

# Audience Feedback Final Report

Shubham Chandak, Maggie Ford, Qingxi Meng, Mai Lan Nguyen, Manan Rai

**Mentors:** Sadjad Fouladi, Prof. Michael Rau, Prof. Tsachy Weissman

CS349T/EE192T Autumn 2020-21

## Table of Contents

<b>Abstract</b>	<b>2</b>
<b>Introduction</b>	<b>2</b>
<b>Background: Puffer platform</b>	<b>3</b>
<b>Latency Research and MSE</b>	<b>4</b>
Sources of Latency	4
Initial Testing and Further Research	5
<b>WebRTC Demos</b>	<b>7</b>
Potential WebRTC libraries	7
Streaming with Janus	8
<b>Emoji Feedback and Audio Feedback</b>	<b>11</b>
Emoji Feedback	11
Animation	12
Canned audio effects	13
Audio Feedback	14
<b>Chat</b>	<b>16</b>
<b>Conclusion and Future Work</b>	<b>19</b>
<b>Acknowledgements</b>	<b>20</b>
<b>References</b>	<b>20</b>
<b>Appendix: Janus Streaming</b>	<b>20</b>
Building Janus	21
Running Janus	22
Streaming audio from file via RTP	22
Streaming video from file via RTP	23
Streaming video from YouTube via RTP	24

# Abstract

Virtual theater performances often lack the real-time audience feedback available in live performances, thus missing a crucial aspect of the theater experience both for the audience and for the performers. In this project, we attempted to close this gap by developing feedback mechanisms that adapt the typical audience feedback interaction in live theater to a virtual setting. In particular, we implemented audience interaction through the use of emoji and audio feedback, as well as a chat system. We also added audiovisual effects to the player to simulate the theater experience for the audience.

An important prerequisite for real-time audience interaction is subsecond round-trip latency from the performers to the audience, however this is lacking in most streaming platforms. We explored the ability of various streaming mechanisms to achieve such a latency, initially focusing on the use of WebSocket based communication and Media Source Extensions (MSE). However this system was unable to achieve the desired latency, and hence we finally developed a prototype for a WebRTC based system which is a better fit for this application.

**Code availability:** The code for the emoji/audio feedback and the chat is available in the audience-feedback branch of the GitHub repository:

<https://github.com/stanford-stagecast/audience>.

# Introduction

The coronavirus pandemic caused significant disruptions in the Theater and Performing Arts industry. With the closures of in-person theater spaces, many working artists found themselves having to shift onto the digital space to continue creating their work - e.g., on Zoom, Twitch, YouTube Live, etc. Nevertheless, none of the currently available technologies were specifically designed to be utilized by theater-makers/-goers and their use for artistic purposes often proves to be sub-optimal and inconvenient.

Through this project we intend to envision how theatrical 'liveness' and interaction translates (or rather becomes redefined) in an online space. We aim to design and develop a system to host a live theatrical production with low-latency audio and video streaming to actors and audience members.

In our design we mainly focus on achieving low-latency actor-audience and audience-audience communication and creating an interactive audience feedback interface.

On the low latency end, our group aimed to stream the video from the source to the audience with sub-second latency. To do so, we first tried to adapt from the streaming platform Puffer (Yan et al., 2010) to achieve low latency. However, we encountered some technical difficulties

with MSE (W3C, 2016). Therefore, we decided to use WebRTC for streaming instead and created a prototype system.

For the audience feedback part, our group tried to collect emoji-based feedback, text feedback and audio feedback from the audience. We designed a nice user interface for the audience to send emojis or participate in text chat. The audio of the audience is also collected for the director to see the feedback.

## Background: Puffer platform

Here we briefly describe the Puffer video streaming platform<sup>1</sup> (Yan et al. 2020) which was developed by Prof. Winstein's research group to explore the use of machine learning to improve video streaming algorithms, especially to reduce glitches and stalls due to unreliable network conditions. The Puffer project transmits free over-the-air television channels and collects network data from users to optimize the streaming algorithm. In this project, we used part of the Puffer platform<sup>2</sup> to deliver the video to the audience, and we expanded the platform to support various forms of audience feedback and interaction.

The Puffer codebase consists of the following components:

- **Media server (C++):** This is responsible for detecting new video segments (added to a directory) and sending these to the users through WebSocket connections. The media server is responsible for adapting the video bit rate, buffer size and other parameters according to the network conditions to achieve the desired performance. The media server uses regular feedback from the users to make these decisions.
- **Web server (Django):** This is responsible for managing the user accounts and serving the content to the user. Puffer uses a Django backend for receiving and storing the audience feedback, and for facilitating the chat interaction.
- **User interface (HTML/JavaScript):** This is the frontend part that connects to the media server using WebSockets and displays the audio/video content on the browser using MSE (Media Source Extensions). It also sends regular feedback to the media server regarding the current video timestamp allowing the server to adapt the video parameters. Finally, the frontend interacts with Django through http requests which we use for sending/receiving audience feedback.

---

<sup>1</sup> Website: <https://puffer.stanford.edu/>

<sup>2</sup> Made available by Sadjad Fouladi at <https://github.com/stanford-stagecast/audience>

# Latency Research and MSE

One of the major issues we had to face in designing a working audience feedback system was **reducing the latency in the video and audio stream**<sup>3</sup>. We recognized that the so-called “liveness” of a performance in the digital space would be best perceived by the audience if they could communicate their reactions/feedback (to the actors and/or other audience members) almost immediately – similar to the experience in a real, physical live theater. Hence, our goal was to **achieve sub-second latency video and audio streaming in the already existing streaming system provided by Puffer**.

## Sources of Latency

As illustrated below in figure 1, our video is first transmitted from the video source (e.g. camera) to Puffer (the video streaming system), which in turn serves the video to the User (e.g. audience member) over the network.



Figure 1: The flowchart of the transmission of video from source to user

In inspecting our streaming system, we identified two key sources of latency: **video chunk size** (on the path from video source to Puffer) and **video buffer size** (on the path from Puffer to user).

The video source, VS, provides the recorded video to Puffer in chunks of predetermined size,  $x$ . Hence, it will take at least  $x$  amount of time for a video to reach Puffer from the video source, which means we will have at least an  $x$ -second delay on the path from video source to puffer (e.g. if the video source is a camera, it has to record 2s of video first and then send that to Puffer for streaming, and so there would be at least a 2s delay).

On the path from Puffer to user, on the other hand Puffer has to read the video chunks and only then can it stream them to the user. In that part of the system, the client builds and keeps

---

3

<https://aws.amazon.com/media/tech/video-latency-in-live-streaming/?fbclid=IwAR03bg8fNtp1w7zE2AiY2CcKfwCkBNqr9zVxLyDKr4xkRaKiDGI8wMEiSJw>

a buffer of size  $y$ . Hence from initialization, we have to wait for puffer to read the  $y$  seconds worth of video first before we can play the video. Hence, overall the system may have latency  $l \geq y$ .

In our original Puffer system, the video chunks we served were 2.0002 seconds long whereas our buffer was of size 15 seconds. Hence, the original system had 15-second latency. Whereas our goal was to achieve latency  $l \leq 0.5$  seconds.

## Initial Testing and Further Research

During initial testing, we found that the buffer size had to be:  $y \geq x$ , where  $x$  is the size of the video chunk and  $y$  is the buffer size. Hence, we were able to easily bring our latency down to 2.0 seconds by simply decreasing the buffer size down to 2.0 in the code for Puffer (as illustrated below in figure 2).

```
static constexpr double MAX_BUFFER_S = 15.0; /* seconds */
```

Figure 2: The line of code that defines the maximum buffer size (line 129 in ws\_client.hh)

To decrease the latency further, however, we had to decrease the served video chunk sizes as well. We were able to serve our video in 0.5 second chunks, which allowed us to decrease our buffer size to 0.5 and therefore achieve a 0.5 second latency.

On this step, however, each of our 0.5 video chunks started with a keyframe. Hence, keyframes appeared every 0.5 seconds in our video stream which we found redundant. To optimize further, we attempted to generate 0.5 chunks from our video stream, where only every other chunk contained a keyframe. We did this by attempting to parse the binary .m4s video files and using ffmpeg. This however seemed to be infeasible as we could not access the information about the location of the keyframe in the file from the metadata. Additionally, we were not able to determine if MSE (Media Source Extension - API for browser video streaming), which is used in the Puffer code, is able to process video chunks that don't start with a keyframe.

00000170	00 00 28 48 6d 64 61 74	00 00 02 ad 06 05 ff ff	.(Hmdat)...? .
00000180	a9 dc 45 e9 bd e6 d9 48	b7 96 2c d8 20 d9 23 ee	^ E 楊 奈 H 寫 , ? ? 頓
00000190	ef 78 32 36 34 20 2d 20	63 6f 72 65 20 31 35 35	x264 - core 155
000001a0	20 72 32 39 31 37 20 30	61 38 34 64 39 38 20 2d	r2917 0a84d98 -
000001b0	20 48 2e 32 36 34 2f 4d	50 45 47 2d 34 20 41 56	H.264/MPEG-4 AV
000001c0	43 20 63 6f 64 65 63 20	2d 20 43 6f 70 79 6c 65	C codec - Copyle
000001d0	66 74 20 32 30 30 33 2d	32 30 31 38 20 2d 20 68	ft 2003-2018 - h
000001e0	74 74 70 3a 2f 2f 77 77	77 2e 76 69 64 65 6f 6c	ttp://www.video1
000001f0	61 6e 2e 6f 72 67 2f 78	32 36 34 2e 68 74 6d 6c	an.org/x264.html
00000200	20 2d 20 6f 70 74 69 6f	6e 73 3a 20 63 61 62 61	- options: caba
00000210	63 3d 31 20 72 65 66 3d	31 20 64 65 62 6c 6f 63	c=1 ref=1 debloc
00000220	6b 3d 31 3a 30 3a 30 20	61 6e 61 6c 79 73 65 3d	k=1:0:0 analyse=
00000230	30 78 33 3a 30 78 31 31	33 20 6d 65 3d 68 65 78	0x3:0x113 me=hex
00000240	20 73 75 62 6d 65 3d 32	20 70 73 79 3d 31 20 70	subme=2 psy=1 p
00000250	73 79 5f 72 64 3d 31 2e	30 30 3a 30 2e 30 30 20	sy_rd=1.00:0.00
00000260	6d 69 78 65 64 5f 72 65	66 3d 30 20 6d 65 5f 72	mixed_ref=0 me_r
00000270	61 6e 67 65 3d 31 36 20	63 68 72 6f 6d 61 5f 6d	ange=16 chroma_m
00000280	65 3d 31 20 74 72 65 6c	6c 69 73 3d 30 20 38 78	e=1 trellis=0 8x
00000290	38 64 63 74 3d 31 20 63	71 6d 3d 30 20 64 65 61	8dct=1 cqm=0 dea
000002a0	64 7a 6f 6e 65 3d 32 31	2c 31 31 20 66 61 73 74	dzone=21,11 fast
000002b0	5f 70 73 6b 69 70 3d 31	20 63 68 72 6f 6d 61 5f	_pskip=1 chroma_
000002c0	71 70 5f 6f 66 66 73 65	74 3d 30 20 74 68 72 65	qp_offset=0 thre
000002d0	61 64 73 3d 31 20 6c 6f	6f 6b 61 68 65 61 64 5f	ads=1 lookahead_
000002e0	74 68 72 65 61 64 73 3d	31 20 73 6c 69 63 65 64	threads=1 sliced
000002f0	5f 74 68 72 65 61 64 73	3d 30 20 6e 72 3d 30 20	_threads=0 nr=0
00000300	64 65 63 69 6d 61 74 65	3d 31 20 69 6e 74 65 72	decimate=1 inter
00000310	6c 61 63 65 64 3d 30 20	62 6c 75 72 61 79 5f 63	laced=0 bluray_c
00000320	6f 6d 70 61 74 3d 30 20	63 6f 6e 73 74 72 61 69	ompat=0 constrai
00000330	6e 65 64 5f 69 6e 74 72	61 3d 30 20 62 66 72 61	ned_intra=0 bfra
00000340	6d 65 73 3d 33 20 62 5f	70 79 72 61 6d 69 64 3d	mes=3 b_pyramid=
00000350	32 20 62 5f 61 64 61 70	74 3d 31 20 62 5f 62 69	2 b_adapt=1 b_bi
00000360	61 73 3d 30 20 64 69 72	65 63 74 3d 31 20 77 65	as=0 direct=1 we
00000370	69 67 68 74 62 3d 31 20	6f 70 65 6e 5f 67 6f 70	ightb=1 open_gop
00000380	3d 30 20 77 65 69 67 68	74 70 3d 31 20 6b 65 79	=0 weightp=1 key
00000390	69 6e 74 3d 32 35 30 20	6b 65 79 69 6e 74 5f 6d	int=250 keyint_m
000003a0	69 6e 3d 32 34 20 73 63	65 6e 65 63 75 74 3d 34	in=24 scenecut=4

Figure 3: Part of a .m4s.file in binary format

After more research about streaming with low latency, we finally decided to switch from MSE to WebRTC. The reason is that MSE is generally not used for ultra low latency because it prioritizes the smoothness of streaming videos rather than extremely low latency. The original purpose of MSE is to replace Flash for playing videos on the browser. However, one downside is that we need to move our current work to WebRTC which is not easily scalable. Another downside is that WebRTC is not traditionally used for video streaming. However, we found a workaround to this, and the details about it are shown in the WebRTC Demos section below.

# WebRTC Demos

To explore the possibility of using WebRTC as the streaming platform, we looked into some of the open source implementations and demos, and built a prototype system showcasing the capabilities of this framework. The main prototype was built using Janus (Amirante et al. 2014), but we first describe certain other libraries we looked into. Note that our aim was to develop a system with one end running native code (preferably C++) which can transmit video on the disk or from a live stream to the other end running on the browser.

## Potential WebRTC libraries

- WebRTC native code from Google<sup>4,5</sup>: This is supposed to be the standard WebRTC library, and we found a couple of tutorials<sup>6,7</sup> on using this for native-to-browser streaming applications. However we found that this was not very well-documented and the tutorials used an older version of WebRTC making it complicated to get it working. In addition, the codebase is quite bulky due to the use of Chromium build toolchain and many dependencies. Overall, we felt that this was not suitable for building the initial prototype system.
- Pion WebRTC<sup>8</sup>: This is a pure Go implementation of WebRTC. We were able to test the demo play-from-disk<sup>9</sup> which can stream a video from a native WebRTC client to a browser. The code suggests that the data is transmitted frame-by-frame by taking the input file in the IVF format<sup>10</sup> which is simply a transport format for VP8 encoded video. They also have several other demos, including a broadcast demo<sup>11</sup> where the video is broadcasted to many peers but the video publisher needs to upload the video only once. This can be useful for applications where the publisher is bandwidth constrained, but we have a high throughput transmission server available. One possible concern here is the use of Go rather than C++.
- Other WebRTC implementations<sup>12</sup>: This webpage discusses the “top five” open source WebRTC media server projects, and we used the top one (Janus) for our main prototype as discussed next.

---

<sup>4</sup> <https://webrtc.googlesource.com/src/+refs/heads/master/docs/native-code/index.md>

<sup>5</sup> <https://webrtc.googlesource.com/src/+refs/heads/master/docs/native-code/development/index.md>

<sup>6</sup> <https://sourcey.com/articles/webrtc-native-to-browser-video-streaming-example>

<sup>7</sup> <https://github.com/brkho/client-server-webrtc-example>

<sup>8</sup> <https://github.com/pion/webrtc/>

<sup>9</sup> <https://github.com/pion/webrtc/tree/master/examples/play-from-disk>

<sup>10</sup> <https://wiki.multimedia.cx/index.php?title=IVF>

<sup>11</sup> <https://github.com/pion/webrtc/tree/master/examples/broadcast>

<sup>12</sup> <https://ourcodeworld.com/articles/read/1212/top-5-best-open-source-webrtc-media-server-projects>

## Streaming with Janus

In this section we describe the WebRTC streaming prototype built using Janus (Amirante et al. 2014). We go through a high-level overview here, and the detailed commands and other details are available in the [appendix](#). The overall system is shown in Figure 4 below.

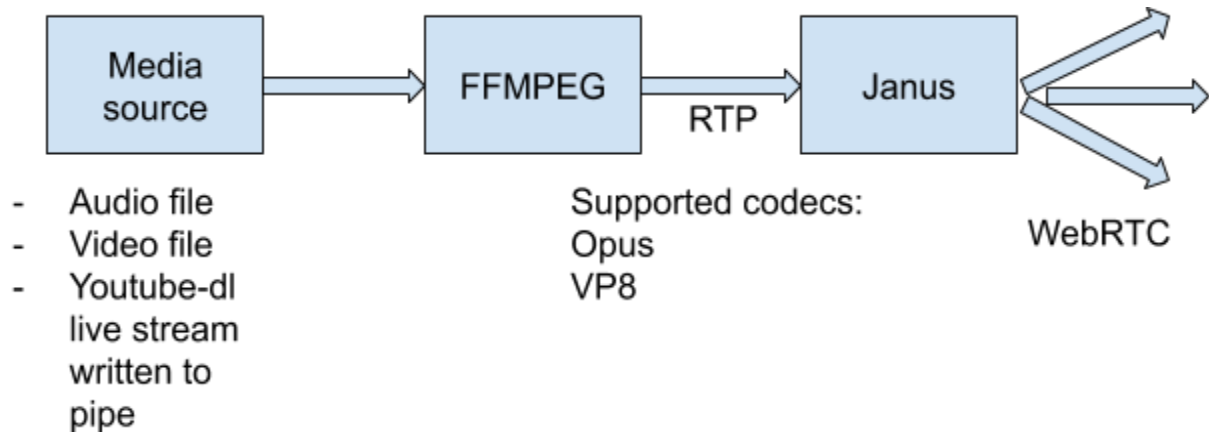


Figure 4: Janus streaming prototype architecture

As shown in the figure, the prototype is capable of streaming an input from an audio file, a video file or from youtube. In all cases, the input is first converted to an RTP (Real-time Transport Protocol) using ffmpeg (other tools such as gstreamer can be used for this task). We use RTP because Janus provides a RTP receiver that can read in the RTP stream as long as the audio is encoded using Opus codec and the video is encoded using VP8 codec (note that WebRTC and the browsers usually support certain other formats as well<sup>13</sup>). This stream is then sent over WebRTC and can be seen on the browser by serving the Janus demo website on a simple HTTP/PHP static server. When relevant, we used the `-re` flag for ffmpeg, which ensures that the input is read at frame rate, so that the communication to Janus properly simulates a real-time application. To ensure that the streaming works well when the input is not from a file but rather a live stream, we used the youtube-dl downloader piped into ffmpeg (youtube-dl<sup>14</sup> is a downloader for youtube videos). This nicely simulates a typical use-case for us where we get the video frames from an external source and need to rapidly transmit this to the audience through WebRTC. We show a few screenshots for the output below.

---

<sup>13</sup>

[https://developer.mozilla.org/en-US/docs/Web/Media/Formats/WebRTC\\_codecs#Supported\\_audio\\_codec](https://developer.mozilla.org/en-US/docs/Web/Media/Formats/WebRTC_codecs#Supported_audio_codec)

<sup>14</sup> <https://youtube-dl.org/>



## Plugin Demo: Streaming Stop


Streams

Stop Opus/VP8 live stream coming from external source (live)

**i** Metadata

You can use this metadata section to put any info you want!

Stream Started 640x360 300 kbits/sec



Janus WebRTC Server © Meetecho 2014-2020

Figure 5: Example of Janus streaming a normal YouTube video

## Plugin Demo: Streaming Stop


Streams

Stop Opus/VP8 live stream coming from external source (live)

**i** Metadata

You can use this metadata section to put any info you want!

Stream Started 266x144 174 kbits/sec



Janus WebRTC Server © Meetecho 2014-2020

Figure 6: Example of Janus streaming a live YouTube video

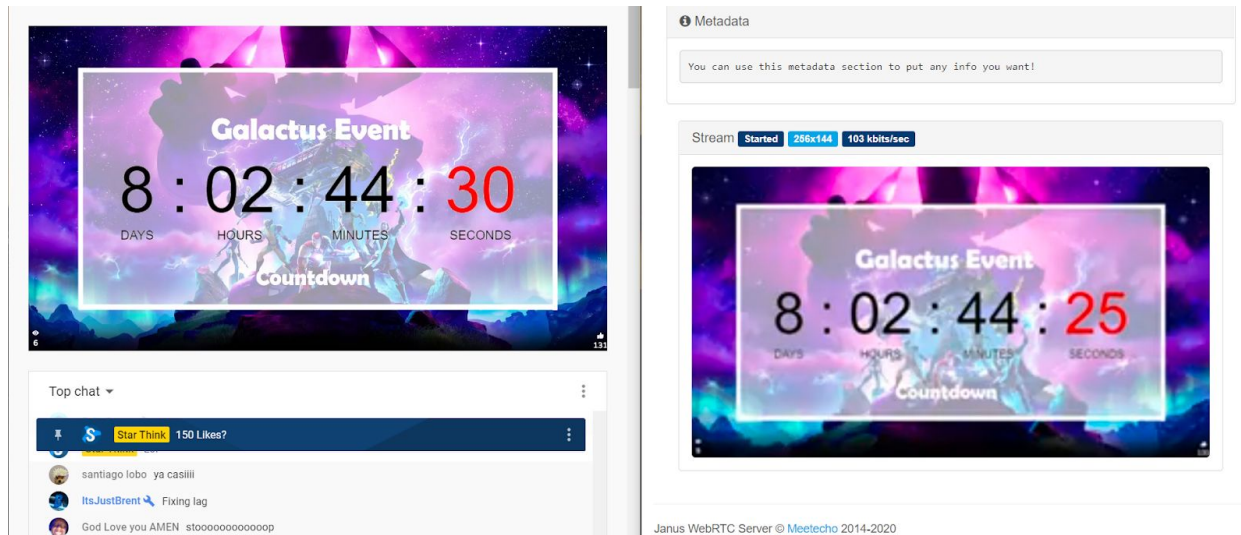


Figure 7: Comparing Janus streaming (right) with original YouTube live stream (left) for a countdown video. Notice that Janus is 5 seconds ahead of the YouTube stream as discussed further in the text.

As shown in Figure 5, we try to use Janus to stream a normal YouTube video<sup>15</sup>, and the bit rate is around 300 kbit/sec. As shown in Figure 6, we try to use Janus to stream a live YouTube video<sup>16</sup>, and the bit rate is around 174 kbit/sec. Notice that the bitrate of streaming a live YouTube video is higher than that of streaming a normal YouTube video. As shown in Figure 7, we also try to compare Janus streaming with YouTube live streaming for a countdown video<sup>17</sup>. Notice that our streaming is 5 seconds ahead of the YouTube streaming. The reason is that YouTube keeps a buffer to ensure that the streaming is less susceptible to the poor internet connection.

Finally, we discuss some of the limitations of the current prototype. We note that these are most likely related to some technical issues and not fundamental limitations of Janus itself. This is because the demos on the Janus website<sup>18</sup> do not suffer from these limitations. One limitation is that the prototype only works on Chrome, for reasons currently unknown. Another limitation is the performance, it's very good for audio, but a bit laggy for video, with the video dropping out sometimes. We could not go into the depths of these issues due to time constraints at the end of the quarter, we hope that the current prototype and research can be useful as a starting point for the actual system to be used in the Winter quarter performance.

<sup>15</sup> <https://www.youtube.com/watch?v=dQw4w9WgXcQ>

<sup>16</sup> <https://www.youtube.com/watch?v=DDU-rZs-lc4>

<sup>17</sup> <https://www.youtube.com/watch?v=NKfPhe245kE>

<sup>18</sup> <https://janus.conf.meetecho.com/>

# Emoji Feedback and Audio Feedback

This section describes the implementation of audience feedback through emojis and audio, as well as updates to the interface to enrich the audience experience. Since the work on reducing the latency is still ongoing, the current feedback system is not as real-time as aimed for in the ultimate system, and the feedback is currently supposed to be visualized and interpreted after the performance. We developed the mechanisms to collect data and test the ideas, noting that much of the front-end interface can be reused when the low latency implementation is available.

## Emoji Feedback

First we look at the implementation of emoji-based feedback shown in the screenshot in Figure 8 below. The list of emoji buttons is shown on the right bottom corner of the video screen, and the corresponding emoji button becomes larger and lights up when the cursor is placed over it. The list of emoji buttons is easy to edit by changing the html accordingly. The buttons were made responsive, using basic CSS media-query-based responsiveness for popular screen-size breakpoints (414px, 768px, 1024px). Note that the design is laptop-first (instead of mobile-first).



Figure 8: Screenshot of emoji feedback buttons

To communicate the button press event to the server, we implemented two mechanisms in JavaScript/Django. In both cases, the username, the video timestamp when the button was pressed and the button emoji text are transmitted.

1. **WebSocket-based feedback:** Here the feedback is sent to the media server through the WebSocket connection used for receiving the multimedia and for relaying latency-related feedback. We implemented a special message type on the media server in C++ to interpret such messages, but currently the messages are just printed on the stdout and not actually stored or shown to the director. We believe that the WebSocket based feedback can have certain advantages later since it can naturally support streaming data (such as audio feedback), but currently we mostly rely on the Django feedback described below.
2. **Django-based feedback:** As described previously, the web server is built using Django which provides an easy-to-use API for collecting and storing the feedback. We store the post-hoc timestamped feedback in a local SQLite database through the Django backend. We expose an endpoint called *feedback* that takes the timestamp and feedback text, and auto-populates the logged-in user to the request, and creates a new instance of the feedback model. The received feedback can be seen using the Django admin interface as shown in Figure 9. Future work includes compilation of the feedback and presentation in a suitable format for the director and performers.

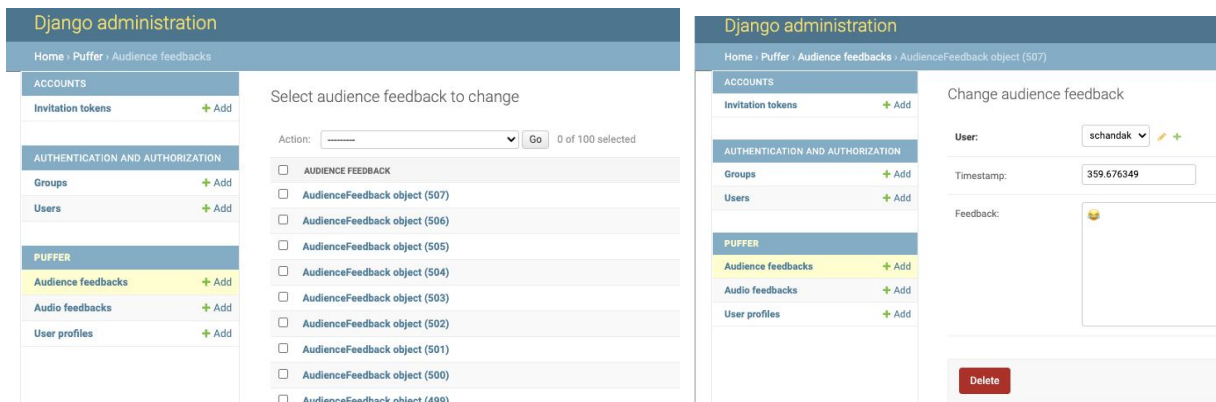


Figure 9: Django admin interface showing the emoji feedback

## Animation

The interface shows animation effects on the video player when the emoji buttons are pressed to enhance the user experience. We use two animation methods, both based on CodePen community contributions. Our aim was to entertain the user without obstructing the actual content, and therefore we display the animations only on part of the player, and use low opacity for them. The first<sup>19</sup> effect is to display emojis (corresponding to the button pressed)

<sup>19</sup> <https://codepen.io/vivinantonypen/gbENBB>

floating towards the top of the screen. The position of the emojis is chosen at random and CSS based animation is used. The second<sup>20</sup> effect is a firework animation that is shown only for emojis with a “happy” sentiment, as defined by an HTML data attribute. To avoid overuse, this is shown only with probability  $\frac{1}{3}$  when a “happy” emoji button is pressed. This uses a JavaScript canvas API for the animation. To make sure that the animations play only for a small duration after the button press, we store the last time the animation was displayed and stop the animation if no new button press occurred in the last 4 seconds. The animation effects are illustrated in Figure 10.

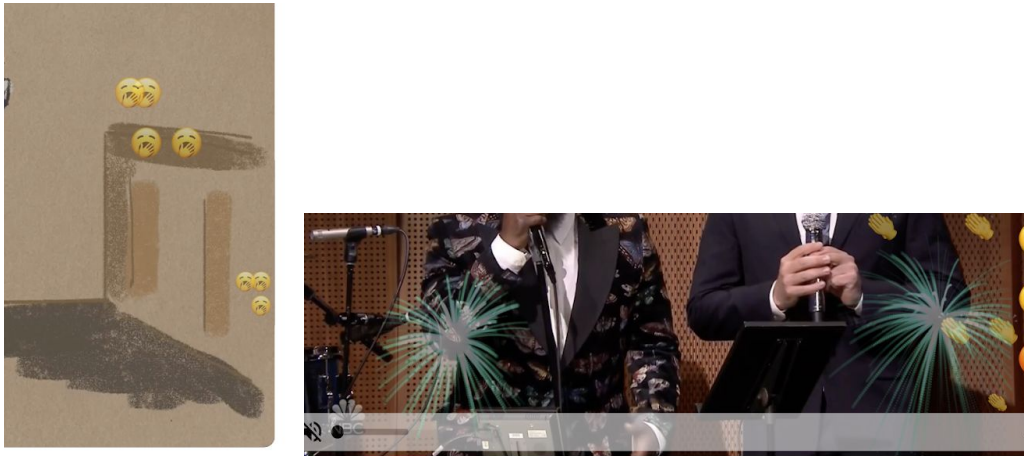


Figure 10: Emoji animation (left) and firework animation (right)

## Canned audio effects

An important component of any live performance is the cheering, clapping, laughing, booing and other sounds generated by the audience. We plan to ultimately implement this in real-time based on the cumulative feedback from the audience members. For example, a laughter soundtrack can be played when more than some threshold number of audience members press the laughing emoji. Having a threshold ensures that individual audience members cannot disrupt the performance. The laughter sound can be played both at the performers’ end and on the other audience machines, superimposed with the actual audio from the performance. However, since the low-latency system is still in development, we could not implement this in full.

Instead we developed a system where the emoji audio is played independently for each audience member according to the emoji button pressed by them. We obtained canned sound effects (as typically heard on TV shows) from an online database<sup>21</sup> and assigned the effects to specific emoji buttons using the HTML data attribute. For playing the audio we use the Howler

---

<sup>20</sup> <https://codepen.io/judag/pen/XmXMOL>

<sup>21</sup> <http://www.realmofdarkness.net/sb/crowd/>

JS library<sup>22</sup>. The effects are of length ~5s and are played at a lower volume than the actual audio to avoid disruption. We currently play the laugh, clap, boo and angry sound effects for the corresponding emojis. The relevant sound clips can be found on GitHub<sup>23</sup>.

## Audio Feedback

In addition to the emoji feedback, we implemented a preliminary version of audio feedback. The eventual goal here is to have live (possibly filtered and moderated) audio from the audience to the performers and other audience members, to simulate the theater experience. Another possibility would be to capture live audio, use algorithms to classify the audio and then play a corresponding canned soundtrack. This has the advantage of not playing any noise or sensitive conversations, while still offering the audience members the convenience of not having to repeatedly press the emoji buttons.

The current system is an initial prototype that collects and stores audio events, which can be listened to later and used for training purposes or to understand the viability of real-time audio feedback. We use the hark JS library<sup>24</sup> that allows audio event detection and hence we don't need to record and store the audio for the entire performance. This is then recorded and uploaded to the Django web server. The various aspects of the system are detailed below:

- **Privacy considerations:** To address privacy concerns, the audio feedback is optional and the rest of the platform works without hitches even if the user blocks the microphone in the browser. In addition, we show an indicator whenever the audio is being recorded and offer the option to mute/unmute the recording system during the performance (muted by default). See Figure 11 for an illustration.



Figure 11: Audio recording indicator, mute and unmute buttons in the player.

- **Audio detection:** The hark library performs polling at a user-defined interval and raises JS events when the speaking starts or stops. It uses a decibel threshold which can be changed. To capture short claps we need a small polling interval (set to 10ms). But a

---

<sup>22</sup> <https://github.com/goldfire/howler.js/>

<sup>23</sup>

<https://github.com/stanford-stagecast/audience/tree/audience-feedback/src/portal/puffer/static/dist/audio>

<sup>24</sup> <https://github.com/otalk/hark>

small polling interval leads to highly fragmented audio capture (e.g., two consecutive claps in different audio files). To fix the issue above, we added logic that makes sure we stop recording only when there has been continuous silence for some time (currently set to 1s).

- **Audio recording:** For recording the audio, we use the MediaStream Recording API<sup>25</sup>, where a separate recording was performed for each detected event. The recording was converted to WebM format using the Opus codec and the file size was around 7 KB per second of recording. We found that in some cases the detection was on for very long durations (see point on limitations below), and this led to very large files being sent to the server. So we limit a single recording to 60s, where we send chunks of size 60s to server (with appropriate suffix for identification and ordering). At the server these can simply be concatenated to recover the original long recording. This provides resilience in case the user closes the browser or the recording gets too long.
- **Audio recording upload:** We use a Django model and endpoint, similar to that used for emoji feedback. The Django model contains the file, the user name and the video timestamp (to enable association of audio with the section of the performance). The file is saved to `media/{username}/audio_{timestamp}_{suffix}.webm` where the suffix is for cases where the file needs to be split into 60s chunks (see previous point). The directory structure at the server and the Django admin interface for the audio feedback are shown in Figure 12 below.

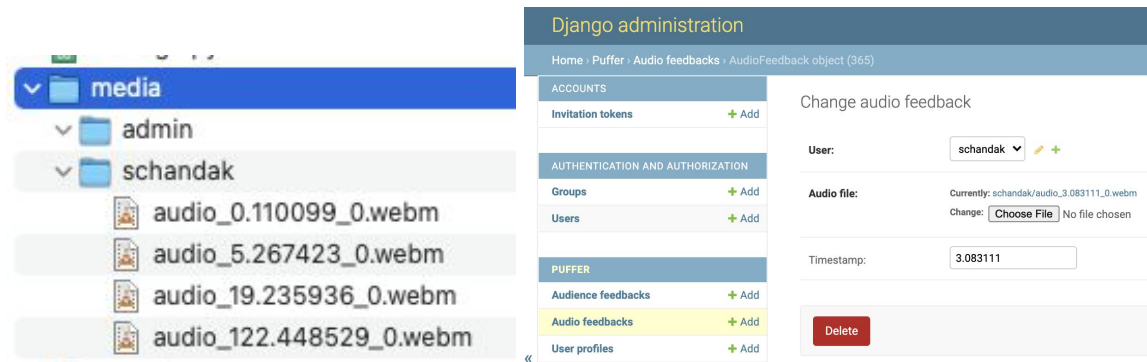


Figure 12: Audio feedback directory structure and Django admin interface.

- **Limitations:** The current system has certain limitations related to audio detection. One is that the detection is somewhat over-sensitive and tends to capture background noises such as a closing door. This can be probably be fixed using careful tuning of the detection threshold, however it might be hard to select a uniform threshold that works

across different users with different mic quality. One solution could be to adaptively select the threshold for each user.

Another major issue is related to echo cancellation. For Chrome and most other browsers, echo cancellation doesn't work for non-WebRTC audio input<sup>26</sup>. This means that if the user listens to the performance on a speaker, that audio is detected and recorded. One solution for this is to build a custom echo cancellation system, another is to separate the original audio at the server. Currently we simply display a warning recommending that the users keep their audio muted unless they wish to provide feedback or use headphones for listening.

## Chat

In order to allow for a more interactive experience, we implement a chat functionality where viewers can interact with one another in real time. The plan for the final version is to have chat messages relayed to the actors/directors in real time as well, allowing for a fluid performance where any feedback floated through the chat can be incorporated as deemed necessary by the performers. For the current version, the chat messages are stored for post-hoc analysis, so that the viewers' comments can reach the performers with little friction in order to inform future productions.



Figure 13: Chat box as it appears alongside the video.

- **Design:** The chat box appears on the main watch page alongside the video player. The trade-off is that it reduces the size of the video player; but more visibility for the chat means that people may be more likely to use it, which would allow us to gather more

<sup>26</sup> <https://github.com/webrtc/samples/issues/1243#issuecomment-626810415>



feedback. Note that the design of the component, including the color palette, is similar to that of the rest of the page. All messages sent via the chat are time-stamped based on the current video time, and displayed in latest-at-the-bottom format. The chat has an auto-scroll feature that triggers when the viewer has not manually scrolled too far towards the top of the component, which allows for a smoother experience, and adds an additional layer of interactivity to the page. Note that timestamps are not displayed in the chat: we only show the sender's username and the message itself.

- **Integration with Emoji-based Feedback:** Without the chat feature, the emoji-based feedback only affected an individual user's experience in that they would see the emoji-associated animation overlaid on the video player. With the chat, however, we make the emoji feedback a more shared experience. All emoji feedback is stored on the backend in the same format as the chat messages, and displayed in the chat box as such.
- **Server and Pipeline:** The Django puffer app serves the views for the player. We use a *feedback* model to store the associated user, the timestamp (as a float), and the feedback text. We expose an endpoint called `feedback` that takes the timestamp and feedback text, and auto-populates the logged-in user to the request, and creates a new instance of the feedback model.

In order to create the chat replay when a user loads the page, we use a new endpoint to collect all chat/feedback messages between a from and a to time (as before, these timestamps correspond to in-video times). This endpoint returns a JSON-list of usernames, timestamps, and feedback texts. This list is then sorted by the frontend in increasing order of timestamps.

The frontend fetches new messages from the server every 500 milliseconds to ensure that different clients are closely synced to new messages sent by other clients. Note that the 500ms delay is also small enough to ensure that a new feedback (both emoji-based and chat) sent by a user seems to have been immediately added to the bottom of the chat for the user. The frontend also tracks the timestamp for the last server call, using this as the "from" timestamp for the next message fetch.

- **Full-screen Chat:** We realize that many viewers would prefer to experience the actual performance in full screen, which would traditionally hide the chat in other streaming services. In order to retain some semblance of an interactive theater experience even when the video player is in full-screen mode, we allow the option of viewing the chat replay as an overlay on the full-screen video player as shown in Figure 14.

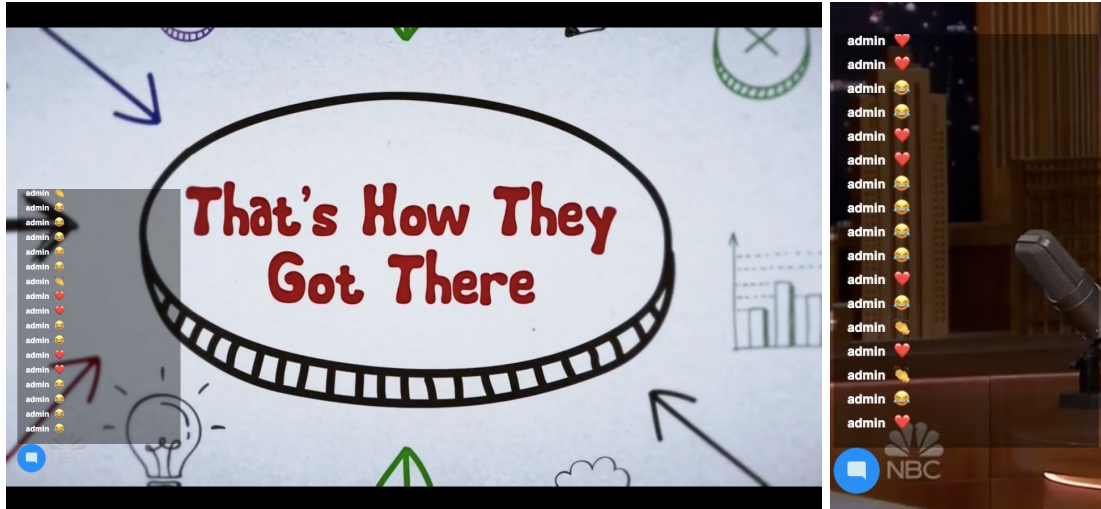


Figure 14: Chat box in full-screen mode.

We also show a floating toggle-button that can be used to hide the full-screen chat as needed. This chat uses the same server-fetching functionality as the screen-side chat, and both chat components are updated simultaneously. Note, however, that the full-screen chat only allows reading a replay of messages, and not sending new ones, primarily because of design constraints. Since the full-screen chat is overlaid on a semi-transparent box, we found it difficult to design a message entry that works full-screen. Given more data on how much viewers use both the normal and full-screen chat features, we can develop an entry design that also works full-screen.

- **Moderation:** A mechanism to filter and moderate chat messages is very important for a feature that allows interaction between viewers, and between viewers and performers. For the current example, we allow blocking users from the chat through the Django admin portal. This is done by creating a *UserProfile* model linked with user accounts that stores a boolean flag. If this flag is set, they can still see messages that other users send to the chat, but cannot send new messages. When a user is blocked from the chat, they see a different component in place of the chat entry as shown in Figure 15.

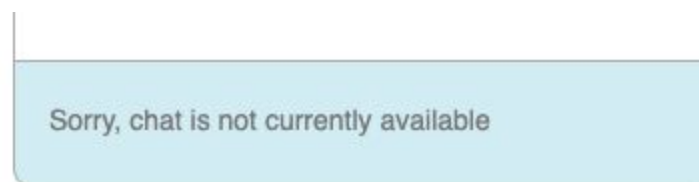


Figure 15: Users who have been blocked from the chat see this message in place of the chat entry.

- **Video Loop:** We considered a possibility of playing the performance on loop over an extended period in order to gather more data about user experiences from a larger cohort of users. In order to support this, the chat messages are only tagged with the video timestamp, and fetched from the server as is. As a result, when the video is played on loop, messages from different re-runs are not distinguished from each other. To a user, this would give the sense of more user interaction, which is likely to further encourage them to engage with the chat functionality.

## Conclusion and Future Work

The interaction and the exchange of energy between the actor and the spectator is what distinguishes a theatrical experience from other types of media consumption. During a live performance, audience feedback is crucial not only for the performers and the creative team but also for the audience members themselves. It is through this feedback that a sharing of space and community is established.

Through this project we intended to envision how theatrical ‘liveness’ and interaction translates (or rather becomes redefined) in an online space. We hope that the current progress on emoji and audio-based feedback, audiovisual effects to enhance user experience, chat functionality, and the exploration of low-latency video transmission using Puffer and WebRTC can act as an inspiration for such a system and provide useful building blocks for achieving this goal. In the future work, we suggest a few ways in which the audience members’ presence can become even more prominent - i.e. through audience auditory and visual feedback - thus facilitating a fuller, more fulfilling, and more “live” theatrical experience.

While the chat function and emoji feedback seemed to be fundamental in creating an interactive online space, many other video streaming platforms already utilize this form of interaction. To enhance the audience experience, we suggest enabling audience members to enter “breakout/chat rooms” with their friends, where they can view the performances together and chat/talk to each other via audio conferencing throughout the performance. Additionally, we encourage creating a digital theater lobby for the audience to interact with each other, perhaps through digital puppets, at beginning, end and intermissions of performance.

We also recognize some technical challenges that require development. Those include: low latency audience feedback which is to be communicated back to the audience, actors and staff, adjustments to the levels of this feedback - i.e. intensity of sound (loudness), visual effects (ensuring minimal disruption). We also suggest a utilization of statistics and compilation of feedback data from the audience. This data can be utilized to generate reactions of the

appropriate intensity back to the audience or can be reported back as a statistic to the director/crew members of the project.

Finally, we encourage exploring ways in which our platform can ensure user safety. We envision our space to be inclusive, non-discriminatory and respectful. It should be our priority to protect our users against any type of harmful and disruptive behavior.

## Acknowledgements

We would like to thank the mentors and instructors Sadjad Fouladi, Prof. Michael Rau, Prof. Tsachy Weissman and Prof. Keith Winstein for their guidance and helpful suggestions.

## References

Amirante, A., Castaldi, T., Miniero, L., & Romano, S. P. (2014, October). Janus: a general purpose WebRTC gateway. In *Proceedings of the Conference on Principles, Systems and Applications of IP Telecommunications* (pp. 1-8).

Francis Y. Yan, Hudson Ayers, Chenzhi Zhu, Sadjad Fouladi, James Hong, Keyi Zhang, Philip Levis, & Keith Winstein (2020). Learning in situ: a randomized experiment in video streaming. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)* (pp. 495-511). USENIX Association.

W3C (2016) Media Source Extensions. <https://www.w3.org/TR/media-source/>.

## Appendix: Janus Streaming

Here we describe the detailed commands used for building and running Janus for our WebRTC prototype, please see the [relevant main section](#) in the report for context. The code for Janus is available on GitHub at <https://github.com/meetecho/janus-gateway>. We tested Janus both on Linux and on Mac. For Linux we used the popeye2.stanford.edu server which is the address referred to in some of the commands below. Also note that the prototype currently only works

on Chrome. But similar demos available on the Janus website<sup>27</sup> work on other browsers, suggesting that this is only a technical issue and not a fundamental limitation of Janus. Finally, note that the Janus streaming demos also have the option to take input directly from a file, however we do not use this since it is limited to only the a-law and mu-law audio formats. Instead we use the demo that uses input from RTP that can support a larger range of inputs.

## Building Janus

After installing the dependencies, run the following:

```
git clone https://github.com/meetecho/janus-gateway.git
cd janus-gateway
sh autogen.sh
./configure --prefix=/home/schandak/janus-gateway/bin/
--disable-aes-gcm
make
make install
make configs
```

On Mac, the `./configure` command should also have

```
PKG_CONFIG_PATH=/usr/local/opt/openssl/lib/pkgconfig
```

### Notes:

- Currently we directly run “`sudo apt-get install libnice-dev`” to install the dependency `libnice`. However, note the warning: “While `libnice` is typically available in most distros as a package, the version available out of the box in Ubuntu is known to cause problems. As such, we always recommend manually compiling and installing the master version of `libnice`.”
- The `--disable-aes-gcm` flag helped in avoiding an `srtp` related compilation error<sup>28</sup>. Also see the next point.
- For the error “undefined symbol: `srtp_crypto_policy_set_aes_gcm_256_16_auth`”, it could be solved by running:  
    `export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/usr/lib`
- `--prefix` in `configure` command allows us to specify the folder where Janus is installed. `$PREFIX` denotes this path in the commands below.

---

<sup>27</sup> <https://janus.conf.meetecho.com/demos.html>

<sup>28</sup> <https://janus.conf.meetecho.com/docs/FAQ#aesgcm>

# Running Janus

## General Comments

- To run janus, type in `$PREFIX/bin/janus` on the command line.
- To run the html demo server, go to the html folder in the repo root directory, and then do `python3 -m http.server [PORT]` (where PORT is optional, and is set to 8000 by default). Then you can open `poppeye2.stanford.edu:PORT` in Chrome. You can also do `php -S 0.0.0.0:8000` to set up a server.
- The relevant code for the streaming is available in `plugins/janus_streaming.c` within the repo root directory.
- The configuration file for the streaming is at `$PREFIX/etc/janus/janus.plugin.streaming.jcfg`.

## Streaming audio from file via RTP

### Set up RTP streaming

```
Run ffmpeg -re -stream_loop -1 -i infile.mp3 -c:a libopus -f rtp
rtp://127.0.0.1:5002
```

### Notes:

- The command streams an input file to RTP in a loop
- `-re` is needed when streaming from a file so that all the file is not sent at once, instead we send at the native frame rate
- `-stream_loop -1` runs the thing in a loop
- `-c:a libopus` is needed to make output opus codec which is needed for janus (we tried `-c a:pcm_mulaw` but that gave an error even though mu-law should be supported for webrtc<sup>29</sup>)
- `-f rtp` sets the output format to rtp protocol
- `rtp://127.0.0.1:5002` is the output address. 127.0.0.1 is just localhost, 5002 is the port number that'll come up below

### Update config file:

- Find `rtp-sample` section in the config file
- Set `audioport` to the port from the `ffmpeg` command.

- In the log of the ffmpeg command, you'll see a line like `a=rtpmap:97 opus/48000/2`: in the config set `audioopt` to 97 and `audiortpmap` to `opus/48000/2`.

After this, just run janus, go to the demo site, and open Streaming demo with mode Opus/VP8 live stream.

## Streaming video from file via RTP

### Set up RTP streaming

```
Run ffmpeg -re -stream_loop -1 -i input.mp4 -an -c:v libvpx -f rtp
rtp://127.0.0.1:5004 -vn -c:a libopus -f rtp rtp://127.0.0.1:5002
```

#### Notes:

- The command streams an input file to RTP in a loop
- `-re` is needed when streaming from a file so that all the file is not sent at once, instead we send at the native frame rate
- `-stream_loop -1` runs the thing in a loop
- `-c:a libopus` is needed to make output opus codec which is needed for janus
- `-c:v libvpx` is needed to make output VP8 codec
- `-an` and `-vn` are used to generate two separate streams (videos without audio and audio without video)
- `-f rtp` sets the output format to rtp protocol
- `rtp://127.0.0.1:5004` is the output address for video.
- `rtp://127.0.0.1:5002` is the output address for audio.
- You can do either video or audio by omitting the appropriate options.
- This doesn't work with mp4 files containing more than 4 audio channels.

#### Update config file:

- Find `rtp-sample` section in the config file
- Set `audioport` to the port from the ffmpeg command.
- Set `videoport` to the port from the ffmpeg command.
- In the log of the ffmpeg command, you'll see a line like `a=rtpmap:97 opus/48000/2`: in the config set `audioopt` to 97 and `audiortpmap` to `opus/48000/2`. Do similar set-up for the video with `videoopt` to 96 and `videortpmap` to `VP8/90000`.

After this, just run janus, go to the demo site, and open Streaming demo with mode Opus/VP8 live stream.

Notes:

- When video is played, it sometimes shows “No remote video available”. This issue is also discussed at <https://github.com/meetecho/janus-gateway/pull/1972>. One potential solution is to comment the code at <https://github.com/meetecho/janus-gateway/blob/8491eb86obf7fdcee94b5fdec9e9e43ofbe2421c/html/janus.js#L1921-L1935>.
- Generally the video quality seems a bit patchy and we also saw some issues with audio/video sync that ideally shouldn't happen. It's not clear if the performance issues are due to (i) ffmpeg encoding speed, (ii) RTP protocol, (iii) Janus, (iv) network conditions of webrtc connection. These issues were experienced both on the Stanford Wi-Fi network and via VPN when the system was running on the popeye2 server located in the EE department at Stanford.
- A few options for ffmpeg that might help in performance: `-threads` to increase number of threads, `-vframes` (for reduced frame size at output<sup>30</sup>).

## Streaming video from YouTube via RTP

Instead of reading from a file one can take input from youtube (either a normal video or a live stream). For this we can use youtube-dl which can be obtained from <https://youtube-dl.org/> (note that the version installed by `sudo apt-get` is old and doesn't work anymore). The following is partly based on an online tutorial<sup>31</sup>. Documentation for youtube-dl can be found online<sup>32</sup>.

First find a youtube video/live stream and copy the url. For example, we'll use <https://www.youtube.com/watch?v=dQw4w9WgXcQ>.

Then run

```
youtube-dl -f worst -q --prefer-ffmpeg -o -  
https://www.youtube.com/watch?v=dQw4w9WgXcQ | ffmpeg -threads 40  
-re -i - -vn -c:a libopus -f rtp rtp://127.0.0.1:5002 -an -c:v libvpx  
-f rtp rtp://127.0.0.1:5004
```

This basically writes output of youtube-dl to a pipe which is read by ffmpeg and sent to RTP. The `-` in the youtube-dl output (`-o -`) and the ffmpeg input (`-i -`) denote this fact. For youtube-dl: `-f worst` selects worst video quality available on YouTube, and `-q` is to quiet down the logging output. Everything else remains the same. If we use a live stream video on

<sup>30</sup> <https://trac.ffmpeg.org/wiki/Scaling>

<sup>31</sup> <https://flashphoner.com/how-to-grab-a-video-from-youtube-and-share-it-via-webrtc-in-real-time/>

<sup>32</sup> <https://github.com/ytdl-org/youtube-dl/blob/master/README.md#readme>



YouTube, then we get the live version. This can be used to test the latency of the overall system using a suitable live stream video, potentially one that includes a timer. The `-re` flag for `ffmpeg` is optional and should be skipped when we are dealing with a live stream on YouTube.