable time (3 seconds). We also keep track of the status of the workers in each phase. If prepared responses from all workers return and signal the done channel before the time-out channel is signalled, we proceed to the commit phase and send commit messages to all workers.

An adaptation of the two phase commit for our system is as follows:

*Phase 1 (voting)*

Client - sends *commitRequest* to host client

Host Client - sends *prepareToCommit* to all clients and starts the time-out goroutine

Client - sends *prepared* to host client, then waits

Host Client - receives *prepared* from all clients before timing out

- A not prepared from any client or timing out aborts the vote

*Phase 2 (completing the transaction)*

Host Client - commits the vote by storing it in memory for committed votes

Host Client - sends *committed* to all clients

Clients - commits the vote by storing it in memory for committed votes

For our system, every client in the network is a worker. The host client is both a coordinator and a worker. Clients vote for the next song to play. Voting opens after a new host is selected and closes when the host is finished streaming a song. The two-phase commit process occurs for each vote from a client. Votes are rejected by the host client if a vote is in the process of committing. Clients will retry their votes until it is accepted by the host client.

Clients can change their vote. It will go through the two-phase commit process to update the vote across all clients in the network.

When the host client has finished streaming a song or if a connected terminal issues a skip, it stops accepting votes and notifies all the other clients that the stream has ended. Since all the clients have the same copy of the committed votes, they can independently determine the new song to stream and the new host client. The new host client will become the new coordinator and starts accepting votes to commit on all clients on the network. Clients will automatically vote for a song when the new stream starts but it can be changed when it receives a vote request from a connected terminal. This ensures that there are always votes before a song ends. When a client becomes disconnected and they voted for a song on another client, they will end up with no votes. To address this, the disconnected client will randomly choose a song, and in this case, becomes the host. A random number bounded by the number of songs that can be voted for is used as the index number to select a random song.

**Tie Breaking**

In the event that more than one song has the most votes, we will choose the song whose hash comes first alphabetically.

**Case where multiple clients has and can host selected song (Not implemented yet)**

In the event that more than one client can stream the same song based on the song metadata, we need a consistent process of determining which host will stream the song. We will use the following order of conditions to reduce the the list of eligible clients to a single one. First of all, the client that just finished streaming will not be eligible. Since there can only be one host a

time, there will be at least one other eligible client. Next, we examine all the songs that are voted for in the current round. Assuming that each client can stream as many of the voted songs as possible based on the songs they can stream, we determine how many peers each client will be able to satisfy. Only the clients that can satisfy the most peers will remain. Lastly, we convert the remaining client IP addresses to integers by removing the dots. The client with the greatest IP as an integer will be selected. By now, there will only be one client remaining to be selected as the next host. Since the information (i.e. voting information) we use in our selection process is committed via the two-phase protocol, it is consistent across all clients. As a result, each client will select the same host client independently.

**Streaming Overview**

The *host client* is the client that streams a song to all terminals in the network. At a given time, there is at most one host client on the network.

The system supports mp3 music files. Songs are music files that originate from a terminal's machine. Music files are uploaded to a client on established connection and stored in the client's disk. A host client loads a song from the disk to its memory, to stream to the terminals that connect to it.

A client that is unable to connect to any of it's configured peers or if there are no configured peers, will end up voting for its own songs and become the host in a network of one client. A client that initializes and is able to connect to a peer will be given a copy of the commited votes, available songs' name and the current host on the network. The current host on the network will be notified of the new client by the new client's peers.

Should the host client crash in the middle of a stream, none of the clients have a copy of the entire music file to continue playing the song. Instead, the lost of a peer is detected by each client. A new song and host client is selected in the network with the system's voting process. Since each client has a copy of the committed votes and only the host client can update or add votes to each client, every client has a consistent copy of the votes. Independently, they can determine the next song to stream and the new host client.

When a song is finished streaming, the host client will notify all the other clients. Each client will then notify their connected terminal of the new host to stream from. The terminal will update its streaming url using the new host address. Unfortunately, local terminals are likely behind a NAT firewall, so we will change this to a polling mechanism. The terminal periodically checks from it's client whether the host has changed.

Latency is an issue when streaming over the network. To address this problem, we created a double buffer using a circular array that can hold two samples or chunks at a time. We alternate between reading (i.e. playing) and writing (i.e. caching) to each index of the array. Every http request for the next chunk or sample of the music will cache the response and playback the last cached response. The sample size of each chunk is large enough so that the requests are

spaced out at an interval that does not saturate the network. Turning on accelerated networking on our Azure virtual machines greatly reduced the upload time of songs from a local terminal.

We use the PortAudio and mpg123 libraries to decode and stream mp3 music between clients and from clients to its connected terminals. To interface with these libraries we will use these Golang libraries:
https://github.com/oandrew/go-mpg123
https://github.com/gordonklaus/portaudio

## Failure Recovery

### General node failure

Clients that aren't the host failing while not in a two-phase commit stage does not affect other clients and will regain state upon re-joining a peer.

Host clients that fail will prompt the other clients to select a new host and song based on votes.

### Voting process (two-phase commit)

Failure of client after *prepareToCommit* is sent by host client

- Host client aborts the vote after a set timeout for the worker to reply
- Clients do not persist state and will follow the join process when it restarts and initializes

Failure of client after it replies with *prepared* to host client

- Clients do not persist state and will follow the join process when it restarts and initializes
- Other clients can safely proceed to commit when host client sends committed afterwards

Failure of host client after client sends *commitRequest*

- Client aborts after timeout to receive *prepareToCommit*

Failure of host client after client sends *prepared* or *aborted*

- Client checks if any of its peers have committed the vote
    - Commit if any peer has committed the vote
    - Abort if none of the peers has committed the vote
- A new host client will be selected among the remaining clients by tallying the committed votes
- The failed host client will restart as a new client joining the network

## Other Work In Progress Items

- We encountered a few bugs regarding the communication between multiple clients and terminals on Azure that will need to be fixed
- As mentioned previously, the terminals are not reachable from the Azure clients so we will need to change the implementation of the terminals to poll the clients to determine when the stream has ended and determine the new host address

## Testing

- We have set up a network of clients and terminals locally and on Azure to validate its functionality

**Demo**

- Demonstrate *normal operation* of your system (no failures/joins) with at least 3 nodes.
    - We will start up 5 nodes on Azure VMs
    - Show that they all become connected to each other and stream the same song
    - Show that voting via the terminals uses 2PC and committed to all nodes
    - Show that "winning" songs to stream next are indeed selected from the most votes
    - Show that tie breaking for voting works
- Demonstrate system can survive at least 3 node failures
    - Terminate 2 nodes during streaming
    - Other nodes should be unaffected
    - Let current song finish playing or trigger a skip from a terminal
    - Voting process should succeed and choose new host without the 2 nodes without their votes or songs
    - Terminate the host
    - A new host is selected and all terminals now stream from the new host
- Demonstrate system can *join and utilize* at least 3 new nodes
    - Restart 3 nodes stopped in the previous part
    - They are able to listen in to current stream, catch up on votes, execute votes, and host.

**References**

Adapted the two-phase commit process from:
https://www.cs.ubc.ca/~bestchai/teaching/cs416_2018w1/lectures/lecture-nov8.pdf