CLEFNET: Recurrent Autoencoders with Dynamic Time Warping for Near-Lossless Music Compression and Minimal-Latency Transmission

Vignav Ramesh[*]
Saratoga High School
20300 Herriman Avenue
Saratoga, CA 95070
`rvignav@gmail.com`

Mason Wang[*]
Saratoga High School
20300 Herriman Avenue
Saratoga, CA 95070
`masonwang025@gmail.com`

Abstract

The onset of coronavirus disease 2019 (COVID-19), an infectious disease caused by severe acute respiratory syndrome coronavirus 2 (SARS-CoV-2), has sparked unprecedented change. Due to the public health guidelines imposed during the COVID-19 pandemic, there is no longer sufficient street traffic for remaining buskers to generate sufficient revenue, leading a majority of street musicians to pursue remote music production. However, real-time music production is notoriously difficult due to the excessively high latencies that current video call platforms such as Zoom and Google Meet harbor. In this paper, we propose an architecture for a platform with end-to-end, near-lossless audio transmission tailored specifically to online joint music production, called Latent Space. We discuss the usage of a recurrent autoencoder with sequence-aware encoding (RAES) and a 1D convolutional layer for audio compression, which we dub CLEFNET, as well as propose a new evaluation metric for naive autoencoders (AEs), MSE-DTW loss, which combines the traditional mean square error (MSE) loss function with dynamic time warping (DTW) to prevent an increase in loss when the target sequence predicted by the AE is strictly a temporal variation of the source sequence. Moreover, we detail the logistics of a live system implementation which uses the Web Audio API to extract raw audio samples in real-time to feed into our client-side model before relaying the traffic using peer-to-peer WebRTC technology. The Latent Space platform can be accessed at `https://latent-space.tech`, and the code and data can be found under the MIT License at `https://github.com/rvignav/ClefNet`.†

1 Introduction

The onset of coronavirus disease 2019 (COVID-19), an infectious disease caused by severe acute respiratory syndrome coronavirus 2 (SARS-CoV-2), has sparked unprecedented change [1-3]. In January 2020, the World Health Organization (WHO) declared the COVID-19 outbreak a global health emergency, and in March 2020, the outbreak was declared a global pandemic. The virus has continued to spread on a global scale with unparalleled rapidity, reaching a total of over 2.5 million deaths as of February 2021 [4, 5]. Stay home orders, quarantine rules, lockdowns, and other public health restrictions have been implemented in order to curb this growth, but these measures have substantial social effects [6-8].

Music is an integral part of human culture, taking a wide variety of forms and evoking a unique human spirit in its creation [9]. Of the approximately 30 million musicians in the U.S., more than 40,000 are street musicians, or buskers [10]. Due to the public health guidelines imposed during the COVID-19 pandemic, however, there is no longer sufficient street traffic for remaining buskers to generate sufficient revenue, leading a majority of street musicians to pursue remote music production.

Real-time remote music production is notoriously difficult. Without being able to meet in person, musicians are forced to connect via video call platforms such as Zoom, Google Meet, and Jitsi Meet. However, these platforms all harbor excessive latencies, or delays of over-the-network transfers of large audio payloads: while humans can easily recognize latencies of 10ms and more, Zoom's latency is 135ms, Google Meet's latency is 100ms, and Jitsi Meet's latency is 500ms. Musician synchronization is vital to music recording and production, yet audio codecs that video calling platforms currently use are not optimized for remote music production. For instance, MP3 audio compression algorithms utilized by various video-calling platforms use carefully-tuned psychoacoustic models to make inferences about which components of a given audio stream are most perceived by a human listener; while these perceptual coding methods exhibit quite impressive compression results within the domain of human speech, they are ineffective and even detrimental when applied in situations for which they were not designed, e.g. music-related tasks [11].

Thus, there is significant value in creating an automated tool for real-time transmission of music with minimal latency. A flexible platform that can support a wide range

[*]Equal contribution

†The provided code and prototype reflect a deprecated version of the proposed architecture; a more complete set of features will be added in the near future.

of video-calling capabilities useful for real-time remote music production is critically necessary. In this paper, we present the prototype of an online video-calling platform that enables minimal-latency audio transfer. The proposed pipeline involves three components: audio compression, transfer, and reconstruction.

There has been progress in music compression and, more generally, audio compression research. Traditional methods of music compression typically use deterministic algorithms, which rely on identifying features and patterns in the frequency domain [12]. Such algorithms include the MP3 compression algorithm described in [11] and similar (often lossy) pipelines.

However, the use of deep learning for music-related applications, such as the tasks of music generation and musical style transfer, has risen in recent years [13-15]. The motivation for using deep learning (and more generally machine learning techniques) for music compression is its generality; unlike deterministic models, such as grammar-based or rule-based music generation systems, a machine learning-based system can be agnostic, as it learns a representation of music from an arbitrary corpus of audio samples and can then extend to a variety of other genres and styles [16].

For instance, Roche, Hueber, Limier, and Gilin proposed a set of non-linear unsupervised dimensionality reduction techniques to compress a music dataset into a low-dimensional representation which can be used in turn for the synthesis of new sounds [17]. Shallow autoencoders, deep autoencoders (DAEs), recurrent autoencoders (with Long ShortTerm Memory cells – LSTM-AEs) and variational autoencoders (VAEs) with principal component analysis (PCA) for representing the high-resolution short-term magnitude spectrum of a large and dense dataset of music notes into a lower-dimensional vector were presented and evaluated on the publicly available multi-instrument and multi-pitch database NSynth. Roche et. al.'s experiments show that DAEs and LSTM-AEs lead to comparatively low music reconstruction error.

Grady et al. investigated the use of a time-domain AE for music compression and genre classification [11]. Compared to the frequency-domain AEs described in [17], learning on a time-domain signal saves space too as the spectral domain of an audio signal is sparse. Thus, Grady et al.'s research centers around the degree of compression (which, in the context of our work, is directly proportional to the latency of audio transfer) rather than the accuracy of reconstruction. Their implemented time-domain AE, with root mean square error (RMSE) as the loss function, successfully captured the basic rhythm from music files and compressed audio, but the reconstructed music sample contained large amounts of white noise due to the lack of a smoothing component in the loss function.

Roberts et al. proposed MusicVAE, a hierarchical latent vector model for learning long-term structure in music [18]. The MusicVAE model follows the general VAE architecture but introduces a novel hierarchical decoder, which is demonstrated to produce a substantially better performance on long sequences. A two-layer bidirectional LSTM network as the encoder, paired with a novel hierarchical recurrent neural network (RNN) as the decoder, obtains 0.919 reconstruction accuracy for 16-bar melodies.

Ramani et al. proposed a framework for audio style transfer, in which audio compression is an intermediate step, using a single convolutional autoencoder trained on spectrograms of speech signals and a single style signal [19]. The signal is preprocessed by applying the Short Time Fourier Transform (STFT) on the raw input audio to generate an audio spectrogram that is passed through the transformation network to generate the stylized spectrogram. The Griffin-Lim algorithm is used to convert the stylized spectrogram to the required stylized audio.

For the problem of learning high-level controls over the global structure of generated sequences in the context of symbolic music generation with complex language models, Choi et al. proposed the Transformer Autoencoder, which aggregates encodings of the input data across time to obtain a global representation of style from a given performance [20]. This latent representation can then be combined with other temporally distributed embeddings to generate new music of similar style with greater control over the separate aspects of performance style and melody. Choi et al.'s method achieves improvements in terms of log-likelihood and mean listening scores when compared to prior state-of-the-art methods.

However, these methods have not been applied to the task of minimal-latency audio transfer, nor have they leveraged the web to develop an online video-calling tool.

1.1 Contributions

The main contributions of this paper can be summarized as follows:

1. We propose an architecture for a platform with end-to-end, near-lossless audio transmission tailored specifically for online joint music production, called Latent Space.
2. We discuss the usage of a recurrent autoencoder with sequence-aware encoding (RAES) and a 1D convolutional layer for audio compression, which we dub CLEFNET.
3. We propose a new evaluation metric for naive autoencoders (AEs), MSE-DTW loss, which combines the traditional mean square error (MSE) loss function with dynamic time warping (DTW) to prevent an increase in loss when the target sequence

predicted by the AE is strictly a temporal variation of the source sequence.

4. We detail the logistics of a live system implementation which uses the Web Audio API to extract raw audio samples in real-time to feed into our client-side model before relaying the traffic using peer-to-peer WebRTC technology.

2 Method

2.1 Background

Encoder-Decoder Architectures. An encoder-decoder model is a type of neural network architecture trained to reconstruct its input. The encoder learns a new, lower-dimensional representation of the input, called the latent space or the embedding, from which the decoder then aims to reconstruct an output as similar as possible to the original input. Encoder-decoder models, along with self-supervised models [21], are popular in that they learn a compact representation of an input domain that can be used for a variety of tasks.

Autoencoders are a class of encoder-decoder architectures, first introduced in [22], that unsupervisedly learn efficient data encodings. As defined in [23], the problem is to learn some encoder $A : \mathbb{R}^n \to \mathbb{R}^p$ and decoder $B : \mathbb{R}^p \to \mathbb{R}^n$, usually implemented as independent neural networks in practice, that satisfy

$$\arg\min_{A,B} E[\Delta(\mathbf{x}, B \circ A(\mathbf{x}))], \tag{1}$$

where E is the expectation over the distribution of x, and Δ is the reconstruction loss function [24].

Recurrent autoencoders, or RAEs, map time series to a latent representation, enabling efficient, large scale unsupervised learning on temporal sequences [25]. Our model architecture utilizes an RAE, exhibiting the temporal dynamic behavior of recurrent neural networks (RNNs) and thereby increasing reconstruction accuracy as previous audio samples can be accessed and utilized for current inference.

Learning Musical Representations. Music data is typically represented in high-dimensional spaces, allowing for an extremely high number of possible sequences. However, only a fraction of these possibilities are likely for real music. This indicates an opportunity to represent musical sequences in a lower-dimensional space. Encoder-decoder models can learn the fundamental characteristics of a given training dataset and thus exclude the unrealistic, incoherent sequences.

Autoencoders, as previously discussed, are capable of learning a low-dimensional representation and reproducing their high-dimensional inputs. Thus, we utilize the latent space as an effective compression technique to reduce audio payloads without reducing audio quality.

In order to learn the best mapping to most effectively use the latent space, there are two primary goals in terms of desirable properties of a latent space: realism and smoothness.

The goal of realism is to ensure the latent space corresponds to the likely points. In other words, mapping a sampled point in the latent space to the original high-dimensional space should result in a likely point in the context of music sequences [26].

The goal of smoothness is to ensure that sampled nearby points in a latent space have similar qualities to one another. That is, two nearby points in a latent space should map to nearby points in the output space [27].

2.2 Model Logistics

Preprocessing. Our training data consisted of 1,486 two-minute music excerpts in WAV format [28]. However, raw WAV files are not of suitable format to be fed directly into our AE architecture; we must first preprocess the training data in order to convert the audio samples into a machine-conducive format. We first use TensorFlow's `audio_ops` library to decode each WAV file into an array of samples that can be processed and batched by the AE [29]. The arrays were separated by channel (left and right - stereo sound). The decoded samples represent the amplitude of the audio at each timestep, and experimental efforts proved that this data format was hard for the AE to learn; thus, we applied the Discrete Fourier Transform (DFT) to each channel as to decompose the audio signals into components that were more conducive to the AE training process. The DFT is an approximation of the continuous Fourier transform for the case of discrete functions [30, 31]. Given a sequence x_n, the DFT generates a sequence X_k of complex numbers that represent the amplitude and phase of the sinusoidal components of the input signal. It can be represented mathematically as follows:

$$X(\omega_k) = \sum_{n=0}^{N-1} x(t_n) e^{-i\omega_k t_n}, k = 0, 1, ..., N-1, \tag{2}$$

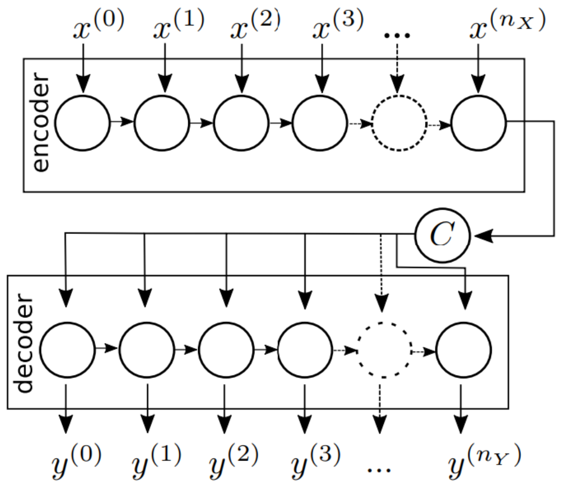
where T is the sampling interval of the signal, $\omega_k = \frac{2\pi k}{NT}$, and $t_n = nT$.

Architecture. One of the primary concerns regarding our model architecture was making sure that our use of an autoencoder to compress the audio was fast enough to decrease latency. That is, we need to ensure that the extra time required to both compress and decompress the audio samples would not negate its benefits. For this reason, we decided to keep the model small and simple, where modifications had minimal impact on runtime.

We utilized an implementation of a RAES and a 1D convolutional layer.

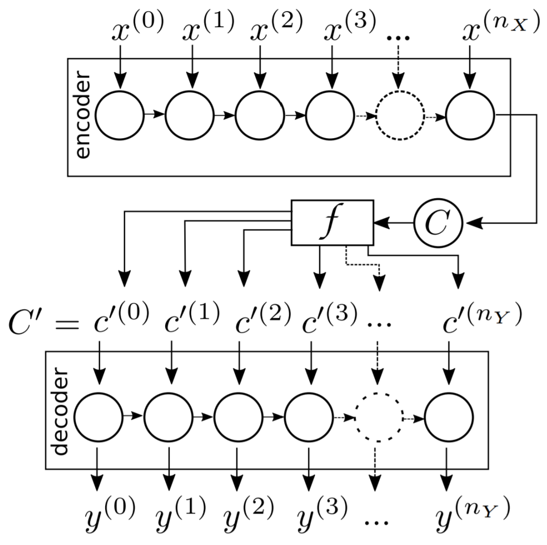
Susik proposed a recurrent autoencoder (RAE) architecture with sequence-aware encoding, employing a 1D convolutional layer to improve its performance in terms of model training time. Details of this implementation can be found in [32]. We provide a brief explanation here. The vanilla RAE (Figure 1) generates an output sequence Y for an input sequence X. $X = Y$ in order for the autoencoder to learn the semantic meaning of our training data. The given fixed-sized context variable C, or the latent space, is then decoded by the decoder. The proposed RAES (Figure 2) instead has the context C interpreted as the transformed sequence C', and this not only speeds up training but also, more importantly, adds sequential meaning to the context.

The limitation of RAES is that the size of the context must be a multiple of the input sequence length, but this is solved by an RAES with a 1D convolutional layer (RAESC) (Figure 3), which adds a 1D convolutional layer and max-pooling layer before the decoder. In essence, this addition allows the number of output channels (filters or feature detectors) to be controlled [32]. Although further pooling and recurrent layers could be added to the architecture, we chose a simplified variant to minimize the model's overhead.



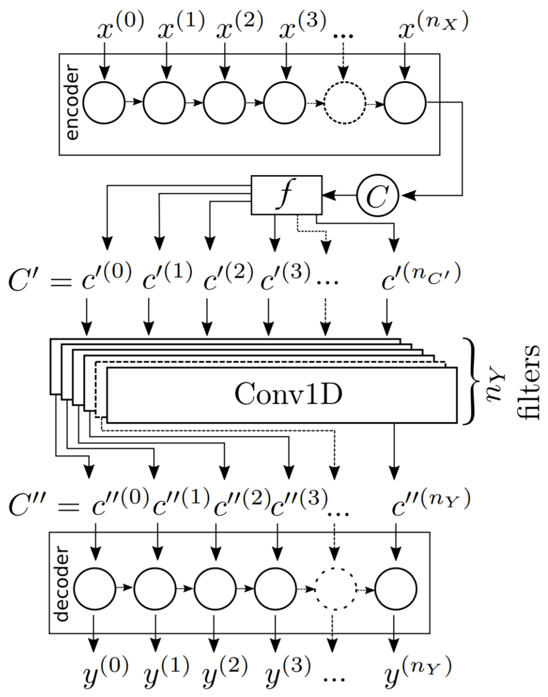
$x^{(0)}$ $x^{(1)}$ $x^{(2)}$ $x^{(3)}$ ... $x^{(n_X)}$
encoder
C
decoder
$y^{(0)}$ $y^{(1)}$ $y^{(2)}$ $y^{(3)}$ ... $y^{(n_Y)}$

Figure 1: A generic recurrent autoencoder (RAE) architecture [32].



$x^{(0)}$ $x^{(1)}$ $x^{(2)}$ $x^{(3)}$ ... $x^{(n_X)}$
encoder
f
C
$C' = c'^{(0)}$ $c'^{(1)}$ $c'^{(2)}$ $c'^{(3)}$ ... $c'^{(n_Y)}$
decoder
$y^{(0)}$ $y^{(1)}$ $y^{(2)}$ $y^{(3)}$ ... $y^{(n_Y)}$

Figure 2: A recurrent autoencoder with sequence-aware encoding (RAES) [32].



$x^{(0)}$ $x^{(1)}$ $x^{(2)}$ $x^{(3)}$ ... $x^{(n_X)}$
encoder
f
C
$C' = c'^{(0)}$ $c'^{(1)}$ $c'^{(2)}$ $c'^{(3)}$ ... $c'^{(n_{C'})}$
Conv1D
n_Y filters
$C'' = c''^{(0)}$ $c''^{(1)}$ $c''^{(2)}$ $c''^{(3)}$ ... $c''^{(n_Y)}$
decoder
$y^{(0)}$ $y^{(1)}$ $y^{(2)}$ $y^{(3)}$ ... $y^{(n_Y)}$

Figure 3: A recurrent autoencoder with sequence-aware encoding and 1D convolutional layer (RAESC) [32].

Minimizing Reconstruction Loss. A critical metric with which autoencoders are evaluated is reconstruction loss (our model is trained using a custom variant of the MSE loss function). The problem of signal compression can be divided into three sub-problems: lossy compression, lossless compression, and near-lossless compression. Lossy compression algorithms compress structured data (in the context of this paper, audio files) in order to reduce file size by eliminating data [33]. Any data deemed expendable by the compression algorithm is removed from the file, thereby allowing for compression but decreasing audio quality in the process. While lossy compression algorithms are popular in the imaging domain (slight decreases in image resolution are often not noticeable), the necessity of high audio quality for remote music production makes lossy compression algorithms unviable. In contrast, lossless compression involves no loss of information; losslessly compressed data can be reconstructed exactly into the original data [34]. However, with lossless compression techniques, compression rates are extremely low as maintaining perfect accuracy of audio reconstruction is, past a certain rate of compression, intractable due to the innumerable amount of subtle aural features in audio files.

Thus, to increase compression rates and cater to the nuanced format of audio files, an ideal autoencoder would achieve near-lossless compression. Near-lossless compression provides quantitative guarantees regarding the type and amount of distortion applied to the source file, which assures that the extracted parameters of interest will only be affected within a bounded range of error [35]. These techniques significantly increase compression rates over lossless compression, thereby decreasing bandwidth usage during audio transfer, while maintaining audio integrity with respect to postprocessing operations.

We also conceptualize a variant of MSE loss, called "MSE-DTW loss," that incorporates DTW [36]. Music can be represented as a linearly composed stream of units (notes, strophes, bars, etc.), where each unit corresponds to a different timestep; thus, if the units of the reconstructed sequence closely or exactly match those of the input sequence, but the reconstructed sequence exhibits temporal differences (i.e. the source and reconstructed sequences differ by a few milliseconds in start and end times, the reconstructed sequence is at a slightly lower or higher tempo than the source sequence, etc.), the model should not be heavily penalized as it would be with MSE loss; the target sequence can be temporally transformed to match the source sequence as a postprocessing operation after reconstruction, so the model should only "focus" on generating latent musical representations that are aurally, not necessarily temporally, accurate. DTW solves this problem; DTW is an algorithm that measures the similarity between two temporal sequences, or time series, that have varied speeds. DTW calculates an optimal batch between source and target sequences via certain restrictions, which include:

- There must be at least one match between each unit in the source sequence and a unit in the target sequence
- The first unit of each sequence must match
- The last unit of each sequence must match
- The mapping of the units of the source sequence to those of the target sequence must monotonically increase, i.e. if $j > i$ are units of the source sequence, then there must not be two units $l > k$ in the target sequence such that i corresponds to l and j corresponds to k

The sequences are "warped," or transformed, in the time dimension to determine a measure of their similarity independent of temporal variations. Thus, the proposed method ensures that, as long as the reconstructed sequence is some temporal modification of the source sequence, MSE-DTW loss remains low.

3 Live System Considerations

Our live system implementation uses the Web Audio API to extract raw audio samples in real-time to feed into our model. After compressing the samples through a client-side model, the packets are then transmitted using the WebRTC real-time communications framework, allowing for end-to-end, minimal-latency transmission.

3.1 Sampling Audio

In order for our model to compress a caller's microphone audio, we need a fast way of extracting low-level audio, but most audio interfaces on the web only support either abstract, high-level stream objects or deal with audio that is already recorded. Furthermore, we want to process the audio on a separate thread in order to allow other code to run on the main thread. We achieve this using the Web Audio API's audio worklets.

Behind all our audio processing is the AudioContext interface, a Web Audio API interface. Inside an audio context, WebAudio API handles audio operations, which are performed with audio nodes. These audio nodes are linked together to form an audio-processing graph, where several sources of audio are supported within a single context [37]. This graph and context are represented by the AudioContext interface, which controls the creation and processing of the AudioNodes (audio-processing modules that make up the graph) [38].

Most importantly, the AudioContext object provides access to an audio worklet to execute scripts off of the main thread. We use the audio worklet to define a custom audio node written in JavaScript. Audio worklet nodes implement the Worker interface, a lightweight version of the Web Workers API [39], and allow us to handle low-level parts of the rendering pipeline [40]. The AudioWorklet interface allows us to execute audio processing scripts on a separate thread to provide low latency audio processing [41].

We define custom AudioNodes in the AudioWorklet with the AudioWorkletNode, and each of these is embedded into an audio graph and passes messages to the AudioWorkletProcessor, which represents the audio processing code [42].

Finally, we connect our custom-defined audio-processing module to two buffers, following the structure of the deprecated ScriptProcessorNode (Figure 4). One of the buffers contains the input audio data, while the other contains the output audio data. An event is sent to our AudioWorkletNode each time the input buffer contains new data and the event handler terminates once the output buffer is filled with data [43].

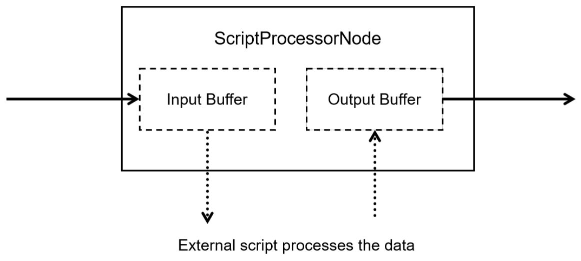


Figure 4: An audio worklet node, following the structure of the depicted ScriptProcessorNode, processes audio samples on a separate thread in real-time [43].

The standard sampling rate of 44.1 kHz, or 44,100 samples per second, best satisfies our prototype's needs. The Nyquist-Shannon sampling theorem states that the maximum frequency that can be represented at any sampling rate is only half the sampling rate [44]. Thus, a sample rate of 44.1 kHz can represent frequencies up to 22.05 kHz, and humans can hear frequencies between 20 Hz and 20 kHz. Given this range and the fact that 44.1 kHz is the standard for most consumer audio, larger sample rates (e.g., 48 kHz, 96kHz) will not provide any noticeable benefit and will instead only increase audio payloads.

In essence, we define our custom audio node inside our audio worklet, which is inside an audio context, to process audio at a low-level and in real-time on a separate thread.

3.2 Client-Side Model

For minimal latency, the model must reside on the client-side, running directly in the browser. Trained models should be saved and imported onto the client-side model. TensorFlow.js is a machine learning framework that can run models using JavaScript fully on the browser [45].

3.3 Transmitting Packets

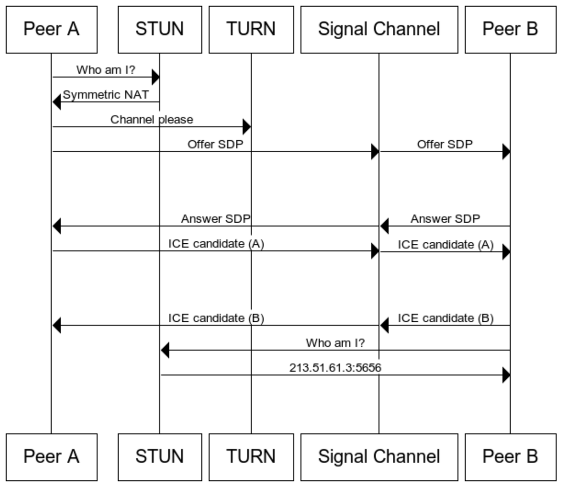


Figure 5: This MDN diagram depicts the discussed exchanges between two peers when establishing peer-to-peer connection [49].

The speed at which packets are transmitted over a network is very important in terms of latency. We use peer-to-peer communications with WebRTC (Web Real-Time Communications) to create real-time connections and establish data channels. WebRTC utilizes plugin-free APIs available in both desktop and mobile browsers and its technologies are implemented as an open web standard. The primary benefit of peer-to-peer communications is that connections are created directly between two hosts without using intermediary servers, thereby playing a crucial role in minimizing latency [46].

Prior to establishing connections between hosts, we access cameras and microphones with JavaScript through the `navigator.mediaDevices` object. More specifically, the function `getUserMedia()` returns a promise which resolves to the `mediaStreams` we use in our video calls [47].

In order for browser-based peer-to-peer communications to succeed, the two browsers must know how to locate one another, bypass firewall protections, and bidirectionally transmit multimedia communications in real-time. Each host makes a request for their public-facing IP address to STUN (Session Traversal Utilities for NAT) servers, allowing the callers to locate one another.

Once the peers are connected to the same signaling data channel, which are created after successful NAT traversal, session negotiation and establishment occurs using SIP (Session Initiation Protocol) and SDP (Session Description Protocol). The initiating peer sends an offer and waits for any receiver connected to the same channel to answer.

Once answered, Interactive Connectivity Establishment (ICE) candidates are shared to represent a combination of IP address, port, and transport protocol to be used, and the initiating peer generates a set of ICE candidates. Once negotiation occurs and the optimal ICE candidates are chosen, the session between the peers will be established and active. A fallback TURN (Traversal Using Relays around NAT) server is able to relay traffic if the peer-to-peer connection fails [48].

Now that the session is active, data streams and data channel endpoints are created by each pair for the data to be bidirectionally transmitted. We create an `RTCDataChannel` on our existing `RTCPeerConnection` to send the video stream and the compressed audio which is structured as binary data. Again, because of the nature of peer-to-peer communications, the overhead for relaying traffic is low as the connection is made directly between two browsers without the use of external servers.

4 Conclusion

In this paper, we propose an architecture for an online video-calling platform with end-to-end, near-lossless audio transmission designed for remote music production. We discuss the application of RAES architectures to audio compression and also propose a custom variant of MSE loss, MSE-DTW loss, that utilizes the technique of dynamic time warping to prevent increases in reconstruction loss based on solely temporal variations between the source and target sequences. Finally, we detail the logistics of a live system implementation based on the Web Audio API and peer-to-peer WebRTC technology. With the proposed architecture, we pave the way towards end-to-end remote music production for musicians affected by the COVID-19 pandemic.

Acknowledgment

The authors would like to thank Mathew Wang for his contributions to the development of the Latent Space web application.

References

1. Yan, Qingsen, et al. "COVID-19 Chest CT Image Segmentation – A Deep Convolutional Neural Network Solution." *ArXiv:2004.10987 [Cs, Eess]*, Apr. 2020. *arXiv.org*.
2. Sutton, Jack, et al. "Homogeneous and Heterogeneous Propagation of COVID-19 from Super-Spreading to Super-Isolation." *ArXiv:2102.13016 [q-Bio]*, Feb. 2021. *arXiv.org*.
3. Bassolas, Aleix, et al. "Optimising the Mitigation of Epidemic Spreading through Targeted Adoption of Contact Tracing Apps." *ArXiv:2102.13013 [Physics]*, Feb. 2021. *arXiv.org*.
4. *WHO Coronavirus Disease (COVID-19) Dashboard.* Accessed 25 Dec. 2020.
5. "Coronavirus Disease 2019 (COVID-19) - Symptoms and Causes." *Mayo Clinic*. Accessed 25 Dec. 2020.
6. Imperial College COVID-19 Response Team, et al. "Estimating the Effects of Non-Pharmaceutical Interventions on COVID-19 in Europe." *Nature*, vol. 584, no. 7820, Aug. 2020, pp. 257–61. *DOI.org (Crossref)*.
7. Cowling, Benjamin J., et al. "Impact Assessment of Non-Pharmaceutical Interventions against Coronavirus Disease 2019 and Influenza in Hong Kong: An Observational Study." *The Lancet Public Health*, vol. 5, no. 5, May 2020, pp. e279–88. *www.thelancet.com*.
8. Peak, Corey M., et al. "Comparing Nonpharmaceutical Interventions for Containing Emerging Epidemics." *Proceedings of the National Academy of Sciences*, vol. 114, no. 15, Apr. 2017, pp. 4023–28. *www.pnas.org*.
9. Dhariwal, Prafulla, et al. "Jukebox: A Generative Model for Music." *ArXiv:2005.00341 [Cs, Eess, Stat]*, Apr. 2020. *arXiv.org*.
10. "How Many Official Street Performers (Musicians, Buskers Etc..) Are in the World, and in the Main Cities NY, London, Paris Etc." *Wonder*. Accessed 9 Mar. 2021.
11. Atreya, Anand R., and D. O'Shea. *Novel Lossy Compression Algorithms with Stacked Autoencoders.* 2009.
12. "Introduction." *Deep Autoencoders for Music Compression and Genre Classification*. Accessed 9 Mar. 2021.
13. "MuseNet." *OpenAI*, 25 Apr. 2019.
14. "Magenta." *Magenta TensorFlow*. Accessed 9 Mar. 2021.
15. Huang, Allen, and Raymond Wu. "Deep Learning for Music." *ArXiv:1606.04930 [Cs]*, June 2016. *arXiv.org*.
16. Briot, Jean-Pierre, et al. "Deep Learning Techniques for Music Generation – A Survey." *ArXiv:1709.01620 [Cs]*, Aug. 2019. *arXiv.org*.
17. Roche, Fanny, et al. "Autoencoders for Music Sound Modeling: A Comparison of Linear, Shallow, Deep, Recurrent and Variational Models." *ArXiv:1806.04096 [Cs, Eess]*, May 2019. *arXiv.org*.
18. Roberts, Adam, et al. "A Hierarchical Latent Vector Model for Learning Long-Term Structure in Music." *ArXiv:1803.05428 [Cs, Eess, Stat]*, Nov. 2019. *arXiv.org*.

19. Ramani, Dhruv, et al. "Autoencoder Based Architecture For Fast & Real Time Audio Style Transfer." *ArXiv:1812.07159 [Cs, Eess, Stat]*, Dec. 2018. *arXiv.org*.
20. Choi, Kristy, et al. "Encoding Musical Style with Transformer Autoencoders." *ArXiv:1912.05537 [Cs, Eess, Stat]*, June 2020. *arXiv.org*.
21. Liu, Xiao, et al. "Self-Supervised Learning: Generative or Contrastive." *ArXiv:2006.08218* [Cs, Stat], July 2020. *arXiv.org*.
22. Press, The MIT. *Parallel Distributed Processing, Volume 1 | The MIT Press*. Accessed 9 Mar. 2021.
23. Baldi, Pierre. "Autoencoders, Unsupervised Learning, and Deep Architectures." *Proceedings of ICML Workshop on Unsupervised and Transfer Learning*, JMLR Workshop and Conference Proceedings, 2012, pp. 37–49. *proceedings.mlr.press*.
24. Bank, Dor, et al. "Autoencoders." *ArXiv:2003.05991* [Cs, Stat], Mar. 2020. *arXiv.org*.
25. Fabius, Otto, and Joost R. van Amersfoort. "Variational Recurrent Auto-Encoders." *ArXiv:1412.6581 [Cs, Stat]*, June 2015. *arXiv.org*.
26. Roberts, Adam, et al., editors. *Learning Latent Representations of Music to Generate Interactive Musical Palettes*. 2018. *Google Research*.
27. "MusicVAE: Creating a Palette for Musical Scores with Machine Learning." *Magenta*. Accessed 9 Mar. 2021.
28. Ángel Faraldo. *Beatport EDM Key Dataset*. Zenodo, 21 Dec. 2017. *Zenodo*.
29. Sherman, Wezley. "Using TensorFlow Autoencoders with Music." *Medium*, 20 Sept. 2018.
30. Fessler, J. *The Discrete Fourier Transform*. 2004.
31. "Anand | Discrete Fourier Transform | Fast Fourier Transform." *Scribd*. Accessed 9 Mar. 2021.
32. Susik, Robert. "Recurrent Autoencoder with Sequence-Aware Encoding." *ArXiv:2009.07349 [Cs, Stat]*, Jan. 2021. *arXiv.org*.
33. *The Official CHFI Study Guide (Exam 312-49)*. Elsevier, 2007. *DOI.org (Crossref)*.
34. *Introduction to Data Compression*. Elsevier, 2018. *DOI.org (Crossref)*.
35. *Handbook of Image and Video Processing*. Elsevier, 2005. *DOI.org (Crossref)*.
36. Olsen, Niels Lundtorp, et al. "Simultaneous Inference for Misaligned Multivariate Functional Data." *ArXiv:1606.03295* [Stat], Dec. 2017. *arXiv.org*.
37. *Web Audio API - Web APIs | MDN*. Accessed 9 Mar. 2021.
38. *AudioContext - Web APIs | MDN*. Accessed 9 Mar. 2021.
39. *Worker - Web APIs | MDN*. Accessed 9 Mar. 2021.
40. *Worklet - Web APIs | MDN*. Accessed 9 Mar. 2021
41. *AudioWorklet - Web APIs | MDN*. Accessed 9 Mar. 2021.
42. *AudioWorkletNode - Web APIs | MDN*. Accessed 9 Mar. 2021.
43. *ScriptProcessorNode - Web APIs | MDN*. Accessed 9 Mar. 2021.
44. Por, Emiel, et al. "Nyquist–Shannon Sampling Theorem." *AOT*, 2019.
45. Ma, Yun, et al. "Moving Deep Learning into Web Browser: How Far Can We Go?" *ArXiv:1901.09388 [Cs]*, Mar. 2019. *arXiv.org*.
46. *Peer-to-Peer Communications with WebRTC - Developer Guides | MDN*. Accessed 9 Mar. 2021.
47. "Getting Started with Media Devices." *WebRTC*. Accessed 9 Mar. 2021.
48. Sredojev, B., et al. "WebRTC Technology Overview and Signaling Solution Design and Implementation." *2015 38th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, 2015, pp. 1006–09. *IEEE Xplore*.
49. *WebRTC Connectivity - Web APIs | MDN*. Accessed 9 Mar. 2021.