BUILDING WEB BASED INTERACTIVE SYSTEMS WITH CSOUND PNACL AND WEBSOCKETS

Ashvala Vinay
Berklee College of Music
avinay [at] berklee [dot] edu

Dr. Richard Boulanger
Berklee College of Music
rboulanger [at] berklee [dot] edu

This project aims to harness WebSockets to build networkable interfaces and systems using Csound's Portable Native Client binary (PNaCl) and Socket.io. There are two methods explored in this paper. The first method is to create an interface to Csound PNaCl on devices that are incapable of running the Native Client binary. For example, by running Csound PNaCl on a desktop and controlling it with a smartphone or a tablet. The second method is to create an interactive music environment that allows users to run Csound PNaCl on their computers and use musical instruments in an orchestra interactively. In this paper, we will also address some of the practical problems that exist with modern interactive web-based/networked performance systems – latency, local versus global, and everyone controlling every instrument versus one person per instrument. This paper culminates in a performance system that is robust and relies on web technologies to facilitate musical collaboration between people in different parts of the world.

Introduction

At the 2015 Web Audio Conference, Csound developers Steven Yi, Victor Lazzarini and Edward Costello delivered a paper about the Csound Emscripten build and the Portable Native Client binary (PNaCl). The PNaCl binary works exclusively on Google Chrome and Chromium browsers. It allows Csound to run on the web with very minimal performance penalties.

However, when using PNaCl at the moment, a downside is that plug-in opcodes do not work out of the box. Thus, this rules out the ability to use the new WebSocket opcodes and UDP opcodes that are necessary in order to build a modern interactive system with Csound. To circumvent the necessity for the plug-in opcodes, we have built a wrapper

around Csound PNaCl's javascript API that allows us to network multiple Csound PNaCl instances allowing for:

1. The use of Csound from devices incapable of supporting the PNaCl architecture.
2. The exploration of long distance collaborative improvisation, composition and performance.

1 Interactive systems that use dynamic orchestras

This method uses no fixed orchestra and relies on the compilation of an orchestra that's sent by the users. To create this system, we start with a simple template, an HTML file that contains all the required elements for the interface.

1.1 Code for the front end

```
<html>
<head>
      <script src="csound.js"> </script>
      <script
src="https://ajax.googleapis.com/ajax/libs/jquery/2.1.3/jquery.min.js"></scrip
t>
      <script src="https://cdn.socket.io/socket.io-1.2.0.js"> </script>
      <script src="main.js"> </script>
</head>
<body>
      <textarea class="orc_editor">
      </textarea>
      <button class="send_orc"> Send </button>
      <button class="send_sco"> Send score event </button>
      <div id="engine"> </div>
</body>
</html>
```

This creates a web page that has a text area, and it imports jQuery, Socket.io and Csound's PNaCl javascript interface. In addition, we have also included a *main.js*. In this, we will write our event handling content and make some simple Csound API calls.

From the code below, we have created event handling for interface elements on the web page. Clicking on the "send" button will get whatever content is inside the text area and emit it out via the Socket. When you click on the "send score event" button, it sends a full score-event string to instrument 1, starting at time 0 for a duration of four seconds.

```
function moduleDidLoad(){
      //Csound module load event
      console.log("Csound Loaded")
}
var socket = io.connect("http://localhost:8181"); // connect to server
//socket code here:
socket.on("connect", function(){
      console.log("Socket.io connection established")
}); // connect event

socket.on("orc", function(msg){
      console.log(msg) //print orc
      $(".orc_editor").val(msg) //update text area
      csound.CompileOrc(msg) //compile if browser is csound PNaCl enabled
}); //orc event

socket.on("sco", function(msg){
      console.log(sco)
      csound.Event(sco)
});

$(document).ready(function(){
      $(".send_orc").click(function(){ //react to click event on the orc button
            var str = $(".orc_editor").val()
            //get string from the text area
            socket.emit("orc", str)
            //emit the string out via the socket to the server
      }); //send orchestra

      $(".send_sco").click(function(){ // react to click event on the score
button
            socket.emit("sco", "i 1 0 4")
            //emit a score event out of the socket to the server
      });
});
```

1.2 Code at the Server

On the server side, we use Socket.io and Node.js to create a simple interface that always emits the strings to everyone. We will call this file *bridge.js*.
The code below creates a server that listens on port 8181, sends back any orchestra sent to it, and then broadcasts it to everyone else who is connected to the server. The same holds true for the score events as well.

```
var express = require('express') // requirements
var app = express();
```

```
var io = require('socket.io').listen(8181) // listen on port 8181
console.log("Listening on port 8181")

io.on('connection', function(socket){
      console.log("connected a client")
      socket.on("orc", function(msg){
            io.emit("orc", msg)
            // broadcasts message to all the clients.
      });
      socket.on("sco", function(msg){
            io.emit("sco", msg)
            // broadcasts message to all the client.
      });

});
```

1.3 Details of the Implementation

Starting with the interface, we will explain the code in more detail. The HTML code creates a text area and two buttons. The two buttons are your primary sources for interaction with the socket. The buttons are distinguished by the classes attached to them; the one that sends the whole orchestra has the class "send_orc" and the other, that sends a generic score event, has the class "send_sco".

In the *main.js* file, we attach event handlers that react to a click event on the aforementioned send buttons. Also, in the *main.js* file, we define an interface that interacts with the messages it receives from the websocket to which it is connected. On a click event, registered by one of the buttons, it broadcasts the appropriate socket message to the server.

For the server side, we wrote code that would respond to a message by sending it to everyone. Figure 1 shows how this works. The example shows *client 1* sending the server an orchestra string. The server receives the string and sends it to both *client 1* and *client 2*. In effect, everyone now has the same orchestra and the client-side code is what determines what happens with the orchestra. In our case, we would update the text area and then compile the orchestra we were sent.

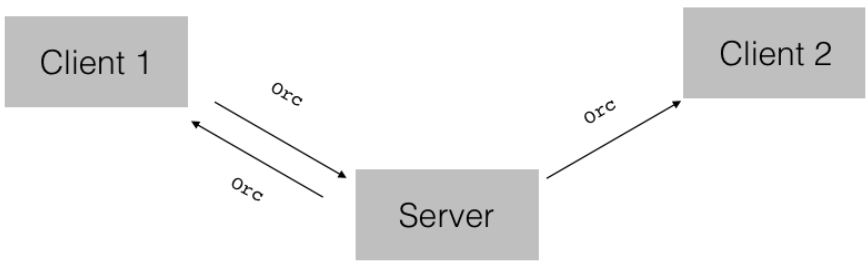

Figure 1 Client 1 sends an orc message

1.4 Observations

In the time we spent with this system, we noticed that latency wasn't necessarily an issue on local area network (LAN) connections. The response to the press of a button on the client side was transmitted instantly to the server and back. However, on the internet, given the overhead associated with sending a message to a remote server and then sending it back to your own or another computer, the latency tended to become more noticeable and more of a musical issue, to a degree, limiting the styles of music one would play and the roles that one could play. We hosted a server in New York and had people from Bangalore, California and Boston connect to it. As you might suspect, the people in Boston were able to hear the sound first, followed by California and then followed by Bangalore.

Another issue was that server broadcasts had no fixed client ordering for score message handling. We can consider latency to be the determinant factor for ordering. However, musically the results can be discouraging, given that there is anywhere between 0 – 300 ms of latency over the internet.

The important and exciting thing was that this approach and system enabled us to control Csound PNaCl from smartphones, tablets and non-Chromium browsers. Opening the website on a smartphone allowed us to type in an orchestra and broadcast it along with the instrument messages thus opening up new musical possibilities and inspiring many new ideas.

2 Static Orchestras and Group performance

In this section we will look at static orchestras and their use in the context of a group performance. We will not go into too much detail with regards to the code in this section, but we will discuss how we approached the problem and what we discovered in the process.

2.1 Approaching static orchestras

Static Orchestras are useful if you want to build:
1. A group environment where everyone gets the same sounds.
2. A laptop orchestra environment where everyone gets to control an instrument.
3. A live-coding system.

These methods are separated as such in relation to where you split your orchestras. The first method has the orchestra split up on the client-side. The second method has the orchestras split up on the server-side and sent to the client individually, and the third method does not rely on splitting up the orchestra.

Socket.io allows us to create "rooms" into which every client can assign themselves. This allows us to correctly and selectively serve orchestras to appropriate rooms and clients and maintain a level of separation. It also allows us to separate score and channel messages based on the group. In effect, multiple groups can use the same orchestra and yet have completely different channel and score messages. This also entails the notion that multiple people could be connected to the same server and yet do a variety of

different things, up to and including: creating their own orchestras; collaborating with others; doing live coding performances; and having interactive orchestra performances.

2.2 Csound live coding environment with Csound PNaCl

As stated previously, using static orchestras allows us to design live coding environments. This section and its subsections will deal with the design and architecture of a simple live-coding setup.

2.2.1 Server-side code

We will work with the *bridge.js* file we used in the first section of this paper. However, we will make a set of small changes to make it work with static orchestras and serve them to a client when they connect to our server.

The changes we have made in the code allow us to now read an orchestra and save it as a string inside of our server. As a way of sending orchestras when requested, we have written a handler for a new event called *request_orc,* and in the next section, pertaining to the front-end code, we will see how this event handling is used in greater clarity.

```
var express = require('express') // requirements
var app = express();
var io = require('socket.io').listen(8181) // listen on port 8181
console.log("Listening on port 8181")
var orc_str;

fs.readFile("StaticOrc.orc", "utf-8", function(err,data){
if (err){throw err;}
orc_str = data
}); //read a file and store it into orc_str - a generic strint

io.on('connection', function(socket){
      console.log("connected a client")
      socket.on("request_orc", function(msg){
            io.to(socket.id).emit('orc', orc_str);
            // send the orchestra just to the client who requests it.
      });
      socket.on("orc", function(msg){
            io.emit("orc", msg)
            // broadcasts message to all the clients.
      });
      socket.on("sco", function(msg){
            io.emit("sco", msg)
            // broadcasts message to all the client.
      });
});
```

2.2.2 Front end code changes

On the front-end side of things, we only manipulate the *main.js* file. We will write an event that will request an orchestra and send it across. We will make a change to the connection event:

```
//socket code here:
socket.on(“connect”, function(){
     console.log(“Socket.io connection established”)
     socket.emit(“request_orc”) //request the orchestra from the server
}); // connect event
```

The new *emit* event allows us to request our static orchestra from the server when we connect.

The remainder of the code in our original *main.js* remains the same. We will instead write a Python-based interface where we will send score events to the browser that has already parsed our orchestra and compiled it. In order to have a more realistic live coding performance, you can still modify the code in the text-area on the fly.

2.2.3 Python interface

This section details the implementation of a simple Python command-line interface, where you can send note events to your server and have a live coding performance.

The sole dependency for this was the socket.io client (*socketIO_client*) for Python, which is available through the official Python package archive.

```
from socketIO_client import SocketIO

def on_connect_function(*args):
    print “connected" # Connected!

class CsSocket:
    def __init__(self, port): #initialization
        self.port = port # Socket.io port
        self.socketIO = SocketIO('localhost', port) # connect to localhost
        self.socketIO.wait(seconds = 1) # check every second
        self.socketIO.on("connect", on_connect_function) # on connect event
function

    def parse(self, str):
        (header, score) = str.split(" “) # split at spaces
        if header == "sco" and len(score) > 0: # check for proper score event
            self.socketIO.emit(header, score) # send score event

if __name__ == "__main__":
    CsSocket_Instance = CsSocket(8181) # create a new instance
    print “sending messages on port 8181”
    while True:
        command_str = raw_input("> “) # User input
        if command_str == "exit":
```

```
        break # exit the loop

    command_str2 = "sco " + command_str # pre-pend sco
    CsSocket_Instance.parse(command_str2) # run through the simple parser
```

The code above creates a command-line interface where you type out score events in the same format as you would have in a Csound file or a score file, e.g "i 1 0 10". Depending on the parameter field (p-field) requirements, you can also expand your score string so that it fulfills these requirements. It also creates a Socket.io client that connects to port 8181 and sends the score message that you enter. Conceivably, the parser function in the CsSocket class can be extended further to send and parse more events – such as chngets, orchestras and influence the interface at the front-end.

2.3 A simple group environment

A simple group environment relies on the client splitting up the orchestra on the client. We will not discuss code examples. We will talk about the details of the implementation instead.

For a simple group environment, we use the same server-side code that we used in the live coding example. However, we made some changes to the orchestra to allow us to divide the orchestra into groups of instruments. We used a separator in the form of a Csound comment "`;- - - - -;`" between instruments that constitute a group.

On the client-side, we wrote a simple function that would look for the separator and split the orchestra into groupings and store the separated groupings into an array. This allows us to map buttons to array indices and display content related to a certain instrument group. An example of this is shown in Figure 2.

Figure 2 A simple Group Environment example

In Figure 2, you can see that we have rendered a knob for the channel “lfo-rate.” Another example of displaying relevant content to a group is shown in Figure 3 where we have rendered buttons for our percussion group that has a kick, a snare and a hi-hat.

Prior to splitting up the orchestra, we compile the Csound code that the client receives from the server. Note that the delimiter/separator is not influential in this scenario as it will be ignored by the Csound parser.

Figure 3 Demonstrating Percussion UI

2.4 A simple laptop orchestra environment

Our simple laptop-orchestra environment is largely similar to the group environment . We have made some changes to the server-side code in this instance. We took the code we wrote on the client-side to split an orchestra and transplanted it into the server-side. This allows us to request a certain orchestra group.

The difference between this and a group environment is that the group orchestra is compiled completely and this means that the client can send score events to every instrument. The laptop-orchestra environment compiles the separated group that is requested by the client. Thus, the client can only send score events to the instruments in the group you requested.

The interface we designed for this is based on the design shown in Figures 2 and 3.

2.5 Observations

We believe that group orchestras work best over a Local Area Network (LAN) connection – given its naturally low latency. Over the Internet, latency is still an issue

and can become an issue when trying to achieve musical results that involve timing events. Clearly, orchestras that have percussion parts and loops that "groove" in time tend to work better on a LAN.

This system can support true live-coding performances. You can implement channel message handlers in your Socket.io code to better handle and parse your *chnset* string and input values. This can be used to affect timbers and the output level of your sounds in real-time.

Musically, we found that separating orchestras by instrument and having a laptop orchestra environment tended to make for a more musical result and a better sense of ensemble playing. This is because everyone is in control of their own sounds and can more tastefully play and mix their instruments with the music and better adapt to the context and flow. And once you implement *chnset* message handlers to mix volume levels correctly, hosting an environment where everyone hears everyone else works out quite well.

Devices that are otherwise incapable of running Csound PNaCl can still participate in a group setting, albeit minimally. They can affect sounds via channel messages and send note events. However, as it stands right now, they are incapable of making any Csound on their own.

It is our hope that this paper gives you some insight and understanding into the many exciting new collaborative composition and group performance possibilities with sockets; and we hope that you will follow this research as we publish an extensive chapter and set of online tutorials in the upcoming MIT Press Book, tentatively titled: *Csound Applied – Csound Inside*.

Conclusions

Through the course of building this system, we have begun to identify and explore new ways of interactively and collaboratively designing, building and performing with Csound. In our daily lives, the Internet is an important tool, and in so many ways, we rely upon, and are literally caught up in, this web. It's fair to say that being able to remotely collaborate with fellow sound designers, performers, composers and coders in real-time over the Internet will allow us to explore music in new, and quite possibly richer and deeper ways. Thus far, we have designed interfaces for iOS, Android, and command-line Python utilities – all to facilitate how we perform with and use Csound. During our presentation, and in the keynote address, we will demonstrate these interfaces and the potential and power of this system as it stands today.