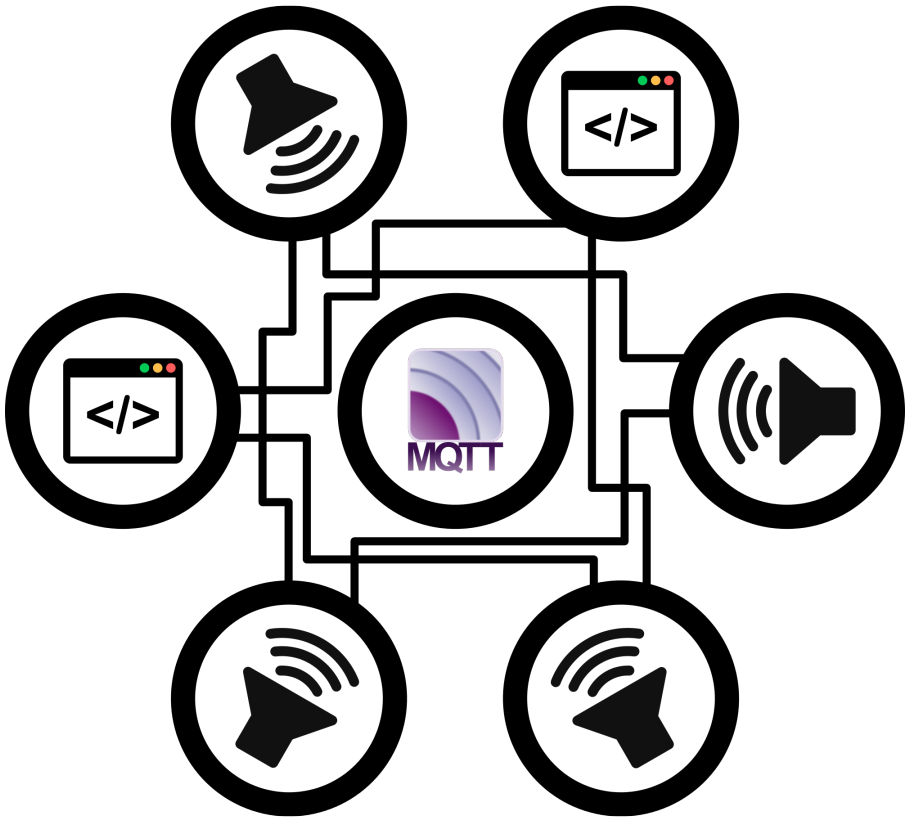
Dynamic Network of Speakers

With MQTT



Ingmar Delsink,
Menno van der Graaf,
Brian van Zwam

HAN University of Applied Sciences
Embedded Systems Engineering

May 2016
v1.0

Summary

This project can be described as a surround sound system. There is support for objects that make sound and support for physical speakers that can play this sound. Via a web interface, the speakers and objects can be drawn into an area. If, for example, an object is closer to speaker A then to speaker B, then speaker A will produce more sound of said object in comparison with speaker B.

A good example of a use case for this system is to use a Multitrack song and use a multiple of speakers. Then each part of this song(drums, bass, guitar, etc..), can be dragged to any speaker. If this would be setup correctly you, the user, can walk between the speakers in real life and enjoy these sounds as if you were in the middle of it all.

The client is written in C++, The website is written in JavaScript, HTML and CSS and the documentation was made in LaTeX.

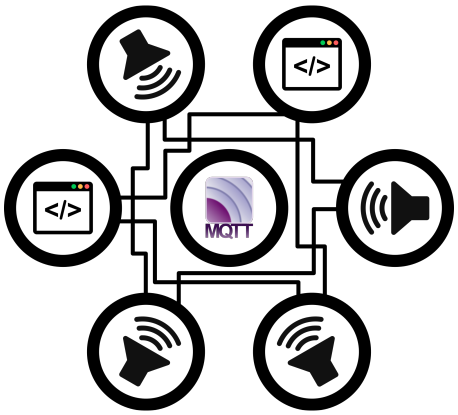


Table of Contents

1 Introduction

1.1 Dynamic Network of Speakers

This project holds a Dynamic Network of Speakers.

The goal for this project is to create a system where a user can setup speakers and tell via a website what sound comes out of each speaker. All the speakers are connected via the internet and are talking with each other via a protocol called MQTT.

This project can be described as a surround sound system. There is support for objects that make sound and support for physical speakers that can play this sound. Via a web interface, the speakers and objects can be drawn into an area. If, for example, an object is closer to speaker A then to speaker B, then speaker A will produce more sound of said object in comparison with speaker B.

In this project an object represents an Ogg music file. A good example of a use case for this system is to use a Multitrack song and use a multiple of speakers. Then each part of this song(drums, bass, guitar, etc..), can be dragged to any speaker. If this would be setup correctly you, the user, can walk between the speakers in real life and enjoy these sounds as if you were in the middle of it all.

1.1.1 Folder structure

The folder structure used in this project and what to expect in each folder:

- `./client/`: Contains the client source files.
- `./config/`: Contains the config file that us used for configuration between website and client.
- `./docs/`: In this folder resides the documentation of the project. See the README in that folder for further information.
- `./misc/`: Some general files that are used in the project, like Logo's, font's, etc…
- `./test/`: The tests for various library's in this system.
- `./website/`: The website of this project.

1.1.2 Installation / Compilation

To compile the various systems, this project uses a makefile. The make commands can be issued from root directory of this project. All the build executables are made in a 'build' folder.

Make commands available:

- `make` or `make dns_client`: Build the DNS client
- `make rwf_test`: Test of the RWF lib
- `make audio_test`: Test of the Audio lib
- `make download_test`: Test of the Download lib
- `make data_parser_test`: Test of the Data Parser lib
- `make logger_test`: Test of the Logger lib
- `make jzon_test`: Test of the JZON lib
- `make config_parser_test`: Test of the Config Parser lib
- `make clean`: Clean the build directory
- `make purge`: Delete the Build directory

Dependencies

The client uses a couple of tools: - pulseaudio - pactl - ogg123 - libMosquitto - LaTeX (See documentation)

1.1.3 Credits

All Authors - Listed Alphabetically.

- Ingmar Delsink
- Menno van der Graaf
- Brian van Zwam

Libraries

Libraries used in this project:
Jzon

Author
Copyright (c) 2015 Johannes Haggqvist Jzon is a JSON parser for C++ with focus on a nice and easy to use interface. https://github.com/Zguy/Jzon

Eclipse Paho JavaScript client

The Paho JavaScript Client is an MQTT browser-based client library written in Javascript that uses WebSockets to connect to an MQTT Broker.
https://github.com/eclipse/paho.mqtt.javascript

jQuery

New Wave JavaScript
jQuery is licensed under GNU GPL and MIT licences, and with the MIT licence you could use it in commercial applications.

marked

Author
Copyright (c) 2011-2014, Christopher Jeffrey. (MIT Licensed)
marked - a markdown parser
A full-featured markdown parser and compiler, written in JavaScript. Built for speed.
https://github.com/chjj/marked

Easylogging++

Author
Copyright (c) 2015 http://muflihun.com (MIT Licensed)
Easylogging++ is single header only, feature-rich, efficient logging library for C++ applications.
https://github.com/easylogging/easyloggingpp

1.1.4 Fonts

Fonts used in this project:
Font Awesome

Author
Fort Awesome
The Font Awesome font is licensed under the SIL OFL 1.1:
http://scripts.sil.org/OFL

Xolonium

Author
Severin Meyer mailto:sev.ch@web.de
This Font Software is licensed under the SIL Open Font License,
Version 1.1. This license is copied below, and is also available
with a FAQ at http://scripts.sil.org/OFL

Nimbus Sans L

Author
(URW)++; Cyrillic glyphs added by Valek Filippov
LICENSE
General Public License (GPL)

1.1.5 License

You can check out the full license here

This project is licensed under the terms of the **MIT** license. The MIT License (MIT)

Copyright (c) 2016 Ingmar Delsink, Menno van der Graaf, Brian van Zwam

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the ”Software”), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED ”AS IS”, WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

2 Topics

Introduction and explanation about the topics used in this project.

2.1 Message model

The topics and messages used in this project are described here.

2.1.1 Public MQTT Brokers

Details of brokers used in this project are listed in the table below. Encrypted ports are marked with * and encrypted where a client certificate is required are marked with ** in the table below.

Table 2.1.1: Public MQTT Brokers.

ID	Server	Broker	Port	Web socket
01	test.mosquitto.org	Mosquitto	1883, 8883*, 8884**	8080 / 8081*
02	iot.eclipse.org	Mosquitto	1883, 8883*	80, 443*
03	broker.hivemq.com	HiveMQ	1883	8000
04	test.mosca.io	mosca	1883	80

2.1.2 Root topic

`ESEiot/DNS/`

2.1.3 topic list

This is a global topic listing. All the headers are a topic that will follow after the Root topic. I.e. `root_topic/header`.

Request

The request topic is a topic where all the requests to devices are handled.

- `request/online`: Request a message from all the online devices. The devices respond with their `clientid` to `info/clients/online`.
- `request/info/clients`: Request all the devices to send their device specific information to a topic. Payload is topic to publish to. All devices respond with the JSON string that they have. (See below) If a device has no information, It wont respond. This command is mostly useful for the website to build up the device information list.

```
{
  "clientid": {
    "objectid1": {
      "distance": 0,
      "angle": 0
    },
    "objectid2": {
      "distance": 0,
      "angle": 0
    }
  }
}
```

Info

A general information topic.

Music status

- `info/music/time`: When a device needs to play a music file, it will send the following information to this topic. The name is the name of the music file (see `info/music/sources`) and the number is the total playtime of this music file in *seconds*.

```
{
  "name": 0
}
```

- `info/music/time/position`: The position of the music file in the time. If send, the speaker will adapt their time to this value. If a speaker gets the play command but doesn't know the time, a value of `0` will be used. The name is the name of the music file (see `info/music/sources`) and the number is the position in *seconds*.

```
{
  "name": 0
}
```

- `info/music/status`: The state of the playing music. [`play`/`p`] / [`s`/`stop`] If a speaker joins, it will wait for a command before it starts to do anything.

- `info/music/sources`: Contains the URIs to the music files in JSON format. The client will download these files. The identifier is the `objectid`, because the sound is bound to an object. So the *identifier* is the *name* of the object and the name on the local file system. See below for an example:

```
{
  "name": "uri"
}

{
  "name_is_objectid": "uri_is_uri"
}

{
  "object_1": "http://www.example.org",
  "object_2": "http://www.example.com",
  ...
  "object_5": "http://www.example.org"
}
```

- `info/music/volume`: The master volume. This value will be of a value between `100` or `0`. `100` is the maximum value and `0` is the minimum value.

Clients

- `info/clients/data/xxx`: Global data for sending information of all the online clients to all devices. `xxx` is the name of the website that is sending the data. This is not important for clients, the website uses this to filter it's own data. The client can subscribe with `info/clients/data/+`. Clients that are online and are not **participating** in the active audio network, won't be in this list. Because this is a 2D field, all the clients will contain all the objects. See below:

```
{
  "clientid1": {
    "objectid1": {
      "distance": 0,
      "angle": 0
    },
    "objectid2": {
      "distance": 0,
      "angle": 0
    }
  },
  "clientid2": {
    "objectid1": {
      "distance": 0,
      "angle": 0
    },
    "objectid2": {
      "distance": 0,
      "angle": 0
    }
  },
  "clientid3": {
    "objectid1": {
      "distance": 0,
      "angle": 0
    },
    "objectid2": {
      "distance": 0,
      "angle": 0
    }
  }
}
```

- `info/clients/object/first/xxx`: Topic for sending the x and y offset of the first object. This is used by the website for synchronization of the data. `xxx` is the name of the website that is sending the data.

Device status

- `info/clients/online`: If a clients comes online, it will post its `clientid`.
- `info/clients/offline`: If a client goes offline, it will post its `clientid`.

2.2 General notes

- In this document and all other documents, the word `speaker` is the same as the word `client`.
- Client name (aka the `clientid`) - Speaker: The name for a speaker will be in the form of `speak_xxx`, where XXX is a random number. The minimal value is `0` and the maximal value of this random number is: `999`. Make sure that the devices can connect with its clientname. If not, that name already could be in use.
- The data for the clients in relation to speaker and object, are in complex notation. So from speaker to client, there is a distance and an angle.

- An MQTT client can create topic strings of up to 65535 bytes.
- Parts between `[]` or the use of `*` after a sentence in this document usually mean that information needs to be added. Or that clarification is needed.
- Values used in client settings are as follows: `0` - `xxx` are legitimate values. `>0` i.e. `-1` are ignore values, these say that the client can ignore this value. See Clients

3 Client

In this chapter the client and it's functions are described. The Client exists of the following subsystems:

- MQTT
- Message/Config Parser
- Audio driver
- Relative Weight Factor
- Logger

The aforementioned parts will be described in the following sections.

3.1 MQTT

The client uses the Mosquitto MQTT C++ library. This is an open source client that implements the MQTT protocol versions 3.1 and 3.1.1. MQTT is a publish-subscribe based "light weight" messaging protocol for use on top of the TCP/IP protocol. It is designed for connections with remote locations where a "small code footprint" is required or the network bandwidth is limited. This makes it suitable for "Internet of Things" messaging.

The client and the website use topics to communicate. The topics for the communication are described in the previous chapter. The DNS class has a public inheritance of the Mosquito class. This means all public and protected members of the Mosquitto class can be accessed by the DNS class. The private members cannot directly accessed by DNS class. In the DNS class the Mosquitto functions are been overwritten by DNS's own implementation. The Mosquitto functions are automatically called by the Mosquitto library.

If the client is executed the Mosquitto library calls the function **on connect**. This function published the client-id in the **online topic** and subscribe itself on the following topics:

- **request online**
- **request client data**
- **clients data**
- **music volume**
- **music status**
- **music sources**

If the client is stopped or disconnected the Mosquitto library calls the function **on disconnect**. This function published the clientid in the **offline topic**.

When the client is running and has executed the function **on connect**, It waits on a message from a subscribed topic. If a message appears, the function **on message** is called. This function takes as parameters a mosquitto message struct, containing:

- **int mid**
- **char *topic**
- **void* payload**
- **int payloadlen**
- **int qos**
- **bool retain**

In the function is decided on which topic the message came from, by comparing the **topic** field with the topics defined in 2. A list of actions is implemented for each topic.

MQTT has a function that is called **last will**. This function is called when a client times out, for example if the systems power cuts out and doesn't properly shutdown. The broker will detect this and publish the clients' last will message. In the client this function is used to publish its message to the **offline** topic.

3.2 Message/Config Parser

A large portion of the messages that are send in this application are in JSON string format. Since the C++ spec does not include a JSON Parser, the library JZON is used to parse/compose the JSON string formats.

3.2.1 Config

The user must provide the client with a config file in JSON format which is used to initialize the client. The data that can be specified includes:

- **Name**
- **Log settings**
- **Topic prefix**
- **Broker**
- **Topics**

It should be noted that the config file has a ".js" extension, this is because the same config is also used by the website. The benefits of doing this is that only on file has to change so that everything adapts, instead of changing the config for the website and client separately.

3.2.2 Message

The messages that are supplied in JSON format, are parsed using an iterative feature, ensuring full dynamic behavior. As an example, the message format of "info/clients/data/+" as specified in topics:

```
{
  "clientid1": {
    "objectid1": {
      "distance": 0,
      "angle": 0
    }
  },
  "clientid2": {
    "objectid1": {
      "distance": 0,
      "angle": 0
    },
    "objectid2": {
      "distance": 0,
      "angle": 0
    }
  },
  "clientid_n": {
    "objectid2": {
      "distance": 0,
      "angle": 0
    }
```

}

This message can have any amount “n” of client-nodes, each containing any number “m” of objects-nodes. Jzon implements an iterator for JSON nodes, these can be used with C++14’s great ranged based for loops. This allows the client to convert the JSON message into an map of objects per client, and a vector with all the distances, which in turn are used by the RWF calculations which will be touched on later.

3.3 Audio driver

Since this project requires the client to play audio files, a small class was implemented to do just that. To keep development time low, a couple of cli tools were used to achieve functionality: vorbis tools’ `ogg123` and pulse audio’s `pactl`. An instance of this class is created for each audio object. Playing the audio is straight forward, execute an instance of `ogg123` with a file path and save the PID. If the music has to be stopped (as per user input) simply call `kill([pid])` and the music stops playing. The volume regulation is less straight forward. It uses `pactl list sink-inputs` to list all applications using pulse audio as output. This list is indexed and contains information such as application name, and process ID. So a loop checks each index in the list for the `ogg123` PID. Once the correct index has been found, it can then be used to set the stream volume with `pactl set-sink-input-volume [index] [volume]`.

3.4 Relative Weight Factor

This system is based on a virtual object that will create sound like it is in some 2D space. This library calculates the volume level that a speaker needs to be in comparison with all the other speakers relative from an object.

3.4.1 The problem

To explain the inner workings lets look at an example. In figure 3.1 is a simple setup. Two speakers “Sp1” and “Sp2”, a virtual object, “Obj”, that “creates” the sound and the real life “user” that listens to this. The speakers are the actual devices that are creating the sound. What the goal of these speaker is, is to recreate the location of the virtual object by means of differing in audio output volume. So that the user is thinking that the sound comes from a certain place. Similar like a surround sound[1] setup. In the aforementioned figure, it is clear that the sound levels need to be of the same volume. This way the user thinks that he stands at the origin of the sound, because the sound has the same volume from all directions.

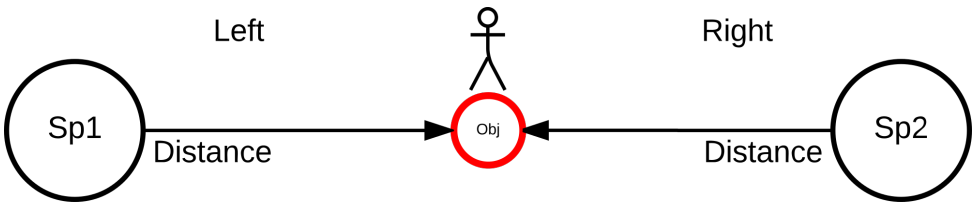


Figure 3.1: Virtual object at center

In figure 3.2 is the next example. Now the virtual object is at an offset from the user. you would normally say that “Sp1” has a shorter distance and “Sp2” has a greater distance. To compensate, “Sp2” needs to be harder then “Sp1”. Let’s think about this for a minute. If the user is standing in the middle and “Sp2” is playing harder then “Sp1”. This means that the “Right” side produces more sound then the “Left” side. The user would hear more sound coming from the “Right” side, so the user would think that the sound comes from the “Right” side.

[1] Surround sound

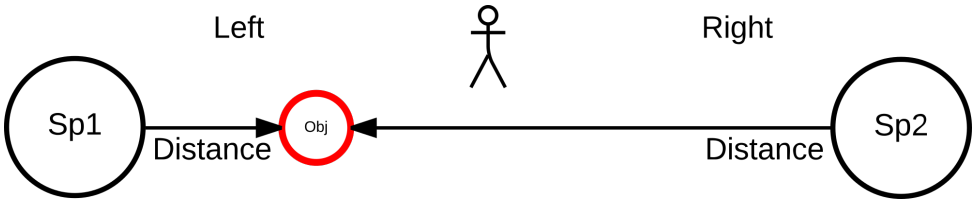


Figure 3.2: Virtual object at offset

In conclusion the speaker where the object is closest to, needs to produce the most sound.

3.4.2 Calculating the RWF

It's is clear that the speaker closest to the object, needs to produce more sound. This way, of the user stands in the middle he will think that the sound is coming from the place where the sound is the hardest.

The value of the volume in this project, is called the relative weight factor or RWF in short. Because there needs to be a value that represents the relative weight each speaker has relative to any object. Table 3.4.1 shows the calculation of the RWF for three objects.

In the left half of the table are the speakers with there distances to an object. The units of measurement don't matter in this case, as long as they are of the same type. The first step is to "invert" all the distances. See the table 3.4.1 "Inverse distance". This is done by adding all the distances and subtracting the distance of a speaker to an object from this total. Next there are the values at the bottom of the table. The *Head* is the maximal value of the RWF factor. The *tail* is the minimal value of the RWF factor. In this project's case, this would be a value between 0% and 100%. Now the number of steps are known.(head-tail) To get the actual RWF value the "value per step" is needed. This is calculated by dividing the "value per step" by the total inverse distance value. Now it is a matter of multiplying the inverse distance with the "value per step" and the RWF values are known.

Speaker	Distance	Inverse distance			Calculated Value		
A	1	6	-1	= 5	5	*8.33	= 42
B	2	6	-2	= 4	4	*8.33	= 33
C	3	6	-3	= 3	3	*8.33	= 25
Total:	6			= 12			= 100
	Head	100					
	Tail	0					
	Volume steps [head-tail]	100	-0	= 100			
	Value per volume step	100	/12	= 8.33			

Table 3.4.1: Calculate RWF for three speakers. Distances: 1, 2 and 3

The tables below show other examples with different distances values. (See table 3.4.2 and 3.4.3)

Speaker	Distance	Inverse distance			Calculated Value		
A	1	3	-1	= 2	2	*16.67	= 33
B	1	3	-1	= 2	2	*16.67	= 33
C	1	3	-1	= 2	2	*16.67	= 33
Total:	3			= 6			= 99
	Head	100					
	Tail	0					
	Volume steps [head-tail]	100	-0	= 100			
	Value per volume step	100	/6	= 16.67			

Table 3.4.2: Calculate RWF for three speakers. Distances: 1, 1 and 1

Speaker	Distance	Inverse distance			Calculated Value		
A	5	16	-5	= 11	11	*3.13	= 34
B	8	16	-8	= 8	8	*3.13	= 25
C	3	16	-3	= 13	13	*3.13	= 41
Total:	16			= 32			= 100
	Head	100					
	Tail	0					
	Volume steps [head-tail]	100	-0	= 100			
	Value per volume step	100	/32	= 3.13			

Table 3.4.3: Calculate RWF for three speakers. Distances: 5, 8 and 3

3.5 Logger

The application logs certain events according to their importance level. The logger used is easylogging++, which is a single header file that implements a full feature logger. A custom configuration is made applied in main, after the initialization. This changes the default output from:

```
2016-05-23 20:05:44,117 DEBUG [default] [User@localhost.localdomain] [int main()]
[./test/log_test.cpp:9] unconfigured DEBUG log
2016-05-23 20:05:44,133 INFO [default] unconfigured INFO log
2016-05-23 20:05:44,133 WARN [default] unconfigured WARNING log
2016-05-23 20:05:44,133 ERROR [default] unconfigured ERROR log
```

to:

```
2016-05-23 20:05:44,133 INFO [default] configured INFO log
2016-05-23 20:05:44,133 WARN [default] configured WARNING log
2016-05-23 20:05:44,133 ERROR [default] configured ERROR log
2016-05-23 20:05:44,133 FATAL [default] configured FATAL log
```

The log output is simultaneously written to the terminal, as well as an output file, if one is specified in the config. Also specified in the config is a minimum loglevel:

- DEBUG
- INFO
- WARNING
- ERROR
- FATAL

4 Website

In this chapter the website and it's functions are described. Furthermore, there will be some explanation about the logic that drives the website.

4.1 Reason of existence

The website is the place where the online speakers and objects are manipulated. The idea of the website is to enable the user to simply drag and drop speakers/objects with platform independency in mind. The website had to be simple, easy to use and not dependent on some kind of webserver. So the website has to be portable[1]. For this reason the website's main logic is made in JavaScript and The layout with simple HTML and CSS. With this implementation most web browser's can open the website, the *index.html* file, and see the site and online speakers.

4.2 Challenges

Some challenges that where faced when making this website.

4.2.1 MQTT with web sockets

This website needs to connect to an MQTT broker. Because of the rule of platform independency and portability, some kind of connectivity with JavaScript is needed. Enter the world of web sockets! For this project, the Paho JavaScript Client is used. The Paho JavaScript Client is an MQTT browser-based client library written in JavaScript that uses Web Sockets to connect to an MQTT Broker. This implementations still has some limits[2], but for this project it was doing it's job quite well.

4.2.2 Website flow

In figure 4.1 is the startup and flow sequence of the website displayed. When the site comes online, the unknown dataset[3]first has to be rebuild.

[1] Platform independent, eg usable on Linux, Windows, etc...

[2] When dealing with non Unicode characters, the Paho client will crash. This was addressed apparently in some fork of the client, but not in this implementation.

[3] The dataset is all the online clients with all the objects and their information. See clients of the topics chapter

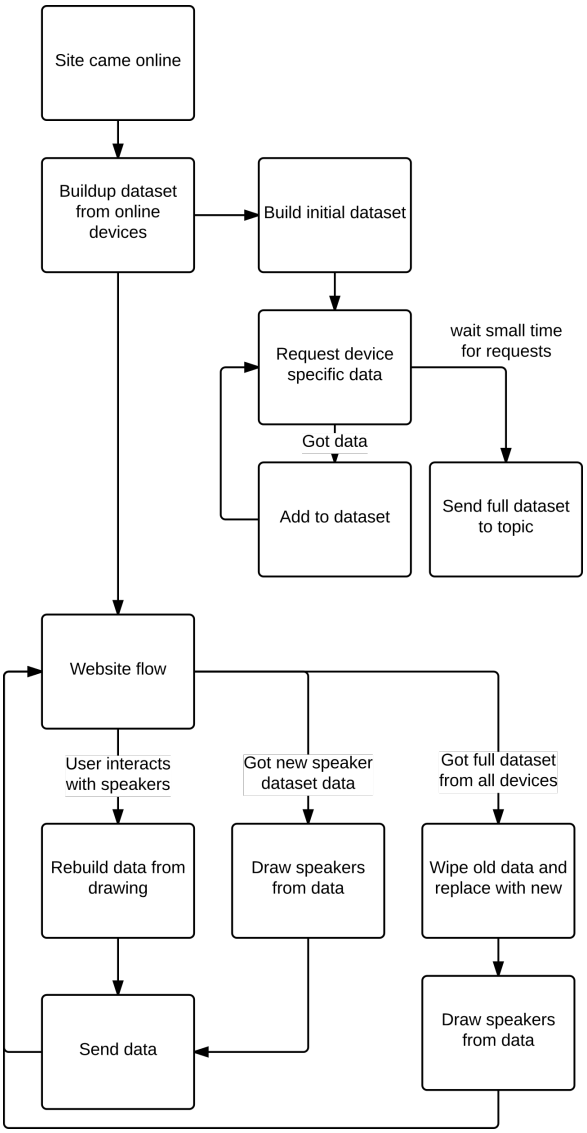


Figure 4.1: The flow of the website

After this initialization, the website is in it's main "*flow*". From here the site has three possibility's:

- User interacts with the drawing (speakers/objects)
- Site got new speaker data (from a single speakers)
- Site got complete new dataset

In the topics chapter, are the topics defined that are used to send and receive the data to and from all the devices.

4.2.3 Draw and synchronize objects

To draw and synchronize the drawn speakers/objects, there are three features needed. One that will draw all the available speakers and objects from the dataset, one that will generate the dataset from the drawn objects and speakers and a way of synchronizing this data between website and speaker.

Draw speakers from data

To draw the speakers and objects, the dataset of all speakers and objects is needed. This data holds the relations between the speakers and all the objects. As seen in figure 4.2, it

will simply be a looping scheme. In this diagram, a somewhat crude software architecture is demonstrated. The drawing is started from an object, and with all the data that is known, the rest is drawn.

Because this is a simple 2D field, all the drawn speakers, will contain all the drawn objects. I.e. if there are two objects(obj1, obj2) and two speakers(sp1, sp2), sp1 will contain the distance and angle from sp1 to obj1 and obj2. Sp2 in this example will also have this data, not the same values of course, but it will have the distance and angle from sp2 to obj1 and obj2. With this knowledge, it is only needed to draw an object or speaker once. So if a object is drawn, and another speaker has that object, the object doesn't need to be redrawn.

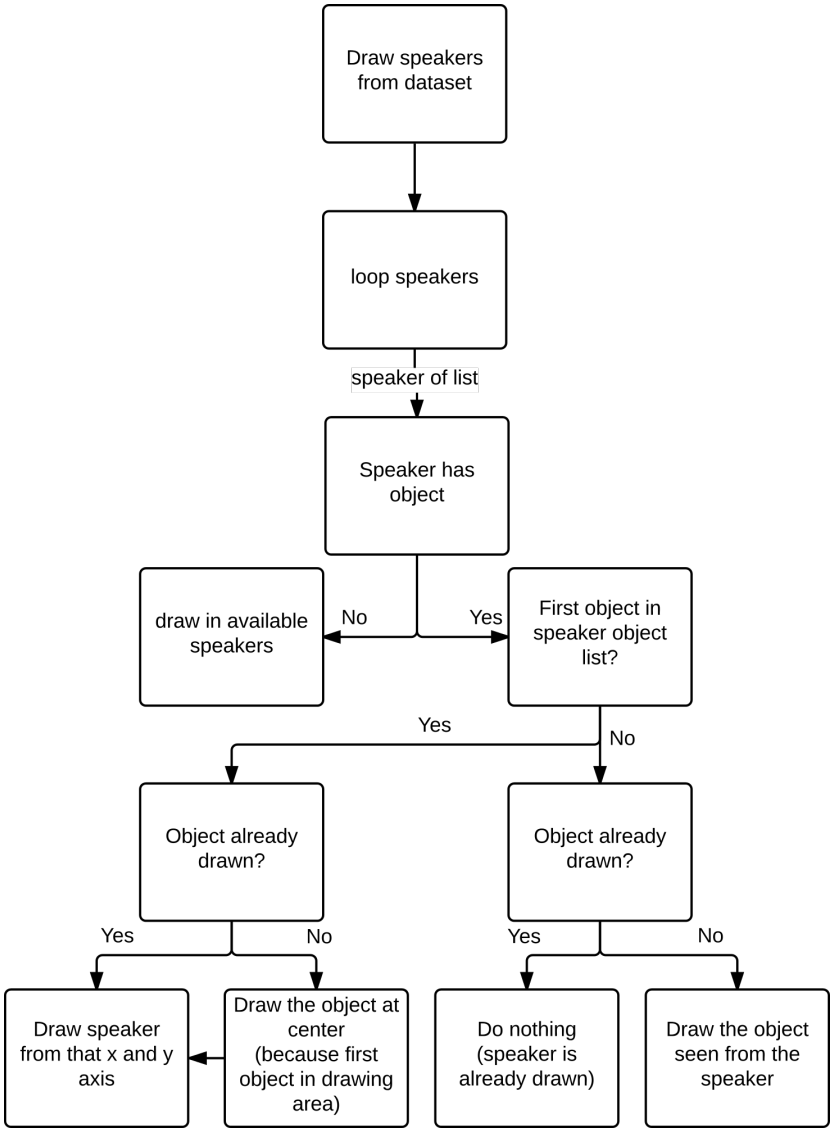


Figure 4.2: Draw speakers and objects from the dataset

Generate dataset from drawing

This feature has a simple task. Loop all speakers and calculate all the distances and angles from speaker to the objects. Figure 4.3 describes this quite well.

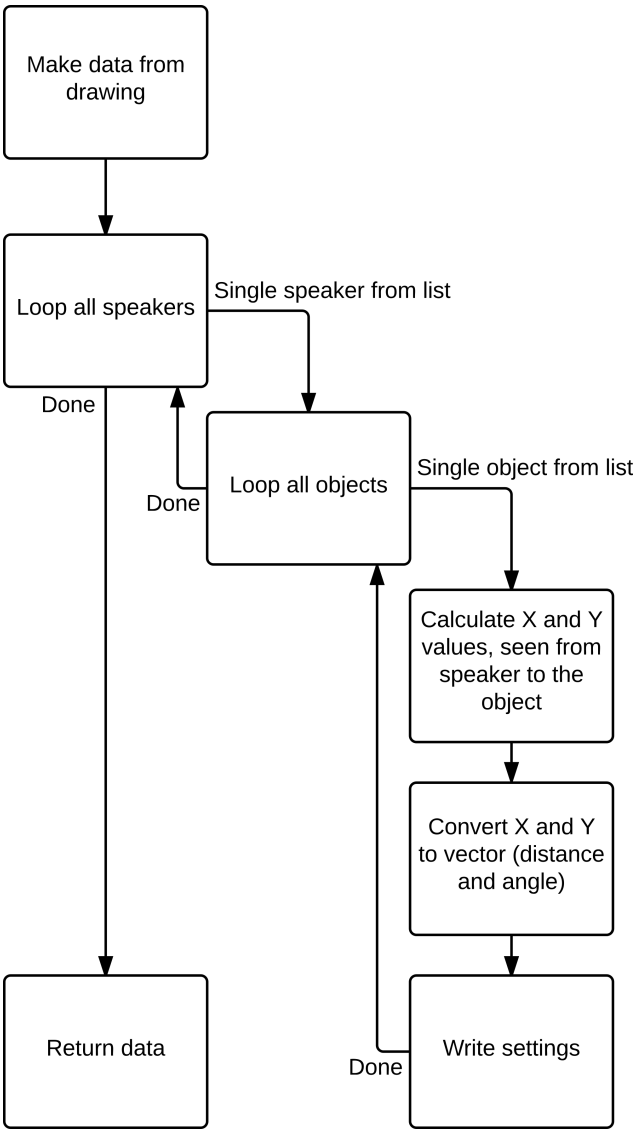


Figure 4.3: Generate the dataset from the drawing

Synchronization

The last part is the synchronization. This is used for a synchronization between two or more websites and clients. The dataset is generated. The moment this is send to a topic, the other websites, not the one sending, will use this data to redraw the speakers and objects. One extra topic is used in this system specific for the website. The moment, a website sends this data, it also sends the x and y offset of the first drawn object. This way, the website can redraw the data like it is displayed on the sending website. This is a reasonable failsafe system with the least amount of overhead.

An other solution would be to send the complete dataset to the clients, but also send one special for the websites. This second dataset would contain all the speakers and objects and their x and y coordinates. This means that for x messages to the clients, x messages to the websites need to be send. Doubling the amount of data. For this reason, making use of the already send dataset and using a small message for the x and y offset of one object, is the most practical solution.

Still, this is by no means the best solution. When that small message with the offset of one object is ignored or not send, all of the drawing will be drawn from the center of the screen instead of the specified x and y offset. However this is a price that has to be paid.

4.2.4 Loading configuration file

The not so beautiful approach

In this system, a single configuration file is used between website and client. This file contains all sorts of data, from the topics used to the name of the project. In JavaScript there is however a problem with this setup. In C or C++ this would be easy, just read it from the local file system. In general, this is not allowed in JavaScript by design. It's a violation of the sandbox. There are API's to load a file via a button etc., but this is not user friendly. Think about it, every time the site is loaded, a window pop's up with the message: "*Hello dear user, would you please be kind and load a configuration file. If you don't, I will not work.*".

The solution for this problem is loading the file on load time. Just like any other JavaScript file, it is loaded via the "script" tags. The config file was saved with a ".js" extension and the configuration file was made in a JSON format, starting with: "var CONFIG =" and ending with ";". The client can filter this out and the website can load in the config file.

It is a solution, maybe not the best, but it works.

4.3 Website layout

In these following paragraphs, the layout and main usage of the site is explained.

4.3.1 Idle

In figure 4.4 is the state of the website shown when it is in "idle" mode. On the right hand side are the available speakers and objects. From here, the user can drag these in the draw area.

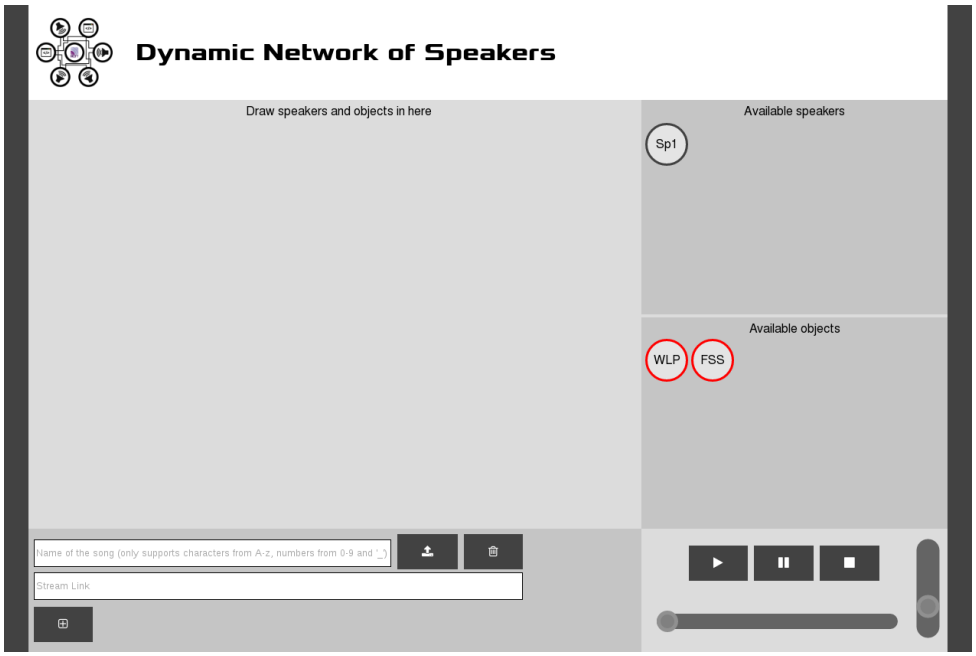


Figure 4.4: Website when idle

The following features are available:

- Available speakers list
  - Listing of all the online speakers
- Available object list
  - Listing off all the available objects

- Music controls (right bottom corner)
  - Music status control (play, pause, stop)
  - Volume control
  - Time playback control (not yet implemented)
- Object controls (left bottom corner)
  - Object name field
  - URI field
  - Upload button
  - Delete button
- Extra controls (+ button)
  - See section about the Extra options

4.3.2 Manipulating speakers and objects

As shown in figure 4.5 and 4.6, the speakers and objects can be dragged over to the draw area. In this example there is first one speaker and two objects and in the second example there are two speaker and two objects. As demonstrated here, the speakers and objects can simply be dragged around with the mouse.

The moment that there is a speaker available, e.g. a speaker comes online, it will show up in it's "available speakers" list. From that moment, the user can simply drag the speaker to it's desired location.

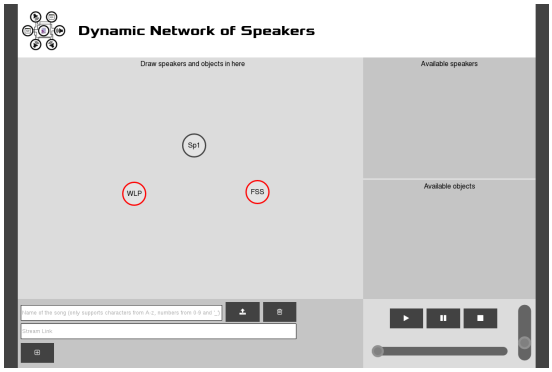


Figure 4.5: One speaker and two objects

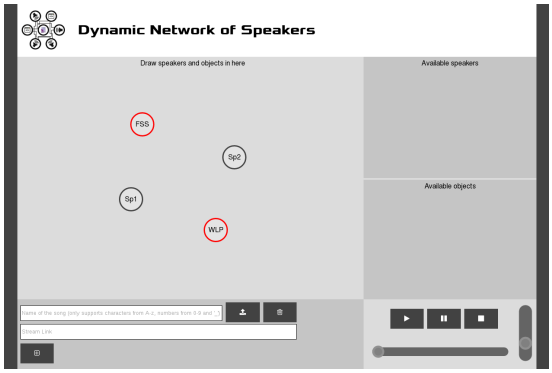


Figure 4.6: Two speakers and two objects

4.3.3 Edit objects

This website also has the ability to add, delete or modify any existing objects. Figure 4.7 shows this process.

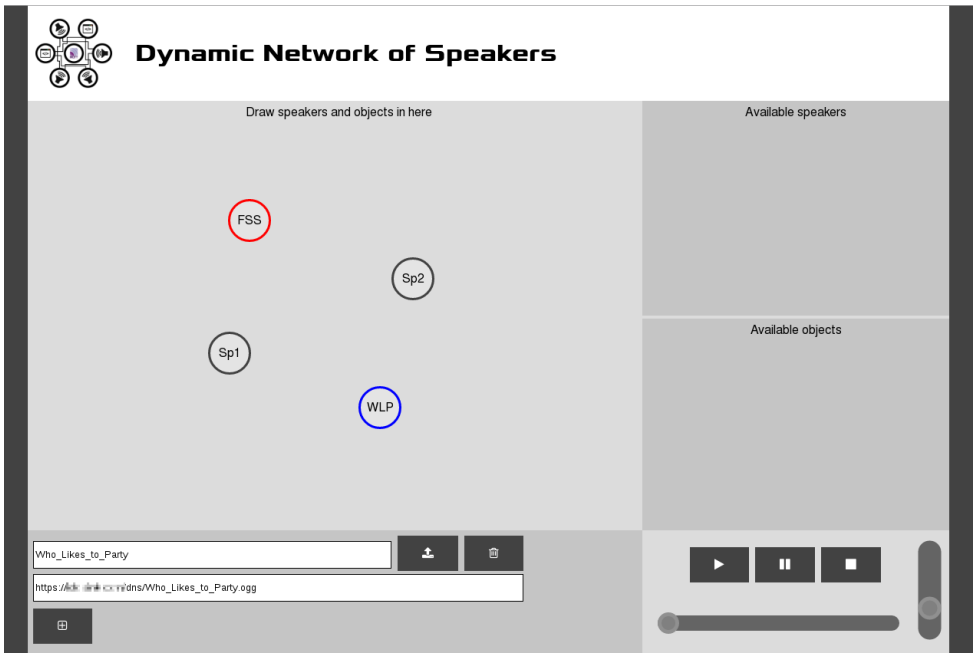


Figure 4.7: Edit an object

The following list will show the steps necessary to complete the listed actions.

- Add an object
  - Fill in the Name of the object
  - Fill in the URI of the object
  - Press the upload button
- Delete an object
  - Select an object (the selected object is marked with a colored circle, blue in this case)
  - Press the delete button
- Modify an existing object
  - Select an object (the selected object is marked with a colored circle, blue in this case)
  - Modify the URI[4]
  - Press the upload button

4.3.4 Extra options

The extra options tab, as shown in figure 4.8, holds some extra options that are not of a great importance for the main function of this site.

[4] Editing the name will result in uploading a new object with that name or overriding an object with that name

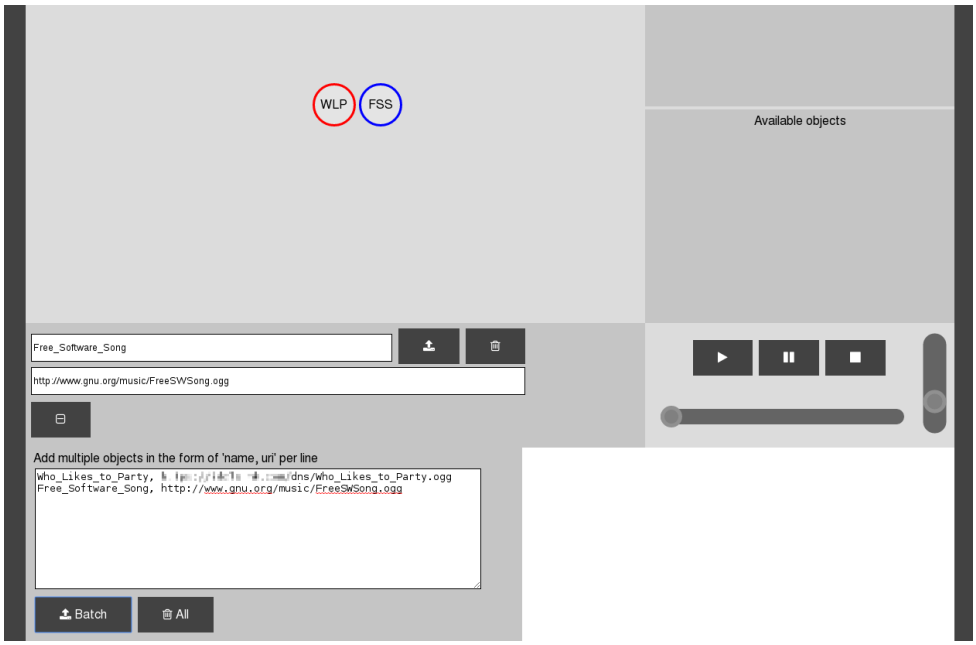


Figure 4.8: Extra options

This tab has the ability to batch upload and delete all objects. This can be helpful when using this website in combination with multitrack[5] songs and the user wants to upload multiple objects at once.

[5] A song split up into multiple stems. A single stem may be delivered in mono, stereo, or in multiple tracks for surround sound.

5 Conclusion

The project functions as a decent proof of concept. The surround sound gives a noticeable effect, with minimal audible latency. It shows the power of MQTT as an “IoT” protocol, where speed and dynamic behavior are key. C++14 + MQTT + JSON makes for easy development, and great flexibility. We, “derpicated”, are proud of this project, as it came out working great and was allot of fun to develop.

5.1 Known issues

Yet some known issues could be solved in future iterations, to name a couple:

5.1.1 Audio latency

The latency is still audible. A system could be designed, that better syncs the audio. Perhaps by pre-planning the actions, and then executing on time-points. A bit of research into the minimal audible latency wouldn’t harm, most likely the media sector has determined this already.

5.1.2 Dedicated audio library

Audio could be improved by using a dedicated library, instead of CLI-tools. To implement this only the audio class needs to be revised, but finding a suitable library might prove difficult, as it needs to support an arbitrary amount of streams, each at a different volume, and should preferably be easy to use.

5.1.3 Start at time

A start time feature could be implemented, where the user can determine a time point in the audio to play from. The difficulty with implementing this is that the website has to know the total length of audio tracks, which can differ from each other. Since the website doesn’t have the computing power to parse an audio file, the client would have to decide the audio length, send it back to the website, which in turn could send a start time back to the clients.

5.1.4 Different hardware

As the project is meant to run on many devices, perhaps with different hardware, some devices might be louder than others. The current setup only scales to the max output, per devices. A system could be implemented, where the user can configure a factor per device. Meaning that a laptop that outputs double the volume as a raspberry pi, could be set to 50% max volume. Only an extra config parameter would be needed.

5.1.5 Website framework

The website is using jQuery for manipulating the UI of the website. This is a bit messy and s a better system could be used. Something that acts more as an object that can be interfered with then DOM elements.

5.1.6 Website logging

The website itself doesn’t tell the user a great deal about it’s status. Sure, it says that a speaker is online, but what happens when the website can’t connect? At this moment, it will log things in the console, but this is not user friendly. Some kind of log window or message box would be appreciated.

List of Tables

List of Figures