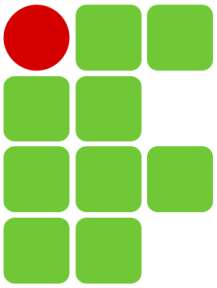

INSTITUTO FEDERAL
SANTA CATARINA
Campus São José

Relatório Final: Funcionamento do *WebRTC* e desafios encontrados na disciplina de Projeto Integrador II.

Disciplina: PJI 2
Guilherme Medeiros
Bruno do Nascimento

Este trabalho serve como recuperação de parte do desenvolvimento do projeto integrador. São objetivos deste documento relatar o que foi estudado sobre *WebRTC* e outras tecnologias envolvendo *streaming* de vídeo embarcado e explorar como o sistema pode ser implementado. As referências estudadas para este relatório são as mesmas que foram utilizadas pelo grupo para compor o trabalho anterior.

1. A escolha do WebRTC

Ao iniciar este projeto, a primeira dúvida foi quanto a qual plataforma usar. Foram feitas referências a Jitsi, MS Teams entre outros softwares, mas todos pareceram mais complicados do que a API que todos esses softwares usam: O WebRTC.

O padrão WebRTC é usado em várias aplicações de streaming de sucesso, por exemplo muito da plataforma de streaming da *Google* (meet, hangouts) usa o *Web Real Time Communication* para fazer o *streaming* de áudio e vídeo, além deles, *Facebook*, *Whatsapp*, *Discord*, e outros aplicativos utilizam *WebRTC* como principal plataforma de streaming (ISABAL, 2022).

Sabendo disso, a escolha do *WebRTC* é bem clara quando o objetivo é ter uma plataforma para desenvolver streaming de vídeo, mas é, primeiro, necessário entender como ele funciona.

2. *Web Real-Time Communication*

WebRTC é (SREDOJEV, 2015) uma plataforma web de código aberto desenvolvida em 2011 pela *Google* entre outras empresas com objetivo de criar uma plataforma que seria desenvolvida para o desenvolvimento de streaming (CHIANG, 2014) em *Real Time Communication* usando *Web*. A plataforma permite aplicações *desktop* e *mobile* (SREDOJEV, 2015) que vão de chamadas de vídeo à transferência de arquivo P2P.

Parte do objetivo do *WebRTC* é a existência de uma API baseada em navegador de vídeo chamada que não necessite a instalação de plugins. Utilizando a plataforma e *HTML5* (CHIANG, 2014) (SREDOJEV, 2015), por exemplo, é possível fazer a criação de uma vídeo chamada sem a adição

de nada ao seu dispositivo, já que todo o sistema funciona em cima de *C++* e *JavasCript*, junto com uma sessão de *APIs* que, com o acesso correto ao sistema do dispositivo, conseguem fazer funcionar uma vídeo chamada em um navegador.

A tecnologia contém os fatores necessários para a comunicação em tempo real na internet (SREDOJEV, 2015), que podem ser acessados pela API em javascript desenvolvida para a plataforma. Navegadores como Google, Mozilla e Opera dão suporte ao WebRTC.

Entretanto, não é fácil compreender que *WebRTC* não é apenas uma API mas sim um conjunto de APIs, e pode ser levemente confuso entrar na plataforma sem saber que cada grupo, cada programador que desenvolveu sua plataforma *WebRTC* tinha seu próprio propósito e funcionamento, por isso pode ser necessário compreender a tecnologia para ser possível construir em cima da plataforma e criar seu sistema, caso as aplicações sejam muito específicas.

2.1. Componentes importantes do *WebRTC*

Pode-se dizer que (SREDOJEV, 2015) o *WebRTC* tem 3 componentes essenciais:

2.1.1. *MediaStream*

Essa API é o que permite ao navegador ter acesso e controle aos dispositivos necessários, normalmente câmera e microfone (CHIANG, 2014). O *MediaStream* faz a coleta da entrada e cria um *output* para a coleta por outra *API* (seção 2.1.2). Em uma chamada com N mídias sendo transmitidas possui N *MediaStreams* funcionando (SREDOJEV, 2015), cada uma com uma entrada - que pode ser um microfone, uma câmera, ou até mesmo uma entrada de vídeo interna ao dispositivo rodando a API - e uma saída, que é disponibilizada ao restante da plataforma para a transmissão e pode ser levada para uma ou mais tecnologias diferentes caso esse seja o objetivo.

Neste ponto a plataforma já oferece (CHIANG, 2014) a informação de qual tipo de mídia está sendo transmitido por esse stream.

2.1.2. *RTCPeerConnection*

RTCPeerConnection é uma API que estabelece e fornece uma conexão *WebRTC* entre duas pontas (CHIANG, 2014). Para fazer esse tipo de conexão P2P, protocolos STUN e TURN são usados pelo *framework ICE*, ambos fornecidos pela *Google*.

2.1.2.1. STUN, TURN e ICE

O grande problema que faz necessário o uso do *framework ICE* é a existência das *NATs* (PAREIN, 2020).

Para se criar uma conexão P2P com usuários na internet moderna, onde uma ponta pode ou não estar usando *NAT* e, caso esteja usando, este pode ser de tipos diferentes, é necessário saber exatamente como se endereçar essa conexão.

O *framework ICE* cria um agente que faz uma requisição para um servidor *STUN (Session Traversal Utilities for NAT)* especificado pelo *RTCPeerConnection* na criação do servidor *WebRTC*. Neste momento, a *NAT* especifica o endereço (PAREIN, 2020) e a porta para receber do servidor *STUN*, requisições de endereços diferentes para checar se o endereço está atrás de uma *NAT*e de qual tipo ela é. Um exemplo de funcionamento de *STUN* pode ser visto na figura 1.

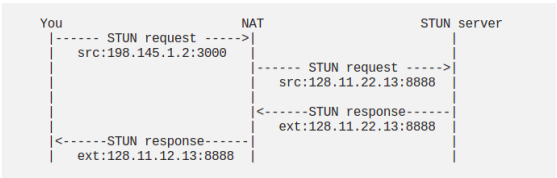

Figura 1 - *STUN* (PAREIN, 2020)

Entretanto, falar de *NAT* não é tão simples como normalmente é feito, principalmente para conexões P2P, já que *NAT*s podem ser implementadas de maneiras diferentes e mudar completamente a forma de se fazer uma conexão. *Full Cone NAT*, único *NAT* que matém a porta permanentemente aberta (AMORIN, SILVA, 2022), permitindo conexão com hosts externo, permitem (PAREIN, 2020) a conexão P2P, entretanto, *NAT* Simétrico (AMORIN, SILVA, 2022) aplica restrições à portas que impossibilita a conexão P2P (PAREIN, 2020).

Nos casos onde a conexão P2P não é possível se torna necessário o uso de um servidor *TURN (Traversal Using Relays around NAT)*, que cria um servidor (PAREIN, 2020) para fazer uma espécie de triangulação ao redor da *NAT*. Para isso, o cliente chama o servidor *TURN* (figura 2) com uma requisição *Allocate*, que sinaliza ao servidor *TURN* para que ele guarde recursos para uma nova conexão, este retorna ao cliente uma sinalização *Allocation Successful* e o designa um endereço “apelido”. Após isso, o servidor *TURN* entra em contato com o segundo cliente e também o designa um endereço “apelido”. Dessa forma, é possível dar a volta no *NAT* e fazer a conexão por meio dessa triangulação via servidor *TURN*.

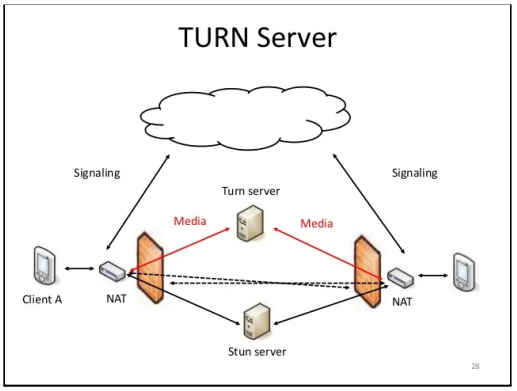

Figura 2 - Triangulação (NEXTCLOUD, 2022)

Em *WebRTC* é necessário para a conexão ser estabelecida (PAREIN, 2020) a especificação do servidor turn dentro do objeto *RTCPeerConnection*.

A compreensão do funcionamento da conexão *P2P* só pode ser compreendida em sua totalidade em *WebRTC* se o mecanismo *STUN*, *TURN* e *ICE* for entendido e é bastante complicado desenvolver a conexão em cima da plataforma sem saber o que o modelo significa.

É quando o *ICE* acha uma forma via *TURN* e/ou *STUN* que a conexão está finalizada e ambos os *peers* estão conectados. A figura 3 mostra uma conexão completa *WebRTC*.

2.1.3. *RTCDataChannel*

Esta *API* cria um canal bi-direcional quando a conexão já foi estabelecida pelo *RTCPeerConnection*, permitindo a troca de informação entre as pontas (SREDOJEV, 2015). Cada canal de dados pode ser configurado para fornecer a informação dentro ou fora de ordem e de uma forma confiável ou não, no sentido da entrega dos dados.

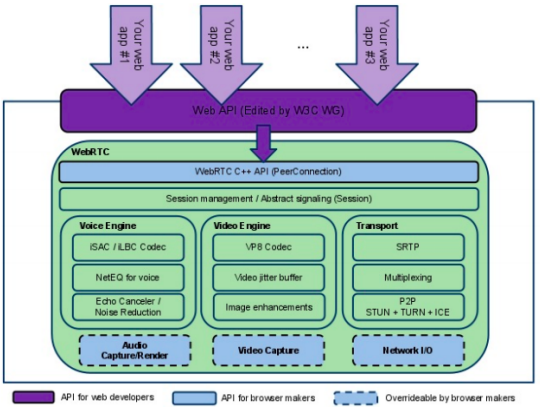

Figura 3 - Conexões *WebRTC* (CHIANG, 2014)

3. Compreendendo o *WebRTC*

Para começar a entender como montar um cenário *WebRTC* primeiro é necessário compreender a sinalização da plataforma, ou seja, a troca de mensagens para estabelecer e manter a conexão e envio de dados (figura 4).

3.1. Sinalização

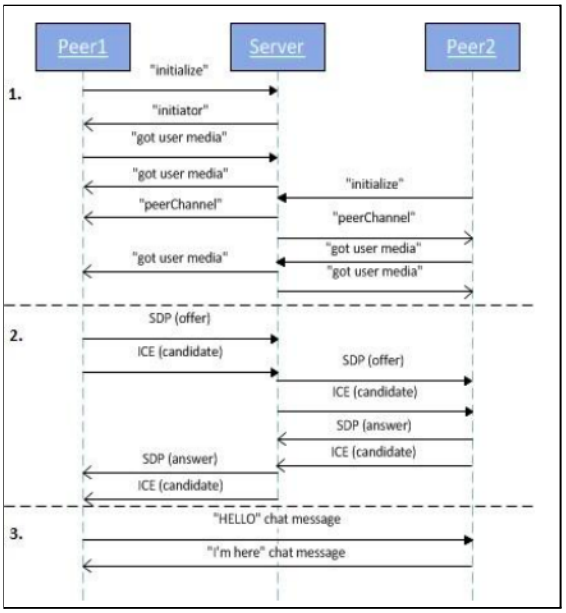

Figura 4 - Sinalização *WebRTC* via *SIP* em *WebSockets* (CHIANG, 2014)

De forma alguma essa é a única forma de se implementar um cenário *WebRTC*, aliás, a plataforma mantém a troca de sinalizações aberta (CHIANG, 2014), que pode ser feita por *SIP, XMPP, XHR*, entre outros. A solução apresentada na figura e uma das mais comuns é por *SIP* funcionando sobre *WebSockets*. As sinalizações em si não serão descritas em detalhes já que estão bem explicadas na figura 4, Entretanto, é importante conhecer o protocolo *SDP* que faz a troca de informações sobre tipo de mídia, formato, *codecs*, entre outras propriedades (CHIANG, 2014) que cada ponta suporta e aceita receber. A sinalização acontece em todo o processo em paralelo ao fluxo de dados.

3.2 Métodos e APIs *WebRTC*

Primeiramente (CHIANG, 2014), os navegadores irão receber, pelo *RTCDataChannel* usando a *API GetUserMedia*, o vídeo de cada usuário para ser transmitido para a outra ponta. Esse *stream* de vídeo é empacotado de acordo com a negociação feita pelo *SDP*, e só depois são transmitidos e desempacotados pelos navegadores do outro lado da transmissão, formando assim um fluxo de dados *P2P* que é transmitido utilizando (CHIANG, 2014) um link *RTP*.

Assim o funcionamento dos protocolos *WebRTC* é feito.

4. Primeira tentativa de estabelecer conexão *WebRTC* pura.

No desenvolvimento do projeto, primeiramente foi testado um servidor *WebRTC* rodando sobre *WebSockets* desenvolvido em *JavaScript*, linguagem que teve de ser estudada pelo grupo para a compreensão dos códigos. A transmissão se baseia em dois arquivos *JavaScript*, um para a implementação do servidor e outra para a recepção do vídeo por parte do cliente, com o servidor sendo executado em localhost, o código do cliente (watcher) pode ser visualizado neste repositório do *GitHub*.

Neste código, primeiramente, se estabelece o servidor *ICE* para a determinação do *STUN* (como visto anteriormente, na seção 2.1.2.1). Seguinte a isso, cria-se um objeto *RTCPeerConnection*. Este objeto passa por toda a configuração necessária para se receber e enviar o fluxo de dados, mesmo que aqui estejamos apenas configurando um visualizador de vídeo. A configuração se baseia em criar uma resposta e uma descrição local com as propriedades *SDP* e sinalizando, via *sockets*, um “apelido” chamado “*watcher*”, que será usado para sinalizar o servidor que quem entrou em contato com ele foi um visualizador. Toda a troca de

sinalização acontece por baixo do código, via *SIP*, os métodos disponíveis em *JavaScript* abstraem uma parte do funcionamento da sinalização.

Já o servidor é bem mais complicado de ser compreendido, ele dispõe de uma porção de funções para fazer a transmissão. Alguns dos mais importantes parâmetros para esse funcionamento são o *getStream*, que possibilita, junto ao *gotDevices*, ao gerente do servidor escolher a origem do fluxo de vídeo e áudio. Dessa forma, não apenas uma câmera e um microfone do computador podem ser usados, mas também um *stream* de vídeo não ao vivo desde que seja simulado interfaces que transmitam ele e que possam ser selecionadas pelo *GetStream*. Além disso, utilizando *sockets* o servidor *WebRTC* ao receber a mensagem *"answer"* e *"watcher"* determinando, como falado anteriormente, o caráter do visualizador do vídeo, cria uma conexão utilizando *RTCPeerConnection* finalmente faz o *stream* de vídeo com sucesso ao visualizador.

Este cenário foi montado e executado em *localhost*, a figura 5 mostra uma execução desse teste.

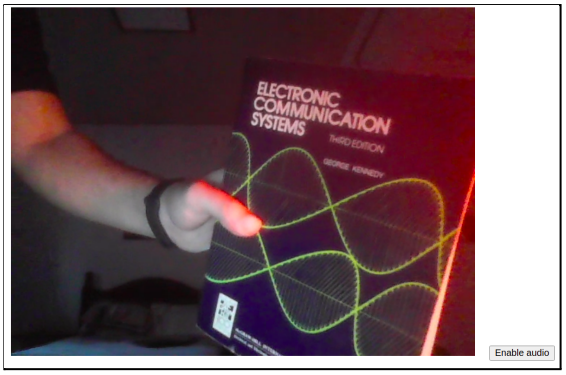

Figura 5 - Execução do *WebRTC*

Assim, depois de compreender o funcionamento do código foi possível montar o teste e ver a transmissão de vídeo completamente.

Entretanto, o problema dessa implementação é que a plataforma para o funcionamento do servidor escolhida foi uma *RaspBerry Pi* que não possui acesso a navegadores, parte crucial para o funcionamento do *WebRTC* puro. Esta aplicação, entretanto, funcionaria em outros cenários, já que uma vez que o *broadcaster* tenha sido iniciado no navegador, acessar o *streaming* de vídeo é simples, deixando livre para o cliente se conectar ao servidor assim que quiser por via de um navegador, que é perfeitamente implementável no contexto de aplicativos ou serviços *web*.

Por isso, para o serviço ser funcional, ele precisava conseguir ser executado por linha de comando e algumas alternativas foram estudadas.

4.1. Node.js

Node.js pareceu primeiramente a solução mais direta. Este (NODEJS, 2022) é um software em código aberto que possibilita que sejam executados códigos escritos em JavaScript sem a utilização de um navegador, sendo uma plataforma assíncrona e orientada a eventos.

Tentativas com *Node.js* foram feitas, entretanto sem sucesso. Para fazer as modificações necessárias na plataforma do *WebRTC* para permitir a utilização do *Node.js* ainda mais estudo seria necessário e não pareceu claro para o grupo como fazer sua utilização, principalmente quanto a orientação à eventos utilizando *callback*.

4.2 Firebase e FirebaseRTC

Depois das dificuldades enfrentadas com o *Node.js* a introdução do *FirebaseRTC* foi promissora. O *Firebase*, plataforma da *Google* que possibilita o *FirebaseRTC* permite que o *backend* de uma aplicação seja completamente substituída por uma plataforma separada, sinalizando seu funcionamento como *Back-end-as-a-service (BaaS)*. A execução (WEBRTCc, 2022) é descrita pela própria organização WebRTC (esta não será repetida aqui e pode ser conferida no link acima) e funciona perfeitamente.

Nenhuma modificação é necessária no código para o funcionamento do projeto, entretanto, se pareceu necessário compreender o código por trás do projeto do *Firebase* para entender o que estava acontecendo. O código base pode ser visualizado aqui.

Logo no começo já são percebidos estruturas já conhecidas, como a configuração do agente *ICE*, afinal essa é uma solução *WebRTC* e o agente *ICE* assim como os servidores *STUN* são necessários para o funcionamento. Além disso, percebemos que a conexão *peer* criada é uma *RTCPeerConnection*, assim como era no código anterior.

Grande parte do código do *FirebaseRTC* não é compreendido pelo grupo, entretanto, o principal da conexão *WebRTC* permanece bastante clara. *Firebase* funciona com uma conexão banco de dados *NoSQL*, que nunca foi explorada por nenhum dos autores deste relatório.

5. Conclusões

O funcionamento de um simples servidor de *streaming* de vídeo pode ser mais complexo do que se imagina. Após decisões e mudanças de escopo, se tornou complicado entregar a solução no prazo requerido e, principalmente, modificar parte dos códigos que foram

incompreensíveis em parte para o grupo que realizou o projeto.

Entretanto, o projeto possibilitou alguns aprendizados interessantes para o grupo que devem ser mencionados:

1. **Fase de pré projeto**: foi compreendido como um grupo de desenvolvimento com o objetivo de desenvolver um produto/solução deve decidir pelas tecnologias utilizadas, linguagens, técnicas, delegar funções e dividir tarefas nessa fase da disciplina. Tornou-se claro, entretanto, que devem existir mudanças quanto a estudar mais a fundo todas as necessidades e limites de todas as tecnologias utilizadas, já que, em começo, não foi percebido pelo grupo que a utilização de um navegador seria impossível pela utilização de uma *Raspberry Pi*.
2. **Estudo da tecnologia a ser utilizada:** O *WebRTC e a compreensão de seu funcionamento* se mostraram um desafio para o grupo, entretanto, após a análise do código e da literatura sobre o tema, grande parte de como a plataforma funciona, suas configurações, sinalizações, tecnologias que usa e limitações foram compreendidos.
3. **Entendendo um código não pessoal:** Grande parte da dificuldade da vida de um desenvolvedor da área de tecnologia é a de compreender um código que em que o profissional não fez desde o princípio. A realização deste projeto possibilitou entrar em contato com programas de tecnologias desconhecidas, feitas por desenvolvedores com quem não é possível ter contato direto e discutir dúvidas, mesmo assim, foi possível entender alguns dos projetos em sua totalidade, outros apenas em parte, entretanto, essa experiência se mostrou valiosa para os membros do grupo.

Por isso, mesmo considerando como o projeto em si, o *streaming* de vídeo via *WebRTC* embarcado em uma *Raspberry Pi*, não foi bem sucedido, o grupo considera o aprendizado da disciplina relevante e importante para a formação do profissional de Telecomunicações.

6. Referências

WEBRTCa. **Introdução ao WebRTC**. Disponível em: https://webrtc.org/getting-started/overview. Acesso em: 17 mar. 2022.

WEBRTCb. **WebRTC samples**. Disponível em: https://webrtc.github.io/samples/. Acesso em: 17 mar. 2022.

WEBRTCc. **Firebase + WebRTC Codelab**. Disponível em: https://webrtc.org/getting-started/firebase-rtc-codelab. Acesso em: 15 mar. 2022.

Sredojev, B., Samardzija, D., & Posarac, D. 2015. ***WebRTC technology overview and signaling solution design and implementation***. *2015 38th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*.

Chiang, C.-Y., Chen, Y.-L., Tsai, P.-S., & Yuan, S.-M. 2014. ***A Video Conferencing System Based on WebRTC for Seniors***. *2014 International Conference on Trustworthy Systems and Their Applications*.

ISABAL, Miguel. **8 Powerful Applications Built Using WebRTC**. Disponível em: https://www.unitedworldtelecom.com/learn/webrtc-applications/. Acesso em: 17 mar. 2022.

PAREIN, Heloise. 2020. **WebRTC: the ICE Framework, STUN and TURN Servers**. Disponível em: https://levelup.gitconnected.com/webrtc-the-ice-framework-stun-and-turn-servers-10b2972483bb. Acesso em: 15 mar. 2022.

AMORIM, Douglas; SILVA, Lucas da. **NAT**. Disponível em: https://wiki.sj.ifsc.edu.br/index.php/NAT. Acesso em: 15 mar. 2022.

NEXTCLOUD. **Help-me-understand-if-both-stun-and-turn-servers-are-required**. Disponível em: https://help.nextcloud.com/t/help-me-understand-if-both-stun-and-turn-servers-are-required/20795. Acesso em: 15 mar. 2022.

NODEJS. **About Node.js**. Disponível em: https://nodejs.org/en/about/. Acesso em: 15 mar. 2022.