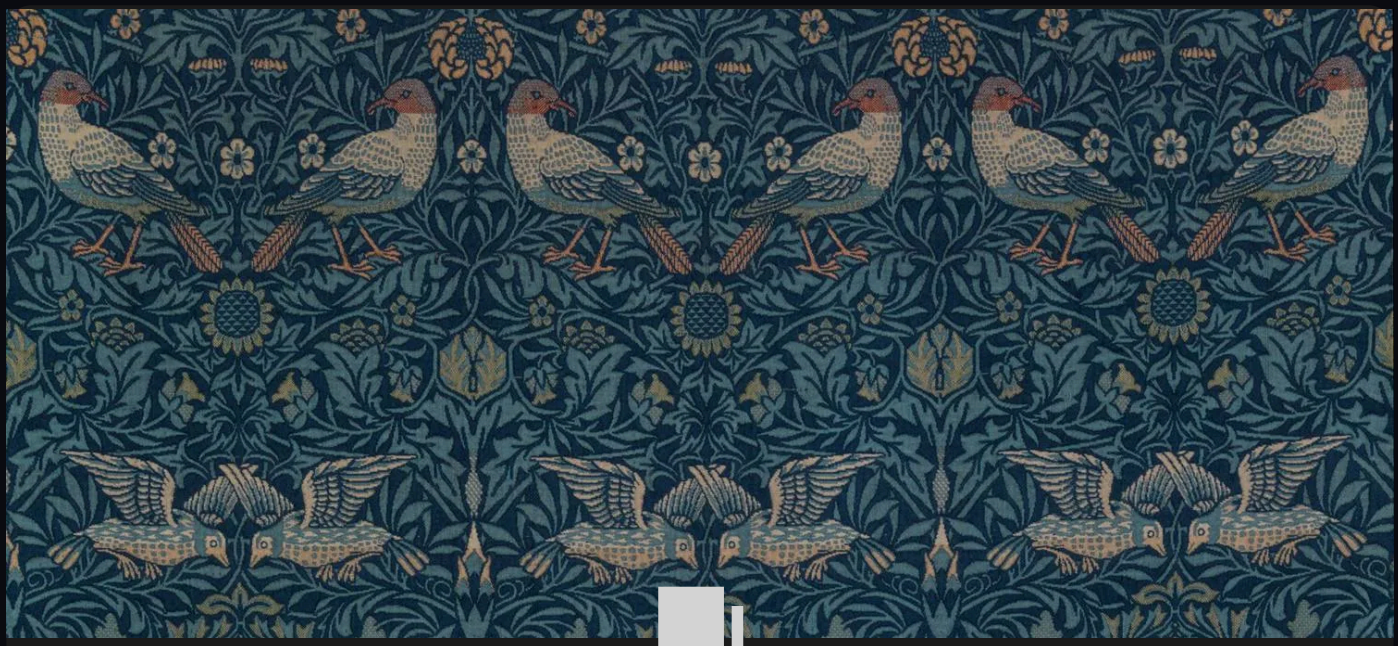
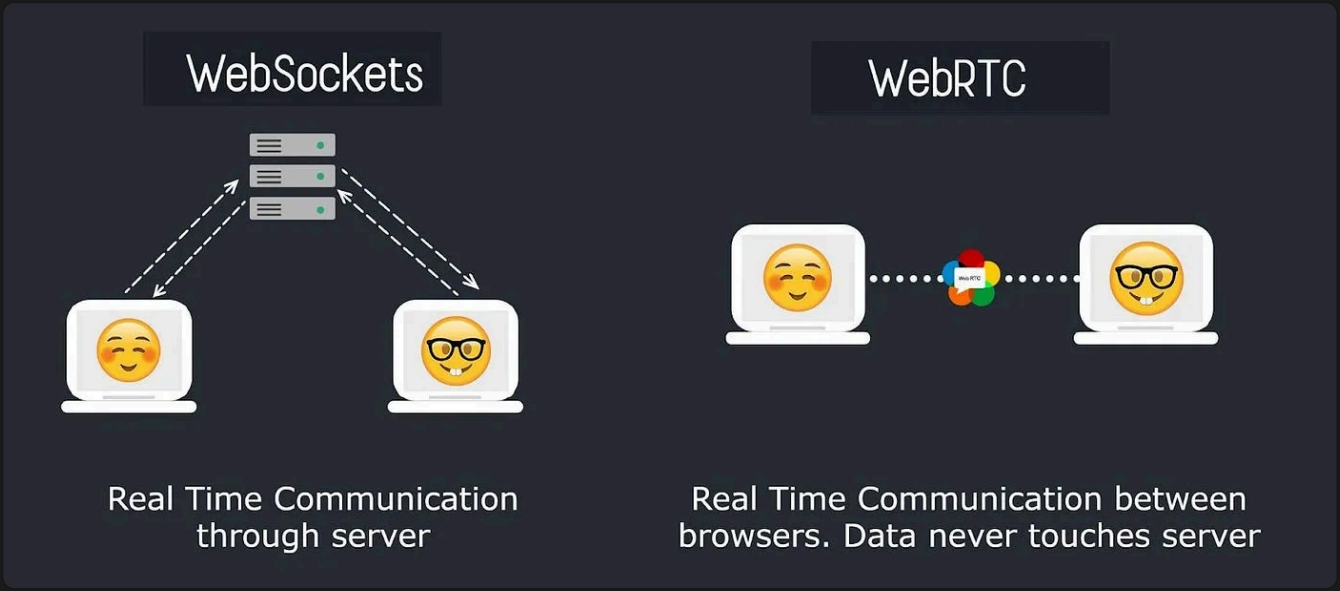


Week 23.1

In this lecture, Harkirat explains `WebRTC` (Web Real-Time Communication), a powerful technology that enables real-time communication between web browsers and devices. He introduces the key jargons involved, such as `peer-to-peer` (P2P) communication, `signaling servers`, `STUN servers`, `TURN servers`, `offers and answers`, `RTCPeerConnection`, `WebRTC Stats` and finally other architectures like `SFU` and `MCU`.

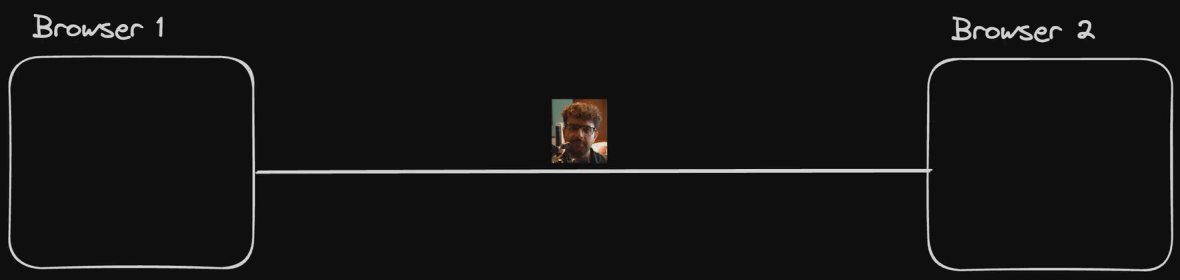
WebRTC (Web Real-Time Communication) is a powerful technology that enables real-time multimedia communication directly within web browsers, without the need for third-party plugins or software. Here are the key reasons why WebRTC is a compelling choice:



1. **Browser-native real-time communication**: WebRTC is the core and only protocol that allows real-time media communication from inside a web browser. It eliminates the need for proprietary plugins or applications, making it a seamless and accessible solution for web-based communication.
2. **Proven implementation**: The provided links demonstrate a successful implementation of WebRTC for a live streaming application, showcasing its practical usage and capabilities.
3. **Sub-second latency**: WebRTC is designed to deliver real-time communication with sub-second latency, making it ideal for applications that require near-instantaneous data transfer. This low-latency characteristic is crucial for various use cases, including:
    - **Video conferencing**: Applications like Zoom and Google Meet leverage WebRTC for multi-party video calls, enabling seamless real-time communication among multiple participants.
    - **One-to-one communication**: Platforms like Omegle and online teaching applications utilize WebRTC for one-to-one video and audio calls, facilitating real-time interactions between two parties.
    - **Real-time gaming**: WebRTC's ability to transmit data in addition to audio and video makes it suitable for real-time gaming applications that require low latency and high frame rates (e.g., 30 FPS).
4. **Peer-to-peer communication**: WebRTC enables direct peer-to-peer communication between browsers, reducing the need for intermediary servers and improving efficiency and scalability.
5. **Open standard and cross-platform compatibility**: WebRTC is an open standard supported by major web browsers and operating systems, ensuring cross-platform compatibility and widespread adoption.

WebRTC Jargons

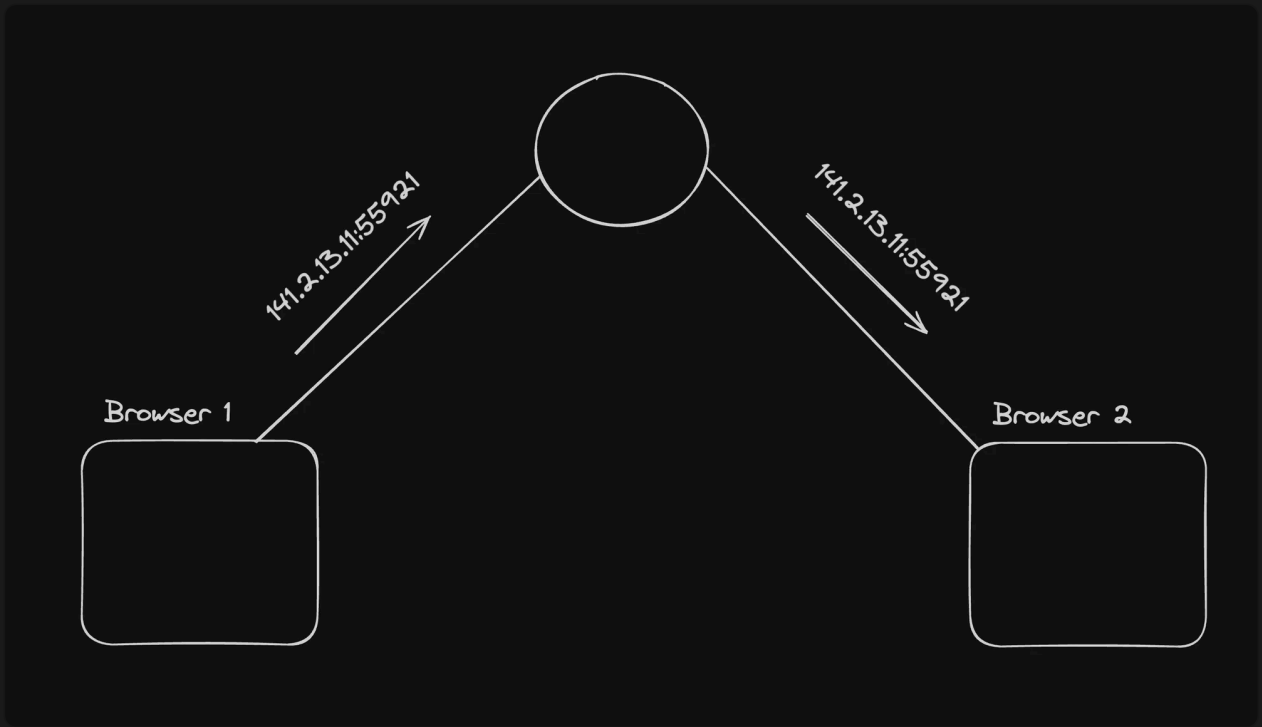
WebRTC (Web Real-Time Communication) enables real-time multimedia communication directly within web browsers through a peer-to-peer (P2P) protocol.



> This image depicts two browsers directly connected to each other in a peer-to-peer (P2P) fashion. In a WebRTC P2P connection, media data (audio, video, etc.) is transmitted directly between the two peers without going through an intermediary server. This direct connection enhances efficiency and reduces latency.

While media data is transmitted directly between peers, WebRTC requires certain servers for specific tasks like signaling, NAT traversal, and media handling.

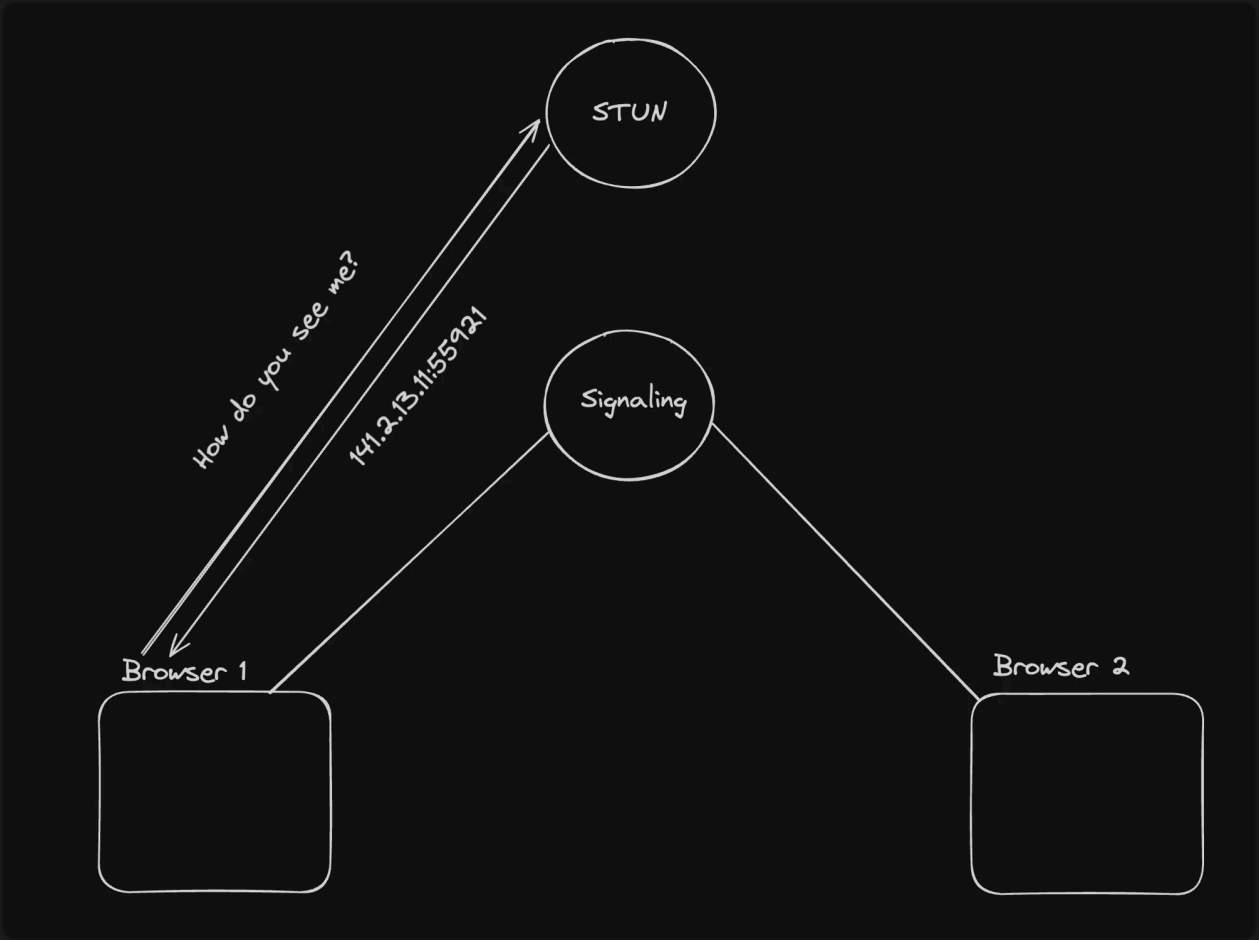
1. **Signaling Server**:
    - Facilitates the initial handshake between WebRTC peers by allowing them to exchange network addresses (IPs) and other metadata required for establishing a direct P2P connection.
    - Typically implemented using WebSocket or HTTP servers.



> This image depicts two browsers communicating through a STUN (Session Traversal Utilities for NAT) server and a signaling server. The STUN server helps determine the

routers.

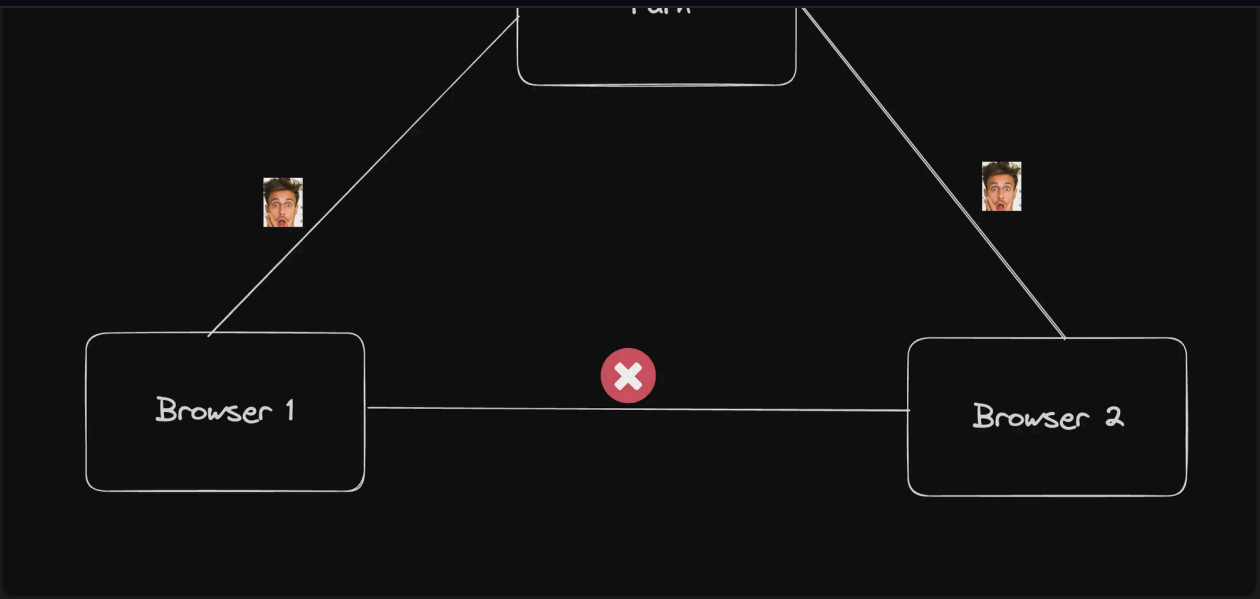
1. STUN (Session Traversal Utilities for NAT) Server:
    - Helps discover the public IP addresses of peers behind NAT devices, enabling direct P2P connections.
    - Shows how the world sees the peers' IP addresses.



> The image illustrates two browsers connected through an intermediary TURN (Traversal Using Relays around NAT) server. When a direct P2P connection cannot be established due to restrictive network conditions or firewalls, the TURN server acts as a relay, forwarding the media data between the two peers. This ensures connectivity even in challenging network environments.

1. ICE (Interactive Connectivity Establishment) Candidates:
    - Potential networking endpoints that WebRTC uses to establish a connection between peers.
    - Represent methods for peers to communicate, such as using private router IPs (on the same network) or public IPs (different locations).

1. TURN (Traversal Using Relays around NAT) Server:
    - Acts as a relay server, forwarding media data between peers when a direct P2P connection cannot be established due to restrictive network conditions or firewalls.



> The image illustrates two browsers connected through an intermediary TURN (Traversal Using Relays around NAT) server. When a direct P2P connection cannot be established due to restrictive network conditions or firewalls, the TURN server acts as a relay, forwarding the media data between the two peers. This ensures connectivity even in challenging network environments.

2. **Offer and Answer:**

- The initiating peer sends an "offer" containing its ICE candidates and session details (encoded in the Session Description Protocol (SDP) format).
- The receiving peer responds with an "answer" containing its own ICE candidates and session details.

Example

```
v=0
o=- 4239044922361546 49 2 IN IP4 127.0.0.1
s=-
t=0 0
m=audio 49170 RTP/AVP 0
c=IN IP4 192.168.1.101
a=rtpmap:0 PCMU/8000
a=ice-options:trickle
a=candidate:1 1 UDP 2122260223 192.168.1.101 49170 typ host
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 49171
a=candidate:4 1 UDP 41819902 10.1.1.1 3478 typ relay raddr 93.184.216.34 rport 49172Copy
```

3. **RTCPeerConnection:**

- A class provided by the browser that gives developers access to the SDP, allows them to create offers and answers, and enables sending and receiving media.

> In summary, while WebRTC enables direct P2P communication between browsers, it requires various servers for tasks like signaling, NAT traversal, and media handling. These servers facilitate the initial connection establishment, address discovery, and relay media data when necessary, ensuring reliable and stable real-time communication over the internet.

Connecting the Two Sides

Establishing a WebRTC connection between two peers involves a series of steps facilitated by the signaling server and the RTCPeerConnection API provided by the browser. Here's a breakdown of the process:

1. **Browser 1 creates an RTCPeerConnection**
    - Browser 1 creates an instance of the RTCPeerConnection object, which represents the WebRTC connection.
2. **Browser 1 creates an offer**
    - Browser 1 generates an "offer" containing its ICE candidates and session details (encoded in the Session Description Protocol (SDP) format).
3. **Browser 1 sets the local description to the offer**
    - Browser 1 sets the local description of the RTCPeerConnection to the generated offer.
4. **Browser 1 sends the offer to the other side through the signaling server**
    - Browser 1 sends the offer to Browser 2 via the signaling server.
5. **Browser 2 receives the offer from the signaling server**
    - Browser 2 receives the offer from Browser 1 through the signaling server.
6. **Browser 2 sets the remote description to the offer**
    - Browser 2 sets the remote description of its RTCPeerConnection to the received offer.
7. **Browser 2 creates an answer**
    - Browser 2 generates an "answer" containing its own ICE candidates and session details.
8. **Browser 2 sets the local description to be the answer**
    - Browser 2 sets the local description of its RTCPeerConnection to the generated answer.
9. **Browser 2 sends the answer to the other side through the signaling server**
    - Browser 2 sends the answer to Browser 1 via the signaling server.
10. **Browser 1 receives the answer and sets the remote description**
    - Browser 1 receives the answer from Browser 2 and sets the remote description of its RTCPeerConnection to the received answer.

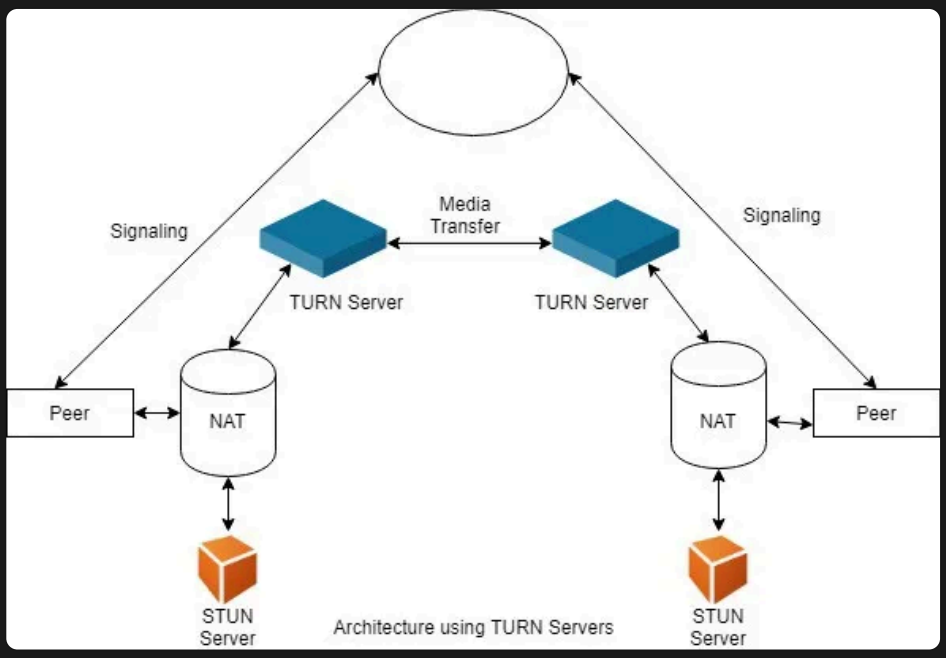
At this point, the WebRTC connection is established between the two peers, and they can start exchanging media data directly in a peer-to-peer fashion.

To actually send and receive media (audio and video), additional steps are required:

1. **Ask for camera/microphone permissions**

- Once permissions are granted, each browser can obtain the local audio and video streams using the getUserMedia() API.

3. **Call addTrack on the RTCPeerConnection**
    - Each browser adds the local audio and video tracks to the RTCPeerConnection using the addTrack() method.
4. **Trigger onTrack callback on the other side**
    - When a track is added to the RTCPeerConnection, it triggers the onTrack event on the remote peer, allowing it to receive and render the incoming media streams.



Implementation

To implement a WebRTC application, we'll be using a combination of Node.js for the signaling server and React with the RTCPeerConnection API on the frontend. The implementation will be a slightly more complex version of the provided JSFiddle example (https://jsfiddle.net/rainzhao/3L9sfsvf/).

Signaling Server (Node.js)

The signaling server will be built using Node.js and will act as a WebSocket server to facilitate the exchange of signaling messages between the peers. It will support three types of messages:

1. **createOffer**: This message will be sent by the initiating peer (Browser 1) to the signaling server, which will then forward it to the receiving peer (Browser 2).
2. **createAnswer**: This message will be sent by the receiving peer (Browser 2) to the signaling server, which will then forward it to the initiating peer (Browser 1).
3. **addIceCandidate**: This message will be sent by both peers to the signaling server, which will then forward it to the other peer. It contains the ICE candidates necessary for establishing the WebRTC connection.

The signaling server will be responsible for relaying these messages between the peers, enabling them to exchange the necessary information for establishing the WebRTC connection.

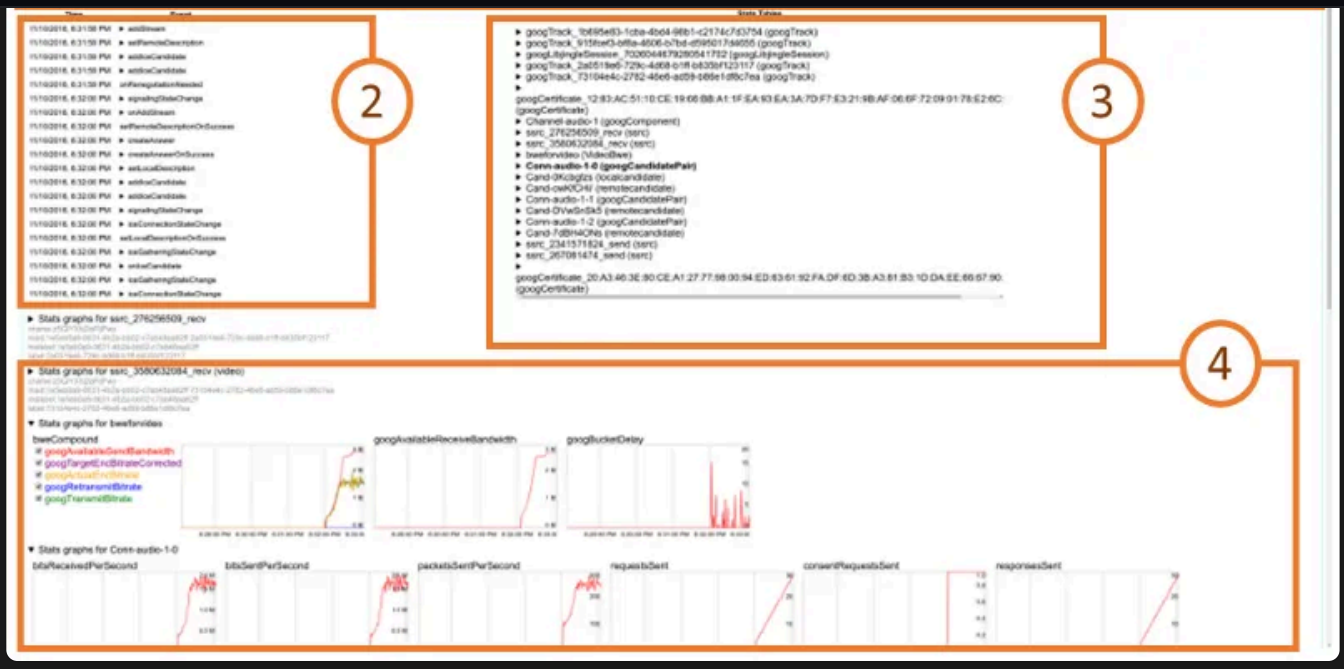
browser to handle the WebRTC functionality.

1. **Establishing the WebRTC Connection:**
    - The initiating peer (Browser 1) will create an instance of the RTCPeerConnection object and generate an offer using the createOffer() method.
    - The offer will be sent to the signaling server using the appropriate WebSocket message (createOffer).
    - The receiving peer (Browser 2) will receive the offer from the signaling server and set it as the remote description of its RTCPeerConnection instance.
    - Browser 2 will then generate an answer using the createAnswer() method and send it back to Browser 1 via the signaling server (createAnswer message).
    - Both peers will set the received answer/offer as their remote/local descriptions, respectively, to complete the connection establishment process.
2. **Exchanging ICE Candidates:**
    - As the peers gather ICE candidates (potential networking endpoints), they will send them to the signaling server using the addIceCandidate message.
    - The signaling server will forward these ICE candidates to the other peer, allowing them to establish the most efficient direct connection.
3. **Sending and Receiving Media:**
    - Once the WebRTC connection is established, both peers can obtain the local audio and video streams using the getUserMedia() API.
    - The local streams will be added to the RTCPeerConnection instances using the addTrack() method.
    - When a track is added to the RTCPeerConnection, it will trigger the onTrack event on the remote peer, allowing it to receive and render the incoming media streams.
4. **User Interface:**
    - The React components will handle the user interface, displaying video elements for rendering the local and remote streams.
    - Appropriate UI elements (buttons, inputs, etc.) will be provided to initiate the WebRTC connection, handle permissions, and control the media streams.

> This implementation will allow users to establish real-time audio and video communication directly within their browsers, leveraging the power of WebRTC and the signaling server for efficient peer-to-peer communication.

WebRTC Stats

WebRTC provides a way to access various statistics and diagnostic information related to the peer connection, which can be extremely useful for debugging and optimizing WebRTC applications. This information can be accessed through the `about:webrtc-internals` page in web browsers or through dedicated tools like the WebRTC Statistics Tool in Unity.



Here are the key points about WebRTC stats:

1. **Accessing WebRTC Stats in Browsers:**
    - In web browsers, you can access WebRTC stats by navigating to `about:webrtc-internals`.[1]
    - This page provides a detailed view of the active WebRTC connections, including information about the local and remote media streams, ICE candidates, and various statistics related to the connection quality and performance.
2. **Debugging with WebRTC Stats:**
    - When users encounter issues with WebRTC applications, developers often ask them to provide a dump of the WebRTC stats from `about:webrtc-internals`.[5]
    - This dump contains valuable diagnostic information, such as the Session Description Protocol (SDP) details, ICE candidates, codec information, and various metrics related to the media streams.
    - Analyzing these stats can help developers identify and troubleshoot issues related to connectivity, media quality, or performance.
3. **WebRTC Statistics Tool in Unity:**
    - Unity provides a dedicated WebRTC Statistics Tool for developers working with the Unity WebRTC package.[5]
    - This tool displays statistics about the WebRTC connections in the Unity editor, including information about peer connections, candidate pairs, and various metrics.
    - Developers can save the collected statistics as dump files, which are compatible with the dump files generated by Chrome's `about:webrtc-internals`, allowing for further analysis and debugging using third-party applications.
4. **Relevant Statistics:**
    - Some of the relevant statistics provided by WebRTC include:
        - ICE candidate information (e.g., candidate types, IP addresses, ports)
        - Media stream statistics (e.g., bitrate, frame rate, packet loss, jitter)

Using Libraries for P2P

While WebRTC provides a powerful framework for real-time peer-to-peer communication, implementing it from scratch can be complex and involve a significant amount of boilerplate code. This is where third-party libraries come into play, abstracting away much of the underlying complexity and providing a more developer-friendly API.One such library is PeerJS

(https://peerjs.com/), which simplifies the process of establishing WebRTC connections and enables developers to focus more on the application logic rather than the low-level details of WebRTC.

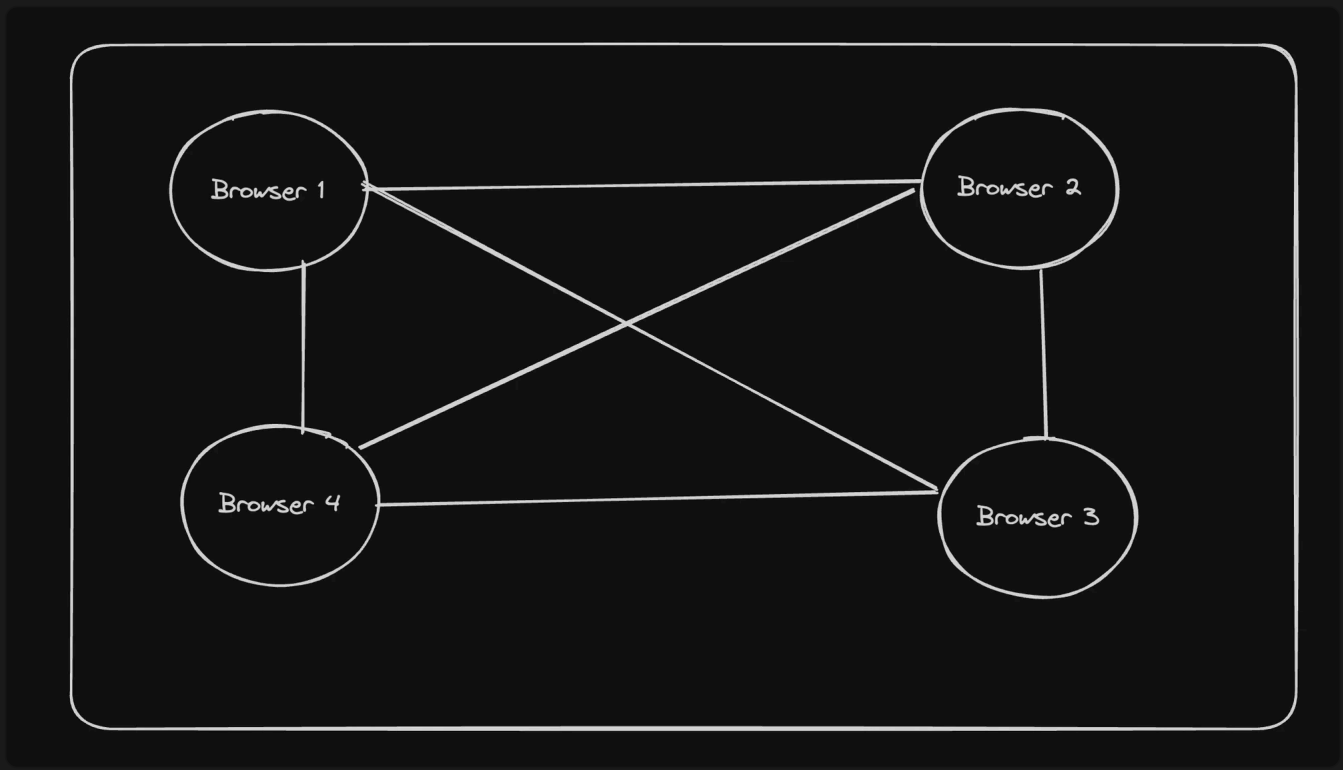
Other architectures

There are two other popular architectures for doing WebRTC

1. SFU
2. MCU

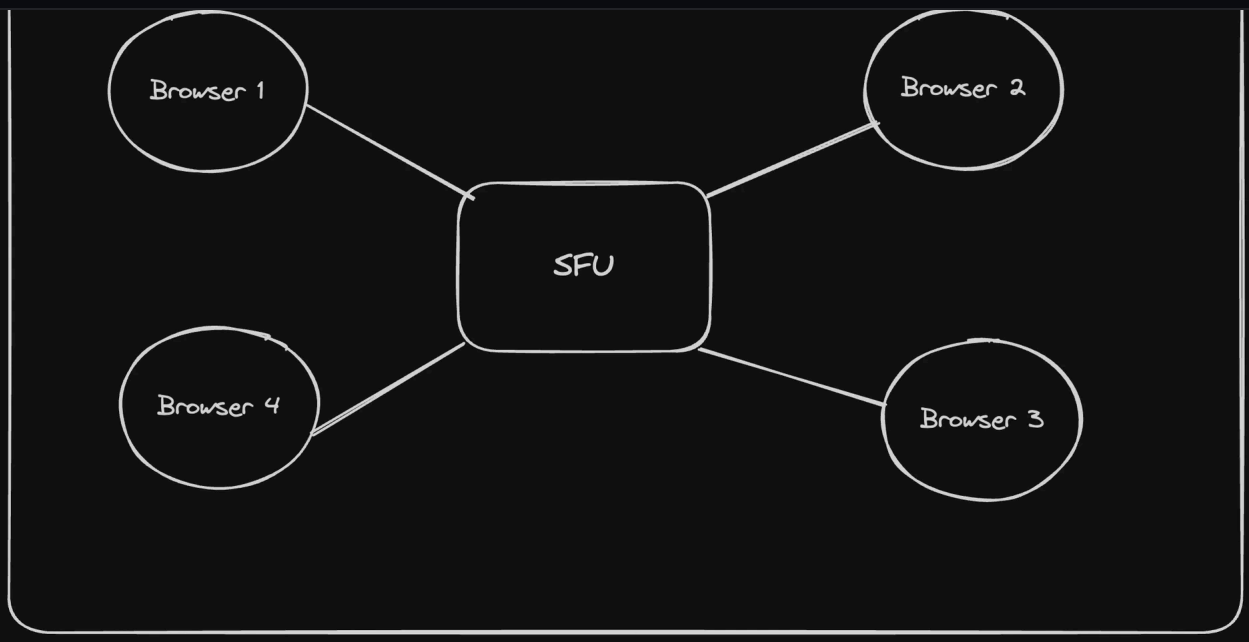
Problems with p2p

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

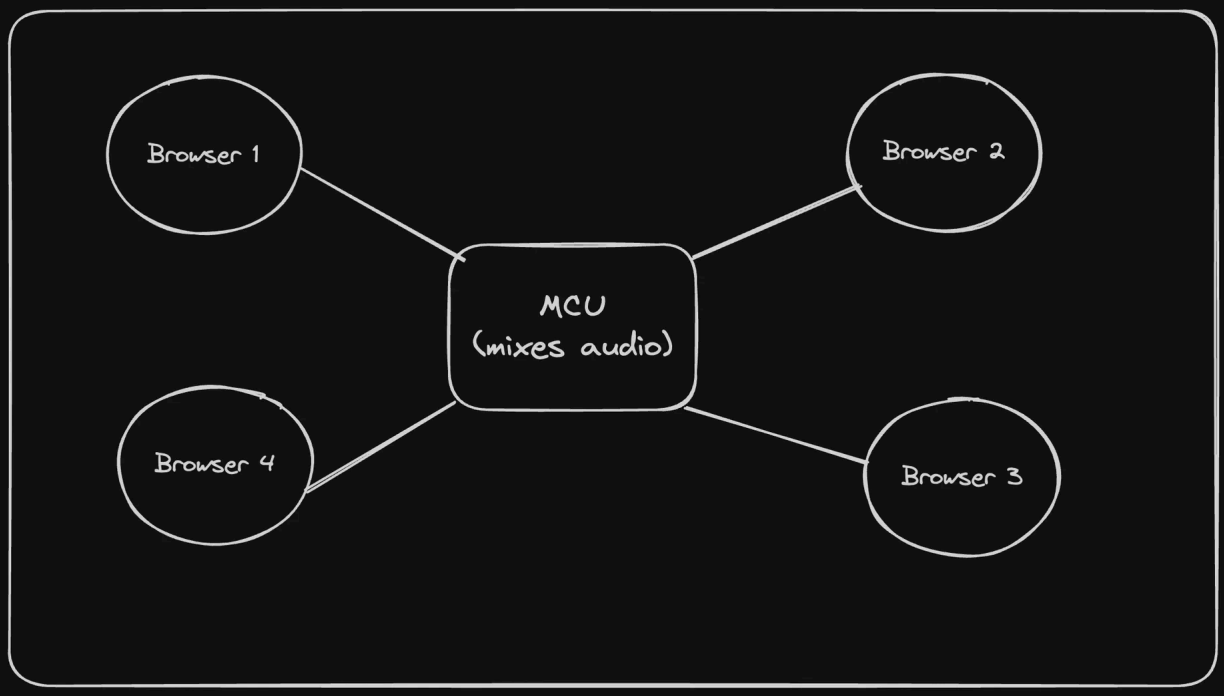


Popular Open source SFUs -

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.



2

This means it needs to

Most upvotes All comments

1. decode video/audio (using something like ffmpeg)

Comment

s **shashank sharma** 16 days ago

Hi

1 0 0 Replies

s **shashank sharma** 16 days ago

Hey Harkirat Sir, My Self Shashank Sharma currently i am in Lucknow, doing nothing did 2+ years job in IT now was tr ying to Upskill my self, I have completed cohort 0-1 thought it was very basic except docker now wanted to learn som ething so could you please tell me is Super 30 i can still enroll and you will be the one who is going to be there for us

0 0 0 Replies