Universidade Federal de Campina Grande
Centro de Engenharia Elétrica e Informática
Unidade Acadêmica de Engenharia Elétrica

Trabalho de Conclusão de Curso

Aplicação de Codificadores Escaláveis de Vídeo

Kaio Ramalho Freire

Campina Grande - PB

Maio de 2021

Kaio Ramalho Freire

Aplicação de Codificadores Escaláveis de Vídeo

Trabalho de Conclusão de Curso submetido à Coordenação de Graduação em Engenharia Elétrica da Universidade Federal de Campina Grande, Campus Campina Grande, como parte dos requisitos necessários para obtenção do título de Graduado em Engenharia Elétrica.

Área de Concentração: Eletrônica

Prof. Marcos Ricardo Alcântara de Morais, D.Sc.

(Orientador)

Campina Grande - PB

Maio de 2021

Agradecimentos

Com grande reconhecimento, agradeço a Deus pela conquista de conseguir terminar um curso como o de Engenharia Elétrica na UFCG. Incluo nos agradecimentos meus pais, Alba Lúcia Soares Ramalho e Waldir Carlos Costa Freire, que me possibilitaram fazer esse curso e sempre apoiaram em todas as decisões, desde o dia em que decidi ir morar fora. Agradeço também a minhas irmãs, que sempre me ajudaram no que eu precisei durante todo o tempo do curso.

Agradeço a todos meus amigos, que foram de muita importância em todos os momentos, sejam eles de comemoração por conta de aprovações ou apoio nos momentos de infelicidades. Agradeço a meus amigos, que me auxiliaram nos testes do aplicativo desenvolvido nesse trabalho.

Agradecimentos em especial a meu amigo Pedro Paulo, que me auxiliou na correção gramatical deste trabalho e que sem ele seria bastante difícil ter conseguido suporte para morar em outra cidade por tanto tempo. Importante agradecer a Telécio, que me auxiliou com a parte da resolução dos problemas a nível de servidor e a Lucas que me ajudou nas defesas de TCC e estágio, fornecendo a câmera para realizar as defesas online.

Agradeço também ao departamento de engenharia elétrica, que esteve comigo nessa caminhada árdua, em especial agradeço a Adail Paz e Tchaikovsky Oliveira, que, em quaisquer momentos em que eu os solicitasse, estiveram lá por mim. São pessoas muito iluminadas por Deus e muitas vezes me aconselharam em relação ao curso. Agradeço ao Professor Gutemberg, atual coordenador do curso, que foi uma pessoa bastante sincera e disponível em todos os momentos em que procurei seu contato.

Finalmente, agradeço ao meu orientador, o professor Marcos Morais, que me orientou e me auxiliou em tudo que foi necessário para concluir este TCC. Desde cada mudança no projeto inicial até o acompanhamento final deste trabalho.

"É possível encontrar a felicidade mesmo nas horas mais sombrias,

basta se lembrar de procurar pela luz". (Alvo Dumbledore)

Resumo

Os estudos sobre codificadores de vídeo são considerados de grande pertinência quando se trata de transmissões ao vivo, seja pela Internet ou pela televisão. Este trabalho tem como objetivo trazer a evolução no que se refere aos codificadores e demonstrar a possibilidade de construção de aplicações sem o uso de softwares externos, utilizando apenas WebRTC. A aplicação foi desenvolvida inteiramente em javascript, nodejs e WebRTC. O WebRTC foi a principal ferramenta da aplicação, que tinha como objetivo estabelecer uma chamada de vídeo entre 2 ou mais integrantes com a possibilidade de compartilhamento de tela.

Palavras-chave: Codec, WebRTC, Conversação Online,

Abstract

Studies on video encoders are consired of huge pertinence when it comes to live broadcasts, whether over the Internet or through television. This work aims to bring up the evolution on what comes to the encoders and the demonstrate the possibility of building applications without the use of external softwares, by the use WebRTC only. The application was developed entirely using javascript, nodejs and WebRTC. WebRTC was the core of the application, which aimed to establish video calls between 2 or more members and the possibility of sharing the screen.

Keywords: Codec, WebRTC, Online Chat

Lista de Figuras

Figura 2.1 – Princípio da codificação escalonável usando camadas T. 4
Figura 2.2 – Estrutura multicamadas com previsão adicional entre camadas para permitir a codificação escalonável espacial. 6
Figura 2.3 – Estrutura de dois laços para escalabilidade *SNR* em um codificador híbrido. 7
Figura 2.4 – Estrutura do codificador de vídeo H.264 / AVC 8
Figura 2.5 – O princípio da codificação de vídeo escalonável no H264. 9
Figura 2.6 – Estrutura do codificador SVC simplificado. 10
Figura 2.7 – Codificador de vídeo HEVC típico (com elementos de modelagem do decodificador sombreados em cinza claro). 12
Figura 2.8 – Possíveis vantagens do uso do codec H266. 13
Figura 2.9 – Aplicação Web utilizando o WebRTC 16
Figura 3.1 – Representação do diretório de uma aplicação de chat utilizando WebRTC 18
Figura 3.2 – Representação do código responsável pela parte do servidor. 18
Figura 3.3 – Representação do código dos pacotes do app. 19
Figura 3.4 – Representação do código de captura da câmera e áudio do navegador. 20
Figura 3.5 – Representação do código de adição de uma *stream* de vídeo. 20
Figura 3.6 – Representação do código de conexão de um novo usuário 21
Figura 3.7 – Representação do código de captura de tela. 22
Figura 3.8 – Representação do código de funções de microfone e câmera. 22
Figura 3.9 – Representação do terminal de início do código. 24
Figura 3.10 – Representação do terminal de início do servidor de comunicação. . . . 24
Figura 4.1 – Representação da página *web* contando apenas com a presença do *host*. 25
Figura 4.2 – Representação da página *web* com a presença do *host* e de outro usuário. 26
Figura 4.3 – Representação da página web com múltiplos participantes 26
Figura 4.4 – Representação da página web com usuário sem vídeo 27

Lista de Abreviaturas e Siglas

WebRTC - *Web Real Time Communication.*

SVC - *Scalable Video Coding.*

Codec - Codificador

AVC - *Advanced Video Coding*

HEVC - *High Efficiency Video Coding*

VVC - *Versatile Video Coding*

H.264 - Outro nome dado para o AVC.

H.265 - Outro nome dado para o HEVC.

H.266 - Outro nome dado para o VVC.

VP9 - Codificador de vídeo VP9.

AV1 - Codificador de vídeo AV1.

P2P - *Peer to peer*

Sumário

1 INTRODUÇÃO . . . **1**
1.1 Objetivos . . . **1**
1.1.1 Objetivo Geral . . . 1
1.1.2 Objetivos Específicos . . . 2
1.2 Estrutura do Trabalho . . . **2**

2 FUNDAMENTAÇÃO TEÓRICA . . . **3**
2.1 Tipos de *Codecs* . . . **3**
2.1.1 Não Escalável . . . 3
2.1.2 Escalável . . . 3
2.1.2.1 Escalabilidade Temporal . . . 4
2.1.2.2 Escalabilidade Espacial . . . 5
2.1.2.3 Escalabilidade de nível de qualidade . . . 6
2.2 *Codecs* . . . **7**
2.2.1 H.264(AVC) . . . 7
2.2.2 H.265 (HEVC) . . . 11
2.2.3 H.266(VVC) . . . 13
2.2.4 VP9 . . . 14
2.2.5 AV1 . . . 15
2.3 *WebRTC* . . . **16**

3 METODOLOGIA . . . **17**
3.1 *App* de conversação com *WebRTC* . . . **17**
3.1.1 Desenvolvimento do app . . . 17
3.1.2 Conexão com servidores externos . . . 22
3.1.3 Execução do aplicativo . . . 23

4 RESULTADOS . . . **25**
4.1 *App* de conversação com *WebRTC* . . . **25**
4.1.1 Problemas encontrados . . . 27
4.1.2 Comparação com outros aplicativos . . . 28
4.1.3 Melhorias futuras . . . 28
4.2 Trabalhos futuros . . . **28**

5 CONCLUSÕES . . . **29**

REFERÊNCIAS BIBLIOGRÁFICAS 30

ANEXOS 31

ANEXO A – CÓDIGO DO SERVIDOR 32

ANEXO B – CÓDIGO DO SCRIPT 34

ANEXO C – CÓDIGO DA PÁGINA HTML 39

ANEXO D – CÓDIGO CSS DO VISUAL DA PÁGINA 43

1 Introdução

Nos dias atuais é observado grande uso do recurso de vídeos sob demanda por várias plataformas, entre elas *Youtube, Netflix, GloboPlay, Amazon*, como exemplos. Para o funcionamento dessas plataformas, o uso de codecs é essencial para diminuir o consumo de banda do usuário e da prestadora de serviços.

Codecs são codificadores de vídeo com a finalidade de reduzir o tamanho do arquivo de vídeo, sem que haja perda de qualidade da imagem e do som. Atualmente existem vários codificadores, dos quais irei apresentar as diferenças entre os escaláveis e os não escaláveis. Codificador escalável é um codificador que permite que vídeos de alta qualidade tenham um ou mais subconjuntos de fluxos de bits.

Hoje em dia temos vários codecs sendo utilizados no mercado e outros ainda em desenvolvimento. Como exemplos de codecs escaláveis podemos citar: *HEVC(H.265), AV1, VP9, e o VVC(H.266).*

Grande parte dos problemas percebidos no uso de vídeos sob demanda e de conversas de vídeo são causados devido a problemas com internet. Por exemplo, quando um usuário sofre perda de dados, em vez de impedir o uso, primeiramente a codificação prioriza reduzir a qualidade para que transmissão não seja interrompida, mesmo que à custa da perda de qualidade.

A solução de reduzir temporariamente a qualidade mostra-se bastante efetiva para lidar com os eventuais problemas na conexão, mas gera também problemas, devido à necessidade do armazenamento de arquivos de vídeo em várias qualidades ou o uso de alguma técnica para armazenar em um só arquivo as várias qualidades de vídeo.

Outro problema advindo do uso de certos codecs é que muitos deles são desenvolvidos por empresas que, para utilizar beneficiar-se dos resultados do desenvolvimento dos codecs, cobram licenciamento para seu uso. Isso gera uma enorme barreira entre os desenvolvedores e os usuários. O problema do licenciamento tem sido contornado graças aos codecs *open source*, disponíveis abertamente na para uso e desenvolvimento, como o próprio WebRTC.

1.1 Objetivos

1.1.1 Objetivo Geral

Neste trabalho tem-se como objetivo principal mostrar as diferenças entre a escalabilidade de codificadores, mostrar os principais codecs em uso no mercado e trazer uma aplicação direta no uso do WebRTC.

1.1.2 Objetivos Específicos

- Explicar a diferença entre um codec escalável e um não escalável.
- Explicar as diferenças entre os vários codecs no mercado.
- Apresentar uma aplicação direta do uso do WebRTC.

1.2 Estrutura do Trabalho

Este Trabalho de Conclusão de Curso é composto por 5(cinco) capítulos, os quais são apresentados a seguir:

- Capítulo 1 - Apresenta-se uma contextualização do tema de forma introdutória, a definição dos objetivos e a estrutura do trabalho.
- Capítulo 2 - Expõem-se os conceitos teóricos utilizados para a pesquisa e desenvolvimento de aplicações.
- Capítulo 3 - Apresenta a metodologia utilizada no desenvolvimento da aplicação.
- Capítulo 4 - Apresentam-se os resultados da aplicação desenvolvida.
- Capítulo 5 - Apresentam-se as considerações finais acerca do trabalho.

2 Fundamentação Teórica

Nessa seção será comentada a teoria utilizada na construção do projeto em si. Nos objetivos foi exposto que o objetivo principal do trabalho foi de estudar as principais aplicações dos codificadores escaláveis de vídeo e realizar comparações com outros *codec*. Então, nesta sessão será explicado o que são *codecs*, seus tipos e suas principais características.

2.1 Tipos de *Codecs*

Nesta seção serão explicados os dois tipos de *codecs* disponíveis no mercado, suas principais características e suas diferenças. Na subseção dos escaláveis serão mostrados também os principais tipos de escalabilidade.

2.1.1 Não Escalável

Um codificador não escalável é um tipo de *codec* que não permite vários fluxos de bits em um mesmo arquivo. Por exemplo, se um *codec* não escalável for feito para a resolução 1920x1080, ao tentar transmitir uma resolução menor perderá bastante qualidade ao exibir ou irá dar erro de compatibilidade.

Já se foi muito comum o uso desses codificadores não-escaláveis, principalmente devido ao fator de sua programação e pelo licenciamento. Hoje em dia é muito mais comum o uso de *codecs* que possuem escalabilidade intrínseca. Os codificadores não escaláveis também são chamados de codificadores de camada única, devido a não permitirem vários fluxos de bits.

2.1.2 Escalável

Os esquemas escaláveis de codificação de vídeo destinam-se a codificar o sinal uma vez na resolução mais alta, porém permitem a decodificação de fluxos parciais, dependendo da taxa e resolução específicas exigidas por um determinado aplicativo. Isso permite uma solução simples e flexível para transmissão em redes heterogêneas, além de fornecer adaptabilidade para variações de largura de banda e condições de erro.

Um codificador escalável é um *codec* que permite que um vídeo de alta qualidade tenha um ou mais subconjuntos de fluxos de bits. Um subconjunto pode representar um conjunto para resolução espacial mais baixa, resolução temporal mais baixa ou um sinal de vídeo de qualidade inferior (cada um separadamente ou em combinação) em comparação com o fluxo de bits do qual é derivado.

Um princípio muito geral de codificação escalonável (em camadas) e decodificação é mostrado na figura 2.1. Nela, é mostrado que o sinal ao chegar pode atravessar por várias camadas de codificação e decodificação, podendo assim produzir vários tipos de resolução e de qualidade intermediários. A resolução espaço-temporal do sinal a ser representada pela camada de base é primeiramente gerada por dizimação (pré-processamento). No estágio de codificação subsequente, uma configuração apropriada do quantizador levará a um certo nível de qualidade geral da informação de base.(Ohm, 2005)

A reconstrução da camada de base é uma aproximação de todos os níveis de resolução de camada mais altos e pode ser utilizada na decodificação das camadas subsequentes. A unidade de processamento intermediário realiza o aumento artificial da taxa de amostragem do próximo sinal da camada inferior para a resolução da camada subsequente. Normalmente, o pré-processamento e o processamento intermediário são realizados por decimação e interpolação ao longo de todos os estágios, enquanto a ação particular a ser tomada pode ser bastante diferente. Dependendo da dimensão da escalabilidade, por exemplo, o processamento compensado por movimento pode ser implementado para aumento artificial da taxa de amostragem de taxas de quadros *(frame-rate)* em escalabilidade temporal.

Figura 2.1 – Princípio da codificação escalonável usando camadas T.

Fonte: (Ohm, 2005), Adaptado pelo autor

2.1.2.1 Escalabilidade Temporal

A escalabilidade temporal ocorre quando é necessário alterar a quantidade da taxa de quadros por segundo. Esse recurso é usado para não se perder nenhum quadro quando se tem um vídeo gravado em alta taxa de quadros por segundo e que será exibido em uma tela com limite de quadros por segundo

A escalabilidade é feita através de um subconjunto de fluxo de bits que determina a quantidade de quadros que será passada para o usuário final. Essas várias opções de

quadros são todas abarcadas em um só arquivo, podendo-se assim diminuir o custo de armazenamento que se teria com cópias do mesmo vídeo.

Um fluxo de bits fornece escalabilidade temporal quando o conjunto de unidades de acesso correspondentes pode ser particionado em uma camada de base temporal e uma ou mais camadas de aprimoramento temporais com a seguinte propriedade: deixe que as camadas temporais sejam identificadas por um identificador de camada temporal T, que começa em 0 para a camada de base e aumenta em 1 de uma camada temporal para a próxima; então, para cada número natural, o fluxo de bits que é obtido da remoção das unidades de acesso de todas as camadas temporais com um identificador de camada temporal maior do que forma outro fluxo de bits válido para o decodificador dado. (Schwarz; Marpe; Wiegand, 2007)

Para *codecs* de vídeo híbridos, a escalabilidade temporal pode geralmente ser ativada restringindo a predição compensada por movimento para imagens de referência com um identificador de camada temporal que é menor ou igual ao identificador de camada temporal da imagem a ser prevista.

2.1.2.2 Escalabilidade Espacial

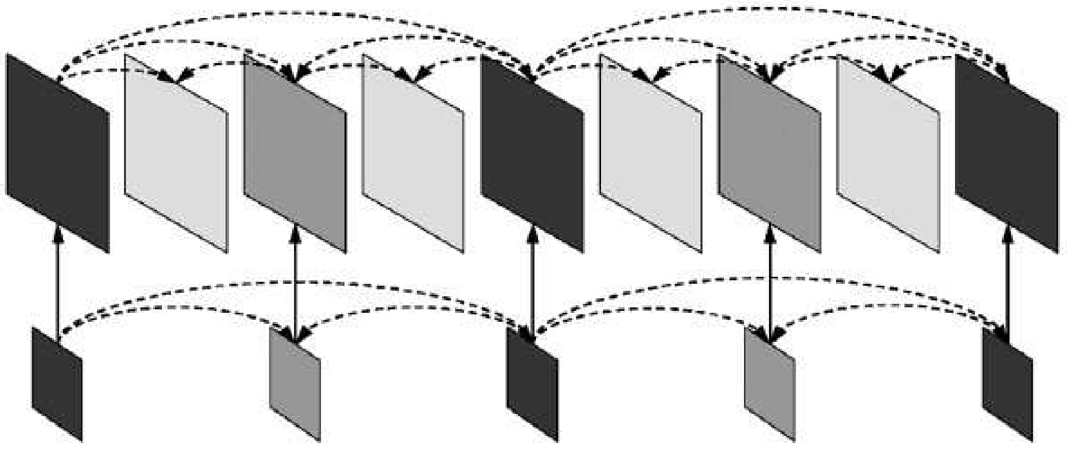
A escalabilidade espacial é quando o tamanho de um vídeo é reduzido para telas menores. No caso um vídeo é feito para uma tela grande, estilo uma televisão e um aplicativo tem que exibir em uma tela de celular. Um *codec* com escalabilidade espacial tem várias resoluções de vídeo disponíveis no mesmo arquivo, diminuindo a quantidade de arquivos armazenados no servidor e permitindo, apenas com a codificação, a escolha de taxas de bits que reduzam ou incrementem a resolução de acordo com o que está sendo exibido.

Para suportar a codificação escalonável espacial, o *SVC* segue a abordagem convencional de codificação multicamadas. Cada camada corresponde a uma resolução espacial suportada e é referida por uma camada espacial ou identificador de dependência D. O identificador de dependência D para a camada de base é igual a 0 e é incrementado em 1 de uma camada espacial para a seguinte. Em cada camada espacial, a predição compensada por movimento e a intra-predição são empregadas como para a codificação de camada única. Mas, a fim de melhorar a eficiência da codificação em comparação ao da transmissão simultânea de diferentes resoluções espaciais, os chamados mecanismos de predição inter-camadas adicionais são incorporados, conforme ilustrado na Figura 2.2. (Schwarz; Marpe; Wiegand, 2007)

O *SVC*, a fim de restringir os requisitos de memória e a complexidade do decodificador, especifica que a mesma ordem de codificação será usada para todas as camadas espaciais com suporte. As representações com diferentes resoluções espaciais para um dado instante de tempo formam uma unidade de acesso e devem ser transmitidas sucessivamente

em ordem crescente de seus respectivos identificadores de camada espacial. Mas, conforme ilustrado na Figura 2.2, as imagens da camada inferior não precisam estar presentes em todas as unidades de acesso, o que torna possível combinar escalabilidade temporal e espacial.(Schwarz; Marpe; Wiegand, 2007)

Figura 2.2 – Estrutura multicamadas com previsão adicional entre camadas para permitir a codificação escalonável espacial.



Fonte: (Wiegand et al., 2003), Adaptado

2.1.2.3 Escalabilidade de nível de qualidade

A escalabilidade de nível de qualidade, frequentemente referida como escalabilidade de relação sinal-ruído *(SNR)*, é bastante utilizada quando um usuário experimenta problemas de conexão e não tem rede suficiente para manter um vídeo em alta qualidade sem ficar com a tela travada. É frequente que na aplicação perceba-se a perda de pacotes e seja reduzida a qualidade do conteúdo exibido para evitar congelamentos e travamentos.

Um exemplo de diagrama de blocos de escalabilidade *SNR* está mostrada na figura 2.3. Nela é mostrada um esquema utilizado em na codificação escalável do H.264.

Figura 2.3 – Estrutura de dois laços para escalabilidade *SNR* em um codificador híbrido.

(Ohm, 2005). Adaptado pelo autor

2.2 *Codecs*

Nesta seção será colocada em questão a fundamentação referente a alguns dos *codecs* mais utilizados no mercado, desde os *codecs* mais antigos aos *codecs* que ainda estão em desenvolvimento.

2.2.1 H.264(AVC)

H.264 é um padrão para compressão de vídeo, baseado no *MPEG-4 Part 10 ou AVC (Advanced Video Coding)*. O padrão foi desenvolvido pela ITU-T Video Coding Experts Group (VCEG) em conjunto com a ISO/IEC MPEG que formaram uma parceria conhecida por Joint Video Team (JVT). A versão final, formalmente chamada por ISO/IEC 14496-10), foi lançada em Maio de 2003.

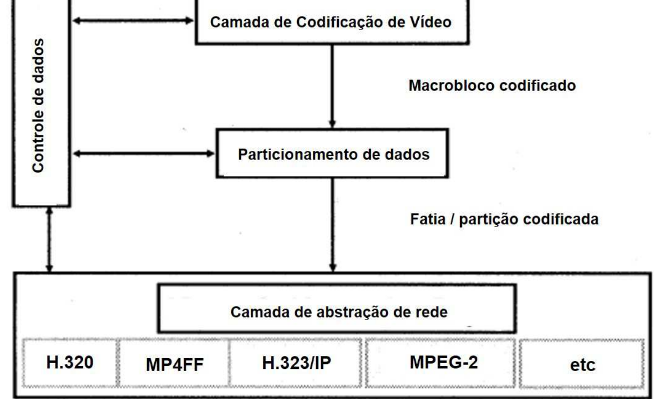
Na época que foi lançado, o padrão foi projetado para soluções técnicas, incluindo pelo menos as seguintes áreas de aplicação:

- Transmissão por cabo, satélite, modem a cabo, DSL, terrestre, etc.
- Armazenamento interativo ou serial em dispositivos ópticos e magnéticos, DVD, etc.
- Serviços de *streaming* de vídeo sob demanda ou multimídia por meio ISDN, modem a cabo, DSL, LAN, redes sem fio, etc.
- Serviços de mensagens multimídia (MMS) sobre *ISDN, DSL, ethernet, LAN*, redes sem fio e móveis, etc.

À época, os criadores do *codec* já pensavam em possíveis novas aplicações a serem implantadas nas redes existentes e planejadas. Isso levantou a questão de como lidar com essa variedade de aplicativos e redes.

Para atender a essa necessidade de flexibilidade e personalização, o design H.264 cobriu uma camada de codificação de vídeo (CCV), que é projetado para representar de forma eficiente o conteúdo do vídeo, e uma camada de abstração de rede (NAL), que formata a representação CCV do vídeo e fornece informações de cabeçalho de maneira apropriada para transporte por uma variedade de camadas de transporte ou meios de armazenamento, assim como é mostrado na figura 2.4. (Wiegand et al., 2003)

Figura 2.4 – Estrutura do codificador de vídeo H.264 / AVC



Fonte: Overview of the H.264/AVC Video Coding Standard, IEEE. Adaptado pelo autor

Na camada de codificação de vídeo (CCV) segue a chamada abordagem de codificação de vídeo híbrida baseada em bloco, em que cada imagem codificada é representados em unidades na forma de bloco de amostras associadas de luma e croma denominadas macroblocos. O algoritmo básico de codificação de fonte é um híbrido de previsão entre imagens para explorar dependências estatísticas temporais e transformar a codificação do residual de previsão para explorar dependências estatísticas espaciais. Não há um único elemento de codificação na CCV que forneça a maior parte da melhoria significativa na eficiência de compressão em relação aos padrões de codificação de vídeo anteriores. É mais uma pluralidade de melhorias menores que se somam ao ganho significativo.

Já na camada de abstração de rede, os dados de vídeo codificados são organizados em unidades NAL, cada uma da qual é efetivamente um pacote que contém um número inteiro de bytes. O primeiro byte de cada unidade é um byte de cabeçalho que contém uma indicação do tipo de dados na unidade NAL e os bytes restantes contêm dados de carga útil do tipo indicado pelo cabeçalho. Os dados de carga útil na unidade NAL são

intercalados conforme necessário com bytes de prevenção de emulação, que são bytes inseridos com um valor específico para se evitar que um determinado padrão de dados denominado prefixo de código de início seja gerado acidentalmente dentro da carga útil.

A definição da estrutura da unidade NAL especifica um formato genérico para uso em sistemas de transporte orientados por pacote e por fluxo de bits, e uma série de unidades NAL geradas por um codificador é referida como um fluxo de unidade NAL.

O padrão h.264 também trouxe inovações comparadas aos padrões antigos, entre essas inovações foi a criação de uma extensão de vídeo escalável (SVC). Essa extensão permitia um mesmo arquivo de vídeo ter várias taxas de quadros por segundo, ou várias qualidades diferentes ou várias resoluções diferentes, tudo isso sem precisar armazenar arquivos diferentes.

Na figura 2.5 podemos entender como funciona essa extensão, de forma básica, ela permite que ao selecionar uma certa quantidade de bits, você possa escolher a forma final do vídeo, se ela vai para uma TV ou para um dispositivo menor como um celular ou um computador com baixa resolução no monitor.

Figura 2.5 – O princípio da codificação de vídeo escalonável no H264.

Fonte: Scalable Video Coding in H.264/AVC, Fraunhofer HHI. Adaptado pelo autor

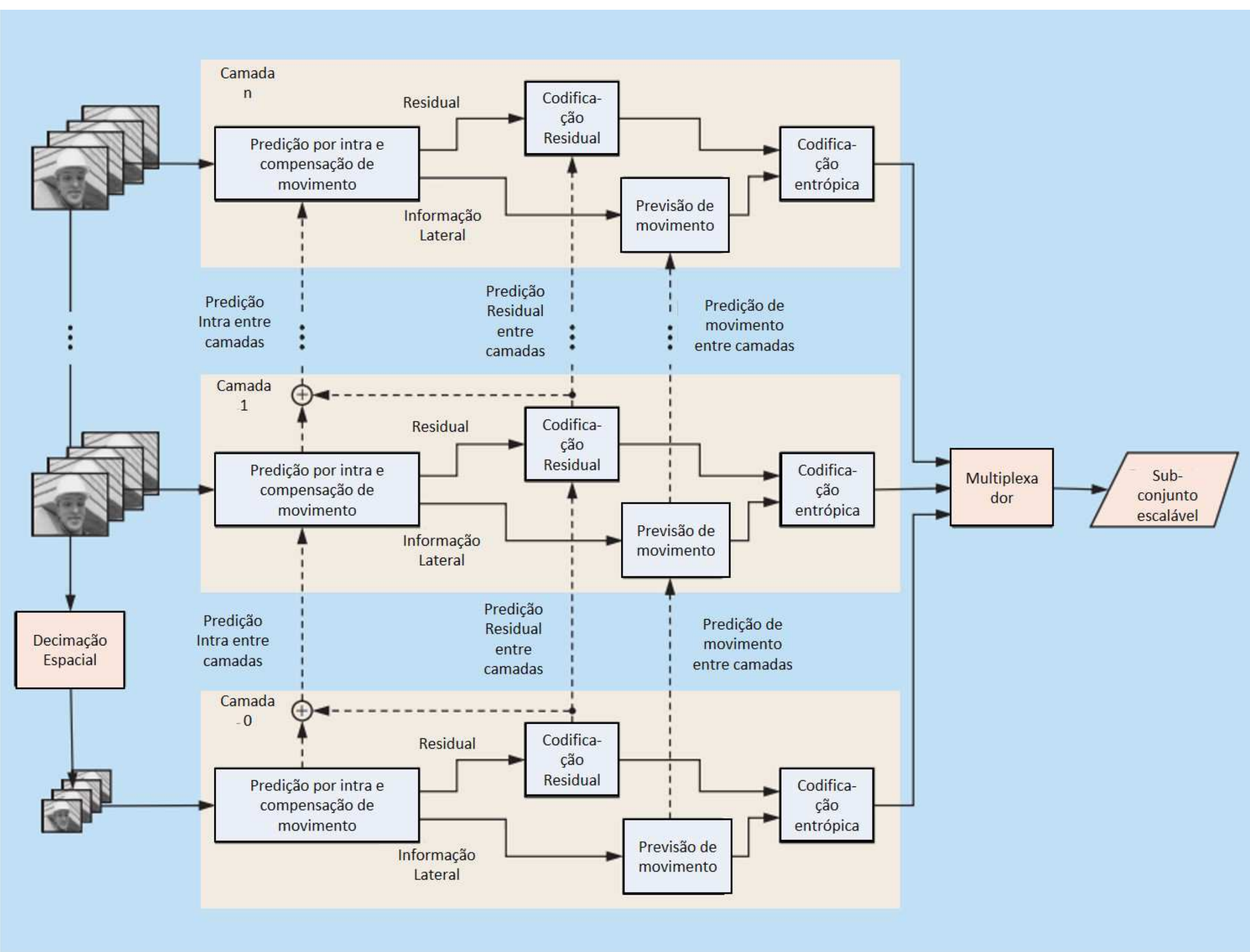
Embora o padrão realmente especifique o processo de decodificação, nos concentramos nas técnicas típicas do codificador para explicar o projeto VCL, uma vez que é mais fácil de entender. Um diagrama de blocos simplificado de um codificador SVC é ilustrado na Figura 2.6. Cada representação da fonte de vídeo com uma resolução espacial particular e fidelidade que está incluída em um fluxo de bits SVC é referido como uma camada e é caracterizado por um identificador de camada. Em cada unidade de acesso, as camadas são codificadas em ordem crescente de seus identificadores de camada.(Schwarz; Wien, 2008)

Para a codificação de uma camada, dados já transmitidos de outra camada com um identificador de camada menor podem ser empregados conforme descrito nos parágrafos a

seguir. A camada a partir da qual prever pode ser selecionada com base na unidade de acesso e é chamada de camada de referência. A camada com um identificador de camada igual a zero, que só pode estar presente em algumas unidades de acesso, é codificada em conformidade com um dos perfis H.264/AVC não escalonáveis e é referida como camada de base. As camadas que empregam dados de outras camadas para codificação são chamadas de camadas de aprimoramento. Uma camada de aprimoramento é chamada de camada de aprimoramento espacial quando a resolução espacial muda em relação à sua camada de referência, e é chamada de camada de aprimoramento de fidelidade quando a resolução espacial é idêntica à de sua camada de referência. (Schwarz; Wien, 2008)

O número de camadas presentes em um fluxo de bits SVC depende das necessidades de um aplicativo. O SVC suporta até 128 camadas em um fluxo de bits. Com os perfis atualmente especificados, o número máximo de camadas de aprimoramento em um fluxo de bits é limitado a 47, e no máximo duas delas podem representar camadas de aprimoramento espacial.(Schwarz; Wien, 2008)

Figura 2.6 – Estrutura do codificador SVC simplificado.



Fonte: (Schwarz; Wien, 2008) adaptado pelo autor

2.2.2 H.265 (HEVC)

O padrão agora conhecido como Codificação de Vídeo de Alta Eficiência (HEVC) reflete a experiência acumulada de cerca de quatro décadas de pesquisa e três décadas de padronização internacional para tecnologia de codificação de vídeo digital. Seu desenvolvimento foi um empreendimento gigantesco que superou os projetos anteriores em termos da quantidade de esforço de engenharia dedicado ao seu design e padronização. O resultado agora é formalmente padronizado como Recomendação ITU-T H.265 e ISO / IEC *International Standard* 23008-2 (MPEG-H parte 2). A primeira versão do HEVC foi concluída em janeiro de 2013 (com aprovação final e publicação formal alguns meses depois - especificamente, a publicação formal da ITU-T foi em junho, e a publicação formal da ISO / IEC foi em novembro). (Sze; Budagavi; Sullivan, 2014)

Como escrito anteriormente, o principal padrão de codificação de vídeo diretamente anterior ao projeto HEVC foi o H.264(AVC), que foi inicialmente desenvolvido no período entre 1999 e 2003 e, em seguida, foi estendido de várias maneiras importantes de 2003-2009. Ele tem sido uma tecnologia habilitadora para vídeo digital em quase todas as áreas que não eram cobertas anteriormente pelo Vídeo H.262 / MPEG-2 e substituiu substancialmente o padrão antigo em seus domínios de aplicação existentes.

É amplamente utilizado para muitas aplicações, incluindo transmissão de sinais de TV de alta definição (HD) por satélite, cabo e sistemas de transmissão terrestre, aquisição de conteúdo de vídeo e sistemas de edição, filmadoras, aplicativos de segurança, Internet e vídeo em rede móvel, discos *Blu-ray* e aplicativos de conversação em tempo real, como bate-papo por vídeo, videoconferência e sistemas de telepresença.

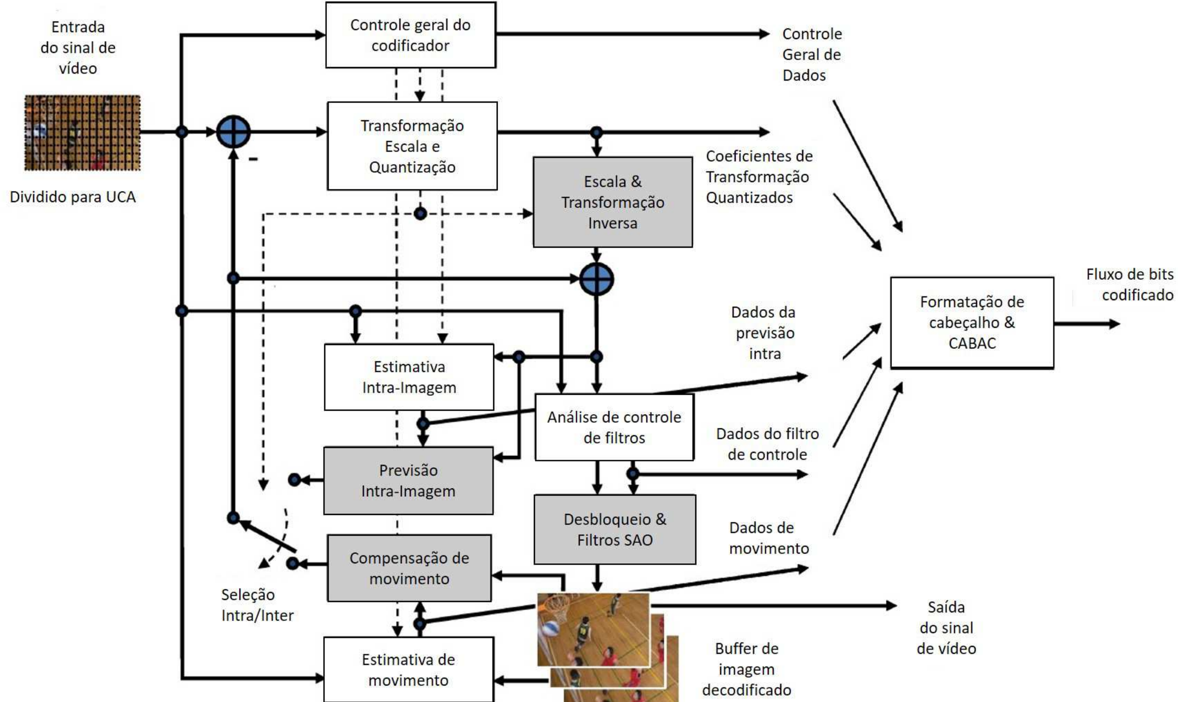
No entanto, uma diversidade crescente de serviços, a popularidade crescente de vídeo HD e o surgimento de formatos além do HD (resoluções 4k e 8k) estão criando necessidades ainda mais fortes de eficiência de codificação superior ao do H.264. A necessidade é ainda mais forte quando uma resolução mais alta é acompanhada por captura e exibição estéreo ou *multiview*. Além disso, o tráfego causado por aplicativos de vídeo direcionados a dispositivos móveis e *tablets*, bem como as necessidades de transmissão de serviços de vídeo sob demanda, estão impondo graves desafios nas redes atuais. Um desejo crescente por maior qualidade e resoluções também está surgindo nos aplicativos móveis.

O HEVC foi projetado para abordar essencialmente todas as aplicações existentes de H.264 e para focar particularmente em duas questões principais: maior resolução de vídeo e maior uso de arquiteturas de processamento paralelo. A sintaxe do HEVC é genérica e também deve ser geralmente adequada para outros aplicativos que não são especificamente mencionados acima.

O padrão HEVC é projetado para atingir vários objetivos, incluindo eficiência de codificação, facilidade de integração do sistema de transporte e resiliência à perda de

dados, bem como implementabilidade usando arquiteturas de processamento paralelo. A camada de codificação de vídeo do HEVC emprega a mesma abordagem híbrida (predição inter/intra e codificação de transformação 2-D) usada em todos os padrões de compressão de vídeo desde H.261. Na figura 2.7 pode ser visto o diagrama de blocos de um codificador de vídeo híbrido, que poderia criar um fluxo de bits em conformidade com o padrão HEVC.

Figura 2.7 – Codificador de vídeo HEVC típico (com elementos de modelagem do decodificador sombreados em cinza claro).



Fonte: (Sullivan et al., 2012) adaptado pelo autor

2.2.3 H.266(VVC)

O H.266(VVC) é a tecnologia de compressão de vídeo mais recente a ser padronizada. Ele foi desenvolvido pela Joint Video Experts Team (JVET) do ITU-T Visual Coding Experts Group (VCEG) e do ISO / IEC Moving Pictures Experts Group (MPEG). A JVET foi fundada como a Equipe Conjunta de Exploração de Vídeo (em codificação de Vídeo do Futuro) da ITU-T VCEG e ISO / IEC MPEG em outubro de 2015. Após uma chamada de propostas bem-sucedida, ela passou para a *Equipe Conjunta de Especialistas em Vídeo* (também abreviado para JVET) em abril de 2018 com a tarefa de desenvolver um novo padrão de codificação de vídeo. O novo padrão de codificação de vídeo foi denominado *Versatile Video Coding* (VVC) e foi finalizado em julho de 2020.

Na figura 2.8 são mostradas algumas das vantagens prometidas para o novo *codec* em fase de lançamento.

Figura 2.8 – Possíveis vantagens do uso do codec H266.

Eficiência do codificador
50% a mais do que o HEVC
Resoluções HD / UHD / 8k
10bit / HDR

Versatilidade
Conteúdo da tela (texto e gráficos)
Mudança de resolução adaptativa
Escalabilidade
Streaming baseado em bloco

Fonte: Fraunhofer HHI, adaptado pelo autor

O VVC foi projetado desde o início para ser eficiente e versátil para atender às necessidades de mídia de hoje. Isso inclui:

- 50% de redução da taxa de bits em relação ao H.265, o padrão predecessor;
- Versatilidade através da codificação eficiente de uma ampla gama de aplicativos e conteúdo de vídeo.
- Vídeo além do padrão e de alta definição, incluindo alta resolução (até 8K ou ainda maior), alta faixa dinâmica (HDR) e ampla gama de cores;
- Conteúdo gerado por computador ou de tela em aplicações de compartilhamento de tela e jogos online;
- Conteúdo envolvente, como vídeo em 360 graus.

Em uma entrevista fornecida pela *Fraunhofer* (uma das empresas parceiras na codificação do VVC), foi falado que por meio da redução dos requisitos de dados, o H.266(VVC) torna a transmissão de vídeo em redes móveis (onde a capacidade de dados é limitada) mais eficiente. Por exemplo, o padrão anterior H.265(HEVC) requeria 10 gigabytes de dados para transmitir um vídeo UHD de 90 minutos. Com esta nova tecnologia, apenas 5 gigabytes de dados são necessários para atingir a mesma qualidade. Como o H.266 foi desenvolvido com conteúdo de vídeo de altíssima resolução em mente, o novo padrão é particularmente benéfico ao fazer *streaming* de vídeos 4K ou 8K em uma TV de tela plana. Além disso, o H.266 é ideal para todos os tipos de imagens em movimento: desde panoramas de vídeo de 360º de alta resolução até conteúdo de compartilhamento de tela.(HHI, 2020)

2.2.4 VP9

VP9 é um formato de codificação de vídeo aberto e livre de royalties desenvolvido pelo *Google*. Ele é o sucessor do VP8 e compete principalmente com a codificação de vídeo de alta eficiência da MPEG (HEVC). No início, VP9 foi usado principalmente na plataforma de vídeo do *Google*, *YouTube*. O surgimento da *Alliance for Open Media*, e seu apoio ao desenvolvimento contínuo do sucessor AV1, do qual o *Google* faz parte, gerou um interesse crescente pelo formato.

Em contraste com HEVC, o suporte a VP9 é comum entre os navegadores modernos. O *Android* tem suporte para VP9 desde a versão 4.4 *KitKat*, enquanto iOS adicionou suporte para VP9 a partir do iOS14. Partes do formato são protegidas por patentes detidas pelo *Google*. A empresa concede o uso gratuito de suas próprias patentes relacionadas com base na reciprocidade, ou seja, desde que o usuário não se envolva em litígios de patentes.

2.2.5 AV1

O AV1 (AOMedia Video 1) é um formato de codificação de vídeo aberto e livre de royalties, inicialmente projetado para transmissões de vídeo pela Internet. Foi desenvolvido como sucessor do VP9 pela *Alliance for Open Media (AOMedia)*, um consórcio fundado em 2015 que inclui empresas de semicondutores, fornecedores de vídeo sob demanda, produtores de conteúdo de vídeo, empresas de desenvolvimento de software e fornecedores de navegadores da web.

A especificação do *bitstream AV1* inclui um *codec* de vídeo de referência. Em 2018, o Facebook conduziu testes que se aproximam das condições do mundo real, o codificador de referência AV1 atingiu 34%, 46.2% e 50.3% mais compactação de dados do que *libvpx-vp9, x264 high profile e x264 main profile*, respectivamente.

Como VP9, mas diferente de H.264(AVC) e H.265(HEVC), AV1 tem um modelo de licenciamento livre de royalties que não impede a adoção em projetos de código aberto.

Em termos de recursos, o AV1 é projetado especificamente para aplicativos em tempo real (especialmente WebRTC) e resoluções mais altas do que os cenários de uso típicos da geração atual(H.264) de formatos de vídeo, onde está espera atingir seus maiores ganhos de eficiência. Portanto, é planejado para oferecer suporte ao espaço de cores da recomendação ITU-R BT.2020 e até 12 bits de precisão por componente de cor. AV1 destina-se principalmente à codificação com perdas, embora a compactação sem perdas também seja suportada. Atualmente, o Youtube forneceu a opção para os usuários escolherem utilizar(ou não) o *codec* AV1 em seus vídeos.

2.3 *WebRTC*

O WebRTC é um novo padrão gerado de um esforço da indústria que estende o modelo de navegação na web. Pela primeira vez, os navegadores são capazes de trocar mídia em tempo real diretamente com outros navegadores de forma ponto a ponto(Loreto; Romano, 2014)

O objetivo do WebRTC é democratizar a comunicação em tempo real. Anteriormente, construir um aplicativo de comunicação de vídeo menor costumava levar meses e envolvia engenharia personalizada para fazer até mesmo o menor dos aplicativos. No entanto, agora podemos fazer isso na metade do tempo ou até menos. Isso também traz a comunidade de código aberto para a comunicação em tempo real (Ristic, 2015)

Uma aplicação web que utilize o WebRTC tem como interface mostrada na Figura 2.9, na qual apresenta-se uma aplicação com várias camadas em seu interior. Cada camada tem sua função, cujo usuário não necessita baixar nenhum *plugin*, já que tudo é processado pelo próprio *browser*.

Figura 2.9 – Aplicação Web utilizando o WebRTC

Fonte: (Ristic, 2015), Adaptado pelo autor

3 Metodologia

Neste capítulo serão descritos os procedimentos utilizados no desenvolvimento de um sistema que utiliza WebRTC. O sistema aqui descrito permite a comunicação entre vários usuários utilizando conexão P2P.

3.1 *App* de conversação com *WebRTC*

Uma das possíveis aplicações do WebRTC é de um sistema que seria muito parecido com *Zoom*, *Google Meet*, entre outros aplicativos de conversação. A principal diferença desse sistema seria que, por ser desenvolvido todo com base em WebRTC, não seria necessário ter cadastro ou instalar qualquer software. Outra diferença bem perceptível é a impossibilidade de se fazerem alterações nas câmeras, devido a ser uma aplicação executada diretamente no *browser*.

Essa aplicação funcionaria da mesma forma que uma reunião do *Zoom*: usuários através de um mesmo *link* da sala poderiam ingressar para conversar e compartilhar tanto a câmera como também a tela de seu computador.

3.1.1 Desenvolvimento do app

Nesta seção serão descritos os procedimentos utilizados para realizar a criação desse sistemas. O sistema foi desenvolvido inteiramente utilizando *WebRTC, JavaScript(js)* e alguns módulos JS, tais como *nodejs* e *peerjs*.

Na figura 3.1 é mostrado o diretório de trabalho da aplicação em desenvolvimento. Nesse diretório podemos ver 3 pastas, que são: *node_modules, public* e *views*. A pasta *node_modules* continha todos os módulos utilizados na execução do programa, a pasta *public* continha o arquivo de script e o estilo utilizando na modelagem da página web. Já a pasta *view* continha o arquivo room.ejs, que definia todas as interfaces utilizadas na página web.

Figura 3.1 – Representação do diretório de uma aplicação de chat utilizando WebRTC

```
KAIO STREAMCHAT
  node_modules
  public
    script.js
    style.css
  views
    room.ejs
  package-lock.json
  package.json
  README.md
  server.js
```

Fonte: Autoria própria

Na figura 3.2 é mostrado o código responsável pela parte do servidor. Na parte inicial da figura podemos ver como são declaradas as solicitações de outros códigos vindos dos módulos que estavam na pasta *node_modules*. Toda a parte de definição dos servidores foi feita utilizando dos módulos *express e socket.io*. No centro do código estavam as definições do servidor e de como seriam criadas as salas de conversas.

Figura 3.2 – Representação do código responsável pela parte do servidor.

```
const express = require('express')
const app = express()
 const cors = require('cors')
 app.use(cors())
const server = require('http').Server(app)
const io = require('socket.io')(server)
const { ExpressPeerServer } = require('peer');
const peerServer = ExpressPeerServer(server, {
  debug: true
});
const { v4: uuidV4 } = require('uuid')

app.use('/peerjs', peerServer);

app.set('view engine', 'ejs')
app.use(express.static('public'))

app.get('/', (req, res) => {
  res.redirect(`/${uuidV4()}`)
})

app.get('/:room', (req, res) => {
  res.render('room', { roomId: req.params.room })
})

io.on('connection', socket => {
  socket.on('join-room', (roomId, userId) => {
    socket.join(roomId)
    socket.to(roomId).broadcast.emit('user-connected', userId);
    // messages
    socket.on('message', (message) => {
      //send message to the same room
      io.to(roomId).emit('createMessage', message)
  });

    socket.on('disconnect', () => {
      socket.to(roomId).broadcast.emit('user-disconnected', userId)
    })
  })
})

server.listen(3000)
```

Fonte: Autoria própria

Ao final do código mostrado na figura 3.2, é possível localizar o método de conexão

dos usuários com as salas, utilizava das funções do *socket.io*. O *socket.io* é uma implementação em node para *web socket*, ou seja, uma forma de fazer comunicação em tempo real, mas mais importante que isso é sua possibilidade de *fallBack*. (FARIA, 2021)

Na ultima linha do código do servidor, é mostrada a porta que foi utilizada para conexão dos usuários. Foi utilizada a porta 3000. No inicio do desenvolvimento, a conexão era permitida apenas de forma local, via *localhost:porta*. Ao final foi possível permitir outras conexões utilizando-se de um servidor alugado e *hosteado* pela *DigitalOcean*.

Na figura 3.3 são mostradas as linhas de códigos contidas no *package.json*. Essas linhas de código basicamente serviam para mostrar as versões das dependências utilizadas no desenvolvimento do app e para mostrar a forma de inicialização da aplicação, que seria através de uma linha de comando *"start"*, que, ao iniciar no *prompt* de comando, o sistema reconhece a inicialização do *"node server.js"*

Figura 3.3 – Representação do código dos pacotes do app.

```
{} package.json > ...
1   {
2     "name": "kaio_webchat_app",
3     "version": "1.1.0",
4     "description": "",
5     "main": "index.js",
      ▷ Debug
6     "scripts": {
7       "test": "echo \"Error: no test specified\" && exit 1",
8       "start": "node server.js"
9     },
10    "keywords": [],
11    "author": "",
12    "license": "ISC",
13    "dependencies": {
14      "body-parser": "^1.19.0",
15      "cors": "^2.8.5",
16      "ejs": "^3.1.3",
17      "express": "^4.17.1",
18      "peer": "^0.5.3",
19      "peerjs": "^1.3.2",
20      "socket.io": "^2.4.1",
21      "uuid": "^8.3.2"
22    }
23  }
24
```

Fonte: Autoria própria

Nas próximas linhas serão mostradas figuras de partes do código contidas no *script.js*. Essas linhas de código tinham como função mostrar toda a parte de captura da câmera de cada integrante da chamada, a forma de conexão entre os usuários, a maneira da qual o usuário conseguia usar as funções de parar (ou de ativar) o vídeo e desativar (ou reativar) o microfone. Também permitia mostrar a forma de captura do chat dos usuários.

- Captura de câmera
- Conexão entre novos usuários
- Captura de tela

- Funções microfone e câmera

Captura de câmera

A captura da câmera do usuário e dos convidados era feita utilizando o WebRTC e de uma função que tinha como objetivo obter a informação recebida do navegador e transferi-la para um elemento de vídeo da página principal.

Figura 3.4 – Representação do código de captura da câmera e áudio do navegador.

```
const myVideo = document.createElement('video')


myVideo.muted = true;
const peers = {}
navigator.mediaDevices.getUserMedia({
  video: true,
  audio: true
}).then(stream => {
  myVideoStream = stream;
  addVideoStream(myVideo, stream)
  myPeer.on('call', call => {
    call.answer(stream)
    const video = document.createElement('video')
    call.on('stream', userVideoStream => {
      addVideoStream(video, userVideoStream)
    })
  })

  socket.on('user-connected', userId => {
    connectToNewUser(userId, stream)
  })
```

Fonte: Autoria própria

Na figura 3.4 é mostrado que a captura foi feita com *navigator.mediaDevices.getUserMedia*, essa função do navegador retornava uma *stream* que era processada através da função *addVideoStream*. Na figura 3.5 podemos ver como era feito o processamento da *stream*. Nela é mostrada que foi capturada a parte de vídeo da *stream*, que foi adicionada à tela em uma janela de vídeo a partir do *videoGrid.append*.

Figura 3.5 – Representação do código de adição de uma *stream* de vídeo.

```
function addVideoStream(video, stream) {
  video.srcObject = stream
  video.addEventListener('loadedmetadata', () => {
    video.play()
  })
  videoGrid.append(video)
}
```

Fonte: Autoria própria

Após processar essa *stream*, o código chamava a função do *peerjs* para responder à solicitação do navegador e assim poder adicionar a imagem à tela. No fim da figura, também podemos ver o início do código de resposta à conexão de um novo usuário, nesse

código podemos ver a chamada da função *connectToNewUser*, que será explicada na próxima seção.

Conexão entre novos usuários

A conexão entre novos usuários foi feita utilizando as funções do *socket.io* através da função *connectToNewUser*. Na figura 3.6 podemos ver que essa função recebia como parâmetros o *id* do usuário conectado e a *stream* captada através do WebRTC. Após receber os parâmetros, era gerado um novo *id* de vídeo e chamava-se a função *addVideoStream* para permitir-se que o vídeo fosse encaminhado para uma nova janela no navegador. É importante notar que essa função também faz com que o vídeo do usuário seja removido após desconectar-se, impedindo assim que reste um vídeo sem fonte de dados nas janelas do navegador.

Figura 3.6 – Representação do código de conexão de um novo usuário

```
function connectToNewUser(userId, stream) {
  const call = myPeer.call(userId, stream)
  const video = document.createElement('video')
  call.on('stream', userVideoStream => {
    addVideoStream(video, userVideoStream)
  })
  call.on('close', () => {
    video.remove()
  })

  peers[userId] = call
}
```

Fonte: Autoria própria

Captura de tela

A captura de tela também é realizada utilizando uma das variáveis de desenvolvedor dos navegadores, a partir do *WebRTC*. Na figura 3.7 pode ser vista a forma com que o código realizava a captura da tela. Primeiramente utiliza-se da função *navigator.mediaDevices.getDisplayMedia*, que retorna ao usuário uma janela de opções para se escolher o que será transmitido. Após o usuário escolher o que será transmitido, é então chamada uma função do *peerjs* para solicitar a criação de um novo conteúdo de vídeo a partir do que foi selecionado pelo usuário.

Figura 3.7 – Representação do código de captura de tela.

```
const shareScreen = () => {

  const myVideo2 = document.createElement('video')
  myStream = navigator.mediaDevices.getDisplayMedia({video: true}).then(stream => {
    displayMediaStream = stream;
    addVideoStream(myVideo2, stream)
    })
    myPeer.on('call', call => {
      call.answer(stream)
      const video = document.createElement('video')
     call.on('stream', userVideoStream => {
       addVideoStream(video, userVideoStream)
      })

     })

  }
```

Fonte: Autoria própria

Funções de microfone e câmera

As funções de silenciar (ou liberar) o microfone e parar de compartilhar o vídeo (ou retornar o compartilhamento) foram feitas a partir das funções *getVideoTracks e getAudioTracks*. Na figura 3.8 é mostrado o trecho de código que realizava as ações.

Figura 3.8 – Representação do código de funções de microfone e câmera.

```
const muteUnmute = () => {
  const enabled = myVideoStream.getAudioTracks()[0].enabled;
  if (enabled) {
    myVideoStream.getAudioTracks()[0].enabled = false;
    setUnmuteButton();
  } else {
    setMuteButton();
    myVideoStream.getAudioTracks()[0].enabled = true;
  }
}

const playStop = () => {
  console.log('object')
  let enabled = myVideoStream.getVideoTracks()[0].enabled;
  if (enabled) {
    myVideoStream.getVideoTracks()[0].enabled = false;
    setPlayVideo()
  } else {
    setStopVideo()
    myVideoStream.getVideoTracks()[0].enabled = true;
  }
}
```

Fonte: Autoria própria

3.1.2 Conexão com servidores externos

Nesta seção serão comentados os procedimentos para usuários de outras redes realizarem a conexão com o servidor. Antes da realização desses procedimentos, só era

possível realizar a conexão estando na mesma rede em que estava sendo executado o programa.

Para permitir a conexão das outras redes, foi necessário a criação e configuração de um servidor que executasse continuamente o código com ip-fixo. Tentou-se a conexão diretamente na rede local, mas não foi possível devido ao IP dinâmico. Logo, a solução implementada foi o aluguel de um servidor *ubuntu 18* que tivesse ip-fixo.

Após realizadas pesquisas em busca do melhor custo-benefício para aplicações de pequeno porte, foi escolhida a *DigitalOcean* como empresa para *hostear* o servidor durante o desenvolvimento das atividades. Após realizar a configuração no site, foi fornecido um ip-fixo e um *login* para acessar o servidor.

A conexão com o servidor era feita via protocolo SSH e utilizando do *software Putty*. Já o acesso para realizar a transferência dos arquivos foi feita via protocolo SSH e utilizando-se do *software FileZilla*.

Após ter sido efetuada a conexão, foi necessário realizar a configuração de um servidor *Apache* para tentar a obtenção de um certificado *https* para realizar a conexão segura entre diferentes pontos. Após uma pesquisa entre amigos, foi possível realizar a conexão de pessoas utilizando o site *kaiotcc.gabr.tel/id*. Onde o ID usado era fornecido por outras pessoas ou criado pelo usuário para ingressar em uma nova sala.

3.1.3 Execução do aplicativo

Para realizar a execução do aplicativo no servidor, foi necessário realizar a abertura de 2 terminais para manter em execução os comandos necessários. Na figura 3.9 pode ser vista a forma de inicialização do aplicativo. No primeiro terminal primeiro era acessada a pasta em que estavam os arquivos, e após isso era iniciado o aplicativo com o comando *sudo npm run start*. Na figura 3.10 pode ser visto o segundo terminal no qual era executado o comando *sudo peerjs –port 3001*, que servia para iniciar o servidor que realizaria a comunicação entre os usuários diferentes.

Figura 3.9 – Representação do terminal de início do código.

```
kaiorf@-s-1vcpu-1gb-nyc3-01: ~/TCC e Estagio/Kaio StreamChat
10 updates can be applied immediately.
1 of these updates is a standard security update.
To see these additional updates run: apt list --upgradable


Last login: Tue May 18 19:46:26 2021 from 187.115.177.141
kaiorf@-s-1vcpu-1gb-nyc3-01:~$ ls
 Python1  'TCC e Estagio'   lalala
kaiorf@-s-1vcpu-1gb-nyc3-01:~$ cd TCC\ e\ Estagio/
kaiorf@-s-1vcpu-1gb-nyc3-01:~/TCC e Estagio$ cd
kaiorf@-s-1vcpu-1gb-nyc3-01:~$ cd TCC\ e\ Estagio/
kaiorf@-s-1vcpu-1gb-nyc3-01:~/TCC e Estagio$ ls
 GitHub                 NodeWebRTC     Teste               WebRTCStreaming
'Kaio StreamChat'       NovoProjeto    TestePythonStream   WebStreamChat
'Kaio StreamChat.zip'   StreamChat     WebRTCPython        novoteste
 Linux                  StreamNoChat   WebRTCStream
kaiorf@-s-1vcpu-1gb-nyc3-01:~/TCC e Estagio$ cd Kaio\ StreamChat/
kaiorf@-s-1vcpu-1gb-nyc3-01:~/TCC e Estagio/Kaio StreamChat$ ls
README.md     package-lock.json  public     views
node_modules  package.json       server.js
kaiorf@-s-1vcpu-1gb-nyc3-01:~/TCC e Estagio/Kaio StreamChat$ sudo npm run start
[sudo] password for kaiorf:

> kaiostream@1.0.0 start /home/kaiorf/TCC e Estagio/Kaio StreamChat
> node server.js

^Ckaiorf@-s-1vcpu-1gb-nyc3-01:~/TCC e Estagio/Kaio StreamChat$ sudo npm run start
```

Fonte: Autoria própria

Figura 3.10 – Representação do terminal de início do servidor de comunicação.

```
kaiorf@-s-1vcpu-1gb-nyc3-01: ~
 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/advantage

  System information as of Wed May 19 20:01:45 UTC 2021

  System load:  0.09               Processes:           98
  Usage of /:   10.4% of 24.06GB   Users logged in:     1
  Memory usage: 22%                IP address for eth0: 68.183.150.164
  Swap usage:   0%                 IP address for eth1: 10.108.0.2

 * Canonical Livepatch is available for installation.
   - Reduce system reboots and improve kernel security. Activate at:
     https://ubuntu.com/livepatch

10 updates can be applied immediately.
1 of these updates is a standard security update.
To see these additional updates run: apt list --upgradable

New release '20.04.2 LTS' available.
Run 'do-release-upgrade' to upgrade to it.


Last login: Wed May 19 19:59:17 2021 from 187.115.177.141
kaiorf@-s-1vcpu-1gb-nyc3-01:~$ sudo peerjs --port 3001
[sudo] password for kaiorf:
Started PeerServer on ::, port: 3001, path: / (v. 0.6.1)
```

Fonte: Autoria própria

Os resultados do aplicativo estarão demonstrados na seção 4 deste relatório.

4 Resultados

Nessa seção serão comentados os resultados das aplicações desenvolvidas utilizando WebRTC.

4.1 *App* de conversação com *WebRTC*

Como visto na seção anterior, o primeiro aplicativo desenvolvido tinha intenção de se assemelhar aos aplicativos de chamadas de vídeo *Zoom* e *Google Meet*. Na figura 4.1 podemos ver a tela do sistema exibido quando apenas o *host* estava na chamada e com o vídeo aberto. Nessa figura, podemos ver as funções que foram citadas anteriormente, tais como desativar microfone, parar vídeo e compartilhar tela.

Figura 4.1 – Representação da página *web* contando apenas com a presença do *host*.

Fonte: Autoria própria

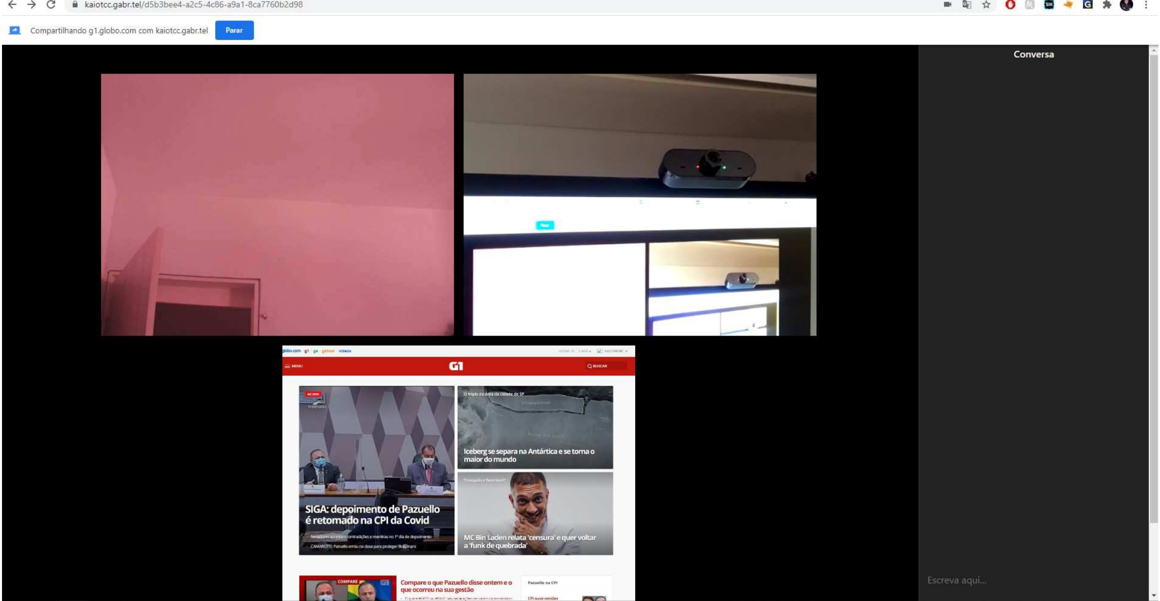
Após um usuário iniciar uma chamada com outro usuário, é possível falar, ouvir e conversar com perfeita qualidade de áudio. Na figura 4.2 pode ser visto o resultado de uma chamada entre 2 usuários diferentes, inclusive com deles utilizando o celular para acessar o sistema.

Figura 4.2 – Representação da página *web* com a presença do *host* e de outro usuário.

Fonte: Autoria própria

Após a chamada ser estabelecida, quando necessário realizar o compartilhamento de tela, a interface *web* se adapta para surgirem mais conexões. O resultado apresentando 2 usuários e uma tela sendo compartilhada pode ser visto na figura 4.3.

Figura 4.3 – Representação da página web com múltiplos participantes



Fonte: Autoria própria

Na figura 4.4 pode ser visto o funcionamento da função parar vídeo. Ao clicar na função, o usuário continuava na chamada ouvindo e se comunicando com todos, apenas com a diferença de que o vídeo estava interrompido. Podemos ver também a interação na página de conversa, na qual os usuários podem mandar mensagens e essas são recebidas pelos dois lados.

Figura 4.4 – Representação da página web com usuário sem vídeo

Fonte: Autoria própria

4.1.1 Problemas encontrados

No desenvolvimento da plataforma, foram encontrados alguns problemas que poderiam ser resolvidos com desenvolvimento mais aprimorado utilizando-se de conhecimentos de *javascript e nodejs*. Os problemas poderiam ser solucionados caso fossem escolhidas outras plataformas que usam *WebRTC*. Estes problemas podem ser listados:

- Impossibilidade de se priorizar conexões-alvo
- Difícil identificação de múltiplos usuários
- Inconsistências de exibição na comunicação entre múltiplos usuários

Em relação à falta de prioridade, pôde-se notar que ao compartilhar tela, a imagem ficava do mesmo tamanho que o das câmeras dos usuários, o que dificulta a leitura do conteúdo compartilhado, diferente de quando compartilhamos tela nas plataformas *Zoom/GoogleMeet*, em que a tela é exibida num tamanho notadamente maior que o das câmeras.

Outro problema notado foi a ausência de identificação nos usuários ao utilizarem o chat: todos apresentavam *ids* diferentes, mas não foi possível mapeá-las em nomes identificadores. A solução seria de ser criada uma função que, na entrada de um novo usuário, lhe solicitasse um nome de usuário e o armazenasse em uma variável própria.

Um outro problema encontrado foi percebido quando um mesmo usuário deseja transmitir sua câmera e a tela do seu computador ao mesmo tempo. Como a conexão deste aplicativo com o WebRTC foi feita em P2P e utilizando-se do *nodejs*, para se transmitir a tela do computador era criada uma nova conexão: portanto quem já estava na chamada não recebia a transmissão da tela, apenas da câmera.

4.1.2 Comparação com outros aplicativos

Uma das principais diferenças entre este aplicativo desenvolvido por mim e outros já existentes consiste em ser no seu todo baseado em *web*, logo não é necessário instalar nenhum software, ou *plug-in* para executá-lo. Outra diferença está no fator de que não se exige nenhum cadastro para que a conversação se inicie.

Uma das desvantagens do *app*, é que, devido a ser todo feito com base em *WebRTC*, não foi possível permitir alterações nas câmeras tais como adicionar efeitos, remover plano de fundo ou outras alterações.

Outra desvantagem observada é que não foi possível exigir uma senha para *login* na sala. Por exemplo, se alguém possuir o mesmo link para a sala que você, não é possível remover o usuário, sendo necessário criar outra sala.

4.1.3 Melhorias futuras

Uma das possíveis melhorias deste aplicativo consiste em criar uma pré-interface em que o usuário possa inserir o código da sala em que estivesse tentando ingressar, além da possibilidade de se criarem salas privadas para que, além do código, seja necessário a validação da senha criada anteriormente.

Outra melhoria possível é de se substituir a conexão *P2P* por outro tipo de conexão com servidor, permitindo por exemplo a conexão múltipla entre vários usuários e o compartilhamento de tela com mais usuários.

4.2 Trabalhos futuros

Um dos possíveis trabalhos possíveis de serem feitos com base em *codecs* e *WebRTC* é o de comparação a nível de rede na questão de simulação de transmissão em grande escala de vídeos e *streams* com utilização de simuladores de rede.

5 Conclusões

Este trabalho teve como objetivo demonstrar um pouco das tecnologias envolvidas nos serviços de *streaming* e do armazenamento de vídeos, além de demonstrar a importância dos estudos sobre codificação de vídeo, um fator essencial para usuários e empresas que trabalham com armazenamento ou edição de vídeo no que se diz respeito a otimizar o consumo de dados e de espaço.

No trabalho também pôde-se mostrar que é possível construir aplicativos, exibir vídeos e realizar outras funções sem a necessidade de um *software* externo, utilizando apenas do próprio *browser*. Isso só foi possível de ser realizado com as tecnologias que envolvem a comunicação em tempo real com a web*(WebRTC)*.

Outra notável importância no trabalho foi a de demonstrar a viabilidade de uma aplicação de chamadas de vídeo e áudio destinada ao fácil acesso de usuários leigos, dado que basta que se acesse através de um mesmo link da chamada, gerado pelo *host*.

Referências Bibliográficas

FARIA, T. *Comunicação em tempo-real com Node e Socket.io*. devpleno, 2021. Disponível em: <https://devpleno.com/socket-io-parte1/>.

HHI, F. *Fraunhofer HHI is proud to present the new state-of-the-art in global video coding: H.266/VVC brings video transmission to new speeds*. 2020. Disponível em: <https://www.digitalmedia.fraunhofer.de/en/mediainformation/PM_2020/vvc-h266.html>.

Loreto, S.; Romano, S. P. *Real-Time Comunnication with WebRTC: Peer-to-Peer in the Browser*. [S.l.]: O' Reilly, 2014. v. 1.

Ohm, J. . Advances in scalable video coding. *Proceedings of the IEEE*, v. 93, n. 1, p. 42–56, 2005.

Ristic, D. *Learning WebRTC*. [S.l.]: Packt Publishing, 2015. v. 1.

Schwarz, H.; Marpe, D.; Wiegand, T. Overview of the scalable video coding extension of the h.264/avc standard. *IEEE Transactions on Circuits and Systems for Video Technology*, v. 17, n. 9, p. 1103–1120, 2007.

Schwarz, H.; Wien, M. The scalable video coding extension of the h.264/avc standard [standards in a nutshell]. *IEEE Signal Processing Magazine*, v. 25, n. 2, p. 135–141, 2008.

Sullivan, G. J. et al. Overview of the high efficiency video coding (hevc) standard. *IEEE Transactions on Circuits and Systems for Video Technology*, v. 22, n. 12, p. 1649–1668, 2012.

Sze, V.; Budagavi, M.; Sullivan, G. *High Efficiency Video Coding*. [S.l.]: Springer, 2014. v. 1.

Wiegand, T. et al. Overview of the h.264/avc video coding standard. *IEEE Transactions on Circuits and Systems for Video Technology*, v. 13, n. 7, p. 560–576, 2003.

Anexos

ANEXO A – Código do servidor

```
    const express = require('express')
const app = express()
 const cors = require('cors')
 app.use(cors())
const server = require('http').Server(app)
const io = require('socket.io')(server)
const { ExpressPeerServer } = require('peer');
const peerServer = ExpressPeerServer(server, {
  debug: true
});
const { v4: uuidV4 } = require('uuid')

app.use('/peerjs', peerServer);

app.set('view engine', 'ejs')
app.use(express.static('public'))

app.get('/', (req, res) => {
  res.redirect(`/${uuidV4()}`)
})

app.get('/:room', (req, res) => {
  res.render('room', { roomId: req.params.room })
})

io.on('connection', socket => {
  socket.on('join-room', (roomId, userId) => {
    socket.join(roomId)
    socket.to(roomId).broadcast.emit('user-connected', userId);
    // messages
    socket.on('message', (message) => {
      //send message to the same room
      io.to(roomId).emit('createMessage', message)
  });
```

```
    socket.on('disconnect', () => {
      socket.to(roomId).broadcast.emit('user-disconnected', userId)
    })
  })
})

server.listen(3000)
```

ANEXO B – Código do script

```
const socket = io('/', {secure: false, transports: ['polling']})
const videoGrid = document.getElementById('video-grid')
const myPeer = new Peer(undefined, {
  host: '/',
  port: ''
})

let myVideoStream;
let displayMediaStream;

const myVideo = document.createElement('video')


myVideo.muted = true;
const peers = {}
navigator.mediaDevices.getUserMedia({
  video: true,
  audio: true
}).then(stream => {
  myVideoStream = stream;
  addVideoStream(myVideo, stream)
  myPeer.on('call', call => {
    call.answer(stream)
    const video = document.createElement('video')
    call.on('stream', userVideoStream => {
      addVideoStream(video, userVideoStream)
    })
  })

  socket.on('user-connected', userId => {
    connectToNewUser(userId, stream)
  })

  let text = $("input");
```

```
  $('html').keydown(function (e) {
    if (e.which == 13 && text.val().length !== 0) {
      socket.emit('message', text.val());
      text.val('')
    }
  });
  socket.on("createMessage", message => {
    $("ul").append(`<li class="message"><b>user</b><br/>${message}</li>`);
    scrollToBottom()
  })
})

socket.on('user-disconnected', userId => {
  if (peers[userId]) peers[userId].close()
})

myPeer.on('open', id => {
  socket.emit('join-room', ROOM_ID, id)
})

function connectToNewUser(userId, stream) {
  const call = myPeer.call(userId, stream)
  const video = document.createElement('video')
  call.on('stream', userVideoStream => {
    addVideoStream(video, userVideoStream)
  })
  call.on('close', () => {
    video.remove()
  })

  peers[userId] = call
}



function addVideoStream(video, stream) {
  video.srcObject = stream
  video.addEventListener('loadedmetadata', () => {
    video.play()
```

```
  })
  videoGrid.append(video)
}



const shareScreen = () => {

  const myVideo2 = document.createElement('video')
  myStream = navigator.mediaDevices.getDisplayMedia({video: true}).
  then(stream => {
    displayMediaStream = stream;
    addVideoStream(myVideo2, stream)
    })
    myPeer.on('call', call => {
      call.answer(stream)
      const video = document.createElement('video')
     call.on('stream', userVideoStream => {
       addVideoStream(video, userVideoStream)
      })

     })

  }

const scrollToBottom = () => {
  var d = $('.main__chat_window');
  d.scrollTop(d.prop("scrollHeight"));
}


const muteUnmute = () => {
  const enabled = myVideoStream.getAudioTracks()[0].enabled;
  if (enabled) {
    myVideoStream.getAudioTracks()[0].enabled = false;
    setUnmuteButton();
  } else {
    setMuteButton();
    myVideoStream.getAudioTracks()[0].enabled = true;
```

```
  }
}

/*const silence = () => {
  const enabled = userVideoStream.getAudioTracks()[0].enabled;
  if (enabled) {
    userVideoStream.getAudioTracks()[0].enabled = false;
    setUnSilenceButton();
  } else {
    setSilenceButton();
    userVideoStream.getAudioTracks()[0].enabled = true;
  }
}*/

const playStop = () => {
  console.log('object')
  let enabled = myVideoStream.getVideoTracks()[0].enabled;
  if (enabled) {
    myVideoStream.getVideoTracks()[0].enabled = false;
    setPlayVideo()
  } else {
    setStopVideo()
    myVideoStream.getVideoTracks()[0].enabled = true;
  }
}

const setMuteButton = () => {
  const html = `
    <i class="fas fa-microphone"></i>
    <span>Desativar microfone</span>
  `
  document.querySelector('.main__mute_button').innerHTML = html;
}

const setUnmuteButton = () => {
  const html = `
    <i class="unmute fas fa-microphone-slash"></i>
    <span>Ativar microfone</span>
  `
```

```
  document.querySelector('.main__mute_button').innerHTML = html;
}
/*
const setSilenceButton = () => {
  const html = `
    <i class="fas fa-volume-off"></i>
    <span>Desativar microfone</span>
  `
  document.querySelector('.main__silence_button').innerHTML = html;
}

const setUnSilenceButton = () => {
  const html = `
    <i class=" fas fas fa-volume-up"></i>
    <span>Ativar microfone</span>
  `
  document.querySelector('.main__silence_button').innerHTML = html;
}
*/
const setStopVideo = () => {
  const html = `
    <i class="fas fa-video"></i>
    <span>Parar vídeo</span>
  `
  document.querySelector('.main__video_button').innerHTML = html;
}

const setPlayVideo = () => {
  const html = `
  <i class="stop fas fa-video-slash"></i>
    <span>Ligar vídeo</span>
  `
  document.querySelector('.main__video_button').innerHTML = html;
}
```

ANEXO C – Código da página html

```
 <!DOCTYPE html>
<html lang="en">
<head>
   <meta charset="UTF-8">
   <meta name="viewport" content="width=device-width, initial-scale=1.0">
   <title>Document</title>
   <script>
      const ROOM_ID = "<%= roomId %>"
   </script>
   <link rel="stylesheet" href="https://stackpath.bootstrapcdn.com/
   bootstrap/4.5.0/css/bootstrap.min.css"integrity=
   "sha384-9aIt2nRpC12Uk9gS9baDl411NQApFmC2
   6EwAOH8WgZl5MYYxFfc+NcPb1dKGj7Sk" crossorigin="anonymous">

   <script defer src="https://unpkg.com/peerjs@1.3.1/
   dist/peerjs.min.js"></script>

   <script src="https://code.jquery.com/jquery-3.5.1.slim.min.js"
   integrity="sha384-DfXdz2htPH0lsSSs5nCTpuj/
   zy4C+OGpamoFVy38MVBnE+IbbVYUew+OrCXaRkfj"
   crossorigin="anonymous"></script>

   <script src="https://cdn.jsdelivr.net/npm/popper.js@1.16.0
   /dist/umd/popper.min.js"integrity=
   "sha384-Q6E9RHvbIyZFJoft+2mJbHaEWldlvI9
   IOYy5n3zV9zzTtmI3UksdQRVvoxMfooAo"
   crossorigin="anonymous"></script>

   <script src="/socket.io/socket.io.js" defer></script>

   <link rel="stylesheet" href="style.css">

   <script src="https://kit.fontawesome.com/c939d0e917.js"></script>
   <script src="script.js" defer></script>
   <style>
```

```
      #video-grid{
         display: flex;
         justify-content: center;
         flex-wrap: wrap;
      }
      video{
         height: 400px;
         width: 533px;
         object-fit: cover;
         padding: 8px;
      }

   </style>
</head>
<body>
<div class="main">
<div class="main__left">
<div class="main__videos">
<div id="video-grid">

</div>
</div>
<div class="main__controls">
<div class="main__controls__block">
<div onclick="muteUnmute()"
class="main__controls__button main__mute_button">
<i class="fas fa-microphone"></i>
<span>Desativar Microfone</span>
</div>
<div onclick="playStop()"
class="main__controls__button main__video_button" >
    <i class="fas fa-video"></i>
        <span>Parar vídeo</span>
</div>
<div class="main__controls__button main__silence_button" >
    <i class="fas fa-volume-up"></i>
    <span>Desligar áudio</span>
</div>
<div onclick="shareScreen()"
```

```
class="main__controls__button share_button">
    <i class="fas fa-share-square"></i>
    <span>Compartilhar tela</span>
</div>
</div>
<div class="main__controls__block">

<div class="main__controls__button">
    <i class="fas fa-user-friends"></i>
        <span>Participantes</span>
</div>
               <div class="main__controls__button">
                  <i class="fas fa-comment-alt"></i>
                  <span>Conversa</span>
               </div>
            </div>
            <div class="main__controls__block">
               <div class="main__controls__button">
                  <span class="leave_meeting">Sair da conversa</span>
               </div>
            </div>
         </div>
      </div>
      <div class="main__right">
         <div class="main__header">
            <h6>Conversa</h6>
         </div>
         <div class="main__chat_window">
            <ul class="messages">

            </ul>

         </div>
         <div class="main__message_container">
            <input id="chat_message" type="text"
            placeholder="Escreva aqui...">
         </div>
      </div>
   </div>
```

```
</body>
</html>
```

ANEXO D – Código css do visual da página

```
    .main {
    height: 100vh;
    display: flex;
}

.main__left {
    flex: 0.8;
    display: flex;
    flex-direction: column;
}

.main__right {
    flex: 0.2
}

.main__videos {
    flex-grow: 1;
    background-color: black;
    display: flex;
    justify-content: center;
    align-items: center;
    padding: 40px;
}

.main__controls {
    background-color: #1C1E20;
}

.main__right {
    background-color: #242324;
    border-left: 1px solid #3D3D42;
}

.main__controls {
    color: #D2D2D2;
```

```
    display: flex;
    justify-content: space-between;
    padding: 5px;
}

.main__controls__block {
    display: flex;
}

.main__controls__button {
    display: flex;
    flex-direction: column;
    justify-content: center;
    align-items: center;
    padding: 8px 10px;
    min-width: 80px;
    cursor: pointer;
}

.main__controls__button:hover {
    background-color: #343434;
    border-radius: 5px;
}

.main__controls__button i {
    font-size: 24px;
}

.main__right {
    display: flex;
    flex-direction: column;
}

.main__header {
    padding-top: 5px;
    color: #F5F5F5;
    text-align: center;
}
```

```
.main__chat_window {
    flex-grow: 1;
    overflow-y: auto;
}

.messages{
    color: white;
    list-style: none;
}

.main__message_container {
    padding: 22px 12px;
    display: flex;
}

.main__message_container input {
    flex-grow: 1;
    background-color: transparent;
    border: none;
    color: #F5F5F5;
}

.leave_meeting {
    color: #EB534B;
}

.unmute, .stop {
    color: #CC3B33;
}
```