# Swarm Player - design / project plan

## Milestone 1: Auto join and time sync

### Summary

When the user switches hotspot, it connects-disconnects automatically.

Non-planned feature: time sync.

### Requirements

Implement auto join procedure:

- The administrator configures public entry page: sets main app URL and hotspot name
- The user opens the public entry page in a browser
- The public entry page asks the user to join the private wifi hotspot
- The public entry page indicates when the user disconnects from the internet
- The public entry page indicates when the user reconnects to the internet (the user selected wrong hotspot, the server is down, etc.)
- The public entry page indicates when the user connects to the private hotspot
- Redirects to the main webapp
- The user is playing with the main webapp
- The main webapp indicates when the user disconnects from private hotspot
- The main webapp indicates when the user reconnects to the private hotspot (pobably only the server went down for a while)
- The main webapp indicates when the user connects to the internet
- The main webapp redirects to the public entry page

It’s possible that the user’s device connects both the internet and the private hotspot (has Ethernet and WiFi or 2x WiFi), in this case “re-connecting” to the internet can be very quick.

Hotspot switching should be emulated for development phase.

### Server

Implement minimal functionality, just to support client auto join feature:

- written in Rust
- clients can connect with websocket
- receive heartbeat or user input from client
- display client requests

### Client

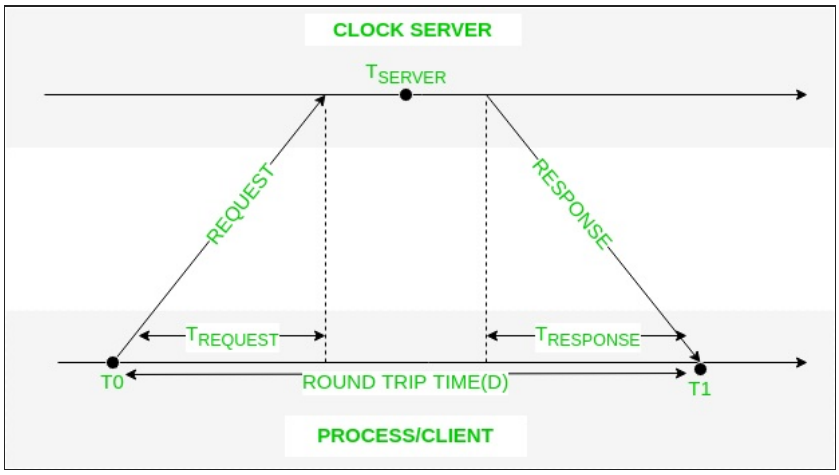
Implement auto join:

- implement hotspot detection in public entry page
- the switch between entry page and main app should be “invisible”
- main webapp should have only minimal functionality
- implement hotspot switch detection in main webapp

### Time sync

Was not planned in this milestone, but finally implemented for its simplicity.

The client uses Cristian's algorithm to synchronize time to the server.




The server only tells actual timestamp for the client's request.

Test features:

- the server simulates slow network, by sleeping the same amount before and after getting timestamp,
- the client's timestamp can be shifted to emulate clock skew.

# Milestone 2: Synchronized broadcast

## Summary

Clients execute commands simultaneously, synchronized with each other.

The pre-requisite is met: time synchronization is already implemented.

## Server

- The server receives message from external sources
- The server adds timestamp to the message: time of receipt plus official lag
- The official lag is cca. 50..100 ms: long enough for all clients to receive it in time, but short enough not to be able to hear the delay
- The server broadcasts all messages to all clients, no filtering applied in this milestone yet

## Client

- Receives message, adds it to a container
- Retreives and processes item at the requested time
- Handle overdue messages
- Performs some simple visible action, e.g. sets background color

# Milestone 3: Fix issues arisen meantime

Make enhancements which can no longer be postponed, and fix non-functionality problems appeared.

## Web proxy

Configure `nginx` for web server and proxy functions.

- Static webserver on port 8000

- Make HTTPS proxy to static webserver (port 8000)
- Make WSS proxy to the server (port 8080)

## Client features

- Reliable audio playback even on Samsung Galaxy A22: measure audio lag and try to correct it
- Avoid screensaver: browser API and invisible video
- Web client update: reload on request
- Logging to server, retry if offline
- Report values to server (admin mode, audio lag), re-send on server restart
- Fix GUO layout for landscape mode

## Server

- Implement logger
- Implement client report messages
- Fix packet stamp

## Logistics

Get some devices.

# Milestone 4: Admin mode

Exactly one of the clients should be put in admin mode. The role of this client:

- control the show,
- play the song: send notes to server,
- receive external MIDI and send to server,
- display dashboard,
- diaplay any other show elements,
- play *Master Channel*.

Milestone features:

- enter admin mode,
- show dashboard.

## Client

- Show active clients with ID and info on the dashboard,
- handle connect and disconnect.

## Server

- Collect dashboard data,
- push data upon change.

## Dashboard data

- client ID
- clock skew
- audio lag
- channels

# Milestone 5: The channel concept

The Player should request the server to split up clients to *Channels*, which are playing different things.

## All about Clients

We can make the following observations about the clients:

- There is a *Master Client* with the audio output is connected to the main amp. The other Clients, *Public Clients* are the mobile devices of the public.
- There will be a guaranteed set of Public Clients (some own devices, friends' devices), 5-8 such devices are expected. We can count with additional 15-40 devices from public, so broad estimate is 5-50 Public Clients. The infrastucture should handle 100 devices. As a realistic value, we expect 20-30 Public Clients.
- Some Public Clients may not be audible: their volume is set too low, even muted, they are at acoustically wrong place, they are outside of the performance room.

## The problem is: addressing

The question is: what should Public Clients play? How should they *addressed*?

The two ends of the scale are that either all clients execute the same commands, or all execute different ones.

Exectuting same thing on all the Public Clients is a safe game. There's no problem with the number of clients, with joining and leaving ones - but as a show, it's boring.

Executing unique commands on all clients has several issues:

- Some Public Clients will be disconnected in the middle of the show. This may cause some gap in the song.
- Some new Public Clients will join in the middle of the show. It's not as serious problem, but it's hard to add them instantly to the show.
- It's hard to write a song for unknown number of clients.

The solution is somewhere halfway, using limited number of *Channels*, which the clients can be assigned dynamically.

## Addressing modes

Currently two addressing schemes are defined:

- *Broadcast*: send command to all clients (trivial),
- *Channel*: send command to Clients belonging to the specified Channel (explained below).

Other addressing modes are expected in the future, e.g. location-based, individual etc.

## Channel addressing overview

- At any point of the show, the *Player* can set the number of *Public Channels*. Upon this request, the server rearranges Public Clients equally into Public Channels.
- There's a special *Master Channel*, which is always exists, can't be deleted, only the *Master Client* is assigned to it. So the Player can address it as the same way as Public Channels.
- The Player addresses all commands to a single Channel, or more Channels.
- The server can rearrange Clients amongst the Channels during the show, as clients detach or new ones attach.
- There are limits for maximum number of Channels and minimum number of Clients per Channel.
- A Client can be assigned to multiple Channels. This is useful when there're too few clients.
- Public Channels can be assigned to Master Client. So, in extreme case, the Master Client itself, without any Public Client, is able to run the whole show.

## Message types

This is a short intermission about message types. There are the following types:

- Client-server Technical Messages: the client sends a request, the server responds to it. There's one such message pair is implemented: the *Time Synchronization* reqeuest-reply. Taxonomically client socket connect and disconnect events belong to this message type.
- Player Content Messages: the Player sends a channel or broadcast message, the server sends it to the addressed clients, which execute it synchronized. There's one message implemented if this type: *Color Change*. (the implementation does not contain addressing, the server broadcasts it to all clients).
- Player Control Messages: the Player sends a control request to the server, which executes it immediately. *Set Channel Number* is such message, not implemented yet.

## The Note Off problem

Sending messages to Clients looks straightforward:

- the message is addressed to one or more Channels,
- the Clients are assigned to one or more Channels,
- so the message should be sent to these Clients.

It's only true, if the messages are *independent*.

The exception is the *Note On - Note Off* pair, it can cause the following issue:

- given a Client, it's assigned to Channel 1,
- a *Note On* message receives for Channel 1,
- the Server sends it to the Client,
- the Client starts playing the note on Channel 1
- a new Client connects, or some disconnects,
- therefore the Server re-organizes the Channel-Client assignements, and as part of it, un-assigns Channel 1 from the Client which is playing the note,
- so when the *Note Off* arrives for Channel 1, the Client, which is not assigned to it anymore, doesn't receive the message, and never stops playing the note.

Incomplete solutions:

- Combine *Note On - Note Off* pair into a single message. No server-side implementation required, btu it only works for pre-recorded score.
- Turn off all pending notes, when a channel is unassigned. It's also can be implemented on client side. May cause glitch in live performance.

The proper solution:

- track pending notes on server-side, for each client,
- upon a *Note Off* message, instead of actual channel assignment, use this information for selecting Clients to send the message to.

The list of pending notes can be constant sized for each Client: number of channels (4, may vary) by number of possible pitches (128, MIDI limitation).

Workaround:

- *Note Off* messages should be sent to all Clients,
- the Clients should track pending notes,
- the Clients should ignore unnecessary messages.

This workaround generates unnecessary network traffic.

## Server requirements

Channel addressing requirements for the server:

- The server can split Clients into Channels.
- One client may assigned to more Channels. This might happen when the number of Clients is low.
- When a new Client connects, the server assigns it to a channel.
- When a new Client connects, and it's added to a Channel, another Client, which is assigned to this Channel and also to another one, might be removed from this Channel.
- The maximum number of Channels is 4 (subject to change).
- Each Channel should have at least 3 Clients assigned to (subject to change).
- When a Client disconnects, thus removes from a Channel, the server may move (remove from another Channel and assign to this) or add (keep it on its actual Channel, and also assign to this) one Client to the Channel, in order to keep the Channels in balance.
- When a synchronized command arrives from the Player, the server adds a timestamp to it (already implemented), and resolves the address (now: Channel), translating it to the list of Clients.

## Player requirements

Minimal player functions should be implemented in order to test server channel addressing functions.

The Player should send - Set Channels command, - a prototype Channel command (e.g. Set Color).

# Notes

## Song

- Sonyi.mod
- Banana song

## Sound engine

- The easy way: use tone.js
- Hard way: native webaudio