WEBRTC에 대한 연구 기록

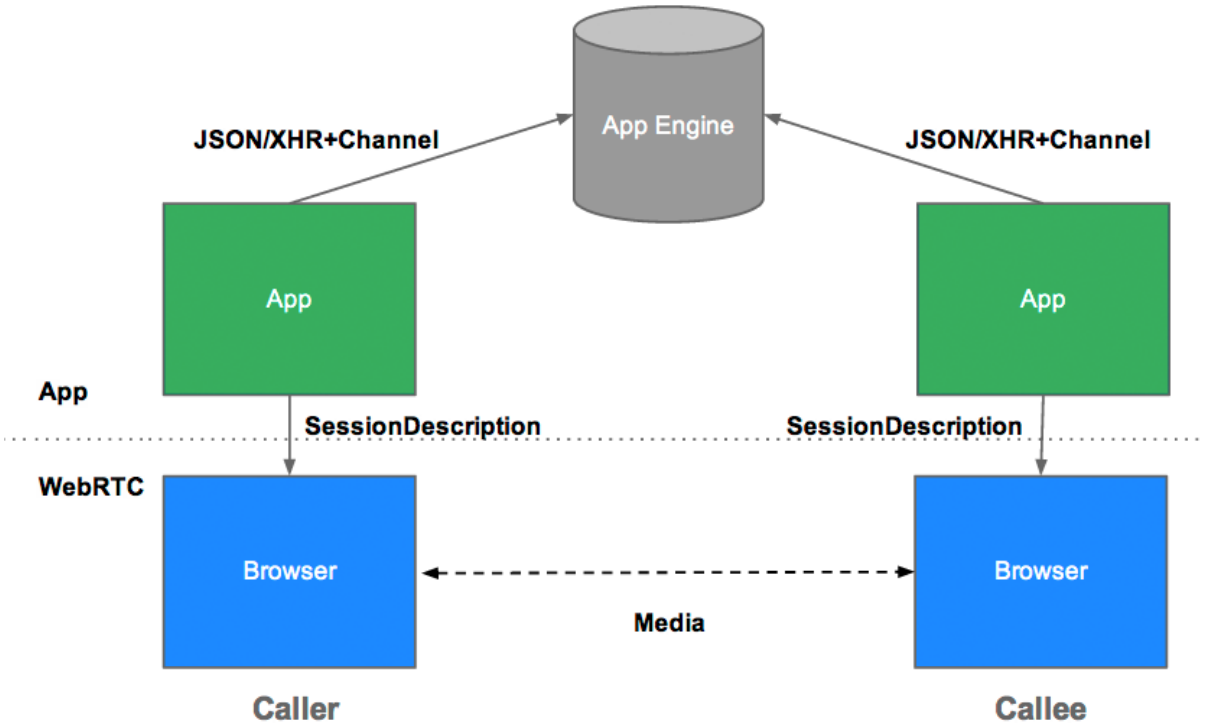
경북대학교 컴퓨터학부 2016116109

윤경록

목차

WEBRTC란 무엇인가?

WEBRTC(Web Real-Time Communication)란, 웹 어플리케이션 및 사이트들이 별도의 소프트웨어 없이 음성, 영상 미디어 혹은 텍스트, 파일 같은 데이터를 브라우저끼리 주고 받을 수 있게 만든 기술입니다. WebRTC로 구성된 프로그램들은 별도의 플로그인이나 소프트웨어 없이 P2P 화상회의 및 데이터 공유가 가능합니다.



가장 기본적인 상업적인 의미로는 Peer와 Peer의 연결에 있어서 거의 대부분을 인터넷 브라우저 자체에서 지원한다는 것입니다. 거의 대부분의 함수 및 기술들을 크롬, Edge, 파이어폭스 등의 브라우저에서 지원하며 심지어는 모바일 브라우저에서도 대부분을 지원합니다. 데이터의 교환을 위한 사전 데이터들은 거의 대부분 JSON 형식으로 이루어지며 이는 SDP, Offer, Answer등등이 있습니다.

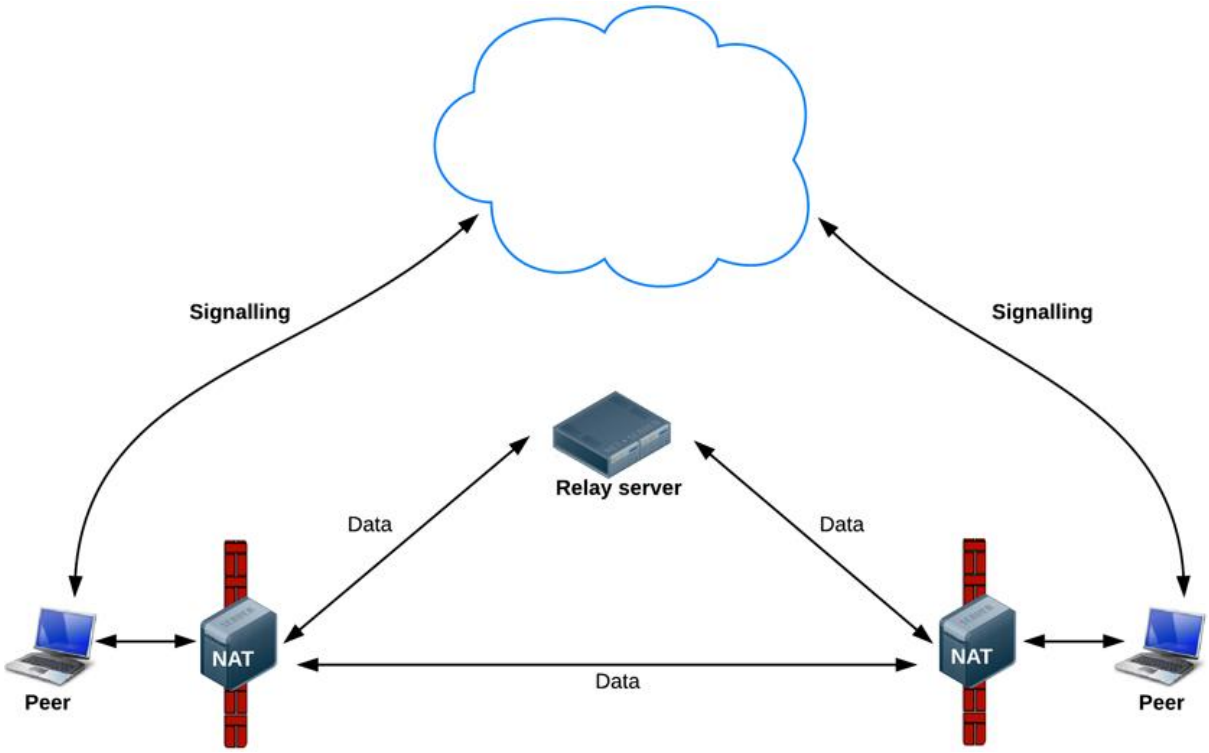
각각의 클라이언트들은 브라우저를 통해 약속된 문서들을 공유하며 그것을 통해 P2P 컨넥션을 만듭니다. 여기에서는 간단히 데이터를 공유할 수도 있으나 가장 중요한 비디오나 음성은 Track으로 관리되며, Track의 확립을 위해서는 SDP공유가 아주 중요합니다. WEBRTC에서는 주요하게 세가지의 클래스를 사용합니다. 이들은 아래와 같습니다.

MediaStream – 카메라와 마이크 등의 데이터 스트림에 접근

RTCPeerConnection – 암호화 및 대역폭 관리 및 오디오, 비디오의 연결

RTCDataChannel – 일반적인 데이터의 P2P통신

이 세가지를 통해 데이터를 교환하는데 이를 위해 각각 클라이언트의 대략적인 위치와 정보를 교환하기 위해 시그널링(Signaling)이라는 과정을 거치게 됩니다. 그것을 위해 서버가 필요하며 가장 기본적인 Signaling Server가 따로 필요합니다. 간략한 모사도는 아래와 같습니다.



WEBRTC의 가장 큰 사용 이유는 NAT의 방어에 어느정도 우회를 가능하게 한다는 것입니다. NAT란 단말에 공개 IP주소를 할당하기 위해 사용되는 것으로, 단말의 비공개 주소와 라우터의 공개 주소와 유일한 포트 연결을 연결해주는 역할을 합니다. 이러한 연결을 위해 STUN 서버가 필요합니다. STUN 서버를 통해 각각의 Peer는 서로의 위치를 찾습니다.

STUN(Session Traversal Utiliteis for NAT)은 공개적 주소를 발견하거나 Peer간의 직접 연결을 막는 등의 라우터의 제한을 결정하는 프로토콜입니다. 클라이언트는 STUN 프로토콜을 사용하여 서로의 위치를 찾습니다.

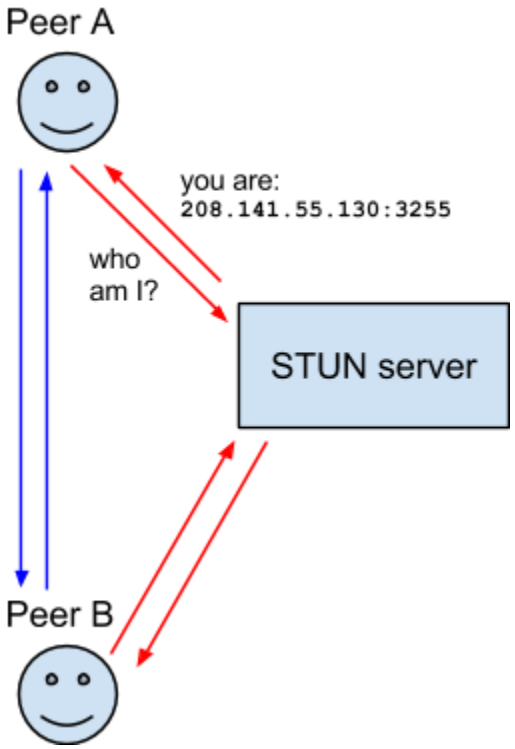


Figure 1 STUN

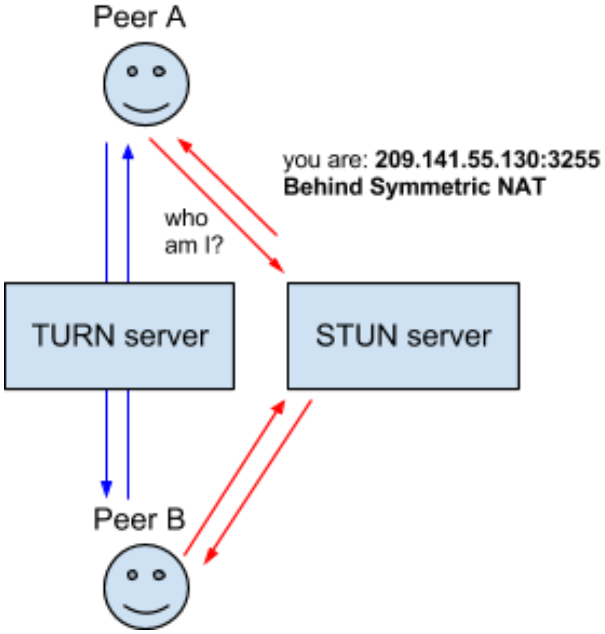


Figure 2 TURN

스턴서버로 연결된 Peer와 Peer들은 어느 정도의 의존성을 띕니다. 이는 만약 Peer가 다른 Peer와 연결을 끊고 새로운 연결을 다른 Peer와 하려할 때, 어느 정도의 제약이 생깁니다. 또, 특정 라우터들은 네트워크 연결을 하기 위한 제한을 가지고 있습니다. 이는 Symmetric NAT라고 하는데, Peer들이 오직 이전에 연결한 적이 있는 연결들만 허용한다는 것입니다. 이 때문에 STUN서버 뿐만이 아니라 TURN 서버가 필요합니다.

TURN(Traversal Using Relays around NAT) 은 TURN 서버와 연결하고 모든 정보를 그 서버에 전달하는 것으로 Symmetric NAT 제한을 우회하는 것을 의미합니다. 이를 위해 TURN 서버와 연결을 한 후 모든 peer들에게 저 서버에 모든 패킷을 보내고 다시 나에게 전달해달라고 해야 합니다. 이것은 명백히 오버헤드가 발생하므로 이 방법은 다른 대안이 없을 경우만 사용하게 됩니다. 하지만 STUN서버로 연결했을 시 약 10건 중 6번이 연결되지 않았습니다. 물론 연결을 확실하게 끊고 다시 연결을 하는 프로세스를 좀더 추가한다면 이 확률이 낮아지겠지만, 통상적인 연결에서는 STUN 서버 단독만 사용시에 부정확한 연결을 보여줬습니다. 이를 위해 TURN서버 연결이 필 수 적이라는 결론에 도달하게 되었습니다. TURN서버로 시그널링을 했을 시에는 10번중 8번의 연결이 성공하였습니다. (성공적인 연결의 척도는 성공적인 비디오 공유로 하였습니다)

또다른 WebRTC의 주요 특징으로 동영상 공유 스트리밍을 위해 SDP를 사용한다는 것입니다. SDP 즉 **세션 기술 프로토콜(Session Description Protocol, SDP)**은 스트리밍 미디어의 초기화 인수를 기술하기 위한 포맷입니다. 본 규격은 IETF(국제 인터넷 표준화 기구)의 RFC 4566로 규격이 공식화 되어 있습니다. 실제로 WebRTC를 이용하여서 SDP format에 맞춰져 음성, 영상 데이터를 교환하고 있습니다. 해상도나 형식, 코덱, 암호화 등의 멀티미디어 컨텐츠의 연결을 설명하기 위한 표준입니다. 올바른 SDP 공유를 위해서는 Offer를 제공하기 전에 미디어 트랙을 추가해주어야 올바른 SDP 생성이 가능합니다.

또, Peer간에 Peer의 특징을 확인하고 제대로 된 연결을 하기 위해 ICE라는 것을 주고 받습니다. ICE즉 **Interactive Connectivity Establishment (ICE)**는 브라우저가 Peer를 통한 연결이 가능하도록 하게 해 주는 프레임 워크입니다. Peer A -> Peer B까지 단순하게 연결하는 것으로는 작동하지 않는 것에 대한 이유는 많이 있습니다. 연결을 시도하는 방화벽을 통과 해야 하기도 하고, 단말에 Public IP가 없다면 유일한 주소 값을 할당해야 할 필요도 있으며 라우터가 Peer 간의 직접 연결을 허용하지 않을 때에는 데이터를 릴레이 해야 할 경우도 있습니다. ICE는 이러한 작업을 수행하기 위해서 설명하였던 STUN, TURN Server 두개 다 혹은 하나의 서버를 사용합니다. 좀더 확실한 연결을 위해서는 두개다 사용하는 것을 권장합니다.

WEBRTC의 특성

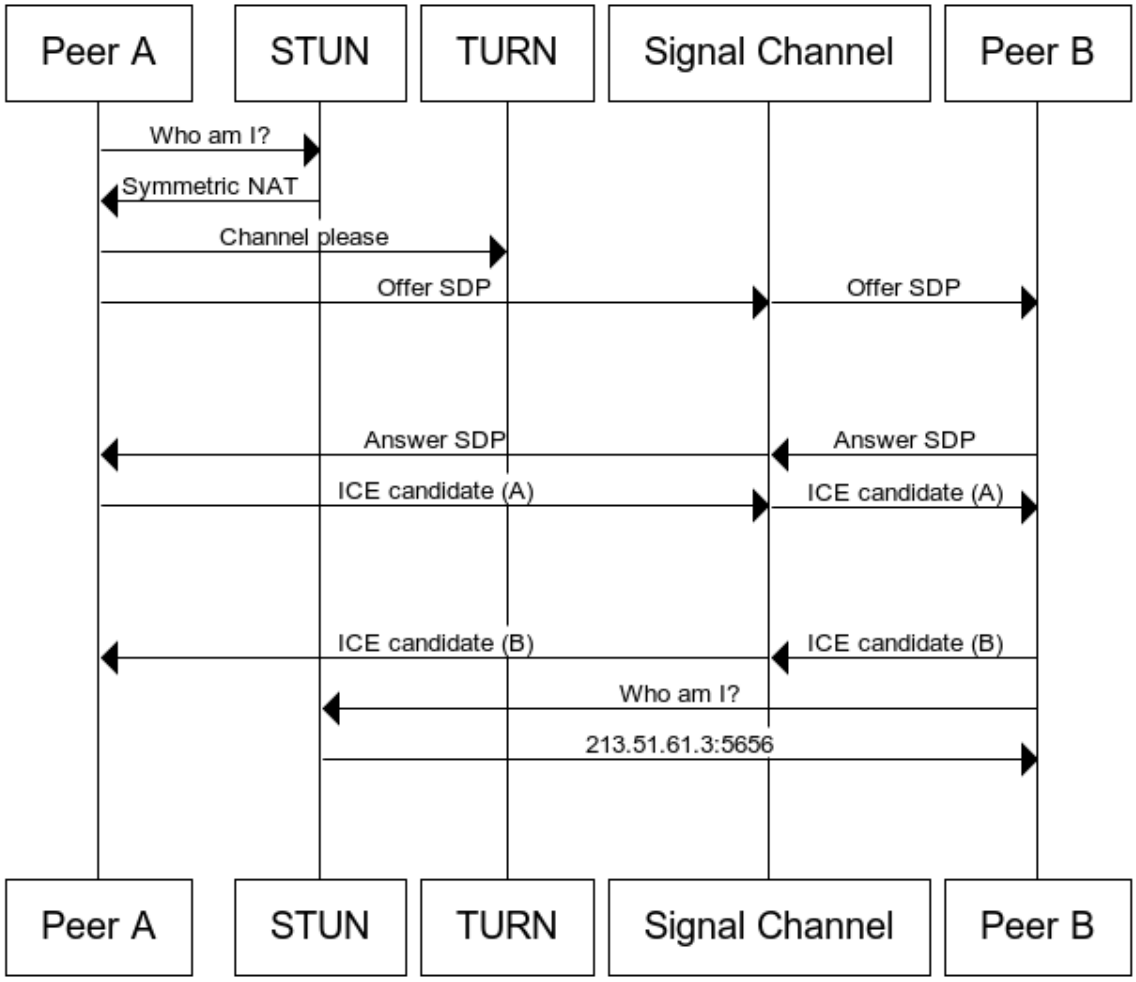
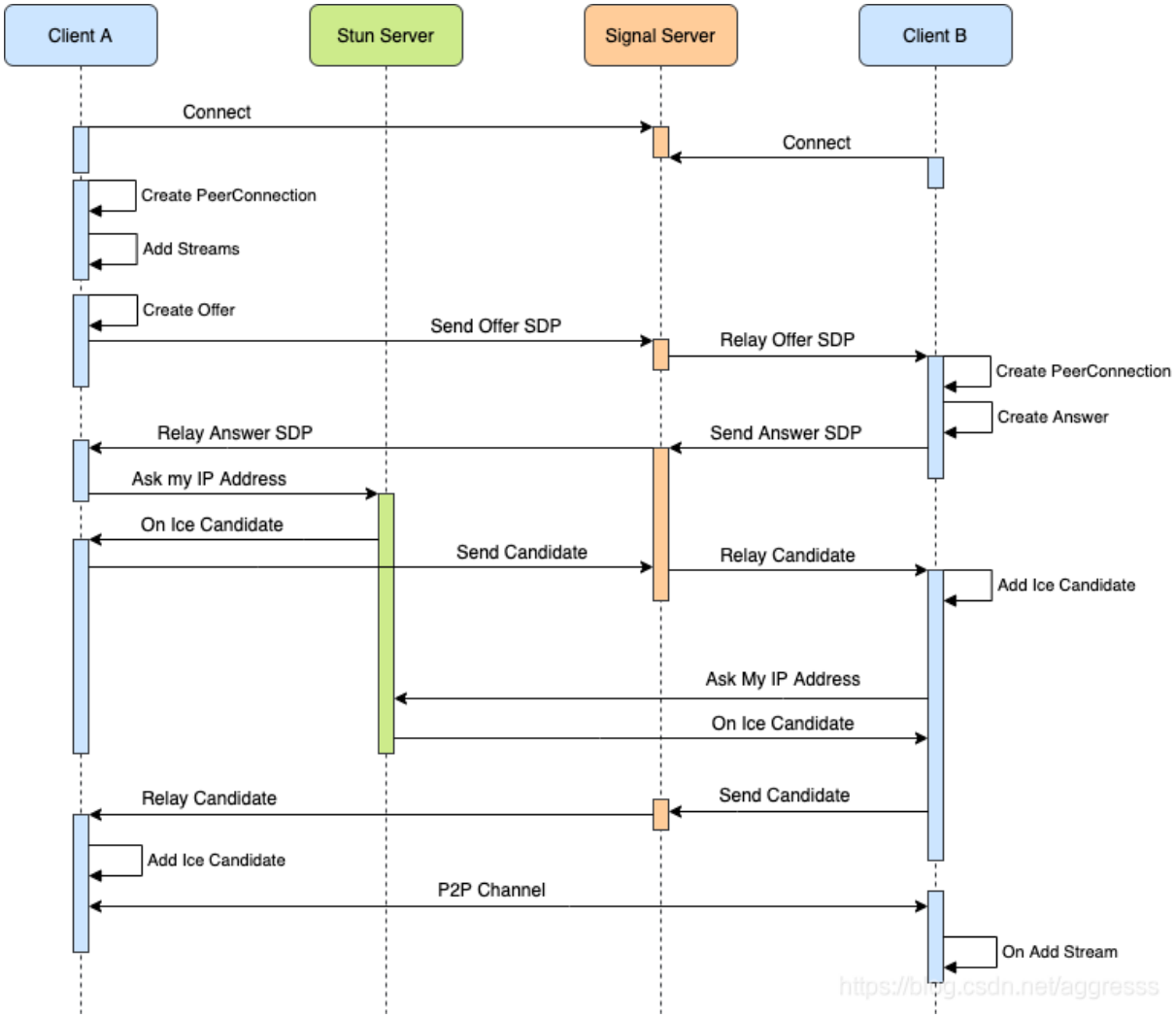


Figure 3 PeerConnection

위 사진은 WebRTC연결을 위한 프로세스입니다. 각 Peer를 연결하기 위해서는 STUN과 TURN 서버를 경유하여 SignalChannel을 통해서 이루어지게 됩니다. Offer와 Answer의 과정을 거치고 난 뒤에는 ICE Candidate를 공유하고 난 뒤 서로 통신을 시작합니다.



위의 모식도에서 표현한 부분들은 대부분의 함수들로 구현됩니다. 대부분의 것들이 JavaScript를 통해서 이루어지지만 Native Language를 통해서도 연결이 가능합니다.

```
navigator.mediaDevices.getUserMedia(gdmOptions)
.then(stream => {
    console.log("Stream Start");
    MyScreenShot.srcObject = stream;

    myStream = stream;

    initCall()

    socket.emit("join_room", roomName);
})
```

먼저 함수의 navigator.mediaDevices.getUserMedia를 통해서 미디어 스트림을 받아 그 스트림을 통해서 영상을 전송하거나 영상을 실행합니다.

```
function makeConnection(){
    myPeerConnection = new RTCPeerConnection({
        "iceServers" : [

            {"urls" : ["turn:ykrdev.tk:3478"], "username" : "test" , "credential" : "test123"}

        ]
    });
    myPeerConnection.addEventListener("icecandidate", handleIce);
    myPeerConnection.addEventListener("addstream", handleAddStream);
    myStream.getTracks()
        .forEach(track => myPeerConnection
        .addTrack(track, myStream));
}
```

RTCPeerConnection 함수를 통해 PeerConnection을 만들고 거기에 icecandidate와 addstream 이벤트에 대한 핸들러들을 추가합니다.

```
function handleIce(data){
    //console.log("sent candidate");
    socket.emit("ice", data.candidate, roomName);
}

function handleAddStream(data) {
    console.log("peerface exec");
    const RemoteScreenShot = document.getElementById("RemoteScreenShot")
    console.log(data);
    RemoteScreenShot.srcObject = data.stream;
    RemoteScreenShot.onclick = () => {

    }
}
```

handleice에서 icecandidate정보들을 서버로 보네 각 Peer에서 icecandidate를 등록합니다.

```
socket.on("welcome", async () => {
    const offer = await myPeerConnection.createOffer();
    myPeerConnection.setLocalDescription(offer);
    //console.log("sent the offer");
    socket.emit("offer", offer, roomName);
});
socket.on("answer", answer => {
    //console.log("received the answer");
    myPeerConnection.setRemoteDescription(answer);
});

// PEER B
socket.on("offer", async offer => {
    //console.log("received the offer");
    myPeerConnection.setRemoteDescription(offer);
    const answer = await myPeerConnection.createAnswer();
    myPeerConnection.setLocalDescription(answer);
    socket.emit("answer",answer, roomName);
    console.log("sent the answer");
});
// RTCCODE

socket.on("ice", ice => {
    //console.log("received candidate");
    myPeerConnection.addIceCandidate(ice);
});
```

시그널링 서버는 socket.io 를 통해 WebSocket으로 실시간으로 정보를 받아들입니다. 서버에서 welcome 이벤트를 받으면 Peer에서 createOffer함수를 통해 sdp를 생성하고 그 정보를 setLocalDescription을 통해 자신에 등록 그후 그 sdp를 서버를 통해 다른 Peer에게 보냅니다. 서버에서 offer라는 이름의 이벤트를 보내면 받은 데이터를 통해 setRemoteDescription으로 다른 Peer의 sdp를 등록하고 그것을 기반하여 자신의 sdp를 생성해 다시 서버로 보냅니다. 다시 그 answer를 다른 Peer에서 받아 setRemoteDescription를 통해 등록합니다. handleAddStream은 만약 서로의 Peer에 등록되어있는 MediaStream을 발견한다면 그것을 어떻게 처리할지를 정하는 이벤트 핸들러입니다. 이를 통해 원격에서 받아온 Stream을 웹페이지에 보여줍니다.

WEBRTC연결에 중요한 사항

WebRTC에서 가장 중요한 사항으로는 Connectivity가 있다. Connectivity는 세가지 유형이 존재한다. 이는 ICE candidates들에 상당히 연관된 것들입니다.

Host Connectivity – Localhost나 동일 네트워크상에서의 연결을 의미한다. 대부분의 연결들이 허용되며 실제 서버의 연결과는 매우 다른 양상을 보입니다. 대부분의 WebRTC관련 예제들이 이와 같은 환경에서 테스트해 많은 예제들이 실제로 작동하지 않았습니다.

Reflexive Connectivity – STUN 서버에서 사용되는 연결 유형으로 Remote PeerConnection이 바로 직접적으로 연결되는 유형입니다.

Relay Connectivity – TURN 서버에서 사용되는 Connection으로 Reflexive Connectivity와 유사하나 TURN서버를 경유함이 차이가 있습니다.

실제 환경에서는 Reflexive 나 Relay 연결해야 하나 이는 상당히 어려우며 TURN 서버와 STUN서버를 올바르게 적용해주어야 합니다. 위 세가지 연결은 UDP 환경에서의 연결을 말합니다. TURN 서버와 STUN 서버를 위해서는 Coturn이라는 서버를 주로 사용합니다. Coturn서버의 설정에 따라 STUN 과 TURN이 제대로 작동하지 않을 수도 있습니다.

SFU란

SFU란 Selective Forwarding Unit이라는 의미로 Peer의 연결들을 서버를 통해 연결하는 것을 의미합니다. N : N 네트워크 환경 및 데이터 공유를 하면 Peer의 개수가 많아질수록 즉 Peer들 간의 Connection이 많아질수록 클라이언트의 부담이 가중화 됩니다. 그것을 해결하기 위해 SFU서버를 대부분 채택합니다.

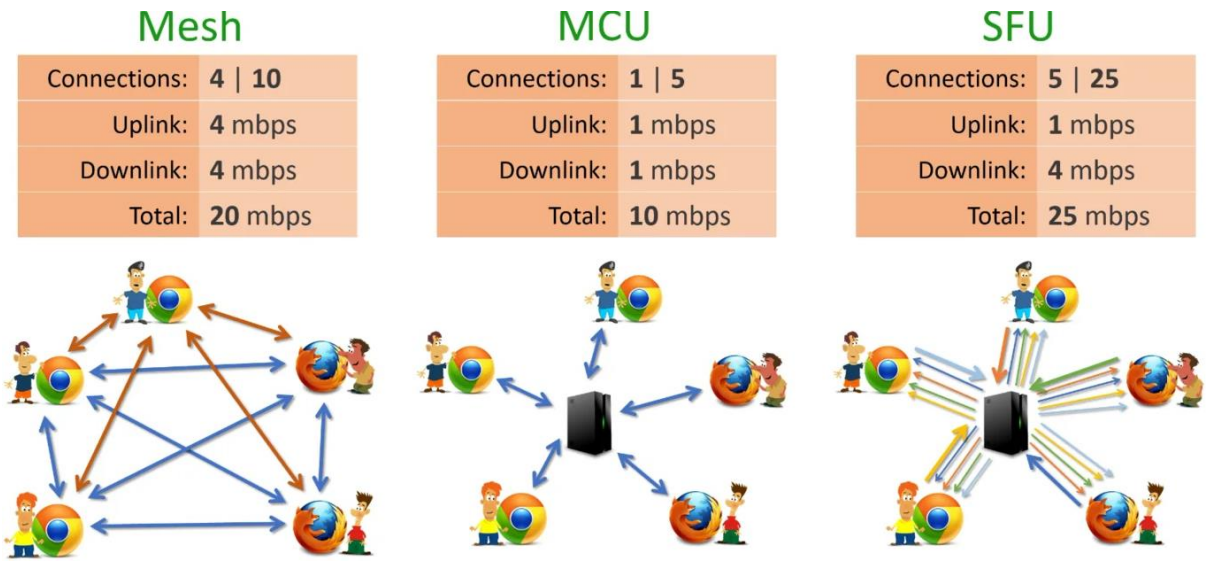
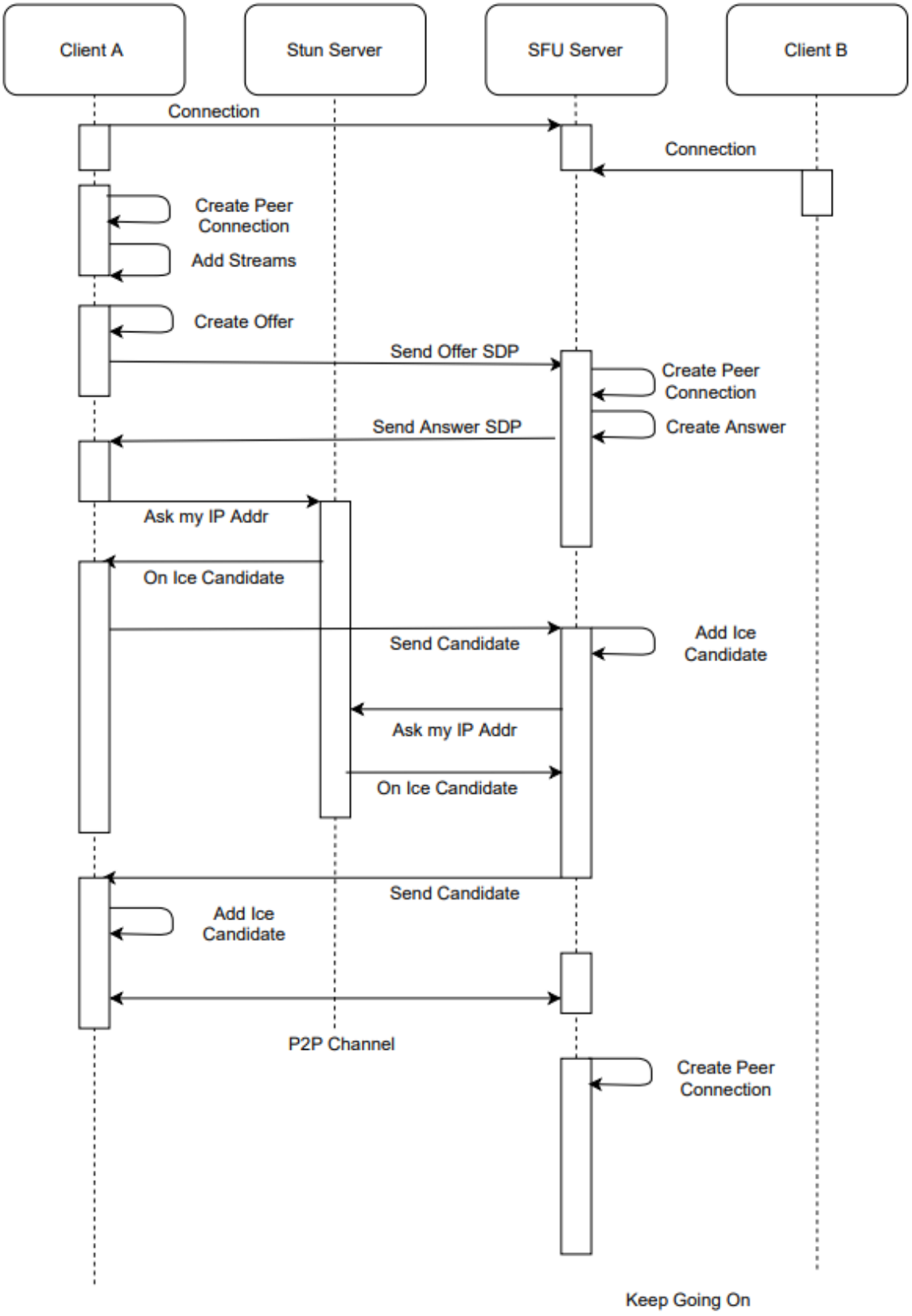


Figure 4 WebRTC 연결 유형

위 Figure 4는 WebRTC의 3가지 유형들을 나타낸 것입니다. MCU같은 경우에는 데이터 공유에 대한 세부 설정이 어려워 잘 적용하지 않고 SFU같은 경우에는 서버에 부하가 심하나 클라이언트의 부하는 적어 많이 사용되고 있고 본 프로젝트에서도 이를 기반하여 계발을 진행하였습니다.

SFU의 구현상 개념은 원래 클라이언트와 클라이언트 간에 형성되던 Connection을 Peer와 Server 간에 연결하는 것입니다. 그런 다음에 다른 클라이언트들에게 서버에서 받은 데이터를 전송할 연결을 다른 클라이언트 들에게 만듭니다. 즉 좀더 간단히 설명을 하자면 클라이언트와 서버간의 PeerConnection을 맺는다는 것입니다.

본 프로젝트에서 설계된 SFU서버



위는 본 프로젝트에서 설계한 SFU 서버의 모식도입니다. Peer Connection은 Server와 생성하고 다른 Peer가 연결된다면 똑 같은 프로세스를 다른 Peer하고도 수행합니다. 여기서 특이한점은 함수상의 관계에서 서버에서 받는 것 만을 위한 PeerConnection함수를 사용한다는 것입니다. 수신을 위한 컨넥션 송신을 위한 컨넥션을 구분하여 다른 변수에 저장한 뒤에 따로따로 Track을 관리하면서 전송 해야할 컨넥션을 관리합니다.

SFU 서버의 한계

SFU서버는 구현상에 여러가지 단점이 존재했습니다. 컨넥션을 연결할 때에 Ice candidate를 주고받았음에도 제대로 TURN서버를 경유하지 않는 경우도 생겼으며, 정확한 함수상에 순서관리가 어렵다 보니 영상이 제대로 보이지 않는 문제점 또한 많았습니다. 하지만 여러 관계를 정리하고 수정하면서 영상을 받아오는데는 성공했으나 실질적인 상업화를 위해서는 상용 mediaServer를 사용하는 것이 좀더 좋은 방법일 수도 있습니다. 하지만 본 연구 프로젝트에 쓰일 WebRTC환경은 매우 작은 리소스를 사용해야 하기 때문에 후에는 C언어를 사용한 WebRTC 클라이언트와 서버를 생성할 필요가 있습니다.