dspNode: Real-time remote audio rendering

Thomas Dodds
thomas@creativelycommon.co.uk

ABSTRACT

The author has been experimenting with various implementations of real-time cloud based audio rendering, keeping the client side application as an extremely light weight remote controller, receiving the fully rendered audio stream from a cloud based audio rendering engine.

The general benefits, drawbacks and conclusions will be discussed with a plausible and functional example application given for the reader's own performance evaluation.

1. INTRODUCTION

Increasing resource requirements of client side rendering can mean complex and innovative rendering is simply unachievable. Although the Web Audio API has provided native processing speed, concurrent audio channel mixing and processing can still put large demands on the devices network connectivity, CPU usage and memory usage. This issue is further compounded when OS level power management and security measures kick in, especially on mobile devices. The client's browser is not always the best place to render audio.

Native local DSP processes are quick and efficient, but when multiple remotely served audio assets (live or prerecorded) are attempted to be served simultaneously bottlenecks begin to surface around network connectivity. The browser's internal javascript engine memory management and the host CPU speed begin to cause issues for the audio engine, causing buffering, dropouts and disruption to playback.

By shifting some of these responsibilities to the server, the intention is to overcome the bottlenecks mentioned, deliver complex audio delivery on mobile devices and to determine its suitability in a production environment exploring the options available around scalability.

For the purpose of demonstration, a simple eight channel audio mixing console has been created, with console functionality such as bussing, pan, reverb, delay and dynamic compression. This example application will be used to demonstrate dspNode and its performance across devices.

Web Audio Conference WAC-2018, September 19–21, 2018, Berlin, Germany.

The intent is to further research and develop the concept to write a generic DSP focused standalone rendering infrastructure that will instantiate, boot and scale simply from a simple client side network request.

As internet speeds have increased, the performance of such systems have now become almost indistinguishable to client-side rendered performance. Given that audio generally requires much less network bandwidth than video, delivery speeds should be sufficient when delivering audio.

Currently, the dspNode client library is available open-sourced on Github [1] with the intent to add a draft of the server rendering code after more testing. A preconfigured arrangement of five dspNodes are set up for the purposes of the demo.

1.1 Related works

Earlier works have explored the field of low latency remotely computed audio streaming such as *CloudOrch: A Portable SoundCard in the Cloud*[2] and open source libraries such as *websockets-streaming-audio*[3]. However, it appears that the published works exploring the subject focuses primarily on the use of HTML5 Websockets to stream the audio data across the internet, being the most appropriate technology at the time. This papers example utalises more recent technologies.

Within the gaming industry, remote rendering has been around since around 2000 and its usage has now become common place with services such as Playstation Now GForce Now[4]. These systems are often proprietary and offer no way to render or stream the audio alone.

2. RENDERING BOTTLENECKS

The performance bottlenecks can generally be broken down into the following areas.

2.1 Memory consumption

The browser has very limited access to the computers storage devices (by design) and must keep all assets within the browser process's sandbox RAM allocation. Careful garbage collection implemented by browser vendors can help to alleviate this issue by removing unneeded resources from memory.

[1] https://github.com/dodds-cc/dspNode

[2] http://www.nime.org/proceedings/2014/nime2014_541.pdf

[3] https://www.npmjs.com/package/websockets-streaming-audio

[4] https://en.wikipedia.org/wiki/Cloud_gaming

However, if we consider multichannel audio mixing, we need all (or at least a lot more) channel data to hand for summing and other related DSP operations. An uncompressed mono audio channel is roughly 50MB, multiply this, by even a moderate eight channels of audio, and we already have nearly 500MB of RAM just for raw audio data assets.

2.2 Networking

Remote media resources are pulled into the client browser session over TCP/IP with the speed subject to the clients own network connection speed. A browser is typically limited to creating ten simultaneous network sockets for TCP/IP traffic.

Innovative streaming solutions such as HLS or DASH streaming have helped to solve this problem by splitting audio and video content into small blocks, fetched from the server on request, removing the need for large memory resources. However, such a system is still limited by the connection bandwidth, unable to pull in large numbers of resources simultaneously to be rendered in real-time.

Browsers typically use TCP protocols to ensure error free delivery of packets over a network. This error handling adds a significant overhead to data transfer speeds especially when it comes to low latency communications.

2.3 Multichannel audio and codec support

Whilst desktop browsers generally have no problem playing back multiple audio objects simultaneously, there are differences in the way audio playback has been implemented on some mobile browsers such as Safari on the iOS.

Multichannel audio and audio codec support varies widely across devices with no standard pattern and limited common ground. Workarounds include downloading uncompressed audio buffers prior to playback, but this puts additional load on the network and memory usage.

Codec support ranges widely across browsers, especially around multichannel support and channel ordering. Although quite tricky to keep track of, a list of supported codecs across devices and browsers is given here [5].

2.4 Host processing speed

While the most common audio DSP operations put a small strain on the host CPU with the Web Audio API, other operations such as real-time convolution reverb or spatial positioning put bigger demands on the hosts processor. This is especially the case when processing multiple channels, each with independent time and frequency based transformations.

Mobile device browsers also often heavily limit the amount of processing speed available. Device specific power saving measures can make this problem worse.

[5] https://developer.mozilla.org/en-US/docs/Web/HTML/Supported_media_formats

3. IMPLEMENTATION

The general aim was to set up an example application that offloads all rendering bottlenecks to the remote rendering engine (dspNode), leaving the client as a simple lightweight control application sending commands upstream, streaming only the fully rendered high quality summed audio mix and status updates downstream.

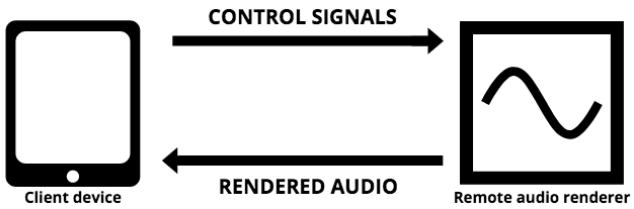

Figure 1. Simple diagram of application concept

Although various technology combinations have been tested to render the audio and send it across the internet, a specific combination as been chosen to focus on as it demonstrates the fundamental aspects of the project well, without too many complexities to get in the way. Other approaches are described later in the paper.

3.1 Control signals

Initially, remote audio engine control signals were implemented over a simple REST interface. This worked well for simple interactions such as transport control but became complicated and slow when finer controls were required. Also, making a new network request for each command added significant delay to the round trip exchange.

WebSockets were tested to communicate between the client and its corresponding remote dspNode. Operating over long polling TCP connection, the protocol reliably allows quick exchange of small data packets to the client and a dspNode instance. Connections are also automatically re-established by both peers if dropped due to network connectivity issues. The problem with Websocket communications is that the signals still need to pass through an additional WebSocket gateway, adding additional latency and infrastructure. Ideally, commands are sent directly to the dspNode.

The RTCDataChannel was chosen as it allows direct communication between the two peers, leaving a WebSocket for peer signaling purposes only, without handling the payloads. The UDP RTCDataChannel can be configured to offer the similar reliability of WebSockets without the need for an external gateway during exchanges of data.

3.1.1 Example control signal payload

```
dspNodeClient.sendCommand({
       channel: 5,
       param: 'gain'
       value: '0.774',
       rampTime: '0.1',
});
```

3.2 Audio Streaming from dspNode

WebRTC is a low latency UDP connection between multiple peers, optimized for real-time audio, video and data communication. This technology is a perfect fit for the proposed system.

Over the past few years, WebRTC library has gained support across all modern browsers and devices, with iOS being one of the last major vendors to join the group of supported browsers in late 2017. [6] Although primarily designed for peer to peer webcam and speech applications, dspNode replaces a peer with a designated remote rendering engine that acts like a typical client.

With WebRTC now consistently supported across all modern browsers and devices (which was not the case at the start of the project), the use of WebRTC is now a viable solution in this context.

3.2.1 WebRTC library

RTCMulticonnection.js [7] is currently used in the project as it provides a very simple interface for creating WebRTC connections and handles the required peer signaling over WebSockets in one simple package. RTCMulticonnection is a simplified wrapper for the browsers native WebRTC API.

3.2.2 Additional WebRTC audio components

WebRTC offers a complete stack for voice communications. It includes not only the necessary codecs, but other components necessary to great user experiences. This includes software-based acoustic echo cancellation (AEC), automatic gain control (AGC), noise reduction, noise suppression, and hardware access and control across multiple platforms.[8]

The additional components are all be disabled at both the dspNode and client ends of the connection as they will interfere with the integrity of the audio signal. Ideally, the audio rendered on the dspNode will match the audio delivered to the client, with the least amount of processing applied in the process. The need for lossy compression in the transport of the audio stream is a given due to client connectivity speeds, so any additional client side processes should be disabled. Additionally, these extra client-side processes are generally aimed at audio containing speech only.

3.3 WebRTC codecs

Table 1. Codec support for WebRTC API[9]

Codec	**Usage**
G.711	Narrowband mono audio, speech, VOIP
DTMF	Telephone control signals
VP8	Video optimised
H.264	Video optimised
Opus	Fullband stereo audio codec

[6] https://caniuse.com/#search=webrtc

[7] http://www.rtcmulticonnection.org/

[8] https://webrtc.org/faq/#audio

[9] https://webrtcglossary.com/codec/

3.3.1 Opus audio codec

Opus[10] is a royalty-free audio codec defined by IETF RFC 6176. It supports constant and variable bitrate encoding from 6 kbit/s to 510 kbit/s, frame sizes from 2.5 ms to 60 ms, and various sampling rates from 8 kHz (with 4 kHz bandwidth) to 48 kHz (with 20 kHz bandwidth, where the entire hearing range of the human auditory system can be reproduced).[11]

The bitrate of the compressed Opus audio feed will ultimately determine the amount of lossy compression applied to the audio signal from source to destination. The frame size will be a large part of determining the packet delivery latency.

Supporting both variable and constant bit rates, various settings can be auditioned to find the optimum audio quality vs latency. Variable bit-rates allow for adaptive streaming, which will adjust its required bitrates based on network conditions.

Intelligent jitter correction buffers implemented in the browser mean packet loss and reordering will not have catastrophic effects on the perceived audio signal.

When initializing a connection with a remote dspNode, the codec parameters are passed in, defining the bitrate ranges, sampling rate and encoding/decoding complexity values.

4. LATENCY

The latency experienced between control signals being sent and the rendered audio being played back on the client device is a crucial factor to the success of the project.

Simple tests have been written to put assess this value across a range of devices and network conditions. We must consider the delay of the control signals from the client, the amount of time taken to apply the DSP parameters and then finally the time taken to deliver the audio stream to the listeners' speaker. All of these delays are variable and can change over time during the stream.

4.1 Control signal latency

A small test was devised to test the round-trip latency of a typical control signal from client to server and back to client. The test can be run by the reader by visiting the link given [12].

The test establishes an RTCDataChannel connection to a simplified dspNode instance via a publicly accessible signalling server. The test client sends a control package to the dspNode over the RTCDataChannel and waits for an echoed reply. The time between when the packet was sent and received is then displayed on screen.

The performance of the RTCDataChannel was more than acceptable and provides the quickest way to send data between peers. Typically, delays observed were under 10ms.

[10] http://opus-codec.org

[11] https://webrtc.org/faq/#what-is-the-vp8-video-codec

[12] https://github.com/dodds-cc/dspNode/blob/master/tests/latencyTests/rtcDataChannelLatencyTest.js

4.2 Audio signal latency

To reasonably accurately determine the audio latency from the dspNode instance to the client, a simple test was devised.[13] Although WebRTC debugging tools are available that can give the reader detailed stats about a connection, a practical real world end to end test gives a more accurate figure. This test will also take into consideration the time required to render the audio stream.

For this test, a dspNode instance is initialised by the dspNode client library which is then used to establish a connection from the browser to the dspNode through a gateway server.

A control signal is sent from the client to the dspNode which loads a test project. A play command is sent from the client which starts one channel of 440 hz test tone that runs for one minute. After the minute, the dspNode process terminates.

Using the Web Audio API, the client then scans all received audio buffers and detects the first non-zero audio sample, which represents the start of the audio signal received from the server. The test compares the timestamp of this non-zero value to the timestamp of the control being sent. The latency is then displayed on screen. The value is also logged by the dspNode and collected. A link to the results recorded can be found at the given link [14]

This is a very simplified test and only begins to accurately assess the latency of the system.

4.1.1 Overview

The latencies experienced were surprisingly low, some as low as below 100ms. The physical location of the dspNode instance and the client connection has a significant effect on the delays experienced. Some additional logic should be added to the gateway to point clients to a physically close dspNode instance (or at least in terms of internet nodes) based on the geolocation information of the client's IP address.

5. DEMONSTRATION APPLICATION

An example application has been created to demonstrate the advantages afforded by the proposed system and test its audio rendering performance. The criteria of the application was to create a multitrack mixer for live sessions to give the audience the opportunity to create their own unique versions of performances.

For the purpose of the demonstration, the audio rendering was limited to eight simultaneous channels of pre-recorded audio, with common mixing console functionality such as volume control, panning, busses, EQ, dynamic compression, delay and convolution reverb, summed to a stereo output.

5.1 Client-side control application

The client-side application acts as a simple remote control, maintaining the state of the mixer controls and using the dspNode client library to send commands to a remote dspNode. The audio that is received from the dspNode is passed into an HTML5 Audio element as a javascript blob and played through the browsers native audio handling.

Each channel has its own individual controls and also controls for the master buss processing for processes such as global EQ, limiting and stereo width.

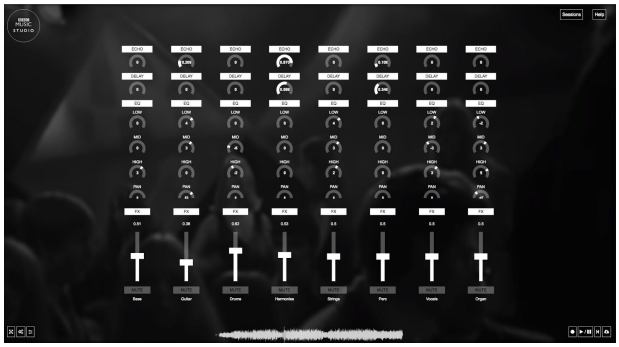

Figure 2. Client side application

The client-side application uses AngularJs[15], peaks.js[16] and jQuery knob[17]. Source code is available on Github[18]

5.2 dspNode audio rendering application

For this particular example, the dspNode audio rendering engine has been implemented with Node Webkit[19]. The focus was to shift network and processing power from the client to the dspNode, testing real world latencies between the two peers.

The audio is fully rendered within NodeWebkit and the rendered audio stream passed over the WebRTC connection established with the client.

The combination of Node and NodeWebkit was chosen as a simple route for demonstration purposes. Experiments with other more efficient audio rendering processes have taken place, examples can be seen in the diagrams supplied[20]. These approaches will be discussed more in the conclusion.

6. INFRASTRUCTURE DEPLOYMENT

For the reader's own performance evaluation, a reservation of five dspNodes have been setup. This setup is being used for demonstrations and performance testing.

WebRTC connections cannot establish links between two peers without a publically accessible signalling server.

WebRTC uses RTCPeerConnection to communicate streaming data between browsers, but also needs a mechanism to coordinate communication and to send control messages, a process known as

---

13 https://github.com/dodds-cc/dspNode/blob/master/tests/latencyTests/audioLatencyTest.js

14 https://github.com/dodds-cc/dspNode/blob/master/tests/latencyTests/README.md

15 https://angularjs.org/

16 https://github.com/bbc/peaks.js/

17 https://github.com/aterrien/jQuery-Knob

18 https://github.com/dodds-cc/dspNode/tree/master/demo/client

19 https://nwjs.io/

20 https://github.com/dodds-cc/dspNode/blob/master/architectures/README.md

signaling. Signaling methods and protocols are not specified by WebRTC.[21] For this example, the Socket.io[22] websocket library is used.

6.1.1 STUN AND TURN

WebRTC is designed to work peer-to-peer, so users can connect by the most direct route possible. However, WebRTC is built to cope with real-world networking: client applications need to traverse NAT gateways and firewalls, and peer to peer networking needs fallbacks in case direct connection fails. As part of this process, the WebRTC APIs use STUN servers to get the IP address of your computer, and TURN servers to function as relay servers in case peer-to-peer communication fails[23]. The dspNode example application employs publicly accessible STUN and TURN servers, hosted by 3rd parties for free.

6.1.2 Demonstration application overview

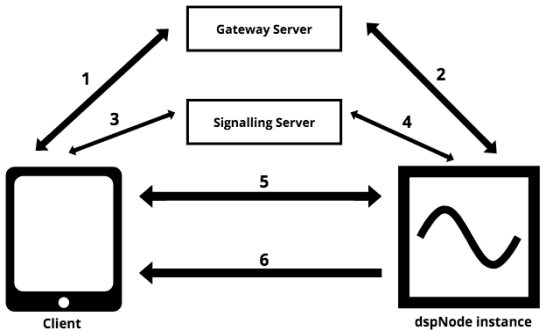

Figure 3. Demo application architecture

1. *HTTP*: The client first makes a request for a new dspNode instance via HTTP to the gateway server.

2. *HTTP*: The gateway server then starts a dspNode instance on AWS Fargate, passing it a randomly assigned unique identifier. When the instance has fully booted, a notification is sent back to the gateway along with the identifier

3. *HTTP*: The client then registers with the signaling server over webrtc, signaling that it is ready for connection.

4. *HTTP*: The dspNode makes a call to the signalling server telling it that it is ready for a client connection.

5. *RTCDataChannel*: Once the websocket signaling server has registered both the client and dspNode and established the simplest route between them, a UDP connection is established between the two peers. The client sends dspNode control signals and receives command acknowledgements and status updates from the dspNode, both in JSON format.

6. *RTCPeerConnection*: A WebRTC audio connection is then established from the dspNode to the client based on the codec settings provided in the clients first request to the gateway server. This is setup as a one-way link, using the getUserMedia API to capture audio on the dspNode.

[21] https://codelabs.developers.google.com/codelabs/webrtc-web/#0

[22] https://socket.io/

[23] https://codelabs.developers.google.com/codelabs/webrtc-web/#0

7. SCALING

Currently, the dspNode process can be run on any platform architecture within in an isolated docker container. The process requires around 500MB RAM, file system access, one vCPU resource and an outbound internet connection. Each single client destination device requires one rendering engine on a one to one ratio.

The docker container can be run anywhere on the internet in theory, but practically the closer the physical location of the dspNode to the clients internet connection, the lower the latency of the control signals and rendered audio will be to the client.

Containers can be run on a variety of 3rd party cloud infrastructure platforms or run on premise. Recently, cloud services have begun to emerge that abstract the underlying host management, providing a simple interface to spin up multiple simultaneous containers. Amazon Fargate has been used for this demonstration, but other solutions are available including open-source options.

8. CONCLUSION

This project has generally validated the concept and potential of remote audio rendering for web applications. The demonstration application shows that high quality audio can be streamed from a remote source with low latency using a client-side control interface.

The latencies observed are surprisingly low across devices and the increasing WebRTC support over the last few years has meant the concept as a means to deliver complex dynamic audio feeds is plausible. This research is the start of further exploration.

9. FURTHER WORK

The first area to continue exploring is the optimisation of the rendering process. The demonstration saw the use of NodeWebkit as a easy to test solution, essentially shifting the Web Audio API from the browser to a dspNode. Ideally, the dspNode will implement a lower level native rendering engine. Experiments with a controllable GStreamer (written in native C) DSP graph running within the dspNode instance have shown that this is viable.

Possible uses of dspNode include the working example given, but could include many other applications. For example, a VR headset rendering environment could send over only its position matrix to the dspNode, allowing the server to compute the complex spatial audio rendering, freeing the headset resources to focus on video rendering.

Another example could be in collaborative audio creation, allowing multiple users to collaborate on a shared audio editor environment running in dspNode in real-time with multiple simultaneous client connections and control signal feeds.

Expanding the scope of the project to video could be an interesting route too, particularly in combination with video rendering libraries like VideoContext.js[24] which are subject to similar client-side bottlenecks.

[24] https://github.com/bbc/VideoContext

10. REFERENCES

[1] dspNode, *Thomas Dodds, April 2018*
https://github.com/dodds-cc/dspNode

[2] CloudOrch: A Portable SoundCard in the Cloud, *Abram Hindle, June 2014*
http://www.nime.org/proceedings/2014/nime2014_541.pdf

[3] Websockets Streaming Audio, *Scott Stensland, 2014*
https://www.npmjs.com/package/websockets-streaming-audio

[4] Cloud gaming, *Wikipedia April 2018*
https://en.wikipedia.org/wiki/Cloud_gamin

[5] Media formats for HTML Audio and Video, *Mozilla April 2018*
https://developer.mozilla.org/en-US/docs/Web/HTML/Supported_media_formats

[6] Can I Use? *Alexis Deveria, April 2018*
https://caniuse.com/#search=webrtc

[7] RTCMulticonnection, *Muaz Khan, April 2018*
http://www.rtcmulticonnection.org/

[8] WebRTC FAQ, *Google Chrome, April 2018*
https://webrtc.org/faq/#audio

[9] WebRTC Glossary, *BlogGeek.com, April 2018*
https://webrtcglossary.com/codec/

[10] Opus Codec, *Xiph,org, April 2018*
http://opus-codec.org

[11] What is the VP8 Codec, *Google Chrome, April 2018*
https://webrtc.org/faq/#what-is-the-vp8-video-codec

[12] dspNode Control Signal Latency Test, *Thomas Dodds, April 2018*
https://github.com/dodds-cc/dspNode/blob/master/tests/latencyTests/rtcDataChannelLatencyTest.js

[13] dspNode Audio Signal Latency Test, *Thomas Dodds, April*
https://github.com/dodds-cc/dspNode/blob/master/tests/latencyTests/audioLatencyTest.js

[14] dspNode Test summary, *Thomas Dodds, April 2018*
https://github.com/dodds-cc/dspNode/blob/master/tests/latencyTests/README.md

[15] Angular.js framework, *Google, April 2018*
https://angularjs.org/

[16] Peaks.js, *BBC April 2018*
https://github.com/bbc/peaks.js/

[17] jQuery knob, *Antony Terrien, December 2015*
https://github.com/aterrien/jQuery-Knob

[18] dspNode Client library, *Thomas Dodds, April 2018*
https://github.com/dodds-cc/dspNode/tree/master/demo/client

[19] Node Webkit, *NW.js community April 2018*
https://nwjs.io/

[20] dspNode Architecture experiments, *Thomas Dodds, April 2018*
https://github.com/dodds-cc/dspNode/blob/master/architectures/README.md

[21] Real time communication with WebRTC, *Google, April 2018*
https://codelabs.developers.google.com/codelabs/webrtc-web

[22] Socket.io open-source javascript library, *April 2018*
https://socket.io

[23] What are STUN and TURN, *Google, April 2018*
https://codelabs.developers.google.com/codelabs/webrtc-web

[24] VideoContext.js, *BBC, March 2018*
https://github.com/bbc/VideoContext