MICROSOFT ENGAGE 2021

VIDEO CONFERENCING WEB APP

Challenge

Building a Microsoft Teams Clone. Your solution should be a fully functional prototype with at least one mandatory functionality - a minimum of two participants should be able connect with each other using your product to have a video conversation.

Tech Stack

- Node.js
- Express
- Socket.io
- WebRTC
- HTML/CSS/Bootstrap

Concept

WebRTC

WebRTC is nothing more than a group of standards and features composed in APIs that can be used to gain access to media devices and establish a peer-to-peer connection with other clients. These APIs used in conjunction with a signaling process and a bunch of other elements, are used to initiate a video/audio call between two or more users.
WebRTC is peer-to-peer by nature. This means that most of the time, there will be no intermediaries in a WebRTC call. The communication will be direct, browser to browser or device to device. This added to the fact that it encrypts the media transport by default, makes it a secure solution for real time communication.

APIs/Interfaces used:

- RTCPeerConnection: Represents a WebRTC connection between the local computer and a remote peer. It is used to handle efficient streaming of data between the two peers.
- RTCDataChannel: Represents a bi-directional data channel between two peers of a connection.
- RTCSessionDescription: Represents the parameters of a session. Each RTCSessionDescription consists of a description type indicating which part of the offer/answer negotiation process it describes and of the SDP descriptor of the session.
- RTCIceCandidate: Represents a candidate Interactive Connectivity Establishment (ICE) server for establishing an RTCPeerConnection
- MediaDevices.getUserMedia() method prompts the user for permission to use a media input which produces a MediaStream with tracks containing the requested types of media. That stream can include, for example, a video track (produced by either a hardware or virtual video source such as a camera, video recording device, screen sharing service, and so forth), an audio track (similarly, produced by a physical or virtual audio source like a microphone, A/D converter, or the like), and possibly other track types.
- MediaDevices.getDisplayMedia(): The MediaDevices interface's getDisplayMedia() method prompts the user to select and grant permission to capture the contents of a display or portion thereof (such as a window) as a MediaStream.

Signalling Layer:

Unfortunately, WebRTC can’t create connections without some sort of server in the middle. We call this the signal channel or signaling service. It’s any sort of channel of communication to exchange information before setting up a connection. Here, socket.io is used for the signalling process.

The information we need to exchange is the Offer and Answer which just contains the SDP. Peer A who will be the initiator of the connection, will create an Offer. They will then send this offer to Peer B using the chosen signal channel. Peer B will receive the Offer from the signal channel and create an Answer. They will then send this back to Peer A along the signal channel.

SDP and ICE :

The configuration of an endpoint on a WebRTC connection is called a session description. The description includes information about the kind of media being sent, its format, the transfer protocol being used, the endpoint's IP address and port, and other information needed to describe a media transfer endpoint. This information is exchanged and stored using Session Description Protocol (SDP).

When a user starts a WebRTC call to another user, a special description is created called an offer. This description includes all the information about the caller's proposed configuration for the call. The recipient then responds with an answer, which is a description of their end of the call. In this way, both devices share with one another the information needed in order to exchange media data. This exchange is handled using Interactive Connectivity Establishment (ICE), a protocol which lets two devices use an intermediary to exchange offers and answers even if the two devices are separated by Network Address Translation (NAT).

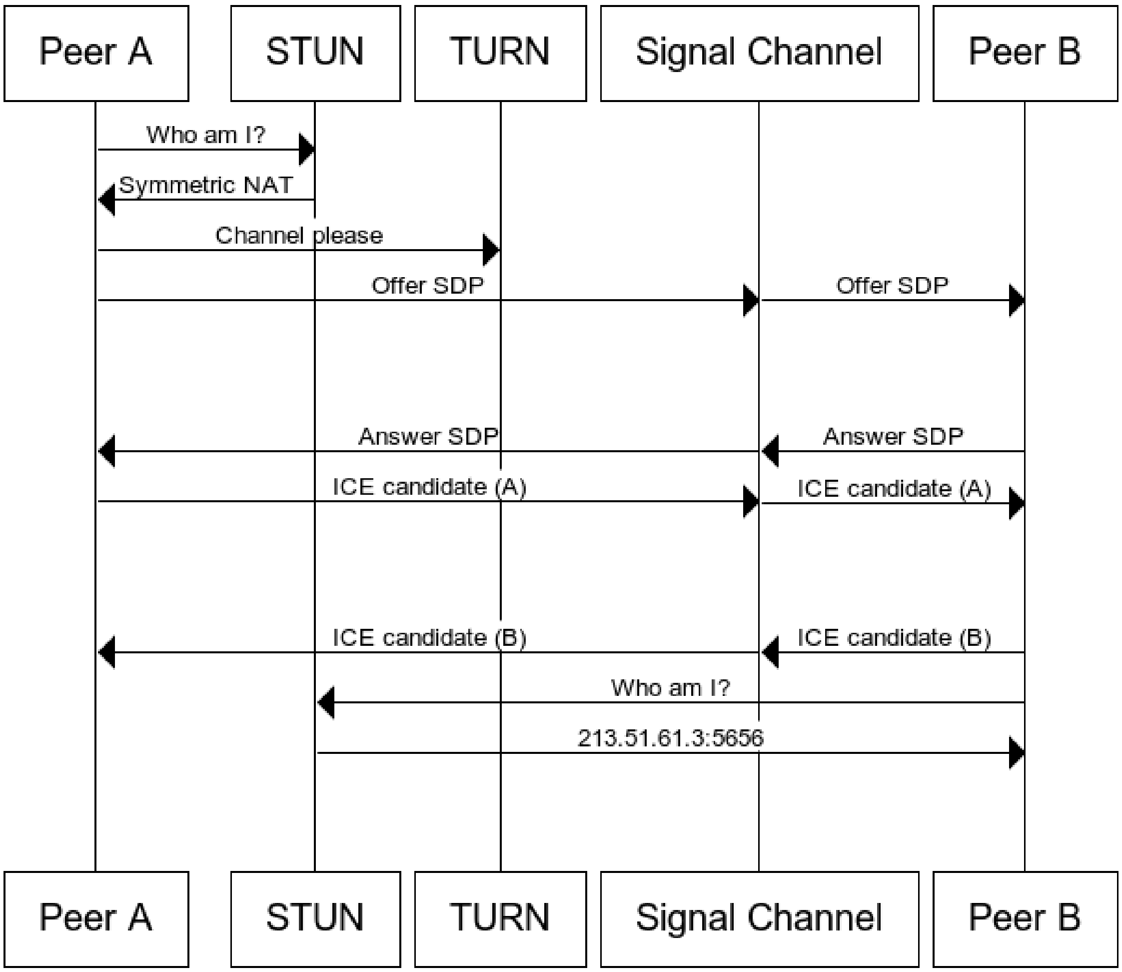
Process

Each peer keeps two descriptions on hand: the **local description**, describing itself, and the **remote description**, describing the other end of the call.

The offer/answer process is performed both when a call is first established, but also any time the call's format or other configuration needs to change. Regardless of whether it's a new call, or reconfiguring an existing one, these are the basic steps which must occur to exchange the offer and answer, leaving out the ICE layer for the moment:

1. The caller captures local Media via `MediaDevices.getUserMedia`
2. The caller creates `RTCPeerConnection` and calls `RTCPeerConnection.addTrack()` (Since `addStream` is deprecating)
3. The caller calls `RTCPeerConnection.createOffer()` to create an offer.
4. The caller calls `RTCPeerConnection.setLocalDescription()` to set that offer as the *local description* (that is, the description of the local end of the connection).
5. After setLocalDescription(), the caller asks STUN servers to generate the ice candidates
6. The caller uses the signaling server to transmit the offer to the intended receiver of the call.
7. The recipient receives the offer and calls `RTCPeerConnection.setRemoteDescription()` to record it as the *remote description* (the description of the other end of the connection).
8. The recipient does any setup it needs to do for its end of the call: capture its local media, and attach each media tracks into the peer connection via `RTCPeerConnection.addTrack()`
9. The recipient then creates an answer by calling `RTCPeerConnection.createAnswer()`.
10. The recipient calls `RTCPeerConnection.setLocalDescription()`, passing in the created answer, to set the answer as its local description. The recipient now knows the configuration of both ends of the connection.
11. The recipient uses the signaling server to send the answer to the caller.
12. The caller receives the answer.

13. The caller calls RTCPeerConnection.setRemoteDescription() to set the answer as the remote description for the end of the call. It now knows the configuration of both peers. Media begins to flow as configured.

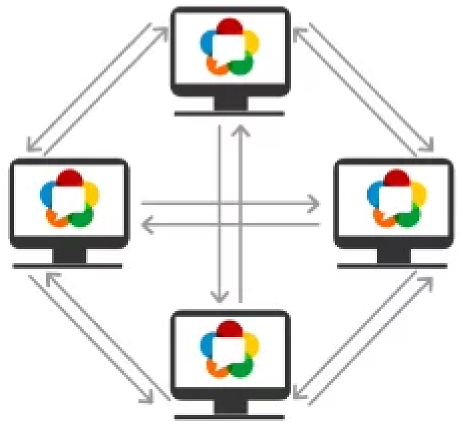


(Schematic Diagram of a WebRTC P2P connection, source: https://developer.mozilla.org/en-US/docs/Web/API/WebRTC_API/Connectivity)

Multiple Users:

Group video calls can be held by extending the above idea to multiple peer-to-peer connections, i.e. a mesh architecture.

In Mesh architecture, each participant in a video room sends and receives video, audio and information about their connection to each participant. This means that if you are connected to 4 people, you have 4 connections open.



(Web Conference with Mesh Architecture, Source: https://webrtc.ventures/2017/11/a-guide-to-webrtc-media-servers-open-source-options/)

Working

STUN/TURN Servers

These were procured by creating a free account on https://xirsys.com/ and obtaining the static STUN/TURN credentials.

Features

- **Group call:** Allowing multiple participants to join a room by sharing the unique room link (unique room ID created by appending a random string to the entered room number) and share media
- **Mute/Unmute:** Allowing users to mute themselves as per convenience
- **Video on/off:** Providing the option of setting video on/off

- **Pin screen:** Users can focus on a particular participant's video by clicking on the pin icon
- **Picture in Picture view:** Allows a floating video experience, where one can keep viewing their video even when on other tabs, using requestPictureinPicture() API
- **Screen Share:** Option of presenting entire screen, a tab or a window on the call, achieved using getDisplayMedia() API
- **Recording:** Users can record either their video or their screen, achieved using MediaRecorder API, and the file is saved as a webm file.
- **Chat:** Users can live chat during the call, with the choice of adding emoticons(using TinyMCE API) along with text. Autolink library is also used to automatically identify URLs in the chat.

Challenges/Road Ahead

Despite a lot of learning and applications, there remain certain challenges/scope of improvement in any program. However, the developer is determined to overcome these in the future versions of the webapp:

- Users have to be careful that their browser does not block their media content, for smooth functioning of webapp, it is thus recommended to view the app on Google Chrome for the best experience.
- Due to limitations of WebRTC mesh technology, scalability beyond a few simultaneous participants becomes a concern. The developer has considered and researched about the use of open source media servers like Kurento and Janus for future versions
- Maximum of the UI has been made responsive to accommodate for various screen sizes, however an element or two, in the chat, might be modified on different platforms.
- The developer would also like to continue working on building the chat feature for availability before and after the video meet as well.

References

- https://developer.mozilla.org/en-US/docs/Web/API/WebRTC_API (Theory)
- https://webrtc.ventures/ (Theory)
- https://gist.github.com/ZachSaucier/8295d9dc926d7064ff0d4f3f04b35b55 (CSS dark theme)
- https://www.tiny.cloud/docs/plugins/opensource/emoticons/ (TinyMCE API)
- https://github.com/amirsanni/Video-Call-App-NodeJS
- https://github.com/bryanwoods/autolink-js (Autolink library for identifying urls in chat)
- https://xirsys.com/ (STUN/TURN servers)