Why WebRTC?

WebRTC is the core/only protocol that lets you do **real time media communication** from inside a browser.

We already did this fairly well in a live stream
https://github.com/hkirat/omegle/tree/master
https://www.youtube.com/watch?v=0MIsI2xh9Zk

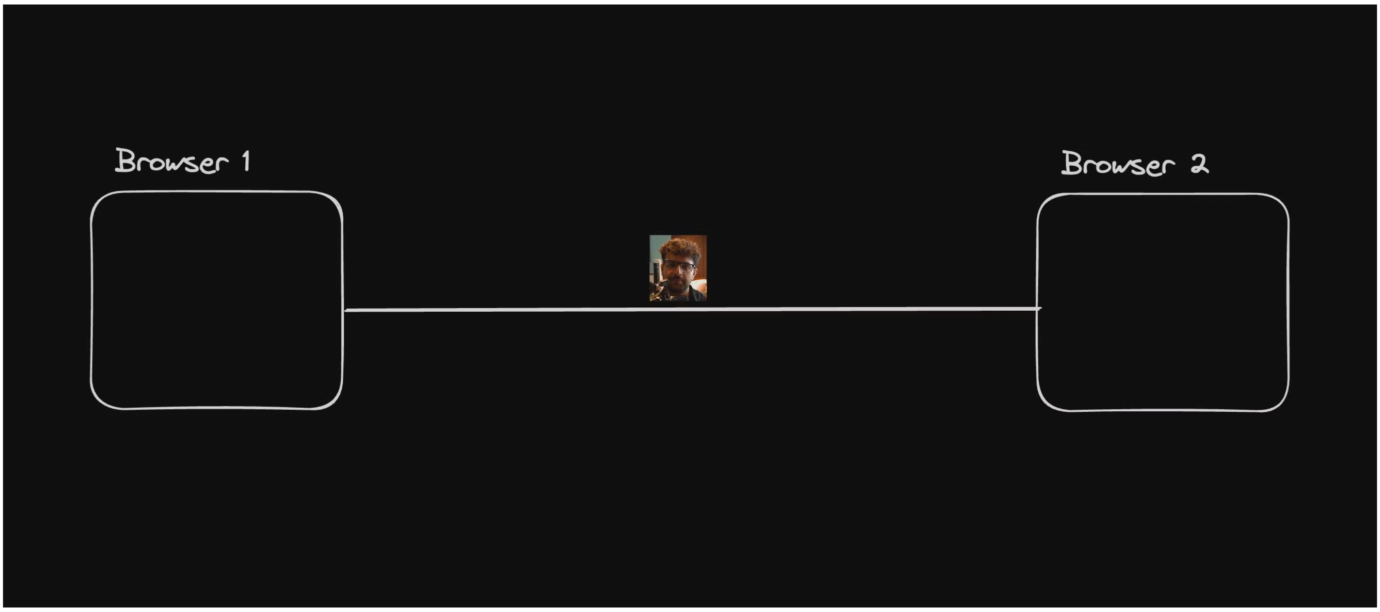
You use WebRTC for applications that require sub second latency.
Examples include

1. Zoom/Google meet (Multi party call)
2. Omegle, teaching (1:1 call)
3. 30FPS games (WebRTC can also send data)

WebRTC Architecture/jargon

P2P

WebRTC is a peer to peer protocol. This means the you directly send your media over to the other person without the need of a central server

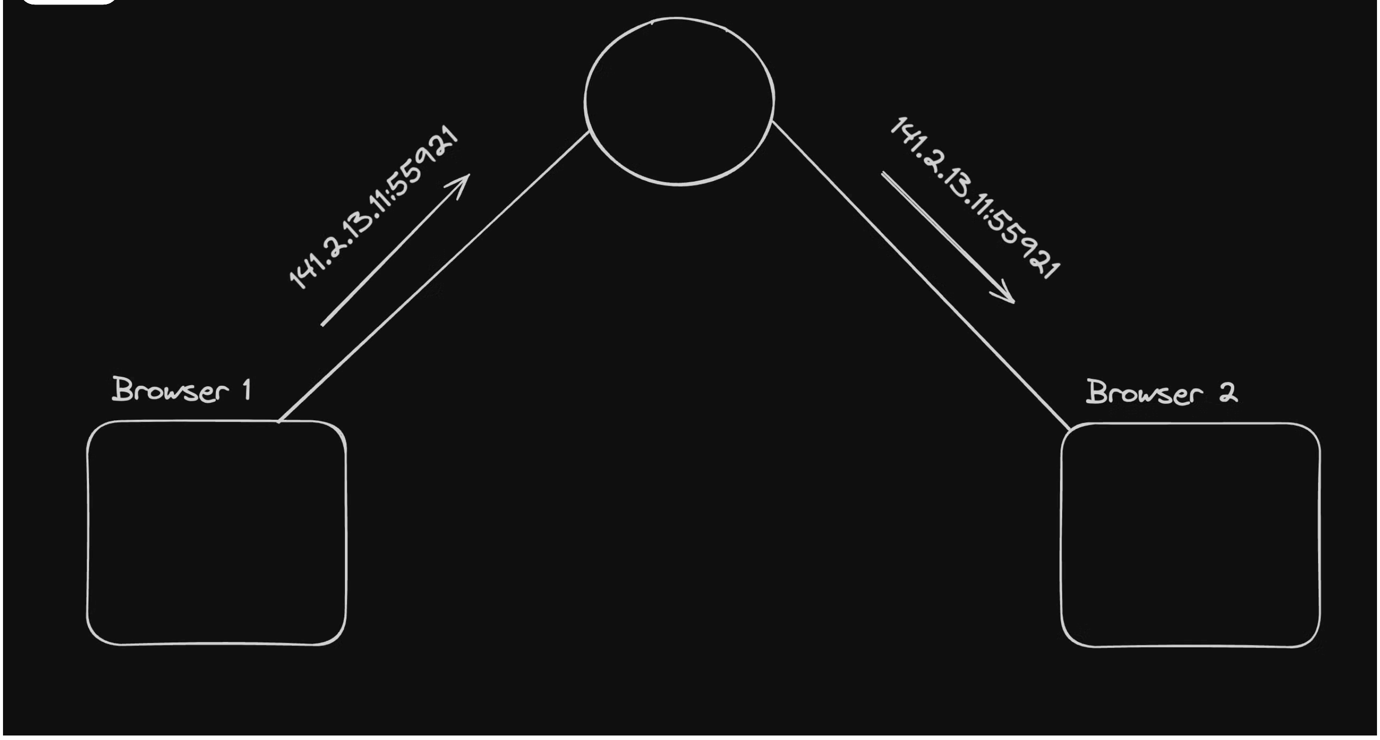


💡 You do need a central server for signalling and sometimes for sending media as well (turn). We'll be discussing this later

Signalling server

Both the browsers need to exchange their address before they can start t [illegible] s used for that.

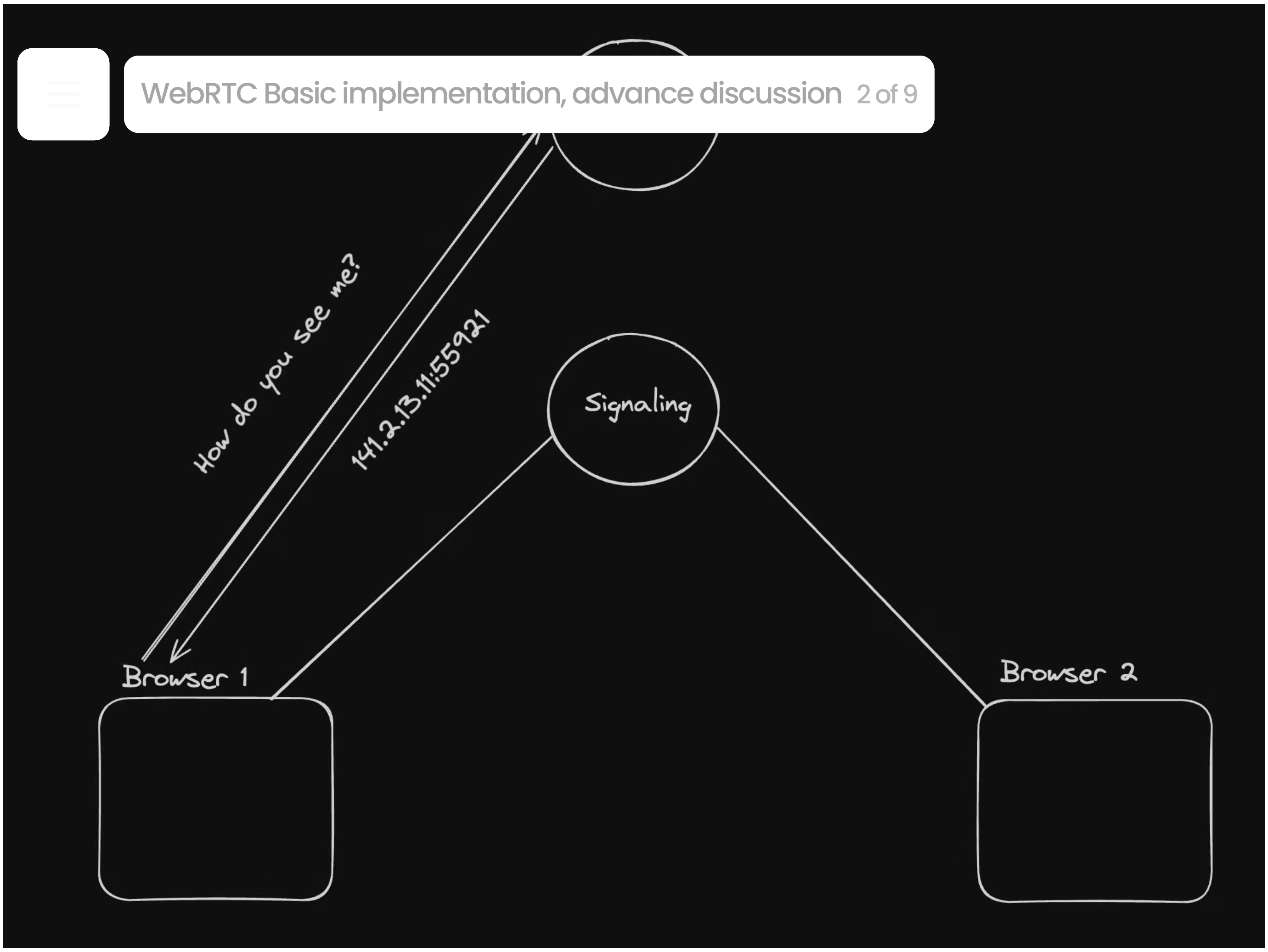
It is usually a websocket server but can be anything (http)



Stun (Session Traversal Utilities for NAT)

It gives you back your publically accessable IPs. It shows you how the world sees you

Check
https://webrtc.github.io/samples/src/content/peerconnection/trickle-ice/



☐ Acquire microphone/camera permissions

Time	Type	Foundation	Protocol	Address	Port	Priority	URL (if present)	relayProtocol (if present)
0.002	host	230873026	udp	1bcea6e8-4c18-4e7b-85d6-3d607eb29417.local	61658	126 | 30 | 255		
0.090	srflx	3706170650	udp	1.52.127.90	61658	100 | 30 | 255	stun:stun.l.google.com:19302	
0.125	Done							

Ice candidates

ICE (Interactive Connectivity Establishment) candidates are potential networking endpoints that WebRTC uses to establish a connection between peers. Each candidate represents a possible method for two devices (peers) to communicate, usually in the context of real-time applications like video calls, voice calls, or peer-to-peer data sharing.

If two friends are trying to connect to each other in a hostel wifi , then they c candidates.

I... ...ct to each other, then they would connect via their public IPs.

Turn server

A lot of times, your network doesn't allow media to come in from `browser 2`. This depends on how restrictive your network is

Since the `ice candidate` is discovered by the `stun server`, your network `might` block incoming data from `browser 2` and only allow it from the `stun server`

Offer

The process of the first browser (the one initiating connection) sending their `ice candidates` to the other side.

Answer

The other side returning their `ice candidates` is called the answer.

SDP - Session description protocol

A single file that contains all your

1. ice candidates
2. what media you want to send, what protocols you've used to encode the media

This is the file that is sent in the `offer` and received in the `answer`

Example

```
v=0
o=- 423904492236154649 2 IN IP4 127.0.0.1
s=-


c=IN IP4 192.168.1.101
```

```
a=rtpmap:0 PCMU/8000
```



```
a=candidate:2 1 UDP 2122194687 10.0.1.1 49171 typ host
a=candidate:3 1 UDP 1685987071 93.184.216.34 49172 typ srflx raddr 10.0.1.1 rport 4
a=candidate:4 1 UDP 41819902 10.1.1.1 3478 typ relay raddr 93.184.216.34 rport 491
```

RTCPeerConnection (pc, peer connection)

https://developer.mozilla.org/en-US/docs/Web/API/RTCPeerConnection

Th to

Connecting the two sides

The steps to create a webrtc connection between 2 sides includes -

1. Browser 1 creates an `RTCPeerConnection`
2. Browser 1 creates an `offer`
3. Browser 1 sets the `local description` to the offer
4. Browser 1 sends the offer to the other side through the `signaling server`
5. Browser 2 receives the `offer` from the `signaling server`
6. Browser 2 sets the `remote description` to the `offer`
7. Browser 2 creates an `answer`
8. Browser 2 sets the `local description` to be the `answer`
9. Browser 2 sends the `answer` to the other side through the `signaling server`

1 ts the `remote description`

This is just to `establish` the `p2p` connection b/w the two parties

To actually send media, we have to

1. Ask for camera /mic permissions
2. Get the `audio` and `video` streams
3. Call `addTrack` on the `pc`
4. This would trigger a `onTrack` callback on the other side

Implementation

We will be writing the code in

1. Node.js for the Signaling server. It will be a websocket server that supports 3 types of messages
    1. createOffer
    2. createAnswer
    3. addIceCandidate
2. React + PeerConnectionObject on the frontend

We're actually building a slightly complex version of https://jsfiddle.net/rainzhao/3L9sfsvf/

Backend

- Create an empty TS project, add ws to it

```
npm init -y
npx tsc --init
npm install ws @types/ws
```

- Change rootDir and outDir in tsconfig

```
"rootDir": "./src",
"outDir": "./dist",
```

- Create a simple websocket server

```
import { WebSocketServer } from 'ws';

t: 8080 });
```

```
let senderSocket: null | WebSocket = null;
```



```
wss.on('connection', function connection(ws) {
  ws.on('error', console.error);

  ws.on('message', function message(data: any) {
    const message = JSON.parse(data);

  });

  ws.send('something');
});
```

- Try running the server

```
tsc -b
node dist/index.js
```

- Add message handlers

```
import { WebSocket, WebSocketServer } from 'ws';

const wss = new WebSocketServer({ port: 8080 });

let senderSocket: null | WebSocket = null;
let receiverSocket: null | WebSocket = null;

wss.on('connection', function connection(ws) {
  ws.on('error', console.error);

  ws.on('message', function message(data: any) {
    const message = JSON.parse(data);
    if (message.type === 'sender') {
      senderSocket = ws;
    } else if (message.type === 'receiver') {
      receiverSocket = ws;
    } else if (message.type === 'createOffer') {
      if (ws !== senderSocket) {
        return;
      }
                                              type: 'createOffer', sdp: message.sdp
    } else if (message.type === 'createAnswer') {
```

```
        if (ws !== receiverSocket) {
```



```
        senderSocket?.send(JSON.stringify({ type: 'createAnswer', sdp: message.sd
      } else if (message.type === 'iceCandidate') {
        if (ws === senderSocket) {
          receiverSocket?.send(JSON.stringify({ type: 'iceCandidate', candidate: mes
        } else if (ws === receiverSocket) {
          senderSocket?.send(JSON.stringify({ type: 'iceCandidate', candidate: mess
        }
      }
    });
  });
```

That is all that you need for a simple one way communication b/w two tabs

To have both the sides be able to send each other media, and support multiple rooms, see https://github.com/hkirat/omegle/

Frontend

- Create a frontend repo

```
npm create vite@latest
```

- Add two routes, one for a `sender` and one for a `receiver`

```
import { Route, BrowserRouter, Routes } from 'react-router-dom'
```

```
import { Sender } from './components/Sender'
```



```
function App() {
  return (
    <BrowserRouter>
      <Routes>
        <Route path="/sender" element={<Sender />} />
        <Route path="/receiver" element={<Receiver />} />
      </Routes>
    </BrowserRouter>
  )
}

export default App
```

- Remove strict mode in `main.tsx` to get rid of a bunch of webrtc connections locally (not really needed)
- Create components for sender

```
import { useEffect, useState } from "react"

export const Sender = () => {
    const [socket, setSocket] = useState<WebSocket | null>(null);
    const [pc, setPC] = useState<RTCPeerConnection | null>(null);

    useEffect(() => {
        const socket = new WebSocket('ws://localhost:8080');
        setSocket(socket);
        socket.onopen = () => {
            socket.send(JSON.stringify({
                type: 'sender'
            }));
        }
    }, []);

    const initiateConn = async () => {

        if (!socket) {
            alert("Socket not found");
```

```
      if (message.type === 'createAnswer') {
        await pc.setRemoteDescription(message.sdp);
      } else if (message.type === 'iceCandidate') {
        pc.addIceCandidate(message.candidate);
      }
    }

    const pc = new RTCPeerConnection();
    setPC(pc);
    pc.onicecandidate = (event) => {
      if (event.candidate) {
        socket?.send(JSON.stringify({
          type: 'iceCandidate',
          candidate: event.candidate
        }));
      }
    }

    pc.onnegotiationneeded = async () => {
      const offer = await pc.createOffer();
      await pc.setLocalDescription(offer);
      socket?.send(JSON.stringify({
        type: 'createOffer',
        sdp: pc.localDescription
      }));
    }

    getCameraStreamAndSend(pc);
  }

  const getCameraStreamAndSend = (pc: RTCPeerConnection) => {
    navigator.mediaDevices.getUserMedia({ video: true }).then((stream) => {
      const video = document.createElement('video');
      video.srcObject = stream;
      video.play();
      // this is wrong, should propogate via a component
      document.body.appendChild(video);
      stream.getTracks().forEach((track) => {
```

```
    });
  }
```



```
  return <div>
    Sender
    <button onClick={initiateConn}> Send data </button>
  </div>
}
```

- Create the component for receiver

```
import { useEffect } from "react"


export const Receiver = () => {

  useEffect(() => {
    const socket = new WebSocket('ws://localhost:8080');
    socket.onopen = () => {
      socket.send(JSON.stringify({
        type: 'receiver'
      }));
    }
    startReceiving(socket);
  }, []);

  function startReceiving(socket: WebSocket) {
    const video = document.createElement('video');
    document.body.appendChild(video);

    const pc = new RTCPeerConnection();
    pc.ontrack = (event) => {
      video.srcObject = new MediaStream([event.track]);
      video.play();
    }

    socket.onmessage = (event) => {
      const message = JSON.parse(event.data);
      if (message.type === 'createOffer') {
        pc.setRemoteDescription(message.sdp).then(() => {
                                              wer) => {
                                              wer);
```

```
                socket.send(JSON.stringify({
```



```
                }));
              });
            });
          } else if (message.type === 'iceCandidate') {
            pc.addIceCandidate(message.candidate);
          }
        }
      }

      return <div>

      </div>
    }
```

Final code - https://github.com/100xdevs-cohort-2/week-23-webrtc

Assignment

Can you change the code so that

1. A single producer can produce to multiple people?
2. Add room logic.
3. Add two way communication.
4. Replace p2p logic with an SFU (mediasoup)

Webrtc stats

You can look at a bunch of stats/sdps in `about:webrtc-internals`

A lot of times you ask users to dump stats from here for better debugging

Using libraries for p2p

As you can see, there is a lot of things we had to know to be able to build a simple app that sends video from one side to another

There are libraries that hide a lot of this complexity (specifically the complexity of the `RTCPeerConnectionObject` from you).

https://peerjs.com/

There are two other popular architectures for doing WebRTC

2. MCU

Problems with p2p

Doesn't scale well beyond 3-4 people in the same call

SFU

SFU stands for `Selective forwarding unit` . This acts as a central media server which `forwards` packets b/w users

Popular Open source SFUs -

1. https://github.com/versatica/mediasoup
2. https://github.com/pion/webrtc (not exactly an SFU but you can build one on top of it)

MCU

It mixes audio/video together on the server before forwarding it.

This means it needs to

1. decode video/audio (using something like ffmpeg)
2. Mix them (create a video canvas/create a single audio stream)
3. Send out the merged audio stream to everyone