VRIJE
UNIVERSITEIT
BRUSSEL

Graduation thesis submitted in partial fulfilment of the requirements for the degree of Master of Science in Applied Sciences and Engineering: Computer Science

DISTRIBUTED MULTI-TOUCH USER INTERFACES FOR MODULAR MUSICAL EXPRESSION

JEROEN NYCKEES

Academic year 2016-2017

Promoter: Prof. Dr. Beat Signer
Advisor: Reinout Roels

Science and Bio-Engineering Sciences

VRIJE
UNIVERSITEIT
BRUSSEL

Proef ingediend met het oog op het behalen van de graad van
Master of Science in de Ingenieurswetenschappen: Computerwetenschappen

DISTRIBUTED MULTI-TOUCH USER INTERFACES FOR MODULAR MUSICAL EXPRESSION

JEROEN NYCKEES

Academiejaar 2016-2017

Promotor: Prof. Dr. Beat Signer
Advisor: Reinout Roels

Wetenschappen en Bio-ingenieurswetenschappen

Abstract

The user interfaces artists use to interact with music software are no longer merely mouse and keyboard based. Instead they are being distributed over dedicated hardware devices that offer a more tangible experience. The introduction of mobile touch devices reinforced this trend in the design of interfaces to interact with software for musical expression. Due to their versatility mobile touch devices have brought a wave of change in the ways of interaction with computer music software. Native applications have been designed and implemented that enable artist to compose widget based user interfaces and to distributed them over multiple devices.

Recently, due to W3C standards such as access to local I/O channels for MIDI, low latency communication over WebRTC data channels and hardware accelerated graphics with WebGL make of the of the browser a viable platform to host more complex distributed user interfaces for musical expression. However, existing work based on web technologies often are limited in the sense of state synchronisation and modularity.

In the context of this thesis we have to develop a distributed, elegant and modular computer music tool, that makes the development, publishing and connectivity accessible, and to discover the opportunities that they enable in performance practice. This is done by designing and implementing a tool that leverages the portability web technologies offer into a framework that eases this process.

Acknowledgements

First of all, I would like to thank my promoter Prof. Dr. Beat Signer and advisor Reinout Roels for giving me the opportunity to write this thesis on the subject that I am passionate about. Furthermore, I would like to thank their help and assistance during the process of this thesis.

I would like to thank my mother for supporting my studies, and my friends and girlfriend for keeping me motivated. Their encouragement kept me motivated throughout these past years.

Contents

1 Introduction

1.1 Context

With the pervasiveness of mobile devices, the number of possibilities for where music can be made and experienced is growing. Modern mobile devices come with computing power performing better than the computers that took Apollo 11 to the moon [42]. This, in combination with the speed of networks, the presence of sensors, and the access to the hardware through web API's on mobile devices, has led to the adoption of mobile devices as musical instruments in distributed audiovisual performance settings, such as the Stanford Mobile Phone Orchestra [24].

Not only in a collaborative setting, but also individually, musicians include mobile devices in their set of tools to control and create music. Their modularity, versatility, and elegance make them extremely suitable devices to be used as distributed user interface (DUI). They function as an extension to the laptop, not only as an extension of the screen, but also as a multimodal interface by exploiting the many sensors on board of contemporary mobile devices.

1.2 Motivation

Modularity

When we look at the current tools for making electronic music, we see a very uniform range of instruments. The classical workstations are mostly abstract panes with knobs and sliders, which require a certain preliminary knowledge of music technology in order to productively create music. These tools are (mostly) very versatile; they give starting musicians so many options creating an extremely steep learning curve for beginners. While current commercial tools offer a wide range of flexibility in composing ones distributed interface, there is still only a restricted amount of widgets available to the user. Usually these tools are platform dependent and limited in the sense of distribution and content sharing.

Interactivity

Additionally, current systems are generally aiming at controlling the music in a unidirectional way. Yet in many cases, it would instead be interesting to have an ever consistent reflection of the state of the parameters the distributed interface aims to control. This also means that the stream of data coming back from the sound source can be used to visualise the state of the piece at any given moment in time. The purpose of this visualisation can be used in multiple scenarios. Either to aid the musician in the process of making music and keeping a mindmap of the piece, or in a concert setting, where the audience is being involved by having visual feedback of the events triggered by the artist.

Mapping Layer

Finally, current tools often have interfaces which are direct replicas of existing physical widgets. For example, the Korg poly-six, a popular synthesizer in modern music, has been virtualised by its manufacturer Korg. As can be seen in Figure 1.1b, the software version of the synthesizer is an attempt to match visually as closely as possible to the original. It is clear that this approach does not contribute to the expressiveness of the instrument. It only becomes interesting when a mapping between input parameters and system parameters is installed [15].

(a) The Korg MS-20 synthesizer

(b) Virtualisation of the Korg MS-20

Research Goals

These problems lead to the research question of this thesis: How to address shortcomings in existing work by designing and implementing an extensible framework for creating customisable interfaces for distributed collaborative musical expression.

1.3 Research Methodology

A research methodology based on Design Science [26] was used to come up with answers to the previously stated research questions.

Problem Identification and Motivation

Out of a general literature study and a personal interest for musical expression and distributed user interfaces the research question has been defined. The contemporary hardware and software used to perform and create music were studied in the context of distributed user interfaces. Finally given the portability of web technologies a consideration of how web technologies could be used to improve these tools was made.

Solution

After this an ideation phase was done on the technical design of the eventual platform to build. The technological constraints were analysed by experimenting with current technology. Finally the user interface design was done by making some sketches and wireframes of what the platform would visually look like.

Development

The solution was implemented as a web application that solely runs in the web browser. Due to the use of cutting edge technology, we encountered and worked around various technical hurdles to meet the requirements.

Evaluation

Eventually the solution was evaluated by doing an experiment. A set of participants was asked to use the system to create a DUI, set up a connection and control music software with the created DUI.

Conclusion

In a final phase reflections are made on the contributions in the studied field. The shortcomings and possible improvements of the proposed solution are summarised for future work.

2
Literature Study

In this chapter we present our literature study on distributed user interfaces for modular musical expression. The focus of this thesis is how to design a framework for distributing user interfaces for musical expression. We will try to find answers to the questions that come to mind when designing such framework, namely "How can computer music interaction be made more simple and elegant using mobile touch devices?", "How can computer music instruments be made accessible to artists with little or no programming background?" and "What new practices are enabled by mobile touch devices to participate in audiovisual experiences?". Finally the findings will form the foundation for the requirements that our framework will have to meet.

2.1 Background

It is significant to this research to give a brief history of how computer music technology evolved to what it is today. This section is by no means a complete summary of the history of electronic music, but merely a gross overview of elements relevant to this research. A complete overview on the history of electronic music technology can be found in chapters 14 and 15 of "The computer music tutorial" [30].

Figure 2.1: Schaeffer's phonogène

2.1.1 The Opening of Music to All Sounds

The way in which music is being created and performed has been the same for a long period of time. A composer would write a score, this score would then be performed by a group of musicians under guidance of a conductor, for an audience. The composer's process was by no means immediate, he would have to wait between writing the music down on paper and hearing it in a concert hall. There were little or no democratic means of recording or reproducing a piece, making a clear demarcation between the production process and the performance.

However, in the early 1940s, not long after the invention of tape, musicians started to include tape manipulation in their music production process. Led by the French intellectual Schaeffer, a new stream in music, known under the name *musique concrète*, arose. By recording sounds of musical instruments, the human voice, or the sound of hitting objects, etc. and then playing them back in reverse or at different speeds, one can argue Schaeffer was the first to use the studio as an instrument itself [25]. Schaeffer's experiments lead to an evolution both on the conceptual level as well as on a technical level. This evolution is known as *the great opening of music to all sounds* [6].

2.1.2 Digital Evolution

At this point in time, the music production process was a very static process. Once audio had been recorded to tape, most of the characteristics of the piece, such as sound timbres, arrangement, relative volume levels etc. were fixed. It was the arrival of MIDI that brought an end to that era. MIDI is an industry standard communication protocol developed in the late 1970s that allows users to connect electronic musical instruments and computers. MIDI was developed by several companies including Roland, Yamaha and Korg and was standardised by the MIDI Manufacturers Association (MMA) [3] in 1983. It is capable of sending event messages specifying pitch and velocity, control signals, and clock signals. By using this protocol to record or program events, musicians are now able to edit a piece after it has been recorded. A common application of MIDI is to use it as a protocol for the sequencer. A sequencer is a piece of hardware or software that is used exactly for the purpose of playback or to record music in time.

Around the same time, computers such as the Apple Macintosh and the Commodore Amiga became powerful enough to handle digital audio editing. More and more the computer became a virtualisation of the studio. From then on there has been a rapid proliferation of new software and hardware designed for musical expression. Software known as digital audio workstations (DAW), such as Steinberg Cubase and Apple Logic Studio, made hardware in the studio virtually obsolete. Instruments and audio effects were being developed as easily interchangeable plugins between different DAW's. Popular commercial examples of technologies used to make these plugins include Virtual Studio Technology (VST) and Audio Unit (AU). This digital evolution made the computer a self contained music production device, thereby democratising electronic music production, thus giving everyone the ability to make music.

Out of the need for abstraction to build systems like this, domain specific languages such as Max/MSP [28, 29] emerged. Max/MSP is a visual programming language for building audio and multimedia applications by connecting low level modular building blocks into sophisticated dedicated components for multimedia software. These low level building blocks are presented in a graph and can be abstracted into other components. This paradigm, known as *dataflow programming*, is very powerful for developing multimedia applications where often certain input events need to be mapped onto some sort of audio or graphic input. The combination of a set of components into a multimedia application is called a Max patch.

As the computer took a central role in the music production process, and as computers became more portable, artists started to integrate the

computer in live performances. However, classical DAWs have a clear focus on the production of music and not necessarily on live performance. Out of the need for audio software that can be used for both live performance and music production, Ableton Live was born. It distinguishes itself from classical DAWs with its so called session view, enabling musicians to create MIDI or audio clips in a 2D matrix grid that can be played back independently, leading to a more improvisational way of creating music.

2.1.3 Reactive Music

Reactive music is an application of a broader term known in computer science as context awareness, a research field of ubiquitous computing originating in 1994. A context aware system adapts according to the location of use, the collection of nearby devices, as well as sensor data [36]. Reactive music is a non-linear format of music, where certain parameters, such as tempo and loudness, react to the listeners behaviour.

For example, in Giant Steps [4], the idea is applied to running. Depending on the body movement of the runner, the application will adapt the tempo of the music and dynamically generate the music to match the tempo of the runner's steps.

Generative music is a particular form of reactive music. The term was popularised by the artist Brian Eno, and is described as a form of music that is algorithmically composed, ever-changing, and created by a system. In the literature several interpretations of generative music are summarised as [40]:

1. **Linguistic/Structural**: music created by grammars or analytic theoretical constructs.

2. **Interactive/Behavioural**: Music emerging from processes that are not inherently musical. For example by using sensor input.

3. **Creative/Procedural**: music emerging from one or more processes started by the musician.

4. **Biological/Emergent**: natural occurring, non deterministic sounds such as wind chimes.

A popular application of generative music is used within the gaming industry. The music during the game play can be adapted according to the context of the game, for example by speeding up the tempo to create tension in difficult parts of the game.

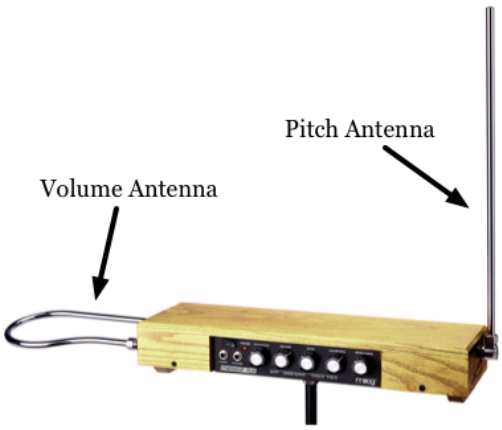


Figure 2.2: The Theremin instrument

Another example of reactive music implementations is RjDj. RjDj[1] is an iOS app for creating reactive musical environments. Artists can create in PureData publish algorithmic compositions that are called *scenes*. These scenes are hence musical experiences that react to the context of the user listening to them. The mobile device of the user will process contextual sensor data such as sound and movement. This data can then be used in scenes for example to start playing more quietly as the environment of the phone becomes more quiet, or adapt the tempo to the amount of movement detected by the devices.

2.2 New Interfaces for Musical Expression

We see that instruments that have settled today have evolved over time into dedicated tools for specific purposes in the process of making music. Guitars, for example, are built a very specific way to enable musicians to play certain chord progressions on strings. The piano keyboard is a direct representation of the system behind western diatonic music, with the white keys being the notes of the C major scale and the black keys the "accidental" notes. Each instrument has been designed as to aid the musician to accomplish a certain task very well. We see the same search for intuitive ways to interact with machines when we look at electronic music instruments. Since 2001, the community interested in this particular research domain is referred to as the NIME[2] community [27] with yearly conferences around the world.

[1] https://github.com/rjdj
[2] new interfaces for musical expression

Figure 2.3: AudioCubes being manipulated

One of the first tangible electronic music instruments was the Theremin [19], invented by russian inventor Leon Theremin in 1928. It is a synthesizer that is controlled by moving your hand up and down in the proximity of two metal antennas. The movement of the hand controls the pitch of the device as can be seen in Figure 2.2. The Theremin was the first of many forms of interaction with electronic instruments, other than piano keyboards, knobs and, buttons that found their way to the music studio.

As technology evolved, the number of possibilities to create new means of musical interaction has only increased. Another more recent example of a tangible interface presented at NIME are the Audio Cubes [35]. Audio Cubes are a set of wireless cubes that have a notion of proximity of other objects. A sound processing network can then combine the properties of the cubes, such as location on a plane, proximity of other cubes or objects, and layout.

One can clearly see a trend where *the assertion of human presence within music produced by and through technology will take many different forms as humans become increasingly alienated from purely physical sound production* [12].

2.3 Related Work

Now that a basic understanding of the evolution of electronic music technology has been established, we will narrow our scope down to how distributed multitouch devices can be interesting in the context of computer music tech-

nology.

Distributed user interfaces are defined as user interfaces with five distribution dimentions [11].

- Input (I). Managing input on a single computational device, or distributed across several different devices (so called input redirection).
- Output (O). Graphical output tied to a single device (display), or distributed across several devices (so-called display or content redirection).
- Platform (P). The interface executes on a single computing platform, or distributed across different platforms (i.e., achitectures, operating systems, networks, etc).
- Space (S). The interface is restricted to the same physical (and geographic) space, or can be distributed geographically (i.e., co-located or remote interactive spaces).
- Time (T). Interface elements execute simultaneously (synchronously), or distributed in time (asynchronously).

One possible way to distribute a user interface is to extend them to mobile touch devices. Because of this, we see an emergence of distributed user interfaces also in the landscape of tools for electronic music production. Touch screens form a very intuitive way to interact with computer systems, especially for performing live and/or aiding in the creative process of making music. They provide new ways of interacting with the environment using basic human gestures.

A recent commercial example of an application where touch screens bring added value to the interface is Beatsurfing[3], an iOS application that enables artists to compose their own interface. An example can be observed in Figure 2.4. Interaction with the interface involves tapping and sliding gestures. As the user's slide gesture collides with widgets of the UI, MIDI events are being generated. These events are mapped to MIDI events, and sent to an audio synthesis device, triggering the playback of sounds. The nature of sliding gestures makes that it is very natural to play rhythmical sequences of notes.

Mobile devices are highly modular. Artists can easily distribute the user interfaces over multiple devices, hence enabling them to control their music exactly as they want, hereby taking advantage of the individual characteristics of the devices. For example, the artist could provoke face-to-face collaboration between peers by opting for a large tabletop multi-touch device,

[3] `http://beatsurfing.net`

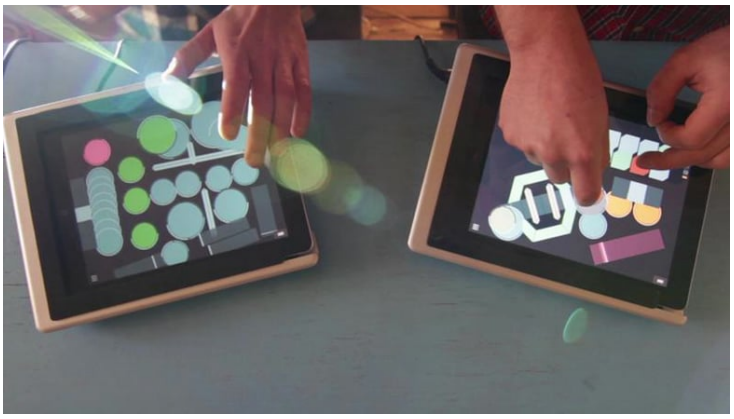


Figure 2.4: Beatsurfing: as the artist performs gestures MIDI events are being triggered.

as the ReacTable [16]. Alternatively, the artist could take advantage of the motion sensor present on mobile devices to engage peers to individually collaborate to a musical performance, as is done in the Stanford Mobile Phone Orchestra [24]. In the remaining part of this section the related work that is implemented as standalone applications is discussed.

Control

Control is a hybrid app that enables users to create interfaces intended for controlling artistic applications by transmitting MIDI or OSC. Control was built by C. Roberts in 2011 with the idea of having more fine grain control over the appearance of the widgets an artist can use to compose it's interface. It does this by using web standards such as HTML, CSS and JavaScript.

It is built on top of the phone gap[4] framework. Phone gap is a framework enabling developers to develop mobile apps strictly using web technologies. HTML, CSS and JavaScript are compiled for multiple platforms such as iOS, Android and Windows Phone. This enables developers to target a much broader audience without having to know each of the technology stacks.

To set up a controller with Control, first of all the user needs to install the app from google Play Store for Android or the App Store for iOS. Secondly the device needs to be connected to the same wireless network as the machine that is running the Audio software.

Once the connection has been set up the user can define his own widgets in a JSON file which is hosted on a server. As a developer one can simply

[4]http://phonegap.com/

host the file on his development machine. In order to load the widget one points the app to the URL where the widget definition JSON file is hosted.

Defining a widget is done by defining a widget object and making it an instance of the Widget class. Next in its constructor method is described how the widget should be rendered. The constructuctor takes two parameters: a context, which is the root DOM element where the widget will be appended. The props variables is where the individual parameters defined in the JSON file will be passed through. In JavaScript this is done like shown in Listing 2.1.

Listing 2.1: Creating a UI in control

```
window.UberWidget = function(ctx, props) {
 this.make(ctx, props);

  this.note = props.note;

 this.ctx = ctx;
 this.container = document.createElement('div');
 this.container.style.position = 'absolute';
 this.container.style.left = (this.x + 3 ) + 'px';
 this.container.style.top = (this.y + 3 ) + 'px';
 this.container.style.width = (this.width - 6) + 'px';
 this.container.style.height = (this.height - 6) + 'px';
 this.container.style.background = '#888888';

 this.ctx.appendChild(this.container);
};
UberWidget.prototype = new Widget();
```

As one can see, control gives access to the underlying DOM on top of which the application is built. This gives the developer of a new widgets the choice to use either classic DOM manipulation i.e. HTML and CSS, or to make use of an HTML5 canvas to render the widget.

To make the application interactive the developer of the widget has to listen to events fired by the application, this is done like shown in Listing 2.2.

Listing 2.2: Creating a UI in control

```
 UberWidget.prototype.event = function(event) {
if (event.type == "touchstart") {
 var touch = event.changedTouches[0];
 if (this.hitTest(touch.pageX, touch.pageY) ) {
     midiManager.sendMIDI("noteon", 1, this.note, 127);
  this.container.style.background =
   '#' + Math.round(Math.random() * 0xFFFFFF).toString(16);
 }
}
};
```

The final step would be to point the app to the correct location of the JavaScript definitions. The end result looks as depicted in Figure 2.5.

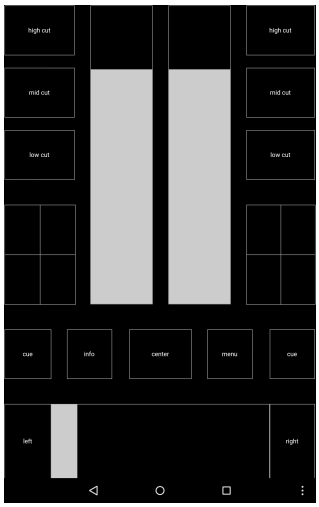


Figure 2.5: Example dj controller

urMus

urMus [13] is a collaborative live coding environment for sound creation. It is an iOS application that can be configured in the web browser. Its design goals are to support interface design and interactive perfomance on multi-touch mobile devices.

It is a three layered system with at the bottom the core low level engines of the system. Sound processing is done on the device itself. On top of that urMus uses the LUA scripting language to offer access to the first layer in a more accessible way. The third layer are instances of programs written in LUA such as scripts for sound creation or UI components. These particular programs combined form a urMus application.

Creating an audiovisual instrument with urMus hence comes down to installing the urMus app on either an iOS or Android device. Users can then collaboratively access via a web browser the urMus editor to chat and define the behaviour of the instrument. The editor, as can be seen in Figure 2.6, is on the one hand used to define the sound synthesis engine and on the other hand for the appearance of the widgets that the instrument will consist off. Parameters changes are mapped to touch events and sensors on the mobile device.

MIRA

The low-level building blocks of a Max patch include UI elements such as buttons and sliders. They can be connected in the graph to other low level

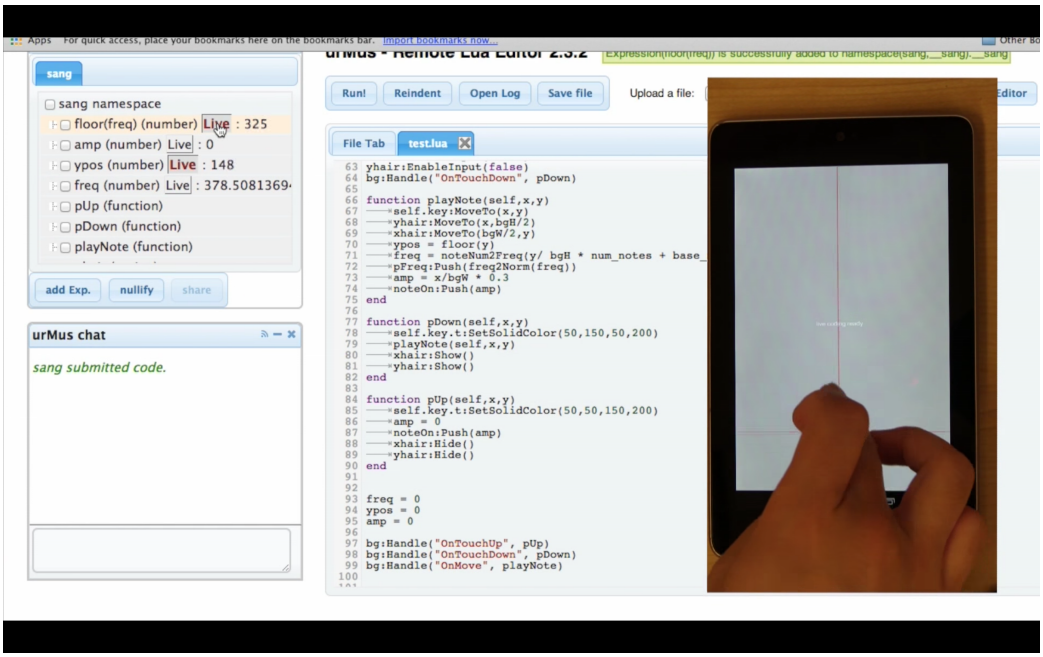


Figure 2.6: The urMus editor

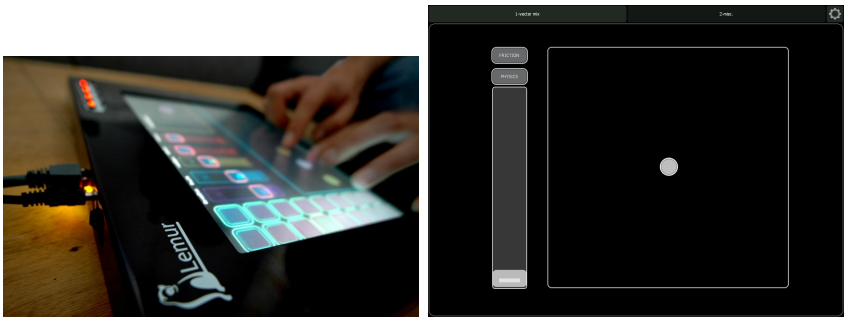
blocks so that they can be used as variables in a patch. With MIRA it is possible to define certain UI components inside of a Max patch and have them reflected on a mobile device. MIRA is an iOS application for mirroring Max patches on iOS devices. It particularly aims at giving more tangible control over Max patches to the user through mobile touch devices.

The main gain with this solution is that it integrates very seamlessly in a users set up. This because it needs almost no network configuration in order to connect the MIRA app to the Max patch. The MIRA app is discovered and automatically configured by the Bonjour protocol. Another benefit of MIRA is the fact that there is no explicit mapping needed between control components and the parameters that need to be controlled by the component. MIRA controllers can be distributed over multiple mobile devices in order to control a central patch.

Lemur

Lemur is a commercial application that originally was developed for dedicated hardware (Figure 2.7a) by JazzMutant[5]. The Lemur application lets one build modular user interfaces to control music software parameters via MIDI or OSC. The control interfaces are created via a desktop application where

[5]`http://www.jazzmutant.com/lemur_overview.php`



(a) Lemur touch screen (b) An XY controller

Figure 2.7: Lemur controller

a user can combine and order predefined widgets. The changes a user makes are automatically reflected on the device.

Moreover, the application has a built-in physics engine that lets users map physical behaviour to certain widget parameters. For example an XY controller (Figure 2.7b) can be configured such that when moved the movement will decay with a certain amount of friction. The friction can be configured to be equal to 0 and thus keep moving infinitely in the direction the user moved the controller. When the controller hits the walls of its surrounding box, it will bounce and continue in the opposite direction. An interesting application of this could be to map the the bounce event to a note trigger, causing a certain sound to be played for each collision with the boundaries the box.

GoOSC

GoOSC is an implementation of a decentralised, serverless and peer-to-peer network for the interchange of musical control interfaces and data using the OSC protocol [5]. The OSC protocol is used to both send meta data and actual control data over the network. The metadata is used to handle editing interfaces on mobile touch-screen devices and auto-discovery of network nodes. Like this it is possible by sending OSC meta messages over the network to create, delete and layout widgets on the devices. The graphical widgets that are mapped to a certain parameter will be synchronised through the network mechanism. Thus guaranteeing that every node in the P2P network is always consistent. The individual widgets are predefined yet customizable in terms of size and colour. The application comes with a set of predefined widgets such as knobs, buttons, sliders and labels. An example of a controller built with GoOSC for Android can be seen in Figure 2.8

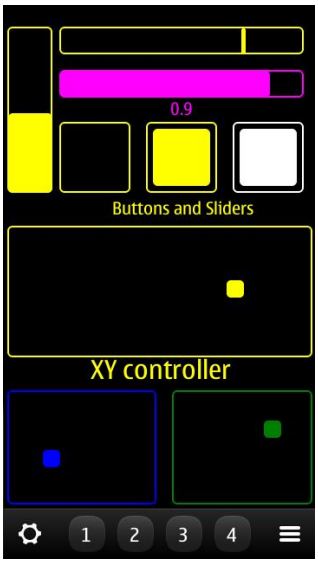


Figure 2.8: GoOSC Interface example

2.3.1 Viability of the Web Browser for Musical Expression

Finally, web technology has made it easier than ever to create and share information on the Internet. We see an immense proliferation of social networks and a shift of desktop applications now making place for Web 2.0 applications. Platforms such as Codepen[6] allow developers to create and share code snippets directly in the browser, hereby including arbitrary JavaScript dependencies. Even complete IDEs, such as Cloud9[7], are slowly getting adopted by the web development community.

This evolution continues in the way musical instruments are being published [34]. The features in HTML5 such as the Web Audio API, Web MIDI, WebSockets, WebGL, and WebRTC have led to a new research field where the web browser functions as a platform for the creation of new forms musical expression, with the COSIMA[8] lab at the Centre Pompidou in Paris being one of the leading figures.

Gibber and Interface.js

Gibber [32] is a live JavaScript coding environment in the browser created by C. Roberts et al. It leverages the Web Audio API with a set of high level

[6] `http://codepen.com`
[7] `https://c9.io/`
[8] Collaborative Situated Media

methods to aid the artist in the process of live coding music. This includes tasks such as:

- audio synthesis options (such as FM synthesis, granular synthesis)
- a variety of sequencing options
- audio effects, such as flanging and buffer shuffling.

The target audience for Gibber would typically be be live music coders, a growing community using programming environments to create (aesthetically debatable) music. Other frameworks used for this purpose include SuperCollider [22], Overtone [1], and Chuck [39]. This way of making music has gained international exposure over the years. People organise gatherings where artists perform live coding musical experiences, with or without visual extensions. Since 2014 this type of gatherings is commonly referred to as Algoraves [8]. During the development of Gibber, Roberts found that existing audio libraries built for this runtime JavaScript node were not efficient enough to realise the complex synthesis graphs envisioned [33]. In the same effort the framework got enhanced with a visual extension using the P5.js library, a JavaScript implementation of the Processing creative coding environment that enables the developer to tie a visual aspect to the experience.

NexusUI

NexusUI [2] is a set of easy deployable creative user interface components. They are implemented exclusively with web technology. This means that a central audio synthesis unit is running a web server such as Ruby on Rails or Node.js. Mobile devices participating to a performance make connection over web socket to the server and communicate using either OSC messages or MIDI. In Node.js setting up a communication server would look like this:

Listing 2.3: Nexus node example

```
var connect = require('connect'),
    http = require('http'),
    app = connect().use(connect.static(__dirname)).listen(8080),
    io = require('socket.io').listen(app);
    osc = require('node-osc'),
    client = new osc.Client('localhost', 4040);

io.sockets.on('connection', function (socket) {

    socket.on('nx', function (data) {
      client.send(data.oscName, data.value);
    });

});
```

The view of the application uses HTML5 canvas elements augmented with a `nx` attribute specifying which component is to be rendered. Events triggered on this web pages are sent with socket.io to the node server. In a Max patch for example, one can then listen for OSC messages coming from the server.

Listing 2.4: Nexus view example

```
<script>
 var socket = io.connect('http://localhost:8080');
 nx.onload = function() {
  nx.sendsTo("node")
 }
</script>
<body>
 <canvas nx="slider"></canvas>
</body>
```

The main advantage of NexusUI is how unobtrusive the framework is. By simply including the JavaScript library and setting up the web socket connection the set up is complete.

In a later phase of this research the author of NexusUI also investigated the generation of NexusUI web pages from a Max patch in *Simplified Expressive Mobile Development with NexusUI, NexusUp and NexusDrop* [37]. In the same effort a drag and drop interface was built for building NexusUI interfaces.

Soundworks

Soundworks [31] is a JavaScript framework for building distributed audiovisual experiences in the web browser where participants use mobile devices to interact, developed at IRCAM in Paris. It is exploring the boundaries of the possibilities to create audiovisual experiences with today's emerging web technology such as the Web Audio API.

It provides a client server architecture that is entirely based on web APIs and Node.js for the server. The sound is generated on each of the individual clients depending on the type of client. The framework identifies 3 different types of clients that are implemented by means of ES6 classes. Next to the client types the framework offers abstractions to facilitate session management, media preloading and clock synchronisation. The client types are divided by the type of role they take up. Three different roles can be distinguished:

1. The **soloist** role is used for participants that play a solo role in the musical application.

2. The **conductor** role is used for the type of client that needs to control the global parameters of the application.

3. The **shared-env** role is used for clients that have a sonic or visual representation in the environment of the application. For example a video projection or a public sound system.

Other than client types, Soundworks introduces the concept of an **experience** as the scenario of an application. A scenario defines what the experience will conclude as a sequence of actions.

Finally, the view of an experience is what the end user eventually will see. It is the graphical rendering and interaction for the clients that are managed in this abstraction. It is possible to define the HTML structure of the view through view templates comparable to underscore templating syntax. The eventual styling of the view can be done using CSS or Sass. Sass is a scripting language that is syntactically similar to CSS and is interpreted into CSS during the build phase of a Soundworks project.

Another possibility is to use HTML5 canvas to render the view.

As a proof of concept the developers of Soundworks used their framework to implement the generative music app, Bloom by Brian Eno and Peter Chilvers. The original app by Eno and Chilvers is an iOS app that plays a low harmonic drone, when the user taps the screen the app plays back higher pitched tones that accord with the drone. The user's sequence is looped over time.

With the abstractions Soundworks provide it is very easy to recreate a distributed version of this app that runs in the web browser. One of the clients gets the conductor role and has to be started before any other participants join in the experience. After that, other clients can join the experience as *soloists* and tap their screen to create a sequence of tones.

2.4 Conclusion

In this chapter we have given our findings from doing a literature study of the field. We have seen how mobile devices, web technology and distributed user interfaces can be used for applications that involve musical expression. However to date we see that there is no system that combines the portability of a web solution with the same degree of interaction or modularity offered by stand-alone native applications. We have seen efforts to ease the process of developing distributed user interfaces for musical expression. Moreover we have seen implementations of systems where mobile devices are used for musical expression in both distributed and non distributed fashion. Finally, we have seen how web technology can be used to ease the process of publishing distributed user interfaces for musical expression.

3 Requirements

3.1 Introduction

Before starting the design of the platform, a consideration of the available technologies was made. This chapter has as main goal to identify limitations of the available technologies that could influence the design and implementation of the eventual tool. At the end of this chapter we combine the gathered insights from our literature study and our technical analysis in a set of requirements that will serve as the requirements for the design and implementation of our solution.

3.2 Communication Protocols

Controllers will have to communicate their instructions to the audio processing unit. There are several different protocols and technologies that can be chosen to do this. In this section we will give an overview of them and finally conclude which one suits our goals the best.

First a summary of communication protocols specifically targeting at communication between multimedia devices will be given. This protocol will be used to communicate user triggered events, such as notes being played and

Command	Meaning	# parameters	param 1	param 2
0x80	Note-off	2	key	velocity
0x90	Note-on	2	key	velocity
0xA0	Aftertouch	2	key	Aftertouch
0xB0	Continuous	2	controller #	value
0xC0	Patch change	1	instrument	
0xD0	Channel Pressure	1	pressure	
0xE0	Pitch bend	2	lsb (7 bits)	msb (7 bits)
0xF8	Timing Clock			
0xFA	Start			
0xFB	Continue			
0xFC	Stop			

Table 3.1: MIDI message specifications

continuous control of musical parameters. Given the nature of these events interoperability, accuracy and flexibility will be important traits in the final conclusion.

3.2.1 MIDI

MIDI is a communication protocol to send messages between music hardware that contains information about pitch and velocity, control signals and clock signals.

MIDI messages are minimum 1 byte and maximum 3 bytes in size. The message always starts with a command byte specifying what type of command is being sent. Possible commands that are relevant to this research is summarised in Table 3.1.

The Note-on and Note-off command are sent when a user respectively plays a note, and releases a note. Typically this will be used to play pitch controlled sounds that correspond to the notes in western music notation. The range of possible keys that can be played is 0 to 127, C-2 being 0 and G8 127. The key parameter for example to send a C3 note is 60, C3# 61 etc. However, notes can also be mapped to the individual parts of a drum kit or to trigger certain events for example on a sequencer.

In most operating systems such as macOS, Windows or Linux distributions connected MIDI devices can be inspected and configured. A developer can query the system to list available MIDI Ports. Audio software can subscribe to these ports and listen for incoming messages. The messages can

then be mapped to one or more parameters in the audio software.

Operating systems typically also allow to register virtual MIDI ports to set up local communication on the same machine via MIDI. This can be useful and good practice to keep the coupling between different components of the software as low as possible.

3.2.2 Open Sound Control

The Open Sound Control (OSC) [41] is a communication protocol designed by the Center for New Music and Audio Technologies (CNMAT), Berkeley, with focus on communication between multimedia devices. It is designed for optimal use with modern networking technology in a client/server architecture. Every node in a network receiving OSC messages is a server, the ones that are sending OSC messages are the clients. Unlike other protocols such as MIDI it is not designed with western notation as underlying idea. It is merely a high level messaging system with a wide range of application areas, such as sensor-based electronic musical instruments [35].

The basic unit for OSC data is a *message*. OSC messages consist of three parts:

- **an address pattern**, specifying the entity or entities within the OSC server to which the message is directed. Address patterns always begin with the character '/' (forward slash).
- **a type tag string**, specifying the data type of each argument.
- **arguments**, the data the message is transferring.

The supported data types are, *int32* and *float32* to send numerical data, *OSC-timetag* to send timestamps, *OSC-string* for textual data and address patterns and *OSC-blob* for arbitrary data.

Popular music programming environments such as Max/MSP, Pd and SuperCollider have implemented the OSC protocol. Even though OSC is being widely adopted in these environments there is no sign today for it to be implemented in the World Wide Web Consortium (W3C) standard. There are however JavaScript libraries[1] that have implemented OSC such that the web browser can be used as a OSC client.

[1] `https://github.com/colinbdclark/osc.js`

3.3 Web Protocols

As the purpose of this framework is to serve as a control interface for musical expression, one of the biggest concerns in making a decision on a communication technology is latency. When playing music it is of very high importance that the delay time between triggering an event and the desired effect is minimised. Therefore a study has been made of the possibilities that browser based solutions have to offer with today's web standards.

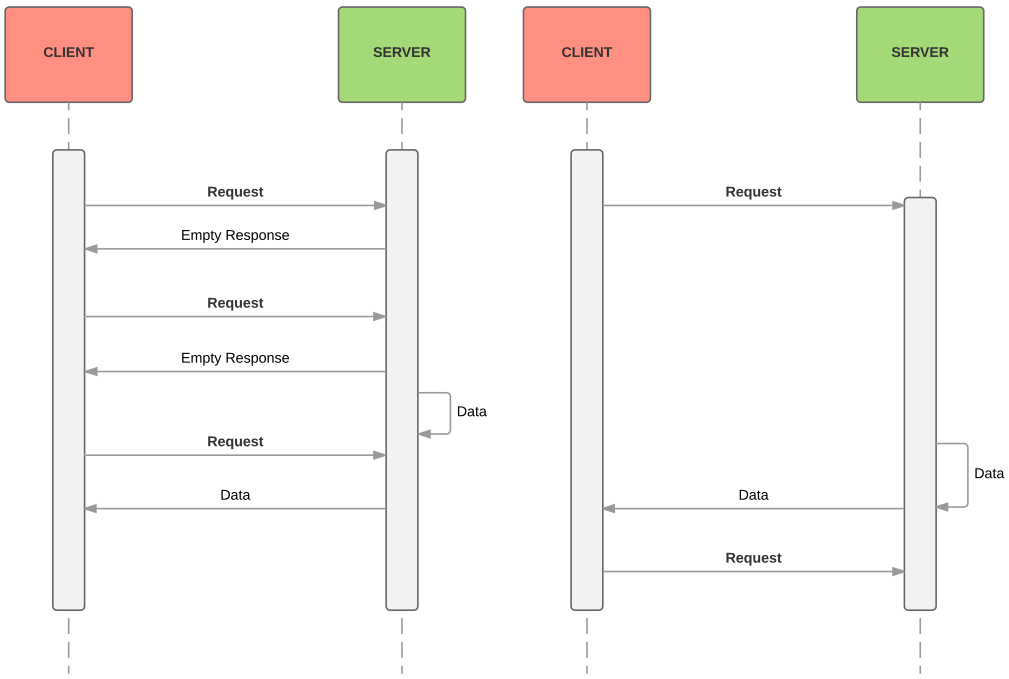
For now the assumption is made that the clients are the nodes in the network that will be controlling a audio processing unit on another node, namely the server node.

Since there will be certain events that will send many events at the same time we will need a communication technology with as little overhead as possible. For example in the case of a slider, when a user manipulates it from its minimum position to its maximum position in 1 second, this means there will be 127 events sent during that second. The user can for example move the slider slowly in the beginning and faster towards the end. In that case we want an accurate reflection on the server side of what the user has been doing on the client side.

3.3.1 HTTP

The HyperText Transfer Protocol (HTTP) is the most widely used communication technology on the World Wide Web. It is implemented in all Web Browsers on top of the Transmission Control Protocol (TCP). HTTP works with a request/response system that has a set of request methods, the most relevant methods being:

- GET, used to retrieve whatever information at the request-URI
- PUT, used to request that the server accept information enclosed in the request and store at the request-URI. Would it be that the information already exist at specified location, then it should be updated.
- POST, used to request that the server accept information enclosed in the request and store at the request-URI
- DELETE, used to request that the server deletes the information stored at the request-URI



(a) Short Polling sequence diagram (b) Long Polling sequence diagram

The HTTP protocol is stateless, meaning that the server stores no information about the client, and thus individual requests will not be associated with each other.

Every request should always be initiated by the client, leading to a response of the server. This means that in certain scenarios where we want to reflect the state of the server over a certain period in time two strategies can be applied.

One being *Short Polling*, this strategy sends over a certain time interval requests to the server. The server will respond with the changes to the state, if any. If no changes occurred in the state the server will respond with an empty message.

Another strategy is *Long Polling*, reducing the amount of overhead that comes with the many requests being sent in short polling. In this strategy the server will only respond to the client when new information is available. While the server waits for new information to be available the communication line remains open. When there is new information available the server sends a response with the new information. As the client receives the new information he restarts the cycle by sending a new request.

We can conclude that as is done in early versions of NexusUI [2] HTTP can be used to send state changes of the UI. The server can then process

incoming HTTP request and translate to the corresponding MIDI message.

However, we see the following issues concerning HTTP:

- Header overhead: since there is no other option using HTTP than to communicate via HTTP request an response messages, the header can create an unnecessary overhead during a long-poll request or long-poll response. Since the aim is to send small amounts of data, but very frequently this could become an issue.

- Allocated resources: establishing a long-poll connection will cause the operating system to allocate a certain amount of resources. Since these connections will remain open until new information is available the cost of this can be high.

3.3.2 WebSockets

WebSockets were introduced to confront the need to low latency applications, due to the overhead that exists in application using HTTP such as long polling, latency can be a problem. It provides a full-duplex channel over a single TCP socket. The WebSocket protocol has been standardised by the World Wide Web Consortium (W3C) and is implemented in most modern web browsers. Because the WebSocket protocol is even-driven the server can broadcast data to the clients at any time. There is no need to poll the server.

A connection is established by sending a HTTP request with an *upgrade header* to inform the server that the connection has to be upgraded to a WebSocket connection. The server response with the 101 response code, indicating that the upgrade of the connection was successful. This phase in establishing the connection is called the handshake.

3.3.3 WebRTC

WebRTC (Web Real-Time Communication) is a collection of protocols that enable real time communication on the web over peer-to-peer connections. It is being standardised by the World Wide Web Consortium (W3C) and the Internet Engineering Task Force (IETF). WebRTC empowers the web browser with real-time communication between other web browsers, this enabling the browser to be used for applications such as real-time video conferencing, file transfer and multiplay gaming.

WebRTC consists of 3 major components:

- **getUserMedia** Allows the web browser to access the system level microphone and camera, as well as to capture these media.

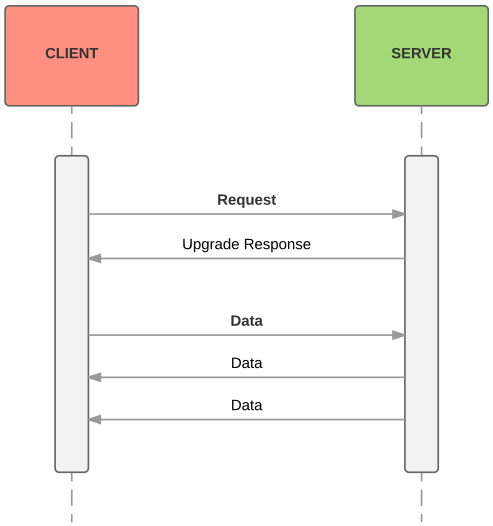


Figure 3.2: WebSocket protocol sequence diagram

- **RTCPeerConnection** Sets up the web browser to have stable and efficient communication of streaming audio or video data between peers.
- **RTCDataChannel** Sets up the web browser to have stable and efficient communication of arbitrary data between peers.

For the purpose of this research RTC datachannels are of main interest. However WebRTC datachannels use P2P to communicate data a server is needed to coordinate the communication between peers. The server is used to do the signalling needed to initialise, manage and close sessions. The WebRTC standard does not require a fixed strategy to be used to do this. Any full duplex communication layer such as HTTP or WebSockets can be used to do this signalling.

In order to provide every computer with a unique IP address to communicate with each other a Network Address Translation (NAT) strategy has to applied, allowing devices such as routers to act as an agent between public and private networks. Using NAT an entire group of computers can be hidden behind one single unique IP address.

To overcome this complexity during the set up of a P2P connection WebRTC uses the Interactive Connectivity Establishment (ICE) framework. Hence, to establish a P2P connection a ICE server address needs to be passed through. The ICE server will try to find the most efficient option to set up this connection. First it will try to make the connection using the IPaddress

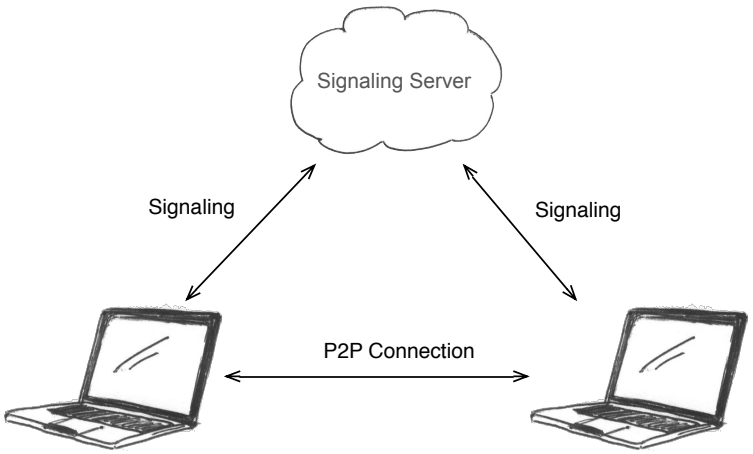


Figure 3.3: WebRTC signalling architecture

	WebRTC (SCTP)	WebSockets
Reliability	configurable	reliable
Delivery	configurable	ordered
Transmission	message-oriented	message-oriented

Table 3.2: Comparison WebRTC and WebSockets

found on the machines OS. If that fails, for example when the device is behind a NAT, it will try via a Session Traversal of User Datagram Protocol Through NAT's (STUN). The STUN server will try to resolve the device's address to it's external address. If that fails, the ICE will try a Traversal Using Relays around NAT (TURN) server to yet resolve the address. A schematic depiction of this is given in Figure 3.4.

Setting up this architecture is not trivial. Several commercial solutions exist that offer signalling services, this making abstraction of the complexity coming with the signalling process. For sake of simplicity we will such service, namely the one provide by peerjs[2] which is free of charge up to limited use.

3.3.4 Conclusion

Given the importance of low latency HTTP polling is excluded from the range of possibilities. To get better insight how WebRTC and WebSockets compare to each other their characteristics are listed in Table 3.2.

[2] `http://peerjs.com`

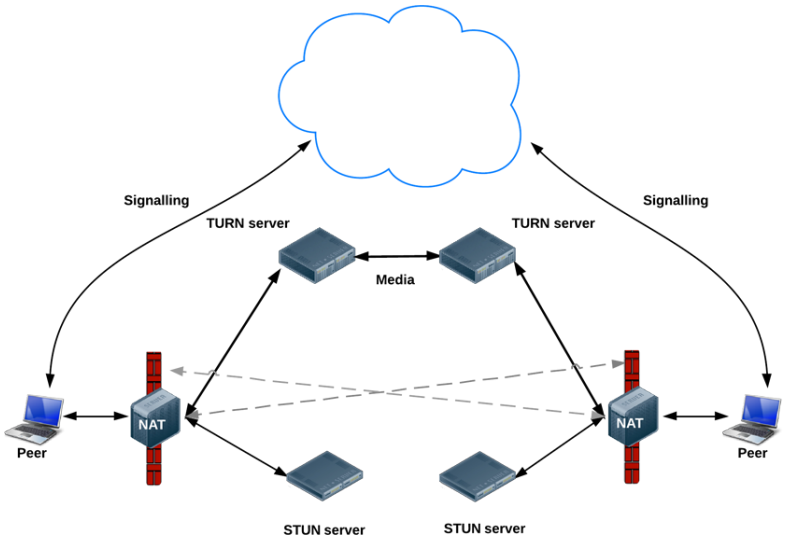


Figure 3.4: WebRTC detailed signaling architecture

In an extensive comparison between WebRTC and WebSockets Karadogan has found that WebRTC is notably faster under similar conditions [17]. Because of this and the lack of need for a server WebRTC was opted for as communication layer for this research.

3.4 Rendering

In this section we will write down our findings from investigating several rendering technologies that exist in a web browser context. This study will help us in deciding to choose the right technology to render widgets when building a DUI for musical expression.

HTML Document Object Model (DOM) elements

Regular HTML DOM elements can be used to build widgets. By styling with CSS and listening HTML DOM events HTML Document Object Model can be used to implement DUI widgets.

Scalable Vector Graphics

Scalable Vector Graphics is an XML based vector image format. They can be styled and animated using CSS and support interactivity. SVG images can be embedded in HTML documents and listen to HTML DOM events such as clicks or hovers. Other than HTML DOM Elements SVG come with

abstractions for drawing shapes such as lines or Bézier curves. However, due to the fact they are DOM elements they require the same resources as DOM elements making them a less attractive option as soon as the amount of nodes is high in quantity. Successful implementations have been done where SVG is being used to create controller widgets for musical expression in WAAX: Web Audio API eXtension [7] and [9].

Several attempts have been made to provide the developer useful abstractions that hide the details of manipulating and creating SVG objects, among which D3.js[3] and TWO.js[4].

HTML Canvas

The HTML canvas element allows the developer to draw shapes using JavaScript. These shapes are rendered using an underlying bitmap. Because the Canvas element only consists of one element it is up to the developer to detect and calculate user events such as clicks for shapes. We have seen the Canvas element being used to implement widgets for musical expression in NexusUI [2].

A JavaScript implementation has been made of the Processing environment named P5.js[5]. A sketch based programming language that by providing abstractions to make drawing to the HTML canvas intuitive attempts to make coding accessible for artists [21].

WebGL

WebGL is a JavaScript API for rendering 3D graphics in the web browser. It is implemented as a feature of the HTML canvas element that allows to use hardware accelerated graphics. This is particularly interesting when rendering computationally intensive graphics such as 3D objects or particle systems.

3.5 Requirements

Out of the literature study a set of requirements can be distilled. Three common requirements for practical DUI application development using a peer-to-peer architecture by Fisher are given [14]. Next, a number of requirements that are based on a set of general design guidelines by Fisher are given.

[3] `https://d3js.org/`
[4] `https://two.js.org/`
[5] `https://p5js.org/`

R1 **Independent Storage and Computation** There should be no dependency between the physical location of storage media, computation, and any input and output devices (besides a network connection) [14].

R2 **Independent Output** Distributed user interfaces should support output on multiple surfaces, potentially each showing only a portion of the visual output to support imagery spanning multiple displays. This requirement is of particular importance for collaboration where multiple individuals may need to observe the output of the distributed user interface application on their own display [14].

R3 **Independent Input** User input should be decentralised so that any device in the network contributes to control over the application. This is particularly important for collaborative settings, where not only a single user but multiples ones may be interacting with the data using multiple devices simultaneously and independently [14].

R4 **Modularity** Widgets should be self contained components that are easy reusable. The developer should have fine grained control over the appearance and behaviour of these components. The user can compose their user interface by combining and distributing these widgets over multiple devices.

R5 **Ease of publishing and connectivity** Users should have no preliminary knowledge about the implementation details in order to distribute the user interface. This means connecting devices, and moving widgets from one device to another should be trivial [18].

R6 **Bidirectional communication** Users should have a consistent reflection of the state of the application at any point in time.

R7 **Visualisation possibility** As a consequence of the requirement that requires bidirectional communication of control messages; the developer should be able to visualise the actions that are coming from either automated events, or events triggered by other users in a collaborative setting [9].

R8 **Mapping Layer** Since we have the ability to design instruments with separable controllers and sound sources, the developer should be able to explicitly design the connection between the two [15].

R9 **Prefer P2P architecture** Performance is a key reason to prefer a P2P network architecture [17]. Multicast communication often uses

UDP instead of for example TCP and hence decreases the bandwidth requirements for parallel streaming of data.

Moreover, the ease of configuration [14] makes of P2P the preferred architecture when designing a DUI. Having to configure and run a dedicated server in order to be able to use a DUI is often not practical.

R10 **Low latency** Playing music is a process that requires very low latency. The time elapsing between a user's action and the sonic sonic response is crucial for the perceived quality of the system.

In Table 3.3 a comparison is given of the systems that are investigated.

framework	R-1	R-2	R-3	R-4	R-5	R-6	R-7	R-8	R-9	R-10
URMUS	✓	✓	✓	✓		✓	✓	✓		✓
control	✓	✓	✓	✓				✓		✓
MIRA	✓	✓	✓	✓		✓	✓	✓		✓
Interface.js	✓	✓	✓	✓	✓			✓		✓
NexusUI	✓	✓	✓	✓	✓			✓		✓
RjDj				✓			✓	✓		
Soundworks	✓	✓	✓	✓	✓	✓	✓	✓		✓
Lemur	✓	✓	✓	✓		✓	✓			✓
GoOSC	✓	✓	✓	✓		✓	✓	✓	✓	✓

Table 3.3: A comparison of related work

3.6 Conclusion

We have seen possible communication protocols for communication between multimedia devices. Both MIDI and OSC are designed to be used in a client-server architecture, this implies that at least one node will be needed to act as a message passing node to which other clients send events. In our implementation we have chosen to opt for MIDI as OSC is not as widely adopted in the platforms we target to control. Most industry solutions still don't support OSC messaging and even though JavaScript implementations exist for most modern web browsers OSC is not supported. This while MIDI is a reliable and well adopted technology in web browsers and industry audio software.

Furthermore we have seen techniques for real time communication such as long polling, Web Sockets and WebRTC. For reasons of performance [17]

and successful earlier implementations with similar functionality [23] we have chosen to implement a peer to peer architecture with WebRTC.

We investigated the possibilities to render widgets in a web context. We have decided to let a degree of freedom to the end user and limit him to use either one of the following three technologies. Depending on the degree of complexity the widgets contain the user can opt to implement widgets as native DOM elements (with SVG), which are easier to implement and style due to the use of JavaScript and CSS. The HTML5 canvas tag, preferably used when the number of separate entities of a widget is high in quantity. Or, WebGL, when rendering is computationally complex and hardware accelerated graphics should be considered in order to have acceptable performance.

Finally based on our literature study and a consideration of possible technologies we defined a set of requirements that our system should meet.

4 Design

4.1 Introduction

Designing a DUI comes with a number of complexities such as synchronisation, resource management, and data transfer [14]. These complexities generally require more effort than when designing a single-device user interface. Moreover as technology evolves, new possibilities arise in the way we publish distributed user interfaces, migrate widgets or synchronize state. Working with cutting edge technology often means sacrifices have to be made in one place to gain either performance or simplicity in other places. In this chapter we will motivate the design choices we have made to deal with these problems.

4.2 Architecture

As we have listed in Section 3.5 a P2P network architecture is preferred. P2P networking is defined as a form of decentralised networking where all nodes participating are treated as equal nodes and can thus function as either server, client or both. All nodes in the network can take up either a server role or a client role. In a more specific context of distributed user interfaces for musical expression we can distinguish the following roles [2]:

- **user interface** The knobs, buttons and sliders that aim to control the musical experience.
- **mapping structure** The translation of user interface state to instrument specific parameters.
- **communication system** The sending and receiving of user events leading to change in the user interface state.
- **audio production engine** The generation of sound.

The design of a tool for the distribution of the user interface of a musical instrument requires deciding which role should be taken up by which individual nodes in the network and how these nodes should communicate. Allison makes the following distinction [2]:

- **Autonomous Collaborative Performance System** Autonomous collaborative performance occurs when each node in the network is a self contained instrument and creates sound autonomously. However, there can be communication between individual nodes or a central server to achieve synchronisation between the sound producing nodes. This form of collaboration is the closest in analogy to a band performing. Real world examples of this type of collaboration include Urmus [13] or the Stanford Laptop Orchestra [38].
- **Centralised Audio Production, Distributed Interface** Another approach could be to distribute the user interface over the devices involved and generate sound on a separate, centralised, dedicate sound engine. In this scenario there are nodes in the network that merely contribute to the state of the sound generation engine. Because of the centralisation of the sound generation a more powerful computing capacity can be used then the one present on current mobile devices.
- **Responsive Server, Adaptive Interface** In the final scenario the state of the server has to be reflected in every node in the network. This requires bidirectional message passing and thus more coordination. By providing bidirectional message passing the user interface nodes will be consistent with the sound production node at any time. Moreover user interfaces could be altered ad hoc depending on the state of the sound production engine. A possible use case of this approach could be when the artist wants to provide a note triggering interface that adapts the key depending on the position in time the performance is at a particular moment. This scenario assumes that the distribution node for the interfaces is on the same node as the message passing node.

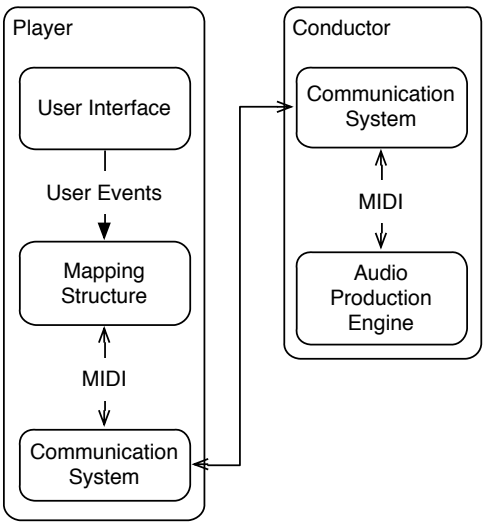


Figure 4.1: Abstract depiction of roles

As requirement R1, R2 and R3 state, we are aiming at designing a tool that aids in the distribution of the user interface role. As shown in Figure 4.1 we will distribute the user interface and mapping layer over multiple devices. User events will there be translated into MIDI messages and sent over a communication system to the device that has the audio production engine. By analogy with the terminology used by DireWolf [18] we will call the network node that communicates with the audio software the *DUI Manager*, nodes that are controlling parameters of the audio software will be referred to as *DUI Clients*.

We will be aiming at controlling a central audio production engine, therefore the architecture will be one with **Centralised Audio Production and Distributed Interface**. A schematic picture of the architecture can be seen in Figure 4.2.

4.3 DUI Client

To enable the user to create MIDI events the environment is enhanced with a number of JavaScript abstractions to create such events and manage the state across individual nodes. These enhancements will handle connectivity set-up and communication with the DUI manager component. These abstractions will be available as a globally exposed JavaScript library.

At load time of a DUI Client the client browser will read the peer id from the URL that is passed through. This peer id will be used in the

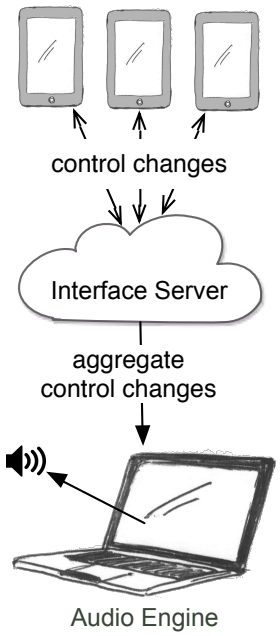


Figure 4.2: Centralised audio production, distributed interface

signalling process to make a real time communication connection with the DUI manager. Moreover the state object will be initialised.

- **initialise** Will parse the peer id that was passed through via the URL and use it to set up a real time communication connection.
- **forEachNote** Will iterate over the notes that are at the time of invocation being played and apply a function to them.
- **getMidi** Will for a certain MIDI value return the MIDI state. This can be for continuous control parameters a certain value, or for notes the velocity value.
- **midi** Will trigger a certain MIDI note to be played or continuous control parameters to be updated to a certain value.

By subscribing to changes of the state of MIDI parameters a visualisation can be made. Each time a MIDI message is received on the DUI manager, the DUI manager will broadcast the state changes to all other DUI clients. Hence a bidirectional MIDI controller can be made (R6). Moreover by observing the MIDI state on the DUI client the state can be visualised in interesting ways (R7).

4.4 DUI Manager

The DUI manager is the central component that handles three tasks. First it handles connectivity with other DUI clients. By setting up a real time communication connection with other DUI clients it enables MIDI message communication and state consistency. MIDI messages received by the DUI manager will be broadcasted to all other connected DUI clients in order to update their state. Widget management, as we want to give the user control over the behaviour and appearance of the widgets R4 that he uses in his DUI we have chosen to let the user implement programmatically individual widgets. The user can use JavaScript, HTML5 and CSS to define custom widgets on the HTML5 canvas tag. Or DOM elements can be used to represent individual widget components that can be styled with CSS. Defining new widgets and persisting them in a dedicated database is the task of the DUI manager component. Finally, widget migration is something that is handled by the DUI manager. The DUI manager is aware of the DUI clients that are connected by a real time communication connection. Previously defined widgets can be migrated from the DUI manager to any connected client.

4.4.1 Security Constraints of the Web MIDI API

Systems like Liine's Lemur[1] or Hexler's TouchOSC[2] have a common architecture where a server is running on the same machine that is running the audio software. The server registers itself as a virtual MIDI device and hence can be discovered for communication with other applications. At the time of writing however it is for security reasons not possible to create virtual output ports with the Web MIDI API. Since it is common practice for MIDI controllers to register itself as a MIDI interface to the operating system, there is a high demand by leading manufacturers in the industry to have that ability[3]. For security reasons however this feature is currently not implemented in common web browsers. This constraint had implications on the design of the architecture for our solution.

The proposed workaround is to augment the browser with an Electron[4] shell. Electron is a framework to build native applications with web technology such as HTML, JavaScript and CSS. It uses a Chromium web browser as a rendering engine and Node.js as platform to do system level operations. Applications can be packaged for macOS, Windows and Linux and an

[1] `http://liine.net/en/products/lemur`
[2] `http://hexler.net/software/touchosc`
[3] `https://github.com/WebAudio/web-midi-api/issues/45`
[4] `https://electron.atom.io`

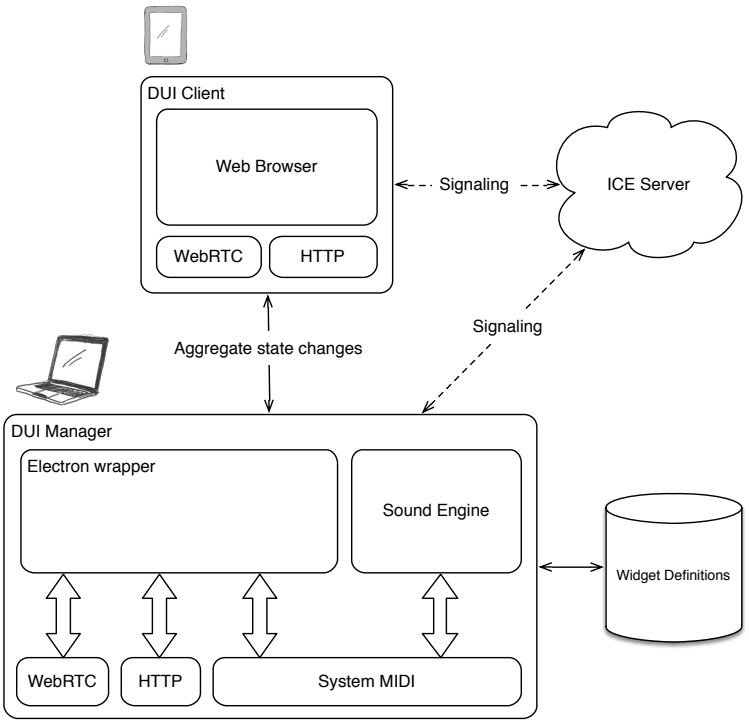


Figure 4.3: Detail of the application architecture

abstraction layer is available to handle inter-process communication (IPC) with these operating systems. Using IPC, MIDI messages can be communicated between the Chromium instance and the Node.js instance running. By registering the application through Node.js as a virtual MIDI channel, the functionality of being able to register as a virtual MIDI from within the browser can be mocked. Because the rest of the application is de facto merely a web site, the problem is neatly isolated and thus easy replaceable by other strategies as technology evolves.

Inside the Electron Shell there will be running an Express[5] web server. This web server will serve the eventual distributed user interface as a web page and will handle communication with the database in order to save and edit widgets.

Finally the ICE server of Peer.js will be used to handle signalling and hence establish a WebRTC data channel connection between individual nodes of the DUI.

When a user saves or runs a widget, the widget definitions will be sent to the server and the application will wait for a response of the server. Once the

[5] `https://expressjs.com/`

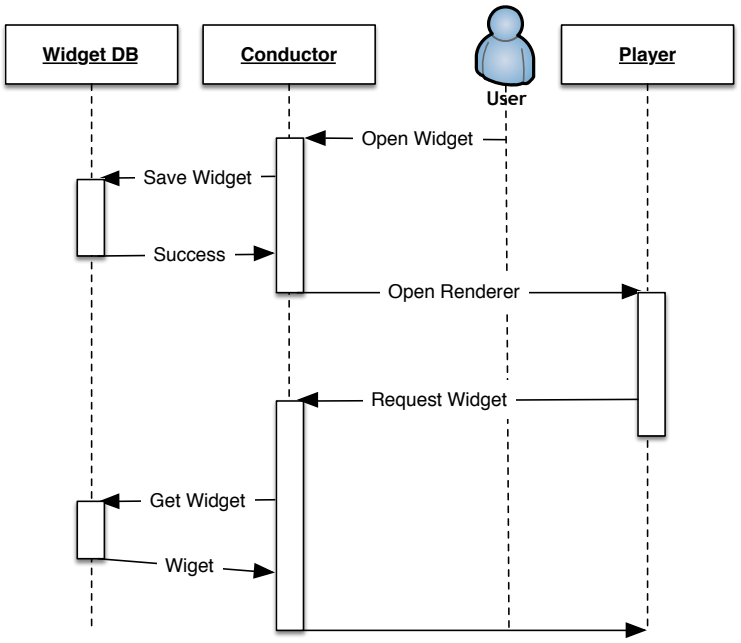


Figure 4.4

server returns an successful response the page will be assembled and served to the user with the widget definitions included. A sequence diagram that illustrates the process of this is included in Figure 4.4.

4.5 User Interface Design

In order to keep the process of distribution and connection as simple as possible, the application is designed to run entirely in the web browser. The user visits the application and can create a new widget by opening a new editor window. A mockup of the home screen is shown in Figure 4.5.

From the home screen the user can connect a mobile device to the application. Therefore you click on the connect button and scan the QR code that is shown. This QR code contains a URL that will redirect the browser on the mobile device to the correct location. Once connected, the connected device will show up in the list of connected devices. The user can drag one of the defined controllers onto the device. When completed, the device will load the selected controller's definitions.

When clicking on one of the defined controllers the editor will open up.

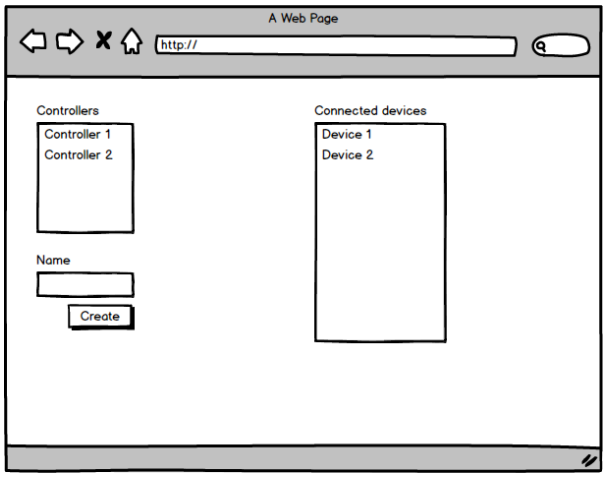


Figure 4.5: The home user interface

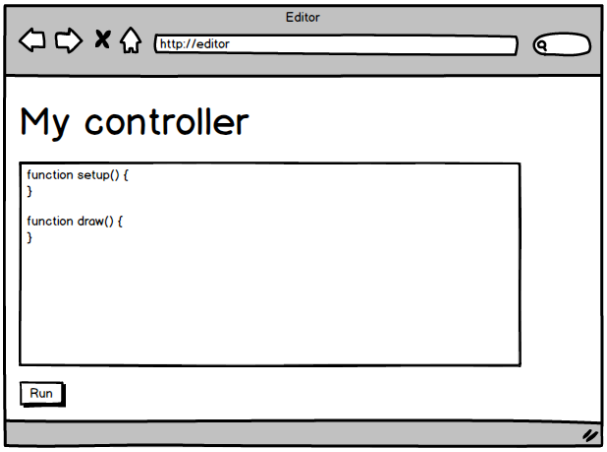


Figure 4.6: Widget editor mockup

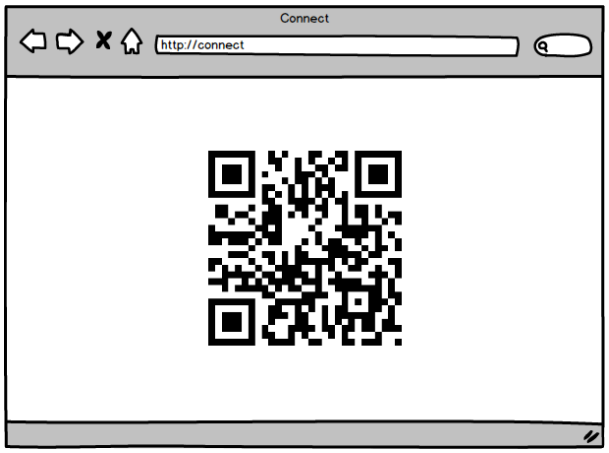


Figure 4.7: Widget connection screen via QR mockup

A mockup of the editor can be seen in Figure 4.6. It contains a text editor where the developer can write the definitions for the widgets. For debugging, the developer can open the widget by clicking the button at the bottom. This will open a new local browser window and load the definitions. As the developer makes changes to the definitions the changes are saved in a database.

4.6 Conclusion

In this chapter we have decided that we will use a P2P architecture with **Centralised Audio Production and a Distributed Interface**. We will have nodes in the P2P network that will take up the role of user interface, mapping structure and communication system that we will call the DUI client component. And there will always be one node that takes up the role of audio production engine and communication system that we will call the DUI Manager. We have seen how new widgets can be added to the system and how they can be migrated from the DUI Manager to the DUI Client.

5 Implementation

5.1 Challenges

In this chapter we discuss the challenges that were faced during the implementation of the DUI Client component as well as the DUI Manager component. In particular we will explain what tools that were used in order to solve the challenges that come with implementing client side web applications, how we structured the project and how eventual new widgets can be implemented.

5.2 Asynchronous Functional Reactive Programming for GUIs

Developing an Electron web app is essentially the same as developing a web application for any other web browser. Hence a large part of the implementation of this system involves JavaScript code. Writing JavaScript web applications can quickly become a very tedious operation if the code is not structured and organised properly. Many efforts are currently being done to aid web application developers with this task. In order to keep the client side code organised Czaplicki presented Functional Reactive Programming (FRP) for GUIs in the Elm programming language [10].

When developing the DUI Manager and Client we followed Czaplicki's data flow model in JavaScript using the React library in combination with a state container. React helps in managing the GUI state by providing abstractions to create reusable components in a declarative way. Because of JavaScript's event-driven, non-blocking nature, Web applications tend to suffer from an overload of callback statements making the application state hard to manage. React overcomes this problem by keeping track of a virtual DOM. Each time the application state changes, React calculates how the changes of the application state map to changes in the real DOM and performs them.

As mentioned React was used in combination with a state container. This means that the entire application state is stored in a single JavaScript object. To change the application state an *action* needs to be dispatched through a *reducer*. Actions are JavaScript objects that contains a description of the event that occurred. Reducers are functions that are guaranteed to have no side-effects and will return an updated state object. Possible asynchronous calls such as for example database API calls will be executed before actions are being dispatched. Each time the state has changed a rerender by React will be invoked. React will thus derive the changes that need to be performed to the DOM and perform them.

5.3 Project Structure

Recently many efforts have been put into adding language constructs to the JavaScript programming language that provide a web application developer with tools that make web application development easier. ECMAScript is a language specification created to standardise the features in JavaScript. At the time of our implementation the latest standard that is supported by most modern web browsers is ES5. This means that in order to use the features of the latest standard (ES7) we need to use a compiler. A compiler will take any valid ES7 code and output semantically equivalent ES5 code.

An important reason why we did this is to be able to use the node package manager in combination with the Webpack[1] JavaScript module bundler. The Webpack module bundler is a node.js applications that takes a JavaScript

[1] `https://webpack.github.io/`

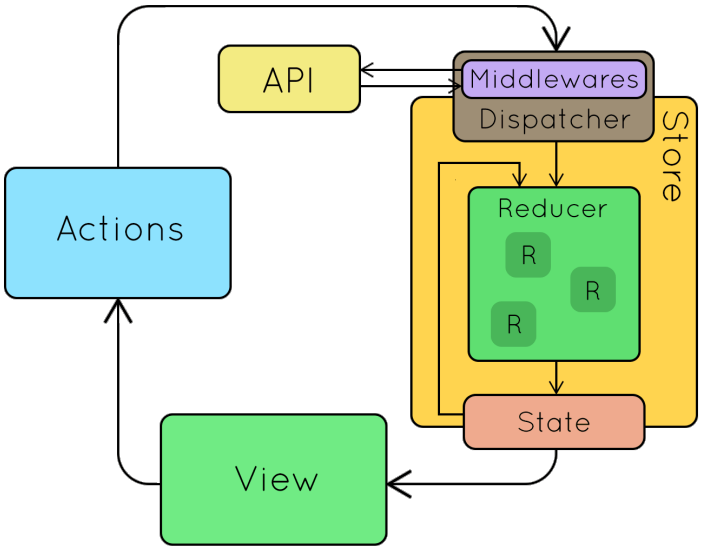


Figure 5.1: Dataflow diagram

input file and bundles all the reachable code from within that file into a single output file. By using ES6 `import` any files can be included in the final bundle. For other file types such as CSS, SVG, images or JSON one needs to specify a so called *loader* that will transform the file as is specified for the loader. A loader for CSS files for example will generate JavaScript code that transforms the css file into a `<style></style>` tag and inlines that tag in the DOM.

This means that our project is structured as three main folders. The first one contains the implementation files for the electron web app (DUI Manager) that is a self contained web application. The second contains the implementation files for the server that will be serving the DUI Manager and the web pages for individual connecting DUI Clients. Finally we have the DUI Client library that is written to automate connectivity and handle state management across the connected DUI Clients. The latter is also a self contained client side web application.

5.4 Widget Definitions

Widgets defined in the DUI manager will be assumed to be valid P5.js sketches. A valid P5.js sketch has either an instance of the P5 object or

has function definitions for the *sketch* and *draw* function. When a web page has no instance of the P5 object, P5.js will assume there are global definitions for the *sketch* and *draw* function, and we say the sketch is running in *global* mode. Otherwise P5 will run in *instance* mode.

P5.js will augment the JavaScript environment with global function definitions that makes drawing to the HTML5 canvas more intuitive for people with no web development background. When the web page is loaded the browser will fire a *'load'* event, when P5.js picks up this event it will execute a initialisation function. The setup function will be invoked when the sketch is loaded. The draw function is an animation loop that will be invoked infinitely. In snippet 5.1 the logic of P5.js's initialisation is included.

Listing 5.1: Initialisation method for P5

```
var _globalInit = function() {
  if (!window.PHANTOMJS && !window.mocha) {
    if(((window.setup && typeof window.setup === 'function') ||
       (window.draw && typeof window.draw === 'function')) &&
       !p5.instance) {
      new p5();
    }
  }
};
if (document.readyState === 'complete') {
  _globalInit();
} else {
  window.addEventListener('load', _globalInit , false);
}
```

As we will be loading sketches dynamically, for example when migrating them from the DUI manager to a DUI client we need the the sketches to be self contained objects. This means that the scope of widgets can not interfere with the scope of widgets that were earlier loaded on the same DUI client. Moreover we might want two widgets as two canvas on the same DUI client and position them absolutely on the web page. Therefore it is needed that we work in instance mode.

To get a better understanding of how a sketch can be used to draw an example of a simple sketch in instance mode that draws a circle is elaborated. The code for the example can be seen in code snippet 5.3.

Listing 5.2: Creating a simple sketch

```
new P5(function (s) {
    s.draw = function () {
        ellipse(50, 50, 80, 80);
    }
});
```

This code will set up the canvas and draw a circle on it. Adding interactivity such as clicks or hovers is done as follows.

Listing 5.3: Creating an interactive sketch

```
var hit = false;
new P5(function (s) {
 s.draw = function (){
    s.noStroke();
    s.ellipse(50, 50, 80, 80);

    hit = collidePointCircle(mouseX,mouseY,50, 50, 80); //see if the mouse
        is in the circle

    if(hit){ //change color!
   s.fill('purple')
    }else{
   s.fill('green')
    }
 }
}
```

5.4.1 State consistency

When we want that user actions invoke MIDI messages being sent we do the following.

Listing 5.4: Creating an interactive sketch

```
var hit, state, note;

new p5(function (s) {
    s.setup = function () {
       state = midi;
       note = teoria.note('C3')
    }
 s.mousePressed = function () {
      hit = s.collidePointCircle(s.mouseX,s.mouseY,50, 50, 80); //see if the
          mouse is in the circle
      if (hit)
        Ludo.midi(state, [Ludo.NOTE_DOWN, note.midi(), 0x7f])
 }
 s.mouseReleased = function () {
   hit = false;
   Ludo.midi(state, [Ludo.NOTE_UP, note.midi(), 0x7f])
 }
 s.draw = function (){
      s.noStroke();
      s.ellipse(50, 50, 80, 80);

      if(hit){ //change color!
        s.fill('purple')
      }else{
        s.fill('green')
      }
 }
})
```

Keeping a consistent state at any time on every device involved is one of the requirements that were listed earlier in the specifications R6. State consistency of MIDI devices commonly happens by sending at the end of a series of state change inducing events emitting a MIDI message with the eventual state of the affected parameter. In Figure 5.2 a depiction of the synchronisation process can be seen of a MIDI device (Controller 1) changing a value from 1 to 63 at channel 1. After the controller has been idle for a short amount of time the audio software will send out to all listening controllers a state change for channel 1 to 63, updating all devices. During a first implementation we ran into the problem of feedback loops in the control messages sent. In order to solve this a decoupling between the widgets and the state must be made.

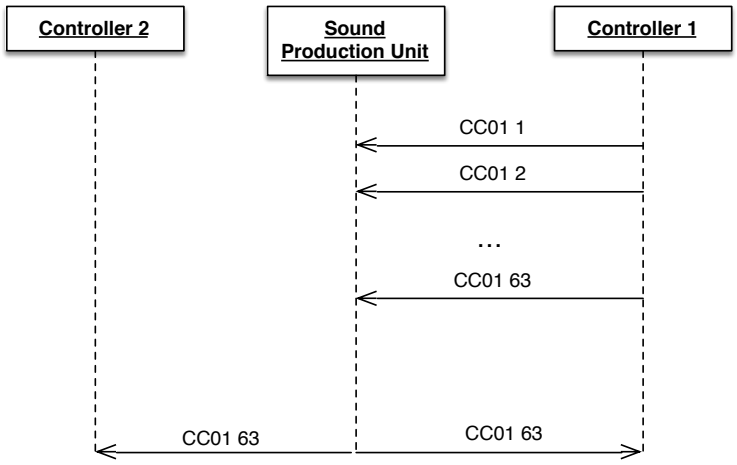


Figure 5.2: Synchronisation sequence example

To explain the problem the example case of the slider will be further elaborated. Typically a callback is subscribed to the event changes occurring to the slider. In this callback the changes will be propagated to the audio software. The audio software will then notify other MIDI devices of the state change by emitting the changes. This change will be picked up by the device where the event is originating from, leading to the callback being triggered again. This will cause jittery behaviour when manipulating the parameters.

The solution opted for is to decouple the application state from the rendering of the view side of the application. This decoupling is done by storing the application state in a JavaScript object. The state of the widgets will be read from the state object. Whenever changes happen locally, the state

object will be adapted and the changes will be communicated to the audio software. State changes originating from the audio software will merely cause a change in the state object. These change can be either read in a next cycle of the rendering loop or one can subscribe a callback to a state change in the state object.

5.5 Electron Inter Process Communication (IPC)

As explained in section 4.4.1 the Electron shell was introduced due to the security constraints of the WebMIDI API. To overcome this shortcoming we used the Node.js library *node-midi*[2], a C++ addon for Node.js that extends the functionality of Node.js with system level operations such as requesting a list of registered MIDI devices or registering a Node.js instance as a virtual MIDI client.

We then use Electron's IPC to communicate through the security layer that web browsers have by registering as a virtual MIDI client and passing through all messages. On the client side of our Electron app we then dispatch the received MIDI message that in their turn will cause them to be forwarded over the WebRTC datachannel to the DUI Clients. Code snippet 5.5 shows how this is done.

Listing 5.5: Electron IPC

```
let midiOut = new midi.output(),
  midiIn = new midi.input(),
  name = 'virtual midi'
try {
 midiIn.ignoreTypes(false, false, false)
 midiIn.openVirtualPort(name),
 midiOut.openVirtualPort(name)
} catch(err) {
 console.error(err)
}
midiIn.on('message', (deltaTime, message) => {
 mainWindow.webContents.send('midi', message)
})
ipcMain.on('message', (event, arg) => {
 midiOut.sendMessage(arg)
})
...
electron.ipcRenderer.on('midi', (event, message) => {
 dispatch(midiMessage(message))
})
```

[2] https://github.com/justinlatimer/node-midi

6 Use Cases

In this chapter we will demonstrate how the tool eventually can be used by artists willing to control musical software. We show how the frameworks works from a practical viewpoint and we document the scenarios with code snippets and screenshots where relevant.

6.1 XY-controller

A very common controller widget used in electronic music production is a touch controller for an audio filter. This type of controller maps the Y coordinates of touch interaction to the amount of resonance off the filter and the X coordinates to the frequency on which the filter will be operating.

First of all we will have to decide which MIDI channels that we will need to reserve for this purpose. In this example we choose to send X-axis events on channel 4, Y-axis on channel 5. As an extra feature we will make the filter be bypassed whenever the user is not operating it. Therefore we reserved channel 6 to send a toggle event whenever the user starts manipulating the controller.

Next, in the draw loop we will draw an ellipse that has as X- and Y-offset the appropriate values out of the state. When the filter is toggled, i.e. when the state queried at toggleChannel is not equal to zero we will change the

background colour and the colour of the circle. At this point the widget will reflect the state of the audio application will be reflected. As the values change the circle will move and as the filter get bypassed the background will change colour. The code for this can be seen in Listing 6.1.

Listing 6.1: XY-controller draw definition

```
new p5(function(sketch) {
  var r = 100
      w = sketch.windowWidth,
      h = sketch.windowHeight;

  var XChannel = 4,
      YChannel = 5,
      toggleChannel = 6;

  sketch.draw = function() {
    sketch.clear();
    sketch.background('rgba(0,255,0, 0.25)');

    if (state.getState().dataChannel[toggleChannel]) {
      sketch.background('rgba(0,255,0, 0.25)');
      sketch.fill("red");
    } else {
      sketch.fill("black");
    }
    sketch.ellipse((Ludo.getMidi(state, [XChannel])/127)*w,
    (1 - Ludo.getMidi(state, [YChannel])/127)*h,r,r);
  };
});
```

We want however also to be able to control the application state in the opposite direction, i.e. the controller controlling the audio software. Therefore we need to define the `touchStarted, touchEnd` and `touchMoved` event handlers. In the `touchStarted` and `touchEnd` we will use the abstractions that the framework offers to handle MIDI communication, and thus send the toggle events to the audio application. The framework will at the same time handle state synchronisation and other devices in the network will get updated.

Finally the `touchMoved` event will map the X and Y coordinates of the touch events to MIDI values ranging between 0 and 127 and send them to the audio application in similar fashion as we did with the toggle event. Code listings are included for the definitions of these event handlers in Listing 6.2. A screenshot of the DUI Manager editor view with the full function definition of the XY controller can ben seen in Figure 6.1.

Listing 6.2: XY-controller event handlers

```
  sketch.touchStarted = function(){
    Ludo.midi(state, [176, toggleChannel, 127]);
    let x = (sketch.mouseX/sketch.windowWidth)
    let y = (sketch.mouseY/sketch.windowHeight)
```

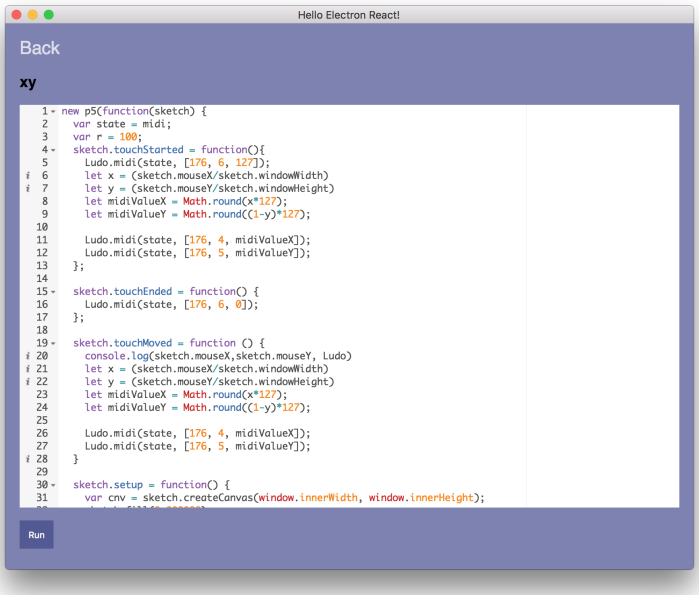


Figure 6.1: XY Widget implementation view in the DUI Manager

```
    let midiValueX = Math.round(x*127);
    let midiValueY = Math.round((1-y)*127);
    Ludo.midi(state, [176, XChannel, midiValueX]);
    Ludo.midi(state, [176, YChannel, midiValueY]);
  };
  sketch.touchEnded = function() {
    Ludo.midi(state, [176, toggleChannel, 0]);
  };
  sketch.touchMoved = function () {
    console.log(sketch.mouseX,sketch.mouseY, Ludo)
    let x = (sketch.mouseX/sketch.windowWidth)
    let y = (sketch.mouseY/sketch.windowHeight)
    let midiValueX = Math.round(x*127);
    let midiValueY = Math.round((1-y)*127);
    Ludo.midi(state, [176, XChannel, midiValueX]);
    Ludo.midi(state, [176, YChannel, midiValueY]);
  }
```

Once the user has defined the widget he can locally preview what it looks like by clicking the run button. After returning to the home screen, the connected peers will show up in the left bottom corner of the home screen. The user can then choose to migrate the widget to the desired device by dragging it to one of the peers in the list. A screenshot of the DUI Manager home screen with a set of predefined widgets is shown in Figure 6.2.

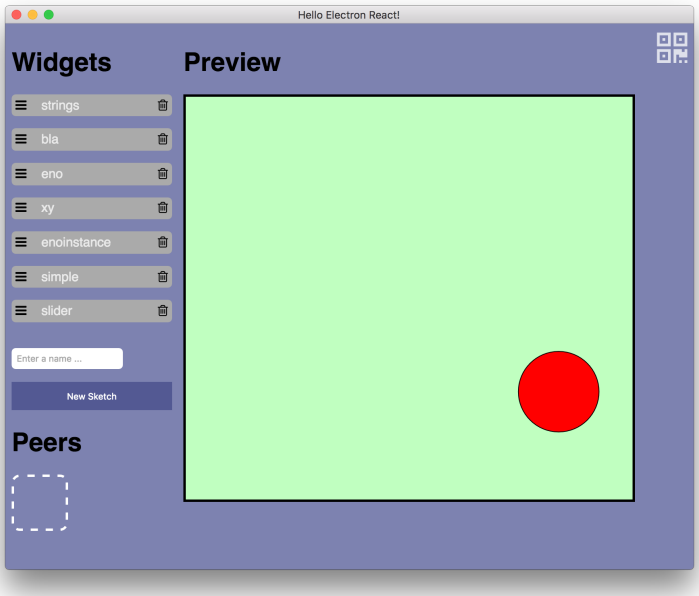


Figure 6.2: DUI Manager with a preview of the XY controller widget

6.2 Visualisation

As requirement R7 states another aspect of the tool we implemented is beside the ability to control music also enable to reflect an artists actions for an audio by means of visualisations. As an example for this we created a widget that is reactive to the actions of an artist playing notes.

As an artist is playing notes in the music software, MIDI events are being broadcasted to the connected DUI clients. Here we transformed the notes being played to their corresponding physical frequency and draw the corresponding physical waves for each note. As the artist plays chords he can see the harmonics occurring in a visual way. An example of the visualisation while two notes are being played can be seen in Figure 6.3.

As another more advanced creative coding use case we made a visualisation that could be used in a concert setting. A common used technique in creative coding is Delaunay triangulation. For this use case we have taken a set of random points on the screen of an audience member and applied Delaunay triangulation to it as can be seen in Figure 6.4. We mapped the filter frequency parameter from the first use case to the parameter that determines the color for each triangle. As a result, when the artist changes the frequency of the filter the visualisation will evolve accordingly.

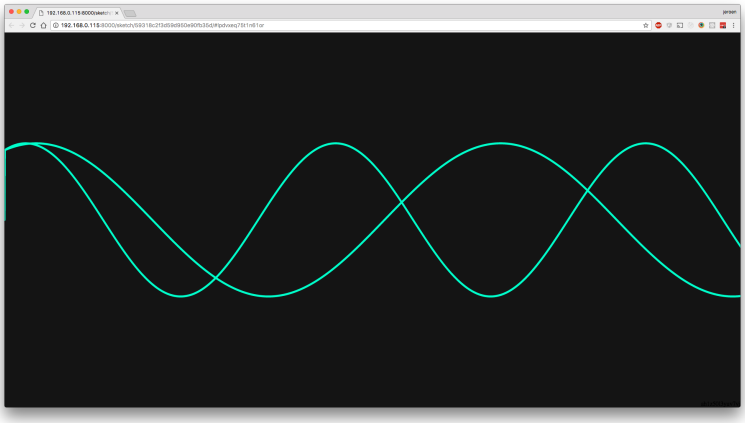


Figure 6.3: Visualisation of harmonics being played by the artist

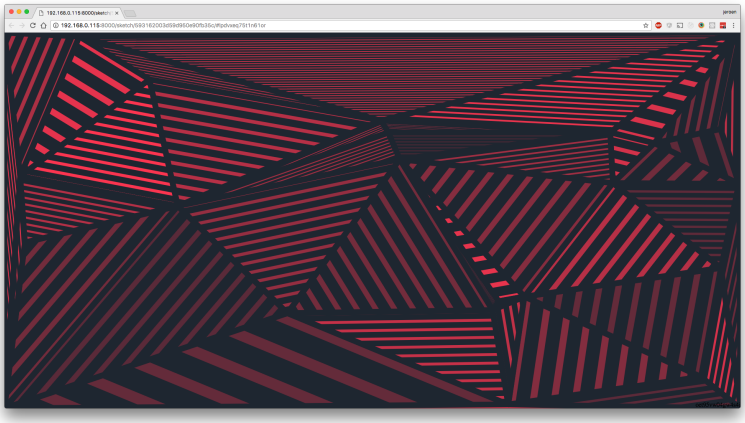


Figure 6.4: Delanauy triangulation mapped to the filter frequency

7 Evaluation

7.1 Introduction

In this chapter the evaluation and its results are described. The evaluation contained an experiment in which several candidates were asked to implement and publish a widget, and finally connect to the widget in order to use it as a piano like controller to play music.

7.2 Setup

The tool we developed in the context of this research was evaluated by letting students, PhD candidates and creative coders use the tool for an imposed task. In total 5 participants were involved in the experiment. First the participants were briefly informed about the research goals of this thesis and the eventual contributions that were made in this context. Not all participants that were asked to implement the widget were familiar with the libraries involved, therefore we gave a short tutorial on P5.js and our abstractions handling MIDI communication. Also a general explanation on how to use the GUI to publish widgets and connect mobile devices was given to the participants. After the tutorial the participants were asked to implement a piano widget based on example code from the tutorial. The whole experi-

ment was timed to see how long it would take the participants to accomplish their task.

We asked participants whether they had used any creative coding libraries before, whether they have programmed in JavaScript before and if they were familiar with music theory. Finally the participants were given a computer system usability questionnaire based on research by Lewis that applies psychometric methods to measure user satisfaction of software systems [20]. The used CSUQ questionaire lets the participants answer questions related to the usability of the system by answering with a score between 1 and 8 for each question. 1 meaning they strongly disagree and 8 they strongly agree.

7.3 Results

None of the participants had extreme difficulties to successfully accomplish their task. It took no one longer then 20 minutes to implement, and publish the widget with the tool. After completion they played on the piano for several more minutes.

The questions of the questionnaire being used can be found in Appendix A.1. In Figure 7.1 we can see a graph representing the average score for each of the questions asked.

From questions q9, q10, q11, q15 and q16 we can see that participants were less satisfied with the amount of feedback that is given in the widget definition process. The participants would have found it useful to get more immediate feedback during the process of implementing a widget, and this in the tool itself. One participant noticed that some form of immediate preview of the widget being implemented could improve the experience.

Questions q3 to q8 teach us about how easy the participants think the system is to learn and to become productive with it. We see good scores for these questions.

7.4 Conclusion

Since we use code as a medium to define our widgets it is important that users get useful feedback on what the possible mistakes are in the definitions they provide. Next to the validation for valid ES6 syntax the participants would suggested to have also semantic validation. Currently the user needs to run the widget in order to know whether the code is valid or not. Running the widget will open the user's default configured browser. After that, it suffices to save the widget and to refresh the browser. A possible solution to this

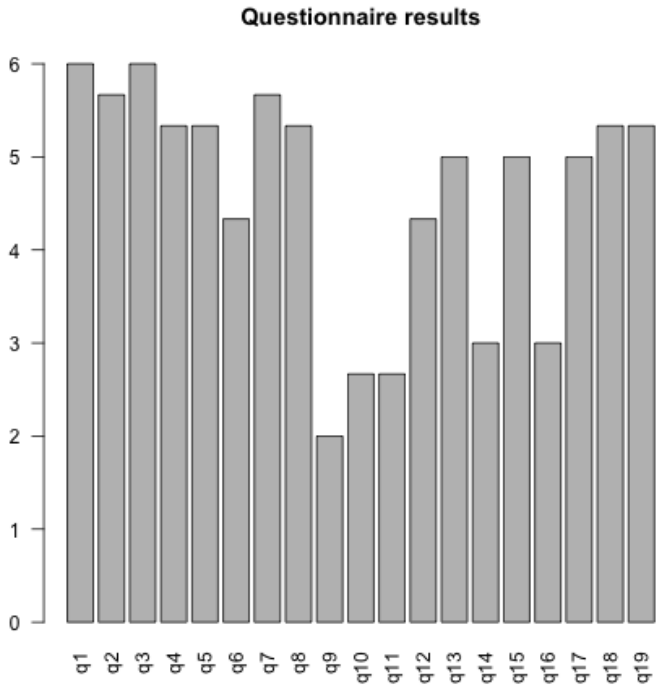


Figure 7.1: Means of the answers to questions being asked to the participants after completion of the experiment

shortcoming can be accomplished by having a real time preview window embedded in the editor view. Other than that, we could look into replacing ES6 with TypeScript. TypeScript is a superset of the ES6 standard that offers optional static type-checking. This would allow us to give better immediate validation of the implementation through type checking and intelligent code completion.

We saw that participants that had previous knowledge of creative coding with systems like OpenFrameworks, Processing or P5.js itself found the task remarkably easier. Even though some participants had no experience with ES6 syntax, no participants had difficulties with it. Some participants remarked that once they felt fluent in using P5.js they were inclined to create more complex widgets. However, since the biggest stumbling block was the learning curve of the libraries involved, a future improvement could be to simplify the implementation of individual widgets. This could be done by splitting up individual widgets in individual sketches, and to allow a controller to exist out of multiple sketches. This would improve the reusability of widgets and free the developer of tasks such as for example collision detection.

8 Conclusions

In this chapter we recapitulate the contributions that the research and development of this thesis has taught us. Finally we go over the limitations and possible future work on this thesis.

While there are many frameworks and tools that distribute user interfaces over mobile touch devices they often lack simplicity in set up. Efforts have been made to overcome this by using the portability of web technology to ease the process of migrating and publishing distributed interfaces for musical expression. However, in contrast to native mobile application, existing work based on web technologies often are limited in the sense of state synchronisation and modularity. The purpose of this thesis was to address these shortcomings in existing work by designing and implementing an extensible framework for creating distributed interfaces for musical expression. In order to achieve this we started by performing a literature study of the field. More specifically we studied the difficulties that arise with modern tools that are used to perform computer music in a solo or collaborative setting. Through this study we gained insights in the history of these tools and technology behind them. In this process we tried to find answers to the questions "How can computer music interaction be made more simple and elegant using mobile touch devices?", "How can computer music instruments be made accessible to artists with little or no programming background?"

and "What new practices are enabled by mobile touch devices to participate in audiovisual experiences?". We gained insights about the matter at hand and found that good musical controllers have well structured instrument and interface design tools. Other than that, we saw that it is important that robust low-latency communication is provided in order to achieve a qualitative user experience. Due to recent W3C standards such as access to local I/O channels for MIDI, low latency communication with WebRTC data channels and the ease of access and portability that the web browser offers, we have seen that the web browser indeed is a viable platform for audiovisual experiences.

Based on these insights we defined a set of requirements that have been used as a base for the implementation of our tool. We created a completely browser based framework that consists of two components. The DUI Manager component enables artists to define widgets using an artist friendly sketch library, P5.js, which we augmented with the functionality to communicate MIDI messages, and artists can migrate widgets to mobile touch devices which will be running the DUI Client component that renders the widgets as defined by the artist in the DUI Manager.

Finally we evaluated the tool that was developed by doing an experiment. Participants of the experiments had to implement a simple piano widget, publish it and eventually connect to it to play music. Prior to the experiment a tutorial was given to the participants that aimed at explaining the context and goals of this research. After the experiments were asked to fill in a CSUQ [20] questionnaire. We learned from the experiment that implementing widgets can be tedious without prior experience in creative coding. However participants with prior experience found the tool comfortable to work with. A general remark was the lack of immediate feedback in the development process of new widgets.

For the future work the tool can be improved by enhancing the experience of developing new widgets. From our evaluation we learned that this can be done by providing the user with immediate feedback in the form of a previewer in the tool that immediately executes the code of the implemented widget combined with possible error messages. Finally it would be useful if a controller consisted of multiple sketch instances that can be arranged on a plane in a drag and drop fashion. Each sketch will be operating on a variable MIDI channel that can be configured in the DUI Manager UI. However due to implementation details of the P5.js library this is currently not trivial.

A Appendix

A.1 CSUQ Questionnaire

1. Overall, I am satisfied with how easy it is to use this system
2. It was simple to use this system
3. I can effectively complete my work using this system
4. I am able to complete my work quickly using this system
5. I am able to efficiently complete my work using this system
6. I feel comfortable using this system
7. It was easy to learn to use this system
8. I believe I became productive quickly using this system
9. The system gives error messages that clearly tell me how to fix problems
10. Whenever I make a mistake using the system, I recover easily and quickly
11. The information (such as online help, on-screen messages, and other documentation) provided with this system is clear
12. It is easy to find the information I needed
13. The information provided for the system is easy to understand

14. The information is effective in helping me complete the tasks and scenarios
15. The organization of information on the system screens is clear
16. The interface of this system is pleasant
17. I like using the interface of this system
18. This system has all the functions and capabilities I expect it to have
19. Overall, I am satisfied with this system

Bibliography

[1] Samuel Aaron and Alan F Blackwell. From Sonic Pi to Overtone: Creative Musical Experiences with Domain-specific and Functional Languages. In *Proceedings of the First ACM SIGPLAN Workshop on Functional Art, Music, Modeling and Design*, FARM '13, pages 35–46, New York, NY, USA, 2013. ACM.

[2] Jesse Allison, Yemin Oh, and Benjamin Taylor. NEXUS: Collaborative Performance for the Masses, Handling Instrument Interface Distribution Through the Web. In *Proceedings of the International Conference on New Interfaces for Musical Expression (NIME)*, pages 1–6, 2013.

[3] MIDI Manufacturers Association and Others. The complete MIDI 1.0 detailed specification. *Los Angeles, CA, The MIDI Manufacturers Association*, 1996.

[4] Christine Bauer and Florian Waldner. Reactive Music: When User Behavior affects Sounds in Real-Time. In *Proceedings of CHI 2013 Extended Abstracts on Human Factors in Computing Systems*, number April, pages 739–744, 2013.

[5] Andrés Cabrera. Serverless and Peer-to-peer Distributed Interfaces for Musical Control. In *Proceedings of the International Conference on New Interfaces for Musical Expression*, pages 355–358, 2015.

[6] Joel Chadabe. Electric Sound: The Past and Promise of Electronic Music. 1997.

[7] Hongchan Choi and Jonathan Berger. WAAX: Web Audio API eXtension. In *Proceedings of the International Conference on New Interfaces for Musical Expression*, pages 499–502, 2013.

[8] Nick Collins and Alex McLean. Algorave: A Survey of the History, Aesthetics and Technology of Live Performance of Algorithmic Electronic Dance Music. In *Proceedings of the International Conference on New Interfaces for Musical Expression*, pages 355–358, 2014.

[9] Nuno N Correia, Atau Tanaka, and Others. User-centered design of a tool for interactive computer-generated audiovisuals. In *Proceedings of 2nd Int. Conf. on Live Interfaces*, 2014.

[10] Evan Czaplicki and Stephen Chong. Asynchronous Functional Reactive Programming for GUIs. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 411–422, 2013.

[11] Niklas Elmqvist. Distributed User Interfaces: State of the Art. In *Distributed User Interfaces*, pages 1–12. Springer, 2011.

[12] Simon Emmerson. *Music, Electronic Media and Culture*. Routledge, 2016.

[13] Georg Essl. Urmus - an Environment for Mobile Instrument Design and Performance. In *Proceedings of the International Computer Music Conference, ICMC 2010*, pages 270–277. International Computer Music Association, 2010.

[14] Eli Raymond Fisher, Sriram Karthik Badam, and Niklas Elmqvist. Designing Peer-to-Peer Distributed User Interfaces: Case Studies on Building Distributed Applications. *International Journal of Human-Computer Studies*, 72(1):100–110, 2014.

[15] Andy Hunt, Marcelo M. Wanderley, and Matthew Paradis. The Importance of Parameter Mapping in Electronic Instrument Design. In *Proceedings of the 2002 Conference on New Interfaces for Musical Expression*, volume 32, pages 149–154, 2002.

[16] Sergi Jordà, Günter Geiger, Marcos Alonso, and Martin Kaltenbrunner. The reacTable : Exploring the Synergy between Live Music Performance and Tabletop Tangible Interfaces. In *Proceedings of Conference on Tangible and Embedded Interaction TEI '07*, pages 139–146, 2007.

[17] Gunay Mert Karadogan. *Evaluating WebSocket and WebRTC in the Context of a Mobile Internet of Things Gateway Evaluating WebSocket and WebRTC in the Context of a Mobile Internet of Things Gateway*. PhD thesis, School of Information and Communication Technology KTH Royal Institute of Technology Stockholm, Sweden, 2014.

[18] Dejan Kovachev, Dominik Renzel, Petru Nicolaescu, and Ralf Klamma. DireWolf - Distributing and migrating user interfaces for widget-based

web applications. In *Proceedings of the International Conference on Web Engineering*, pages 99–113, 2013.

[19] Ssergejewitsch Theremin Leo. Method of and Apparatus for the Generation of Sounds, 1928.

[20] James R Lewis. IBM computer usability satisfaction questionnaires: psychometric evaluation and instructions for use. *International Journal of Human-Computer Interaction*, 7(1):57–78, 1995.

[21] Sierra Magnotta, Anushikha Sharma, Jingya Wu, and Darakhshan J Mir. Creative Computing and Society: When Undergraduates Design a Curriculum for an Introductory Computing Course. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '17, pages 773–774, New York, NY, USA, 2017. ACM.

[22] James McCartney. Rethinking the Computer Music Language: SuperCollider. *Computer Music Journal*, 26(4):61–68, 2002.

[23] Jérémie Melchior, Donatien Grolaux, Jean Vanderdonckt, and Peter Van Roy. A Toolkit for Peer-to-Peer Distributed User Interfaces - Concepts, Implementation, and Applications. In *Proceedings of the 1st International Symposium on Engineering Interactive Computing Systems (EICS'09)*, number April 2017, pages 69–78, 2009.

[24] Jieun Oh, Jorge Herrera, Nicholas J Bryan, Luke Dahl, and Ge Wang. Evolving The Mobile Phone Orchestra. In *Proceedings of the 2010 Conference on New Interfaces for Musical Expression (NIME-10)*, pages 82–87, 2010.

[25] Carlos Palombini. Machine Songs V: Pierre Schaeffer: From Research into Noises to Experimental Music. *Computer Music Journal*, 17:14–19, 1993.

[26] Ken Peffers, Tuure Tuunanen, Marcus A. Rothenberger, and Samir Chatterjee. A Design Science Research Methodology for Information Systems Research. *Journal of Management Information Systems*, 24(3):45–78, 2007.

[27] Ivan Poupyrev, Michael J Lyons, Sidney Fels, and Tina Blaine Bean. New Interfaces for Musical Expression. In *Proceedings of ACM Conference on Human Factors in Computing Systems (CHI)*, pages 491 – 494, 2001.

[28] Miller Puckette. The Patcher. In *International Computer Music Association*, pages 420–429, 1988.

[29] Miller Puckette. Combining Event and Signal Processing in the MAX Graphical Programming Environment. *Computer Music Journal*, 15(3):68–77, 1991.

[30] Curtis Roads. *The Computer Music Tutorial*. MIT press, 1996.

[31] Sébastien Robaszkiewicz and Norbert Schnell. Soundworks - A Playground for Artists and Developers to Create Collaborative Mobile Web Performances. In *Web Audio Conference (WAC)*, 2015.

[32] Charles Roberts and Joann Kuchera-morin. Gibber : Live Coding Audio in the Browser. In *Proceedings of the International Computer Music Conference*, pages 64–69, 2012.

[33] Charles Roberts, Graham Wakefield, and Matthew Wright. The Web Browser As Synthesizer And Interface. In *Proceedings of the International Conference on New Interfaces for Musical Expression*, pages 313–318, 2013.

[34] Charlie Roberts, Matthew Wright, JoAnn Kuchera-Morin, and Tobias Höllerer. Rapid Creation and Publication of Digital Musical Instruments. In *Proceedings of the International Conference on New Interfaces for Musical Expression*, pages 239–242, 2014.

[35] Bert Schiettecatte and Jean Vanderdonckt. AudioCubes: a Distributed Cube Tangible Interface based on Range for Sound Design. In *Proceedings of Tangible and Embedded Interaction (TEI)*, pages 3–10. ACM Press, 2008.

[36] Bill N Schilit, Norman Adams, and Roy Want. Context-aware Computing Applications. In *Proceedings of the 1994 First Workshop on Mobile Computing Systems and Applications*, pages 85–90, 1994.

[37] Benjamin Taylor, Jesse Allison, William Conlin, Yemin Oh, and Daniel Holmes. Simplified Expressive Mobile Development with NexusUI, NexusUp, and NexusDrop. In *Proceedings of the International Conference on New Interfaces for Musical Expression*, pages 257–262, 2014.

[38] Ge Wang, Nicholas Bryan, Jieun Oh, and Robert Hamilton. Stanford laptop orchestra (slork). In *Proceedings of the International Computer Music Conference (ICMC 2009)*, pages 505–508, 2009.

[39] Ge Wang and Perry R Cook. ChucK : A Concurrent , On-the-fly , Audio Programming Language 2 . The ChucK Operator 1 . Ideas in ChucK. In *Proceedings of the International Computer Music Conference (ICMC)*, pages 1–8, 2003.

[40] Rene Wooller, Andrew R Brown, Eduardo Miranda, Joachim Diederich, and Rodney Berry. A framework for Comparison of Process in Algorithmic Music Systems. *Proceedings of Generative Arts Practice 2005 - A Creativity & Cognition Symposium*, 2005.

[41] Matthew Wright, Adrian Freed, and Ali Momeni. Open sound control: State of the art 2003. *Proceedings of the 2013 International Conference on New Instruments for Musical Expression (NIME-2013)*, pages 153–159, 2013.

[42] Nicholas C. Zakas. The Evolution of Web Development for Mobile Devices. *Queue - Mobile Web Development*, 11(2):30–39, 2013.