UNIVERSIDADE FEEVALE

MATHEUS RODRIGUES KAUTZMANN

# ANÁLISE SOBRE O USO DE WEBRTC PARA COMPARTILHAMENTO DE ARQUIVOS P2P

Novo Hamburgo

2021

MATHEUS RODRIGUES KAUTZMANN

# ANÁLISE SOBRE O USO DE WEBRTC PARA COMPARTILHAMENTO DE ARQUIVOS P2P

Trabalho de Conclusão de Curso apresentado como requisito parcial à obtenção do grau de Bacharel em Ciência da Computação pela Universidade Feevale

Orientador: Dr. Gabriel da Silva Simões

Novo Hamburgo

2021

MATHEUS RODRIGUES KAUTZMANN

# ANÁLISE SOBRE O USO DE WEBRTC PARA COMPARTILHAMENTO DE ARQUIVOS P2P

Trabalho de Conclusão de Curso apresentado como requisito parcial à obtenção do grau de Bacharel em Ciência da Computação pela Universidade Feevale

APROVADO EM: ___ / ___ / _____

DR. GABRIEL DA SILVA SIMÕES
Orientador – Feevale

DR. PAULO RICARDO MUNIZ BARROS
Examinador interno – Feevale

ME. VANDERSILVIO DA SILVA
Examinador interno – Feevale

Novo Hamburgo
2021

## AGRADECIMENTOS

Gostaria de agradecer a todos os que, de alguma forma, contribuíram para a realização desse trabalho de conclusão, em especial: Aos meus pais, aos meus amigos e às pessoas que convivem comigo diariamente.

# RESUMO

Compartilhar arquivos de forma confidencial e segura entre dispositivos é essencial no dia a dia dos usuários de sistemas computacionais. Hoje existem dois modelos principais que possibilitam o compartilhamento de arquivos: o modelo cliente-servidor e o de conexão ponto a ponto (P2P). Serviços como o Google Drive, Dropbox e OneDrive possibilitam o armazenamento de arquivos de forma remota e com fácil acesso por parte do usuário e terceiros utilizando o modelo de cliente-servidor, já aplicativos como Telegram e WhatsApp permitem a transferência direta de arquivos entre dispositivos utilizando seus serviços de mensagens, porém ainda utilizando o modelo de cliente-servidor. Já *softwares* como a rede BitTorrent, o Wi-Fi Direct, o Apple AirDrop e o Google Files realizam a tarefa utilizando conexão ponto a ponto e o armazenamento seguro dos arquivos fica a cargo do usuário. Este trabalho busca analisar o uso de WebRTC para o compartilhamento de arquivos de forma direta entre dispositivos. A análise é feita comparando um aplicativo *web* desenvolvido pelo autor, chamado Fylor e que faz uso de WebRTC, com alternativas desenvolvidas em outros trabalhos, algumas baseadas em soluções conhecidas no mercado que utilizam tanto o modelo cliente-servidor quanto o modelo P2P. Ao comparar os trabalhos, busca-se encontrar as vantagens e desvantagens de cada abordagem. Por fim, considera-se que, atuando como um conjunto de APIs que permitem a criação de aplicações que utilizam comunicação em tempo real no *browser*, o WebRTC pode ser uma boa alternativa para o compartilhamento de arquivos de forma segura e confiável.

Palavras-chave: WebRTC. Compartilhamento de arquivos. P2P. Fylor.

## ABSTRACT

*File sharing in secure and confidential ways is essential in the day to day use of computer systems. Today, there are two main file sharing models: the client-server model and the peer to peer (P2P) model. Services like Google Drive, Dropbox and OneDrive enable remote storage of the user's files while still allowing them to be shared to others. Apps like Telegram and WhatsApp allow users to share files between devices through messages, although they still use the client-server model as a bridge between clients. On the other hand, softwares like the BitTorrent network, Wi-Fi Direct, Apple Airdrop and Google Files provide sharing through P2P connections and the user has the responsability to control how their files are stored. This work aims to study the usage of WebRTC for file sharing between devices. In order to do the analysis, the author developed an application called Fylor using WebRTC as the base set of APIs for its operations. Fylor is then compared with alternatives already known in the field or that were developed in other papers and articles that use either the client-server model or peer to peer connections. The main advantages and disadvantages of each work are considered when comparing the different approaches to the problem. In the end, WebRTC is understood as a valid alternative when safe and reliable file sharing is needed. That is specially true on the browser, where the technology can be used natively without plugins or any third party software.*



*Keywords: WebRTC. File sharing. P2P. Fylor.*

# LISTA DE ILUSTRAÇÕES

## LISTA DE TABELAS

## LISTA DE ABREVIATURAS E SIGLAS

| | |
|---|---|
| µTP | *Micro Transport Protocol* |
| AES | *Advanced Encryption Standard* |
| API | *Application Programming Interface* |
| AWDL | *Apple Wireless Direct Link* |
| BLE | *Bluetooth Low Energy* |
| DIMSE | *DICOM Message Service Element* |
| DICOM | *Digital Imaging and Communications in Medicine* |
| FTP | *File Transfer Protocol* |
| HTTPS | *Hyper Text Transfer Protocol Secure* |
| ICE | *Interactive Connectivity Establishment* |
| IETF | *Internet Engineering Task Force* |
| IP | *Internet Protocol* |
| IPFS | *InterPlanetary File System* |
| JNI | *Java Native Interface* |
| LAN | *Local Area Network* |
| mDNS | *Multicast DNS* |
| NAT | *Network Address Translation* |
| NDK | *Native Development Kit* |
| NFS | *Network File System* |
| PWA | *Progressive Web Application* |
| REST | *Representational State Transfer* |
| TCP | *Transmission Control Protocol* |
| TURN | *Traversal Using Relays around NAT* |
| P2P | *Peer to Peer* |

| | |
|---|---|
| SSDP | *Simple Service Discovery Protocol* |
| STOW | *STore Over the Web* |
| STUN | *Session Traversal Utilities for NAT* |
| UDP | *User Datagram Protocol* |
| UPnP | *Universal Plug and Play* |
| VoIP | *Voice over Internet Protocol* |
| VPN | *Virtual Private Network* |
| WAN | *Wide Area Network* |
| WebRTC | *Web Real-Time Communications* |
| WICG | *Web Incubator Community Group* |

## SUMÁRIO

# 1 INTRODUÇÃO

O modelo de cliente-servidor é amplamente utilizado (TANENBAUM; WETHERALL, 2010) e serve para vários casos de uso na internet, inclusive para compartilhamento de arquivos, onde o servidor disponibiliza arquivos a serem acessados por diversos clientes diferentes. Para este fim, o protocolo comumente usado é o *Hypertext Transfer Protocol* (HTTP) ou sua variante segura *Hypertext Transfer Protocol Secure* (HTTPS). No entanto, existem cenários onde um compartilhamento efêmero é mais interessante, como no caso de uma transferência de arquivo pontual ou em um cenário descentralizado que não pode contar com servidores fixos sob controle de alguma entidade específica.

Atualmente estão disponíveis os mais diferentes serviços que funcionam como disco virtual na nuvem como Google Drive, Dropbox, Onedrive e outros. Porém, todos eles dependem de servidores fixos e sob controle das entidades que gerenciam os respectivos serviços.

Apesar de possuírem uma transferência de dados segura entre o cliente e servidor, usando HTTPS e também criptografando os dados em descanso, os provedores desses serviços ainda controlam as chaves necessárias para criptografar os arquivos, e não possuem a chamada criptografia de ponta-a-ponta. Segundo Ermoshina, Musiani e Halpin (2016), esta permite que somente o emissor e o destinatário dos dados consigam ler seu conteúdo e está presente em serviços de mensagem como Telegram, WhatsApp e Signal. No entanto, estes serviços de mensagem possuem servidores armazenando os arquivos enviados, já que operam de forma assíncrona, necessitando aguardar que o destinatário apareça e faça o download do arquivo para então decidir se o mesmo será removido ou não.

Por outro lado, há também os serviços de compartilhamento local de forma direta, que servem para transferências pontuais, como o AirDrop e Google Files. Estas soluções contemplam dispositivos específicos, no caso do AirDrop a compatibilidade se restringe a dispositivos rodando iOS ou macOS e o Google Files funciona entre dispositivos Android.

O nível de comunicação entre os modelos também diferem, no modelo cliente-servidor dos discos na nuvem utiliza-se a internet como meio pelo qual os arquivos são trafegados, sendo necessária uma conexão constante à internet por parte do cliente e do servidor. Já no compartilhamento via AirDrop ou Google Files só é possível realizar a transferência em rede local. No caso do AirDrop, é criada uma rede Wi-Fi localmente entre os dispositivos de maneira proprietária (HEINRICH et al., 2021), ambos os dispositivos devem estar na mesma rede e o protocolo utilizado é de código fechado e desenvolvido pela Apple.

O Google Files, assim como o AirDrop, cria uma rede Wi-Fi privada a partir de

uma comunicação inicial via Bluetooth e utiliza um protocolo próprio desenvolvido pela Google para a transferência de arquivos. Um item em comum entre as soluções deste tipo é que exigem proximidade física entre os dispositivos, pois utilizam comunicação direta entre os hardwares dos aparelhos.

A fim de possibilitar o uso de comunicação direta *Peer to Peer* (P2P) e em tempo real de forma nativa nos navegadores de *internet*, a *World Wide Web Consortium* (W3C), que é a organização responsável pela criação e manutenção de padrões na *web*, criou a especificação *Web Real Time Communication* (WebRTC). O WebRTC nada mais é do que um conjunto de *Application Programming Interfaces* (APIs) JavaScript implementadas pelos navegadores responsáveis com o sistema operacional com o objetivo de tornar possível a programação de aplicativos que necessitem de baixa latência e que façam uso de tráfego em tempo real, como *chats online*, *softwares* de *Voice over Internet Protocol* (VoIP) e muitos outros sem a necessidade de *plugins* (MOZILLA, 2021). Com isso, esta tecnologia consegue ligar dois ou mais *browsers* entre si diretamente, sem ser necessário o uso de um servidor *web* dedicado a isso.

Acredita-se que o WebRTC pode ser uma boa alternativa para compartilhamento de arquivos P2P, pois permite a flexibilidade de enviar os arquivos tanto por rede local ou através da internet. Sendo necessária apenas uma conexão momentânea a um servidor inicial, a fim de estabelecer a conexão P2P entre as partes, este servidor é conhecido como servidor de sinalização. Após esta conexão, a comunicação é direta e síncrona entre os dispositivos participantes, com a transferência de arquivos ocorrendo através de APIs do WebRTC em cima dos protocolos da camada de transporte, utilizando um protocolo de aplicação a ser desenvolvido para cada caso de uso.

Segundo Loreto (2014), a suíte WebRTC conta com criptografia ponta-a-ponta através de *Datagram Transport Layer Security* (DTLS), que é uma variação compatível com *User Datagram Protocol* (UDP) do conhecido *Transport Layer Security* (TLS) usado no HTTPS. Com isso, proporciona privacidade aos pares, pois toda a ligação e comunicação se dá entre as partes diretamente e depois de feita a transferência dos dados não há registros salvos em qualquer servidor. Portanto, pode-se dizer que a utilização de WebRTC é segura, privada e confiável.

O conjunto de APIs WebRTC proporciona maneiras de escolher a melhor rota para a conexão entre os pares, então se os mesmos estiverem em uma mesma *Local Area Network* (LAN), estes usarão o caminho mais curto para se comunicar, sem precisar passar pela *Wide Area Network* (WAN). A escolha do melhor caminho é feita através de um processo de seleção de candidatos, sendo o melhor deles o escolhido para realizar a conexão.

Existem cenários em que uma conexão P2P direta pode não ser possível, como no caso da presença de um *firewall* na rede ou de alguma configuração de *Network Address Translation* (NAT) que impossibilite a conexão entre as duas partes. Alguns tipos de NAT

necessitam do serviço *Session Traversal Utilities for NAT* (STUN) para que os endereços e portas de conexão sejam conhecidos e reportados à aplicação. Já outros tipos de NAT, como o NAT simétrico, impedem conexões de entrada de forma direta. Para estes casos se faz necessário o uso de algum servidor de encaminhamento, que apenas servirá de ponte entre as partes, o tipo mais comum é o servidor *Traversal Using Relays around NAT* (TURN).

Mesmo em cenários de utilização de servidores de encaminhamento, os dados trafegados não podem ser entendidos por um ente no meio do caminho, pois, como descrito anteriormente, a criptografia do DTLS garante a confidencialidade dos dados em tráfego. Os serviços de STUN ou TURN apenas usam o cabeçalho do UDP para obter o endereço do destino para onde os pacotes devem ser enviados. Como o WebRTC é flexível, existe também a possibilidade de usar *Transmission Control Protocol* (TCP) no lugar de UDP, que apesar de mais lento, pode auxiliar a contornar a existência de *firewalls* bloqueando tráfego UDP.

Atualmente o uso de WebRTC é bastante popular para aplicações que necessitam de comunicação em tempo real, como aplicações de chat, VoIP, chamadas de vídeo e até mesmo em aplicações destinadas à realização de conferências virtuais. Já pode-se observar o uso das tecnologias em programas conhecidos como o Twilio (SUMRAK, 2021) e Discord (VASS, 2018).

Graças à acessibilidade abrangente do WebRTC, que vai desde os navegadores modernos, que já possuem o suporte às APIs de forma embarcada, até bibliotecas de código aberto que implementam a suíte de APIs, é possível se observar que essa tecnologia é uma alternativa viável para estudo visando obter uma forma confiável, privada e segura para o envio de arquivos P2P.

Portanto, o objetivo deste trabalho é verificar a viabilidade e o potencial do uso de WebRTC para transferências de arquivos entre dois ou mais clientes de forma direta. Isto será feito através do desenvolvimento de um aplicativo real utilizando a suíte de tecnologias do WebRTC e posterior comparação dos pontos positivos e negativos da solução com alternativas comuns para o caso de uso proposto.

## 1.1 OBJETIVOS ESPECÍFICOS

Para atingir o objetivo de avaliar a viabilidade do uso de WebRTC para o envio e recebimento de arquivos foram estabelecidos os seguintes objetivos específicos em ordem:

- Pesquisar e identificar soluções existentes para compartilhamento de arquivos, levando em conta trabalhos publicados nos últimos cinco anos;

- Apresentar um aplicativo alternativo às soluções encontradas, utilizando WebRTC como base;
- Analisar comparativamente os diferentes meios de trocas de arquivos, incluindo a alternativa desenvolvida, considerando variáveis qualitativas e métricas comuns a todos os meios;
- Demonstrar os resultados de forma clara e objetiva;
- Verificar se existe viabilidade no uso de WebRTC através da análise dos resultados obtidos.

Este trabalho, baseando-se nos objetivos propostos anteriormente, busca proporcionar uma contribuição válida para desenvolvedores buscando alternativas para compartilhamento de arquivos ou mesmo servir como estudo base para futuros estudos sobre a área abordada. O texto está dividido da seguinte forma: o capítulo 1 é a presente introdução, que procura explicar a motivação que originou este trabalho. O capítulo 2 procura fundamentar os conceitos relevantes para o entendimento deste trabalho, incluindo assuntos específicos da suíte WebRTC. O capítulo 3 lista os trabalhos relacionados e suas contribuições para a área de estudo. O capítulo 4 aborda o desenvolvimento do aplicativo de testes que é usado como alternativa às soluções existes e é o foco deste trabalho, aplicando o estudo na prática. O capítulo 5 analisa comparativamente algumas das soluções existentes com o aplicativo desenvolvido. Por fim, o capítulo 6 traz as considerações finais do autor, buscando verificar se existe viabilidade ou não no uso de WebRTC para compartilhamento de arquivos.

# 2 FUNDAMENTAÇÃO TEÓRICA

O conjunto de tecnologias por trás do WebRTC é extenso e para entender como as diferentes partes da suíte se relacionam deve-se compreender, ao menos superficialmente, como cada tecnologia contribui para o funcionamento geral das aplicações web em tempo real.

## 2.1 OBJETIVOS E USOS DO WEBRTC

As soluções disponíveis nos navegadores pelas APIs que existiam antes do WebRTC ser desenvolvido não permitiam o desenvolvimento de aplicações que mantivessem conexão direta via *socket* TCP ou UDP. O que existia até o momento eram APIs como a do Web Socket que é um protocolo feito para navegadores usarem que opera em cima de TCP, mas que ainda dependia de envio e recebimento de mensagens. Não suportando aplicações que operam com fluxo de dados contínuo.

O uso de fluxo contínuo de dados é crucial para operar aplicações do tipo VoIP, aplicações de chamadas de vídeo, ensino a distância e muitas outras. Para estes tipos de aplicações, a baixa latência é crucial e o protocolo UDP seria o mais indicado devido à necessidade de se ter uma entrega contínua de pacotes sem necessitar da garantia de entrega proposta pelo TCP.

Como até o momento não havia meio de se utilizar UDP de forma nativa nos navegadores, muitas das aplicações populares da época como Skype, Google Talk e outras dependiam de plugins baseados em Flash ou Java ou até mesmo só funcionavam com aplicações dedicadas instaladas na máquina do usuário.

A primeira versão do WebRTC foi proposta por Bergkvist et al. (2011) após uma das reuniões do grupo de trabalho do *World Wide Web Consortium* (W3C). O documento surgiu de uma necessidade em comum dos membros do W3C, que possui representantes de empresas de tecnologia, desenvolvedores de *browsers* e provedores de *internet*. Os participantes buscavam uma solução para aplicações que necessitam de baixa latência de operação e que enviam e recebem fluxos de dados contínuos, conhecidas no meio como *real time applications*. Este documento inicial foi utilizado como base para todas as implementações futuras do WebRTC.

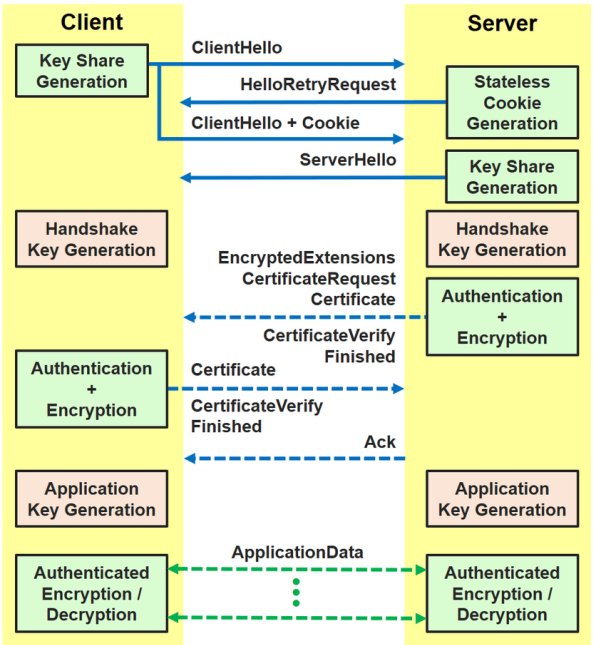
A grande vantagem desta nova suíte de APIs é que finalmente foi proposto um padrão único para todos os *browsers* implementarem, o que deixaria o desenvolvimento de aplicações em tempo real mais simples e sem a necessidade do uso de *applets* Java ou qualquer outro tipo de *plugin*.

Além de estar presente em todos os navegadores modernos atualmente, a especificação da suíte de tecnologias também é disponibilizada como código aberto e pode ser reimplementada em diferentes linguagens, isso permite que aplicações nativas conversem com os *browsers* diretamente e vice-versa. Hoje a tecnologia é utilizada em diversas aplicações presentes na *web* como o Google Meet, Facebook Messenger, Blackboard Collaborate e muitas outras.

## 2.2 DTLS: O PROTOCOLO BASE DO WEBRTC

A suíte WebRTC foi concebida considerando baixa latência, segurança e privacidade como prioridade. Na especificação proposta pelo W3C a criptografia é obrigatória para o uso da suíte de tecnologias. Para conseguir atingir os requisitos apresentados, o grupo de trabalho optou por utilizar o protocolo DTLS como base. Este protocolo é baseado no conhecido protocolo UDP que apresenta características importantes para se obter baixa latência de comunicação, pois ele é bem simples e abre mão da necessidade de estabelecimento de uma conexão prévia e também abdica da confiabilidade de entrega dos pacotes em favor de reduzir o tempo total até a entrega de cada pacote (TANENBAUM; WETHERALL, 2010). Estas características tornam o UDP ideal para cenários de aplicações em tempo real como jogos multijogador, aplicações de VoIP e outras.

**Figura 1 – Diagrama do *handshake* DTLS**




Fonte: (BANERJEE et al., 2017)

No entanto, usar somente o UDP não resolve a necessidade de se obter segurança e privacidade no envio dos dados pela rede. O principal objetivo do protocolo DTLS é adicionar a camada de segurança do TLS, já conhecida para o protocolo TCP, ao

protocolo UDP. Os inventores do protocolo DTLS optaram por deixar o uso de TLS o mais semelhante possível à versão presente no TCP (RESCORLA; MODADUGU, 2006).

Como é possível ver na Figura 1, o *handhshake* de abertura da conexão DTLS é feito em lotes de mensagens, onde primeiro ocorre o início da sessão com a geração de chaves e então a comunicação criptografada começa, na prática dois pares de chaves são usados na comunicação, um par para estabelecimento do *handshake* e validação de certificados e outro par utilizado na criptografia dos dados em trânsito da aplicação.

Segundo Rescorla e Modadugu (2006), o protocolo DTLS fornece segurança e privacidade equivalente à sua implementação em TCP e ainda fornece ganhos na velocidade do *handshake*, permitindo que o protocolo atue com operações em lote, que são validadas com uma única mensagem *ACK* por lote. O protocolo também lida automaticamente com retransmissões e perdas utilizando temporizadores que determinam o tempo máximo até uma mensagem ser respondida pela entidade remota, o DTLS abstrai suas operações para o usuário final, que só precisa ser responsável por estabelecer a conexão, bem como enviar e receber os dados pertinentes para sua aplicação.

## 2.3 *SIGNALING*: NEGOCIAÇÃO PARA O INÍCIO DA CONEXÃO

De acordo com Grigorik (2016), para iniciar uma comunicação direta WebRTC entre dois pares é necessária uma negociação prévia sobre como a conexão será feita. Esse processo é conhecido como *signaling* e é a primeira parte da conexão WebRTC e que se estende até que a conexão chegue ao fim. Esse procedimento serve para que as partes cheguem a um acordo sobre quais interfaces de rede, endereços e portas usarão na conexão, bem como se utilizarão conexão direta ou passando por serviço de TURN e se será utilizado UDP/DTLS ou TCP/TLS como protocolo da camada de transporte. Além de definir os parâmetros da conexão, as partes também se comunicam para saber que tipo de conteúdo será trafegado, pois o WebRTC possui canais específicos para conteúdos de áudio e vídeo, bem como permite a criação de canais de dados, que é o foco deste trabalho.

**Figura 2 – Representação da negociação inicial do WebRTC**


Fonte: (GRIGORIK, 2016)

A Figura 2 representa a primeira comunicação entre as duas partes, onde uma das partes realiza uma oferta de conexão com os termos desejados e a outra parte responde se aceita ou não esta conexão. A escolha de como se dará essa comunicação de negociação fica

a cargo de cada desenvolvedor, a especificação do WebRTC não abrange a implementação do serviço de *signaling*, apenas especifica as etapas que devem ser cumpridas para que a conexão seja estabelecida.

Este serviço normalmente é implementado como um dos blocos de uma aplicação WebRTC e pode usar desde um servidor *web* comum, alguma aplicação usando Web Sockets ou até mesmo uma conexão via *socket* prévia antes de estabelecer a conexão final. Em alguns casos pode ocorrer a troca do transporte dos serviços de *signaling*, essa prática é útil quando se quer o máximo de privacidade na comunicação, pode-se iniciar com uma comunicação a um servidor remoto para se definir a conexão direta e após isso usar um canal de dados do WebRTC para trocar metadados durante a conexão.

### 2.3.1 Descoberta de candidatos

Muitas vezes há mais de uma forma de se obter conexão direta entre duas partes. Cada uma das possíveis alternativas é chamada de candidato à conexão. Um computador pode estar conectado através de mais de uma interface de rede, como por exemplo estar disponível através do Wi-Fi e também através de *ethernet*, ou ainda pode possuir mais de um endereço IP, sendo IPv4 ou IPv6, ou até mesmo estar conectado através de uma *Virtual Private Network* (VPN). Para decidir qual a melhor forma de conectar há o processo de descoberta e escolha de candidatos, esse processo é feito na camada de *signaling* através de trocas de mensagens, onde o ofertante da conexão envia suas alternativas de conexão ao receptor da oferta e vice-versa.

Existe ainda a possibilidade de ser necessário atravessar um NAT restritivo ou até mesmo não ser possível estabelecer uma conexão direta. Todas essas opções devem ser levadas em conta pelo mecanismo de descoberta de candidatos. O mecanismo de descoberta de candidatos no WebRTC é conhecido como *Interactive Connectivity Establishment* (ICE), esse processo é responsável por selecionar o candidato mais adequado para a conexão (GRIGORIK, 2016).

Cada candidato possui uma variável de custo atribuída, que determina o quão boa seria a conexão através daquele meio. Isso permite que em caso de múltiplas opções a melhor escolha seja feita, um exemplo seria a simples verificação de latência da conexão, no caso de existir uma conexão através de VPN e uma conexão direta a conexão direta será sempre preferível pois possui latência menor. Conexões que passam por *relays* do tipo TURN também possuem uma variável de custo maior, pois precisam passar por um servidor remoto antes de chegar ao destino.

O procedimento de escolha de candidatos não termina quando a conexão é estabelecida, podem existir casos de renegociação, no qual se a conexão sofrer instabilidade ou cair o processo de escolha é refeito.

## 2.4 CANAIS DE COMUNICAÇÃO

Como o WebRTC é direcionado para aplicações em tempo real, ele possui três tipos de canais de comunicação: os canais de vídeo, os canais de áudio e os canais de dados. Os canais de vídeo e de áudio são usados para aplicações VoIP, transmissões de vídeo ao vivo, entre outros casos de uso. Já os canais de dados são o foco de estudo deste trabalho, pois permitem o tráfego de dados binários de uso genérico e isto viabilizaria a transferência de arquivos, por exemplo. Os canais de dados utilizam o protocolo *Stream Control Transmission Protocol* (SCTP) para transporte dos dados da aplicação, este protocolo já está encapsulado dentro do DTLS no caso do WebRTC, garantindo a segurança e privacidade dos dados trafegados (GRIGORIK, 2016).

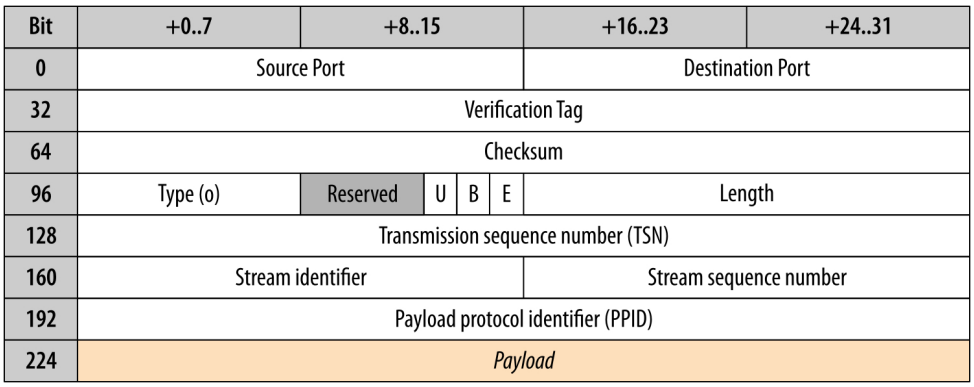
**Tabela 1 – Comparação de características do TCP, UDP e SCTP**

| **Característica** | **TCP** | **UDP** | **SCTP** |
|---|---|---|---|
| **Confiabilidade** | confiável | não confiável | configurável |
| **Entrega** | ordenada | desordenada | configurável |
| **Transmissão** | bytes | mensagens | mensagens |
| **Controle de fluxo** | sim | não | sim |
| **Controle de congestionamento** | sim | não | sim |

Fonte: Com base em Grigorik (2016)

SCTP é uma escolha boa para canais de dados do WebRTC, pois permite a configuração de entrega das mensagens, que podem ser ordenadas ou desordenadas, bem como se o desenvolvedor deseja que o canal lide com perdas de pacotes e retransmissões automaticamente. Como neste caso o SCTP está implementado em cima do DTLS, que por sua vez está implementado em cima do UDP, algumas de suas características já são atendidas pelos protocolos anteriores. Na Tabela 1, as principais características dos protocolos TCP, UDP e SCTP são comparadas.

**Figura 3 – Representação do pacote SCTP**




Fonte: (GRIGORIK, 2016)

A Figura 3 é uma representação gráfica do interior de um pacote SCTP, é possível observar que o protocolo usa 28 bytes de cabeçalho e este *overhead* está presente em todos os pacotes de dados, o restante é usado para o *payload* que carrega os dados da aplicação. Como no caso do WebRTC o SCTP é encapsulado pelo DTLS/UDP, alguns itens do cabeçalho são redundantes, como as portas de origem e destino, uma vez que essas informações já são tratadas pelas camadas anteriores.

Como o SCTP suporta configuração se o transporte deve operar de forma confiável ou não e se os pacotes devem ser entregues em ordem ou não, há no cabeçalho bits especiais que identificam se esse pacote faz parte de um conjunto ordenado e seu número sequencial, simulando uma implementação parecida com o que vemos no TCP no caso de transportes ordenados e confiáveis. O fato disto ser configurável torna o protocolo bem flexível e seu uso fica transparente para o desenvolvedor no WebRTC, que apenas escolhe o funcionamento via uma configuração simples antes de abrir o canal de dados.

A partir do bit 192 estão os dados relativos ao *payload*, primeiro há o identificador do protocolo do *payload* ou PDID, este dado identifica o tipo de dado que será trafegado na região seguinte, no WebRTC temos dois tipos de dados para o canal de dados, o dado pode ser textual, sendo representado como uma *string* simples em UTF-8 ou pode ser do tipo binário, que se traduzem em um *buffer* de dados a ser consumido pela aplicação destino. Finalmente, a partir do bit 224, há o conteúdo da aplicação respeitando o tipo fornecido no PDID.

**Figura 4 – Exemplo de código JavaScript que cria um canal de dados em WebRTC**

```
const channel = pc.createDataChannel('data', {
  ordered: true
});

channel.onopen = () => console.log('Channel open.');
channel.onmessage = (msg) => console.log(msg);
channel.onclosing = () => console.log('Channel closing.');
channel.onclose = () => console.log('Channel closed.');
```

Fonte: Autor

Todo esse conhecimento sobre o funcionamento interno do WebRTC nos canais de dados é útil para estudos na área, porém para o desenvolvedor utilizar isso na prática é muito simples, e não necessita de conhecimentos dos protocolos envolvidos, pois tudo fica abstraído da camada de aplicação. A Figura 4 mostra uma implementação de abertura de canal de dados de forma simples em JavaScript, considerando que a conexão inicial ao *peer* remoto já foi estabelecida e instanciada como objeto da variável **pc**, que identifica um *peer connection*, ou seja, uma conexão remota a um par. Depois de aberto por uma

chamada simples ao método *createDataChannel* passando um identificador para o canal de dados e configurando-o de forma a informar que ele deve ser do tipo ordenado, o canal está pronto para ser utilizado para troca de mensagens.

A partir do momento que o canal é aberto, o WebRTC e seus protocolos internos controlam o envio de mensagens, garantindo a ordenação e o reenvio de mensagens perdidas no caso de canais confiáveis como o criado neste exemplo. Para o desenvolvedor final o canal atua como um emissor de eventos, chamando as funções que são indicadas pelo desenvolvedor quando algo acontece com o canal, seja o recebimento de uma mensagem, a abertura ou o fechamento do mesmo.

## 2.5 STUN E TURN: LIDANDO COM *FIREWALLS* E TIPOS DE NAT RESTRITIVOS

Nem sempre o mecanismo de descoberta de candidatos do WebRTC consegue uma conexão direta entre dois dispositivos. Por este motivo os candidatos ICE podem fazer uso de serviços como STUN ou TURN (GRIGORIK, 2016). O serviço do tipo STUN serve para descobrir qual o endereço de IP externo do dispositivo para a aplicação obter uma conexão direta a um membro fora da rede local, por esse motivo o serviço STUN é considerado de baixo custo, ele não carrega dados da aplicação, apenas auxilia a abrir a conexão direta entre duas partes. Já o serviço do tipo TURN serve como ponte entre dois computadores que não conseguem se comunicar diretamente e neste caso os dados passam por um servidor externo, que apenas encaminha os dados da origem até o destino final.

**Tabela 2 – Comparação de características do STUN e TURN**

| **Característica** | **STUN** | **TURN** |
|---|---|---|
| **Função** | descobrir IPs externos | ponte de tráfego |
| **Usos** | presença de NAT | não há conexão direta |
| **Custo de operação** | baixo | alto |
| **Impacto** | baixo, atua na abertura | alto, sempre ativo |

Fonte: Autor

A Tabela 2 compara de forma resumida os serviços de STUN e TURN. Devido ao seu alto custo de operação, o serviço de TURN geralmente fica como último recurso a ser utilizado, sendo aplicado somente em casos em que seja o único candidato disponível. O processo de captura de candidatos sempre dará preferência para a descoberta de endereços locais primeiro e depois realizará consultas a servidores STUN, se estiverem presentes na configuração inicial. Caso a negociação via *signaling* ainda não resultar em uma conexão direta será utilizado um serviço TURN, se assim estiver configurado.

Felizmente, o serviço STUN consegue lidar com três tipos de NAT diferentes e garante uma conexão direta nestes casos (ROSENBERG et al., 2003):

- *Full cone NAT*, também conhecido como NAT um para um, é um tipo simples de NAT onde o endereço e porta interno é mapeado diretamente para um endereço e porta externo. Qualquer pacote enviado para o endereço e porta externos será encaminhado para o respectivo endereço e porta internos;

- *Restricted cone NAT*, possui as mesmas características do *full cone NAT*, exceto por um comportamento: um *host* externo só pode enviar um pacote para o endereço interno mapeado se o endereço interno tiver enviado um pacote ao *host* externo primeiro;

- *Port restricted cone NAT*, parecido com o *restricted cone NAT*, porém também limitando a porta mapeada, um *host* externo só pode enviar um pacote à porta interna se o *host* interno tiver enviado um pacote à mesma porta mapeada do *host* externo;

Existe ainda um tipo de NAT que o STUN não consegue atender, que é o *Symmetric NAT*. Este tipo de NAT usa um mapeamento de endereço e porta externo diferente para cada requisição interna, e o *host* externo só pode responder para o mesmo mapeamento. Este tipo de NAT é mais restritivo e pode ser contornado somente com o uso de um serviço do tipo TURN.

Claro que além de configurações de rede com diferentes tipos de NAT, há ainda a possibilidade de existirem *firewalls* no caminho entre a origem e o destino da conexão. *Firewalls* simples, com filtros de porta podem ser contornados com uso de portas aceitas na rede, se protocolos especiais estiverem sendo filtrados, como por exemplo, o banimento do UDP, o TCP pode ser usado como segunda opção. No entanto, se o *firewall* inspecionar os pacotes trafegados de forma mais detalhada e não aceitar conexões WebRTC, aí o usuário não poderá usar soluções baseadas na tecnologia nesta rede, ao menos não diretamente.

No momento da escrita deste documento, o cenário corrente consiste na grande maioria dos usuários comuns utilizando uma rede simples com um IPv4 externo e roteador com serviço de NAT simples realizado os mapeamentos, nestes casos uma combinação dos protocolos do WebRTC com o serviço de STUN já consegue fornecer uma conexão direta. O IPv6 está ficando popular e facilita conexões P2P em alguns casos onde o NAT nem é mais utilizado, com cada dispositivo com um ao menos um IP externo único. Redes corporativas costumam usar o recurso de *firewalls* mais complexos que àqueles encontrados nas redes comuns, e nestes casos é possível usar o recurso do serviço TURN.

# 3 TRABALHOS RELACIONADOS

Neste capítulo estão apresentados trabalhos que exploram o tema de compartilhamento de arquivos. Para encontrar os trabalhos fez-se uso de pesquisa em bases de artigos científicos conhecidas no meio acadêmico. Para cada resultado da busca foi feita uma filtragem e seleção dos trabalhos pertinentes com base numa série de regras a serem descritas no transcorrer do capítulo. Feito isso, os artigos selecionados foram separados em dois segmentos. O objetivo do primeiro segmento é obter uma visão sobre o problema de compartilhamento de arquivos e algumas possíveis abordagens para diferentes cenários de rede. Já o segundo segmento traz dois trabalhos que já utilizam o padrão *WebRTC* como tópico de pesquisa e servem de referência para o presente trabalho.

## 3.1 METODOLOGIA DE PESQUISA

Como o objetivo deste trabalho é comparar o uso de compartilhamento de arquivos em diferentes cenários para, a partir de então, avaliar o uso de *WebRTC*, é útil separar os arquivos pelos modelos de rede abordados. Para tal, três modelos de rede foram propostos:

- Cliente-servidor via *internet*, onde há um ou mais clientes consultando arquivos em um servidor local ou na nuvem;
- Transferência local e pontual de um para um com uso de rede espontânea;
- Compartilhamento P2P via *internet*;

No final da busca, o objetivo foi obter cinco artigos seguindo um dado padrão, utilizando os modelos de rede apontados acima:

- Escolher três textos que abordem técnicas de compartilhamento de arquivos alternativas ao *WebRTC*;
- Indicar um artigo abordando compartilhamento de arquivos utilizando *WebRTC* no modelo um para um;
- Obter um periódico que demonstre o uso de *WebRTC* no modelo N para N;
- Possuir com um conjunto total de cinco artigos onde cada texto aborde pelo menos um dos modelos de rede propostos, sendo três deles alternativas ao *WebRTC* e dois deles aplicações do *WebRTC*.

Para reduzir o escopo da busca, os seguintes critérios foram considerados ao selecionar os artigos a serem filtrados:

- Artigos publicados nos últimos 5 anos, em estado da arte;
- Publicações em inglês.

Foram realizadas buscas em duas bases conhecidas de artigos científicos: *Web of Science* e Google Scholar. Estas bases foram escolhidas pois possuem uma boa coleção de artigos recentes ou pré-publicados na área da tecnologia da informação.

**Figura 5 – *String* de busca utilizada na base *Web of Science***

```
TS=(file sharing OR file transfer OR data accessibility OR data transmission OR local file sharing
OR WebRTC) AND TI=(*file sharing* OR *cloud* OR *file transfer* OR *WebRTC*) AND AK=(WebRTC OR
data accessibility OR file sharing OR sharing file) AND AB=(file sharing OR data sharing)
```

Fonte: Autor

Uma *string* de busca para cada base de conhecimento foi elaborada. Na base *Web of Science* foi utilizada a *string* demonstrada na Figura 5. O intuito por trás da utilização dos termos ali presentes foi tentar encontrar artigos que mencionassem transferência de arquivos, tanto localmente como na nuvem e também achar trabalhos que abordassem WebRTC diretamente. Para isso, a ferramenta de busca avançada foi usada, filtrando os textos por tópico, presença de palavras no título, palavras-chave específicas e palavras presentes no resumo.

Na base *Web of Science* foram encontrados 57 artigos utilizando os parâmetros de busca mencionados anteriormente. Para cada um dos resultados procurou-se ler o resumo a fim de descartar artigos que não abordem compartilhamento de arquivos diretamente ou que já estivessem cobertos por uma alternativa mais sólida, já encontrada anteriormente.

Ao fim da leitura, 10 trabalhos restaram e foram encaminhados para a leitura completa e destes quatro foram selecionados, restando um para a busca na outra base de dados. Como ainda estava pendente o trabalho que abordasse transferências locais com redes espontâneas, a busca foi direcionada à outra base, neste caso a base Google Scholar.

No Google Scholar, muitas vezes é possível encontrar trabalhos em um estado anterior aos que estão na base Web of Science, muitos ainda não publicados em revistas científicas ou que ainda não foram enviados a congressos. Como havia o interesse em buscar um trabalho que abordasse compartilhamento direto e com casos de uso recentes, optou-se por procurar um trabalho que abordasse o protocolo AirDrop, que é um ótimo exemplo de transferência local e direta. A busca foi feita utilizando a *string* de busca *AirDrop*, seguindo os parâmetros que foram utilizados na busca na Web of Science, filtrando somente por documentos no estado da arte e no idioma inglês.

Dentre os principais resultados, o trabalho de Heinrich et al. (2021) foi selecionado pois consegue detalhar o funcionamento do AirDrop enquanto propõe alterações ao protocolo.

## 3.2 TRABALHOS SELECIONADOS

Com a busca concluída e os cinco trabalhos devidamente selecionados, é possível explorar o que cada texto agrega para o tópico em questão.

### 3.2.1 Rio: a personal storage system in multi-device and cloud

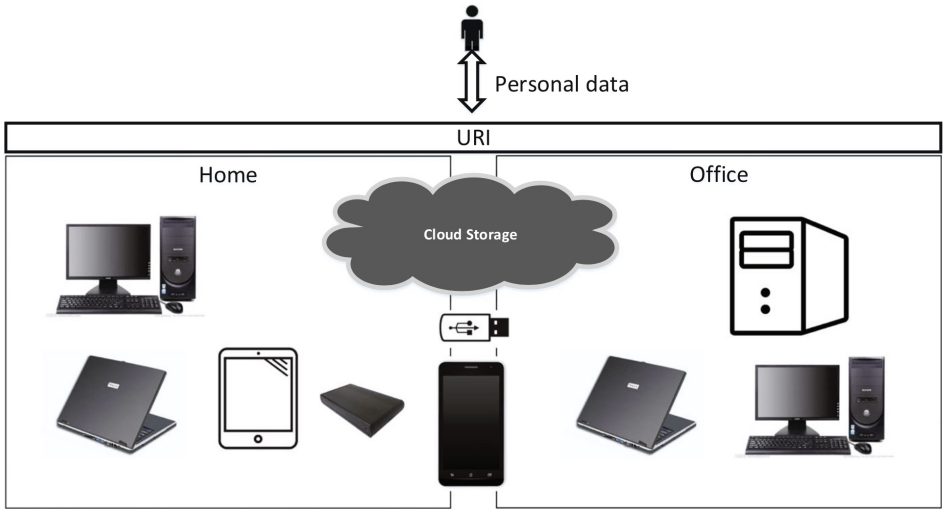
O trabalho de Wang, Wu e Huang (2020) aborda o modelo cliente-servidor tradicional, já muito utilizado em aplicações de compartilhamento de arquivos. Neste modelo temos N clientes podendo adicionar, alterar e excluir arquivos salvos em um servidor. Este servidor pode estar presente em uma rede local ou estar na nuvem com acesso através da *internet*.

Rio é uma aplicação proposta pelos autores para servir como armazenamento remoto de arquivos, similar aos serviços do Google Drive e Dropbox. O principal objetivo deste *software* é permitir que os usuários acessem seus arquivos de qualquer lugar, desde que esteja disponível o acesso à mesma rede do servidor ou à *internet*. O grande diferencial da Rio é que ela conecta diversos meios de armazenamento tradicionais em uma única API, permitindo acesso unificado aos arquivos independentemente de onde estejam armazenados. A Rio permite configurações usando todo o tipo de armazenamento, seja disco rígido local, dispositivos removíveis como *pendrives*, *Hard Disks* (HDs) externos ou mesmo outros serviços da nuvem como Google Drive, Dropbox e iCloud Drive. Esta configuração é demonstrada na Figura 6.

A API do programa é baseada em *Representational State Transfer* (REST) e utiliza HTTPS como protocolo de aplicação, permitindo um tráfego seguro dos dados do usuário. O compartilhamento de arquivos é possível através da delegação de direitos de acesso à outros usuários, expondo os arquivos em questão para acessos de terceiros. Como o armazenamento dos arquivos está no servidor e os acessos podem acontecer simultaneamente por diversos clientes, a Rio teve que implementar um algoritmo para garantir a consistência dos dados, de modo que nenhum cliente ficasse em um estado inválido, sendo o servidor a fonte da verdade.

Este tipo de problema existe pois a Rio mantém os arquivos de forma persistente, devido a natureza do objetivo da aplicação. Então pode-se dizer que para garantir a segurança completa dos dados deve-se, além de garantir a segurança do trânsito dos arquivos, ter criptografia em descanso dos arquivos armazenados no servidor e isto não está no escopo atual da Rio. Isto deixa a cargo do usuário da solução garantir que seus

**Figura 6 – Demonstração do acesso unificado dos arquivos pela plataforma Rio**




Fonte: Wang, Wu e Huang (2020)

dados estejam seguros na máquina.

Os autores também implementaram uma solução de descoberta baseada em *Simple Service Discovery Protocol* (SSDP) quando em rede local para descobrir onde o servidor se encontra e receber requisições de clientes novos. Segundo Albright et al. (1999), o SSDP é um protocolo simples usado para descoberta de dispositivos rodando um mesmo tipo de *software* em uma rede. O protocolo faz parte da suíte *Universal Plug and Play* (UPnP) e permite o disparo de um datagrama do protocolo UDP customizado a um endereço de *Internet Protocol* (IP) especial, a partir disso o roteador fará o *multicast* desta mensagem na rede, esperando respostas de clientes compatíveis.

Percebe-se que isso não exclui a necessidade de estar na mesma rede que o servidor e este deve estar conectado à internet para ser compatível com APIs externas como os acessos ao Google Drive, Dropbox e iCloud Drive.

**Tabela 3 – Comparativo de desempenho da Rio com Ipmsg e FTP**

| | Rio | | Ipmsg | | FTP | |
|---|---|---|---|---|---|---|
| | File | Folder | File | Folder | File | Folder |
| Linux | 3 min 50 s | 4 min 28 s | 3 min 50 s | 4 min 28 s | 3 min 50 s | 4 min 52 s |
| Android | 3 min 50 s | 4 min 28 s | 3 min 50 s | 4 min 28 s | 3 min 50 s | 5 min 04 s |

Fonte: Wang, Wu e Huang (2020)

Wang, Wu e Huang (2020) também demonstram testes de desempenho da solução proposta utilizando um arquivo de 2,51 GB utilizando dispositivos Linux e Android, de mesma capacidade computacional e usando a mesma rede, comparando o *software* com o protocolo *File Transfer Protocol* (FTP) rodando via servidor vsftpd e o aplicativo de

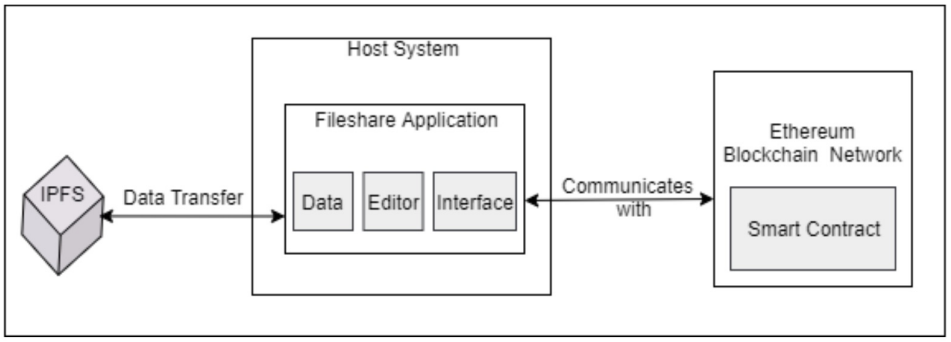
trocas de arquivos em LAN Ipmsg. É possível observar na Tabela 3 que a solução não acrescenta *overhead*[1] em relação ao Ipmsg, enquanto supera o FTP.

Os autores ressaltam que o foco do Rio não é ser um sistema de arquivos distribuído, como *Network File System* (NFS) ou Ori[2] e sim focar em solução simples para armazenar e acessar arquivos via uma API simples voltada ao usuário final. Portanto, conclui-se que Rio é uma alternativa viável para compartilhamento de arquivos quando se deseja manter os arquivos em um local de acesso remoto permanente, de modo a manter a sincronia dos arquivos entre diferentes clientes.

### 3.2.2 FileShare: A Blockchain and IPFS framework for Secure File Sharing and Data Provenance

Khatal et al. (2021) trazem uma abordagem baseada em *blockchain* e *InterPlanetary File System* (IPFS) como base para o compartilhamento de arquivos chamado de FileShare. O IPFS é um sistema de arquivos distribuído e P2P, sem ponto de falha único e com algoritmos de controle de versões e garantia de consistência de dados (BENET, 2014). A *blockchain* utilizada é pública e baseada na rede Ethereum, nesta camada estão armazenados metadados dos arquivos e códigos de contratos inteligentes[3] que gerenciam o acesso e troca de arquivos entre as partes.

**Figura 7 – Diagrama de funcionamento do FileShare**




Fonte: Khatal et al. (2021)

Como mostra a Figura 7, a aplicação do FileShare funciona em comunicação tanto com IPFS, para transferência de dados, quanto com a rede Ethereum, para comunicação de dados adicionais dos arquivos e criptografia dos documentos salvos no IPFS. Este arranjo possibilita tanto criptografia dos arquivos em descanso quanto em trânsito, já que as chaves são gerenciadas pela aplicação cliente utilizando criptografia simétrica AES-256 e o IPFS só armazena os arquivos já criptografados.

[1] Dados que não são parte da carga útil ou *payload*.
[2] Sistema de arquivos distribuído desenvolvido por um grupo de pesquisa da universidade de Stanford.
[3] Códigos que rodam em cima da rede Ethereum e controlam as trocas de ativos na rede.

Além disso, os autores se preocuparam em implementar uma auditoria para todos os eventos de acesso aos arquivos. Estes são salvos como metadados através da execução de contratos inteligentes especialmente feitos para isso na rede Ethereum. Com a característica imutável da *blockchain* esta prática acaba sendo bem sólida, pois não há como editar o histórico de acessos.

O modelo de rede desta proposta conta com N nós atuando como servidores em uma rede P2P, levando em conta os nós que estão rodando IPFS e nós Ethereum, e aceita N nós clientes também rodando o software do FileShare. É altamente acessível e disponível, porém depende de conexão à *internet* e da manutenção e disponibilidade dos dois serviços externos que usa como base.

Uma grande vantagem da solução é possuir apenas uma cópia de cada arquivo, pois seus metadados são gerenciados na *blockchain*, então só arquivos novos são salvos no IPFS. Arquivos existentes são apenas referenciados.

O compartilhamento ocorre por meio de direitos de acesso concedidos sob escolha do usuário que publicou o arquivo. A autenticação acontece fazendo uso de chaves públicas e privadas, onde as chaves públicas são usadas como base para a criptografia da chave simétrica que descriptografa o arquivo, assim garantindo a segurança no compartilhamento com terceiros.

Por mais que tanto a rede IPFS e a rede Ethereum sejam públicas e não exclusivas, não há como interferir no funcionamento do FileShare sem as chaves necessárias, então a solução consegue manter segurança criptográfica mesmo em um ambiente aberto a qualquer outro uso. O uso de chaves como único meio de autenticação é seguro, porém não é tão prático ao usuário final, pois ele precisa levar as chaves consigo em qualquer dispositivo com o qual deseja ter acesso aos arquivos, ou criar compartilhamentos para cada aparelho diferente.

Também é importante notar que os arquivos incluídos na rede do FileShare são persistentes, não há como apagar rastros dos arquivos, e pessoas com as chaves corretas conseguirão acessar os dados. Esta abordagem não é adequada para transferências pontuais de arquivos entre dispositivos, pois obrigatoriamente ao incluir um arquivo é necessário criar o mesmo no IPFS e deixar os metadados salvos na rede de forma permanente.

### 3.2.3 PrivateDrop: Practical Privacy-Preserving Authentication for Apple AirDrop

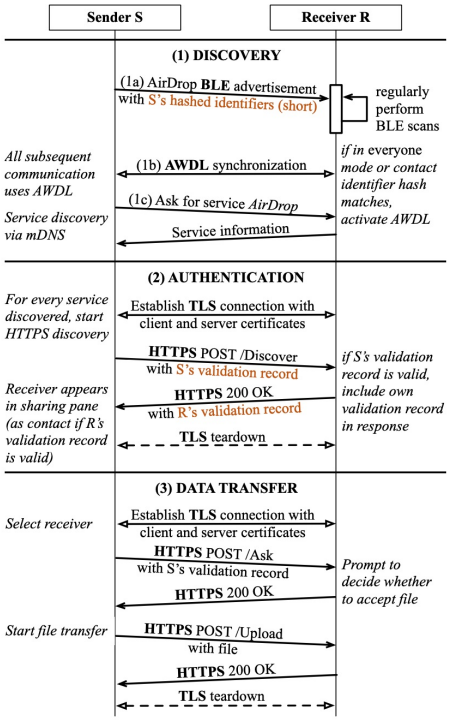
Por mais que o artigo de Heinrich et al. (2021) aborde primariamente uma falha de segurança do protocolo Apple AirDrop, ele se mostra interessante, pois explica o funcionamento do protocolo de código fechado desenvolvido pela Apple. No texto, é proposto um protocolo revisado chamado de PrivateDrop como forma de aconselhar à Apple cor-

reções que devem ser feitas no AirDrop para evitar o vazamento de dados pertinentes do usuário, como seu nome, telefone e endereço de *e-mail*.

Porém o que mais interessa neste texto para o tópico deste presente trabalho é o funcionamento interno do AirDrop e os autores conseguem obter estas informações através de engenharia reversa do protocolo. Os mesmos exploraram a maneira como o AirDrop cria uma rede espontânea e local para transferências pontuais de arquivos entre dois dispositivos.

O protocolo desenvolvido pela Apple funciona de maneira *offline*, necessitando apenas da proximidade de dois dispositivos que desejam trocar arquivos. A Apple optou por desenvolver compatibilidade apenas para os sistemas macOS e iOS, que rodam nos Macs e outros dispositivos como iPhones e iPads. Também há dois requisitos de hardware para o AirDrop funcionar: a presença de suporte a *Bluetooth Low Energy* (BLE) e suporte a Wi-Fi.

**Figura 8 – Diagrama que mostra a sequência de eventos do protocolo AirDrop**




Fonte: Heinrich et al. (2021)

Como é possível perceber na Figura 8, o protocolo AirDrop possui três etapas principais. A primeira etapa é disparada pelo usuário, ao realizar a intenção de compartilhar um arquivo, a partir disto o sistema irá ativar o *bluetooth* do aparelho e começar a enviar pacotes aos dispositivos próximos para descobrir quais entendem e respondem ao protocolo AirDrop. Assim que os dispositivos próximos são identificados, é requisitado ao usuário que ele indique o dispositivo com o qual ele pretende compartilhar o arquivo.

Somente neste ponto que a rede espontânea é criada com a finalidade de enviar o arquivo em maior velocidade, já que o BLE não possui altas taxas de transferência, sendo destinado a mensagens curtas e periódicas. O protocolo que a Apple criou para a estabelecimento da rede sem fio espontânea é chamado de *Apple Wireless Direct Link* (AWDL), e é uma alternativa ao padrão aberto Wi-Fi Direct e antecede a criação do mesmo.

Através da rede AWDL que a transferência de dados é feita, utilizando HTTPS com comunicação via TLS utilizando troca de certificados. O protocolo HTTPS demonstra-se uma escolha boa para este caso de uso pois garante uma transmissão simples e segura dos dados. Antes de começar a transferência do arquivo em questão, o protocolo AirDrop precisa saber se o usuário destino autoriza o recebimento do mesmo e se o usuário está dentro da lista de usuários confiáveis cadastrada pelo remetente. A troca de informação nesta etapa de autenticação se dá através de mensagens REST.

Se tudo estiver correto, a transferência do arquivo, ou conjunto de arquivos selecionados, é efetuada através de um método POST do HTTP, que é adequado neste caso e bem comum para *upload* de arquivos em sites na web também. Uma vez encerrada a transferência, toda a rede AWDL é desfeita e os recursos alocados são liberados.

O modelo descrito e executado pelo protocolo AirDrop é bastante útil para transferências de arquivos pontuais, pois não utiliza um servidor de ponte e não deixa rastros, como *logs* de transferência ou metadados salvos em algum serviço externo. Portanto, este meio acaba fortalecendo a privacidade da transferência dos arquivos. No entanto, este mesmo modelo exige proximidade e conexão local entre os dispositivos envolvidos, pois o protocolo só funciona através do AWDL, que por sua vez, só é válido em redes sem fio ponto a ponto.

### 3.2.4 Comparison of Data Transfer Performance of BitTorrent Transmission Protocols

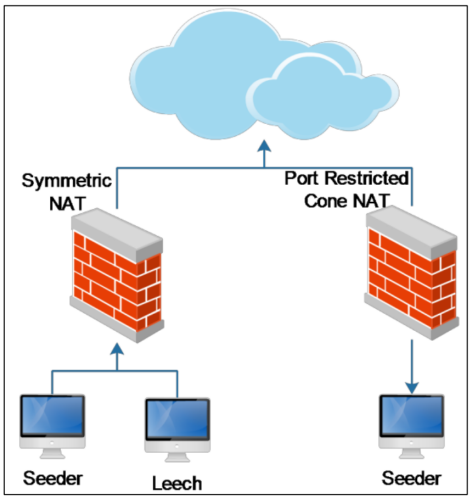
O trabalho de Özkan Canay e Halil Arslan (2019) demostra o uso do protocolo BitTorrent rodando em cima de *WebRTC*. Normalmente o protocolo BitTorrent funciona com base nos protocolos TCP ou *Micro Transport Protocol* (µTP) para transferência de arquivos P2P entre os participantes (COHEN, 2017). Neste trabalho, os autores demonstram a ideia da execução do BitTorrent através do uso de canais de dados, os *RTCDataChannels*, do WebRTC.

Apesar do protocolo BitTorrent poder funcionar sobre TCP, o protocolo µTP sempre tem preferência por causar menos congestionamento na rede. Pela própria natureza do BitTorrent, várias conexões simultâneas são realizadas a diversos computadores remotos. Manter conexões TCP abertas continuamente com todos os pacotes de controle inerentes ao protocolo causa um *overhead* desnecessário. Já o µTP funciona em cima de UDP e é

mais eficiente para o caso de uso presente. Este comportamento é o mesmo adotado pelo *WebRTC*, que pode funcionar em cima de TCP ou UDP, preferindo UDP quando possível.

No texto, os autores comparam a implementação do WebTorrent, que funciona sobre WebRTC com a implementação tradicional do BitTorrent através de análises de eficiência na rede, medindo a quantidade de pacotes trafegados e a largura de banda utilizada. O uso de WebTorrent pelo trabalho na comparação de tecnologias é bem vindo, pois é a biblioteca mais utilizada para se compartilhar arquivos na rede BitTorrent diretamente no browser, sem necessidade de nenhum programa adicional instalado ou *plugins* de terceiros (ABOUKHADIJEH, 2021).

**Figura 9 – Estrutura da rede usada no ambiente de testes com BitTorrent**




Fonte: Özkan Canay e Halil Arslan (2019)

O cenário de testes utilizado por (Özkan Canay; Halil Arslan, 2019) constitui-se de três membros conectados em rede, sendo dois deles em uma mesma rede local e outro presente em outra rede. Como mostra a Figura 9, as duas redes se conectam através da *internet* e contam com NAT fazendo o mapeamento do endereço e portas públicos para os respectivos endereços e portas privados. Destas três máquinas presentes, duas são *seeders*, fornecedoras de conteúdo, e uma é *leech*, consumidora de conteúdo. Destas, um *seeder* e um *leech* estão em rede local conectando à *internet* através de um *Symmetric NAT*. A outra máquina seeder está em outra rede, conectando à *internet* através de um *Port Restricted Cone NAT*.

Os autores optaram por fazer o teste dessa forma, pois esta topologia permite que se faça o teste das capacidades de conexão do WebRTC mesmo quando não há conexão direta entre as máquinas. O obstáculo está presente entre a máquina *seeder* e a máquina *leech*, que estão em redes diferentes e com uma configuração de NAT restritiva. Quando não há como estabelecer conexão direta, como no caso de uma máquina remota tentando conectar em outra que está atrás de um *Symmetric NAT*, o serviço TURN é utilizado

pelo WebRTC para estabelecer a conexão de forma indireta, com pacotes passando por um servidor de encaminhamento.

**Tabela 4 – Largura de banda utilizada e pacotes trafegados utilizando o protocolo µTP**

| Tam. do arquivo | Largura de banda | Número de pacotes |
|---|---|---|
| **100KB** | 119.772 bytes | 197 |
| **1MB** | 1.120.156 bytes | 1.010 |

Fonte: Özkan Canay e Halil Arslan (2019)

Na sua implementação tradicional, o protocolo BitTorrent prefere sempre conexões mais simples e não contempla servidores de encaminhamento. O mesmo opta por estabelecer conexões diretas sempre, *peers* que estão em redes sem alcance se conectam às outras, mas não recebem conexões. Outro ponto importante é que o protocolo não garante privacidade nem confidencialidade, pela própria natureza do compartilhamento público dos arquivos. Os dados trafegam abertamente na rede e analisadores de tráfego podem inspecionar o que está sendo transmitido, bem como origem e destino.

Na Tabela 4 é possível observar que o protocolo µTP acrescenta poucos dados além do *payload* do arquivo que está sendo transmitido e é eficiente na transmissão de dados P2P. Os dados extras correspondem a pacotes de controle que exercem a função de garantir que o arquivo esteja íntegro no destino ao final da transmissão. Segundo Özkan Canay e Halil Arslan (2019), em um arquivo de 1MB, o *overhead* foi de cerca de 17KB.

**Tabela 5 – Largura de banda utilizada e pacotes trafegados utilizando WebTorrent**

| Tam. do arquivo | Largura de banda | Número de pacotes |
|---|---|---|
| **100KB** | 136.246 bytes | 345 |
| **1MB** | 1.680.905 bytes | 5.467 |

Fonte: Özkan Canay e Halil Arslan (2019)

O uso de WebRTC pelo WebTorrent adiciona uma camada de abstração ao protocolo BitTorrent, pois agora o tráfego passa pelo DTLS. Neste cenário, a criptografia dos dados já é garantida na camada do WebRTC e não é opcional, pois faz parte do protocolo. Somado a isso, ainda existem pacotes de controle especiais do WebRTC para que a conexão de estabeleça, assim como uso de serviços STUN e TURN quando necessários.

Graças a estes fatores, o WebTorrent acaba sendo menos eficiente que o uso direto de µTP, como é possível observar na Tabela 5. A abordagem que utiliza as APIs nativas do *browser* troca mais de cinco vezes mais pacotes para transferir o mesmo arquivo exemplo de 1MB, além de possuir *overhead* de cerca de 33KB para este caso segundo Özkan Canay e Halil Arslan (2019).

Apesar de ser menos eficiente, a implementação do WebTorrent utiliza apenas ferramentas nativas do *browser* e graças ao WebRTC foi possível trazer o protocolo Bit-

Torrent aos navegadores utilizando apenas a linguagem *JavaScript*. O uso de WebRTC acaba sendo mais pesado que µTP, pois a suíte inclui diversos recursos extras, como a criptografia de ponta a ponta embarcada e obrigatória, além da a possibilidade do uso de servidores de encaminhamento para transpor ambientes em NAT restritivo.

### 3.2.5 Exploring WebRTC Potential for DICOM File Sharing

O artigo científico de Drnasin, Grgic e Gledec (2020) traz uma abordagem aplicada de WebRTC para compartilhamento seguro de arquivos médicos no formato *Digital Imaging and Communications in Medicine*[4] (DICOM). Arquivos médicos necessitam de sigilo absoluto e devem ser armazenados e trafegados com segurança. Os autores comparam o uso de WebRTC com outras duas alternativas existentes para troca de arquivos médicos: os protocolos *DICOM Message Service Element* (DIMSE) e DicomWeb STOW-RS.

Os próprios autores desenvolveram o aplicativo de teste para o estudo, utilizando WebRTC diretamente via *browser*. Para testes com o DIMSE foi utilizado o software médico ClearCanvas e o experimento com o protocolo DicomWeb STOW-RS foi feito através do serviço DICOMCloud. Todos os softwares foram instalados em um mesmo computador, para mitigar variações de ambiente em cada teste.

As duas alternativas testadas são fundamentalmente diferentes do modelo que utiliza WebRTC, pois utilizam cliente-servidor, com arquivos médicos armazenados no servidor de forma segura e sendo enviados através de TCP. Podendo também estar disponíveis através de APIs REST via HTTP como no caso do padrão DicomWeb.

O aplicativo de testes feito pelos autores depende de conexão síncrona entre dois *peers*, que se encontram em uma sala virtual e dali trocam arquivos de forma segura, privada e direta. Para estabelecer a sala e permitir que os participantes se encontrem é usado um serviço de sinalização ou *signaling*, que nada mais é do que um servidor web que coordena as conexões ponto a ponto, lidando com a negociação WebRTC. Para este fim, os autores escolheram a biblioteca de código aberto SignalMaster.

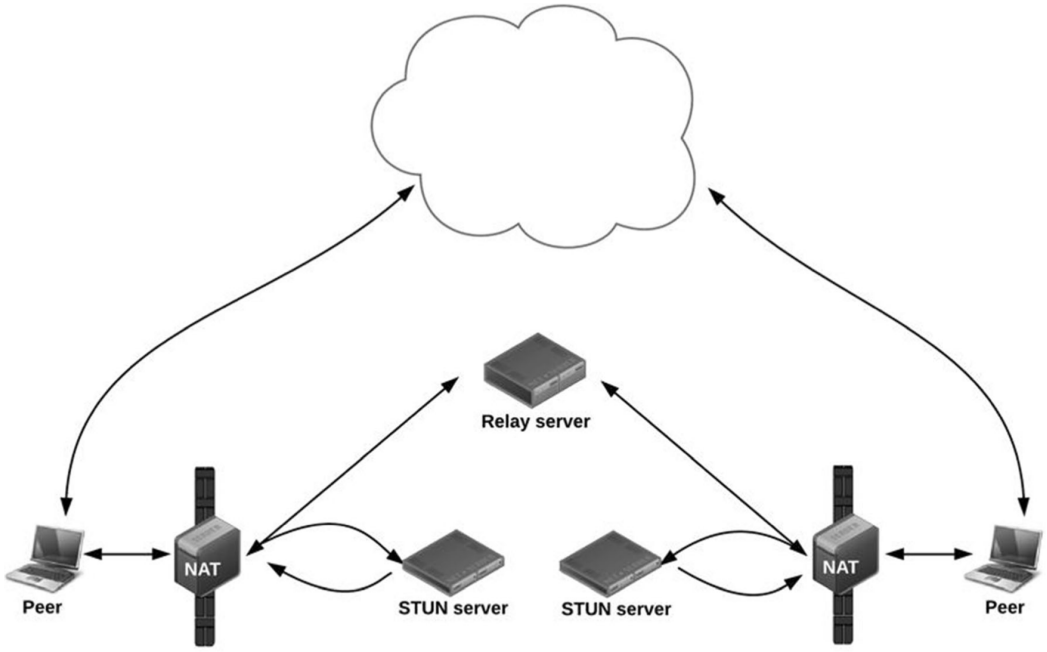
Por padrão o framework *Interactive Connectivity Establishment* (ICE) do WebRTC tenta sempre uma conexão direta entre as partes, porém, caso não seja possível, outros serviços são usados para obter a conexão, como STUN e TURN. É possível observar o diagrama de conexão entre as partes e com o servidor de sinalização na Figura 10.

A interface da aplicação dos autores é bem simples, o software apenas solicita ao usuário algum arquivo ou conjunto de arquivos que sejam compatíveis com o formato DICOM, feito isso a outra parte aceita a transferência e o aplicativo começa o envio do arquivo na rede.

Os testes foram realizados com quatro conjuntos de dados diferentes, presentes

[4] Formato de arquivo padrão utilizado para compartilhamento de informações médicas de pacientes.

**Figura 10 – Diagrama da rede utilizada no aplicativo de testes DICOM WebRTC**




Fonte: Drnasin, Grgic e Gledec (2020)

**Tabela 6 – Datasets utilizados nos testes de transferência de arquivos DICOM**

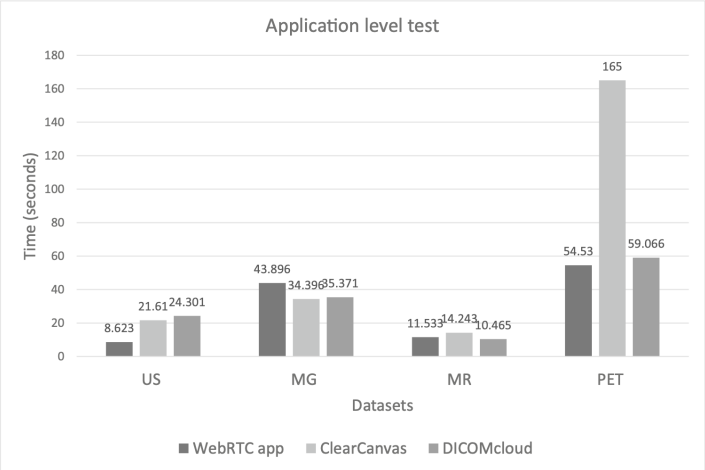
| ID | Dataset | Compression | Size of 1 file | Files number | Total size | Comment |
|---|---|---|---|---|---|---|
| A | US | No | 127 MB | 1 | 127 MB | Single large file |
| B | MR | No | 516 KB | 232 | 65 MB | Small MR case |
| C | MG | No | 54 MB | 4 | 217 MB | A small number of large files |
| D | PET | Yes, Lossy | ~ 68 KB | 4333 | 298 MB | A large number of small files |

Fonte: Drnasin, Grgic e Gledec (2020)

na Tabela 6. Cada conjunto contém uma configuração diferente de dados. O conjunto A possui apenas um arquivo grande, de 127MB e sem nenhum tipo de compressão. O B tem 232 arquivos somando 65MB a serem transferidos. O C contém apenas quatro arquivos de tamanho maior, totalizando 217MB. O último conjunto, com identificador D, possui arquivos comprimidos e em grande número, totalizando 4333 arquivos e 298MB a serem trafegados. Esta variação entre os conjuntos contribui para os testes a serem realizados, pois introduz diferentes casos de uso válidos. Uma única transferência de um arquivo grande se comporta de forma bem diferente de várias transferências de arquivos menores.

A Figura 11 mostra os resultados dos testes realizados pelos autores em uma conexão entre dois computadores utilizando um *link* de 50 Mbps. WebRTC se sai muito bem nos testes, sendo o mais rápido em dois dos quatro cenários. O protocolo DIMSE usado pelo ClearCanvas mostra seu gargalo no cenário com muitos arquivos pequenos,

**Figura 11 – Tempo de processamento de cada *dataset* DICOM entre as 3 aplicações testadas**




Fonte: Drnasin, Grgic e Gledec (2020)

sendo muito mais lento que os demais. Já o padrão DicomWeb também se sai bem, mostrando números próximos aos do aplicativo desenvolvido pelos autores.

Conforme Drnasin, Grgic e Gledec (2020), o uso de WebRTC para transferência de arquivos médicos é promissor, pois a tecnologia já é testada e provada em diversas aplicações da *internet* e garante confiabilidade e segurança como padrão, sem precisar configurações adicionais. Os autores ainda sugerem que a suíte de tecnologias pode ser usada para a conexão entre diferentes entidades médicas e possivelmente servir como alternativa ao atual padrão de transferências de arquivos DICOM.

# 4 FYLOR: WEBRTC NA PRÁTICA

O Fylor é o aplicativo desenvolvido pelo autor como forma de estudar e demonstrar o uso de WebRTC para troca de arquivos na prática. Feito como uma *Progressive Web Application* (PWA) com possibilidade de rodar também nativamente em iOS e Android, o *software* conta com capacidade de enviar e receber arquivos entre dispositivos através da rede, tanto via *internet* quanto via rede local quando ambos os clientes estão na mesma rede. A fim de auxiliar no entendimento e desenvolvimento de aplicações de compartilhamento de arquivos usando WebRTC, o Fylor está sendo disponibilizado como um *software* livre, passível de contribuições da comunidade, sob licença MIT. Como o Fylor utiliza vários componentes para funcionar, o código foi separado em vários repositórios sob controle de uma organização no GitHub específica do projeto[1].

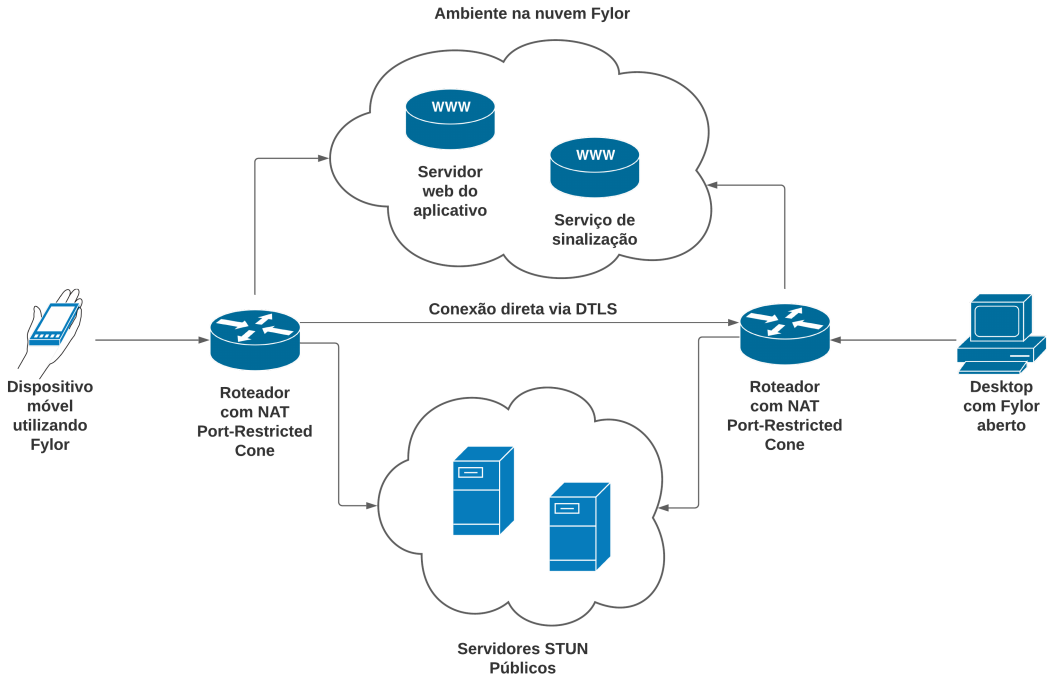
O *software* funciona em todos os *browsers* modernos que já implementam a suíte de APIs do WebRTC, além de fornecer binários que podem ser instalados no Android ou iOS para testes locais. O componente chave do conjunto para este aplicativo funcionar é o *RTCDataChannel*, que permite tráfego de dados via DTLS de forma direta entre os clientes. Isso já garante de forma nativa a criptografia do que está sendo trafegado e sua utilização é obrigatória.

O aplicativo conta com dois modos de operação, quando instalado ele pode funcionar em rede local sem necessitar de conexão à internet utilizando um serviço próprio de descoberta de pares na rede. Além disso, tanto no *browser* quanto em sua forma instalada é possível operar através da *internet* com um serviço de sinalização próprio sendo disponibilizado a partir de um servidor *web* na nuvem. Como é possível ver na Figura 12, o serviço de sinalização é responsável por indicar a intenção de conexão entre dois dispositivos e coordenar o início da conexão direta. A decisão sobre qual modo utilizar é transparente ao usuário, que simplesmente enxerga dispositivos próximos automaticamente e tem a opção de compartilhar um link para aparelhos remotos conectarem.

Ainda na Figura 12, é possível ver que a conexão consegue superar alguns cenários de uso de NAT ou *firewalls*, utilizando uma lista de servidores STUN pré-carregados no código fonte do aplicativo, quem têm o objetivo de mapear os endereços e portas públicos corretos aos dispositivos que estão tentando se conectar, entre os servidores informados na lista há o serviço de STUN público fornecido pelo Google. A lista é utilizada de forma a encontrar, entre os vários conhecidos, um serviço viável para uso no momento da conexão, estando ele o mais próximo possível da origem, disponível e respondendo às requisições que chegam.

[1] Repositórios disponíveis em: https://github.com/fylorapp

**Figura 12 – Diagrama de exemplo de conexão entre clientes Fylor usando serviço STUN**




Fonte: Autor

Quando não é possível estabelecer a conexão direta, como no caso de um NAT simétrico entre os clientes, é usado um servidor TURN sob domínio do autor que encaminha as mensagens entre os clientes. Este tipo de servidor é somente usado quando não há outra alternativa disponível, pois o mesmo acaba gerando tráfego de dados no servidor da nuvem e consequentemente aumenta a latência de transferência de dados entre os clientes e custos de operação para os mantenedores do servidor.

## 4.1 DESENVOLVIMENTO

O *software* foi desenvolvido utilizando o *framework* Expo[2], que permite um desenvolvimento paralelo de aplicações React para *web* e nativas para iOS e Android através do uso de React Native. Essa ferramenta se encaixou muito bem no desenvolvimento do Fylor pois um dos desafios era justamente operar também com a descoberta de pares na rede local, que ainda é uma barreira para aplicativos feitos no *browser* sem o uso de *plugins* de terceiros.

A linguagem padrão de desenvolvimento utilizada é JavaScript, com alguns módulos nativos no caso das aplicações instaladas, que permitem lidar com *sockets* e descoberta de dispositivos locais, bem como rápida manipulação de arquivos no sistema de arquivos local. Estes módulos utilizam Objective-C e C++ no iOS e Java e C++ no Android, sendo

[2] Framework para criação de aplicações React Native e Web disponível em https://expo.dev

que código em C++ é compilado para funcionar com o *Native Development Kit* (NDK) presente no Android e utiliza *Java Native Interface* (JNI) para comunicação entre Java e C++.

O aplicativo nativo também tem como dependência a versão nativa do WebRTC, compilada a partir do projeto original, biblioteca esta que já é inclusa nos *browsers*. Uma das funções do *framework* React Native é possibilitar que as funções JavaScript escritas consigam se comunicar com essa camada nativa de código, permitindo o envio e recepção de dados binários e textuais.

### 4.1.1 Descoberta de dispositivos locais

A descoberta de dispositivos locais está disponível somente na variante instalada do Fylor. Este recurso permite que instâncias do aplicativo Fylor reportem sua presença na rede utilizando técnicas de Zeroconf ou *Zero Configuration Networking*. Segundo Cheshire e Steinberg (2006), o uso destas técnicas surgiu a partir de um grupo de trabalho formado em 1999 e hoje já é comum em implementações de serviços de rede local que desejam ser descobertos sem configuração prévia, como impressoras, aparelhos de rede e dispositivos de *internet* das coisas em geral.

Existe mais de um protocolo de descoberta que pode ser caracterizado como Zeroconf, os mais conhecidos são o SSDP e o *Multicast DNS* (mDNS). O mDNS foi escolhido para o uso no Fylor devido à sua simplicidade de implementação e à disponibilidade de bibliotecas tanto em Android quanto no iOS. O mDNS foi desenvolvido originalmente para uso da Apple em seus dispositivos e depois se tornou um padrão. Ele funciona de forma simples onde pacotes UDP são direcionados à um endereço de *broadcast* especial e todas as máquinas da rede recebem os pacotes e decidem se respondem ou não (CHESHIRE; KROCHMAL, 2013).

O protocolo mDNS permite que seja registrado um domínio local e que consultas sejam feitas na rede para descobrir quais dispositivos já estão registrados com o padrão do domínio, na prática ele funciona de maneira bem semelhante ao DNS, porém sem servidor fixo respondendo as mensagens, as mesmas são direcionadas à toda a rede.

O Fylor, quando instalado no Android ou iOS, utiliza mDNS para registrar o domínio especial do tipo _fylor._tcp.local, juntamente com o nome do dispositivo disponível. A função deste registro é publicar a existência de uma nova instância do Fylor na rede, esperando que futuramente, um outro dispositivo faça uma busca por este registro. Quando uma a busca por domínios do tipo _fylor é realizada, todos os aplicativos ativos na rede respondem e então ficam visíveis entre si.

No entanto, somente realizar um registro de mDNS não seria suficiente para estabelecer uma conexão local, o registro mDNS apenas fornecerá informações úteis para

uma futura conexão, como endereço de IP e porta do serviço local de *signaling* do Fylor, este sim ficará responsável por negociar a conexão WebRTC. O serviço de *signaling* atua como um serviço do tipo TCP e isto também é indicado pela entrada mDNS cadastrada quando o aplicativo entra em operação.

**Figura 13 – Uso de mDNS no Fylor**

```
> dns-sd -B _fylor local.
Browsing for _fylor._tcp.local.
DATE: ---Thu 04 Nov 2021---
14:09:16.295  ...STARTING...
Timestamp     A/R    Flags  if Domain               Service Type         Instance Name
14:11:12.683  Add        2  16 local.               _fylor._tcp.         M21s de Matheus
14:11:12.811  Add        2   6 local.               _fylor._tcp.         M21s de Matheus
14:13:10.516  Add        2  16 local.               _fylor._tcp.         iPad de Matheus
14:13:10.646  Add        2   6 local.               _fylor._tcp.         iPad de Matheus
```

Fonte: Autor

A Figura 13 mostra a ferramenta *dns-sd*[3] consultando o domínio local especial (_fylor._tcp.local) que a aplicação Fylor utiliza para descoberta local. Na imagem, é possível observar que dois dispositivos foram encontrados. Neste exemplo, a aplicação Fylor está aberta em um dispositivo Android nomeado como M21s de Matheus e também em um dispositivo rodando iPadOS nomeado iPad de Matheus. Como é possível observar através da diferença dos *timestamps*, os dois aparelhos foram identificados assim que entraram na rede. O aplicativo foi aberto primeiro no Android e cerca de dois minutos depois aberto também no iPad. Os dispositivos aparecem repetidos pois são acessíveis através de duas interfaces diferentes da máquina que rodou o comando *dns-sd*, já que a máquina estava conectada tanto via Wi-Fi quando via rede cabeada.

### 4.1.2 Descoberta de dispositivos em redes remotas

Para casos que possuem conexão à *internet*, há a descoberta de dispositivos através do serviço de *signaling* na nuvem Fylor que podem ser utilizado em qualquer variante do aplicativo, seja ela instalada ou rodando como PWA no *browser*. A descoberta remota é bem simples de ser utilizada e usa o conceito de salas virtuais. Na implementação atual do Fylor, com transferências de um para um, cada sala pode ter no máximo dois membros.

A sala virtual é representada por uma URL única gerada automaticamente pelo servidor da nuvem Fylor, esta URL pode ser compartilhada com um outro dispositivo, que, ao acessar, entrará na sala virtual gerada anteriormente. A partir do momento que ambos os membros estão na mesma sala virtual, o processo de negociação da conexão WebRTC é iniciado. Esta tarefa é realizada pelo serviço de *signaling*.

O objetivo do Fylor de manter uma conexão P2P se mantém, o serviço de nuvem é usado apenas para abrir a conexão inicial entre as duas partes, pois o elemento em comum

[3] Ferramenta que consulta dispositivos usando mDNS disponível para sistemas Unix

entre as partes remotas é o servidor de descoberta. No entanto, o *software* não obriga o uso do servidor da nuvem Fylor, é possível configurar outro servidor de *signaling* desde que o mesmo possua a mesma API de interação. Se outro servidor for usado, é possível alterar a configuração do aplicativo apontando para o domínio deste servidor alternativo. Esta prática auxilia na descentralização do serviço, pois o serviço de *signaling* remoto não deixa de ser um ponto central de conexão inicial para cenários de conexão através da *internet*.

### 4.1.3 Controlando a comunicação com *signaling*

O Fylor usa dois tipos de serviços de *signaling*, o primeiro utiliza um servidor TCP local para a negociação e é usado apenas nos aplicativos da variante instalada do Fylor e quando é necessária a comunicação sem uso de rede externa (*internet*). O segundo método utiliza um servidor de Web Socket na nuvem Fylor que faz a negociação em tempo real de forma criptografada e segura, pois os Web Sockets seguros proporcionam o mesmo nível de segurança do HTTPS.

No caso de comunicação totalmente em rede local, o protocolo de *signaling* do Fylor inicia atuando em cima de uma conexão TCP direta entre os dois pares. Depois de aberto o canal de rede o processo de negociação e troca de metadados da transferência é feito pelo próprio canal de dados, com a segurança de ponta a ponta fornecida pela suíte WebRTC através do DTLS. O canal de dados que é aberto serve tanto para troca dos *buffers* binários de dados quanto para mensagens de negociação e é chamado no aplicativo de *File Transfer Channel*.

A Figura 14 mostra um trecho do código em JavaScript do Fylor, na imagem está representada a função *openFileTransferChannel*, que é a função responsável por iniciar o canal de dados entre as duas partes. Neste caso a variável *socket* segura a conexão TCP direta entre as partes, que serve como ponte inicial até o canal de dados ser aberto. O canal TCP de *signaling* direto entre as duas partes não conta com criptografia no momento, mesmo que este *socket* seja usado somente até a abertura do canal de dados, este sempre criptografado, o ideal seria a implementação futura de TLS para obter criptografia de todos os canais no uso local do aplicativo.

Na variante do Fylor disponível para navegadores de *internet* o serviço de *signaling* está presente em um servidor web disponível através de Web Sockets. Atualmente não há API direta para uso de *sockets* nativos no navegador, então o uso de Web Socket é uma alternativa para troca rápida de mensagens, no entanto não é possível servir um Web Socket direto do navegador, somente o uso como cliente é permitido, por isso aplicações WebRTC fazem o uso de um serviço de *signaling* em servidor.

A Figura 15 mostra mais um trecho de código do Fylor, na imagem é possível observar a implementação da geração de sala virtual. Nesta sala os membros podem

**Figura 14 – Início do canal de dados com troca de mensagens via *socket* TCP ou Web Socket**

```
export async function openFileTransferChannel({
  address,
  port,
  deviceName,
  local,
}) {
  return new Promise(async (resolve, reject) => {
    try {
      console.log(`Connecting to ${address} on port ${port}`);
      const socket = await createSocket({ address, port, local });
      socket.on('greeting', async () => {
        const channel = await startDataChannel(socket);
        channel.onopen = () => {
          resolve(channel);
        };
      });
      socket.sendMessage({ type: 'greeting', deviceName });
    } catch (e) {
      reject(e);
    }
  });
}
```

Fonte: Autor

**Figura 15 – Criação de sala virtual no serviço de *signaling* do Fylor**

```
io.on('connection', function(socket) {
  console.log('New socket connected');
  let roomId = socket.handshake.query.linkId || null;
  const rooms = Array.from(socket.rooms);
  if ((roomId === null || roomId === 'null') && rooms.length === 1) {
    // No room, create random room
    roomId = uuid();
  }

  socket.join(roomId);
  console.log('Assigned to room ' + roomId);
});
```

Fonte: Autor

trocar mensagens como se estivessem conectados diretamente, este processo é utilizado até a abertura do canal de dados do WebRTC. Na prática, o serviço de *signaling* server como *broker* ou intermediador das mensagens trocadas pelos pares, tornando-se uma maneira de simular conexão direta TCP através de Web Sockets.

Para cumprir os requisitos de uma PWA, a aplicação do Fylor é servida através

de HTTPS e a transmissão dos pacotes de *signaling* no Fylor através de Web Sockets é segura, pois utiliza a variação segura dos Web Sockets, conhecida como WSS.

### 4.1.4 Implementação de STUN e TURN

A fim de manter um bom nível de compatibilidade com diferentes tipos de rede, o Fylor faz uso tanto de serviços STUN como de TURN. Como visto no capítulo 2, serviços STUN são responsáveis por ajudar a transpor barreiras de NAT, enquanto serviços do tipo TURN auxiliam com NAT simétrico e bloqueios impostos por *firewalls*. O aplicativo desenvolvido conta tanto com serviços STUN públicos como um serviço STUN de uso próprio, além de possuir acesso a um servidor TURN de hospedagem própria na nuvem Fylor.

**Figura 16 – Lista de serviços STUN utilizados pelo Fylor**

```
export default [
  'stun:stun.l.google.com:19302',
  'stun:stun1.l.google.com:19302',
  'stun:stun.3cx.com:3478',
  'stun:fylor.app:3478',
];
```

Fonte: Autor

A Figura 16 mostra os servidores STUN utilizados pelo Fylor para sua operação, observa-se que três servidores são de domínio público, sendo dois hospedados pelo Google e um pela 3CX, que é uma empresa especializada em VoIP e que também tem aplicações WebRTC. Já o último servidor da lista é um servidor próprio hospedado para executar os testes propostos neste trabalho. O WebRTC segue a lista de servidores forma ordenada, de cima para baixo, então se houver alguma falha de conexão com o algum servidor, o próximo da fila será consultado e assim por diante.

**Figura 17 – Serviços TURN utilizados pelo Fylor**

```
export default [{ urls: 'turn:fylor.app', username: 'fylor', credential: 'turnfylor' }];
```

Fonte: Autor

Não é comum encontrar servidores TURN públicos disponíveis para uso geral, pois devido a própria natureza do serviço TURN de servir como ponte para os pacotes

de dados, o custo de manter o servidor aumenta muito com o uso contínuo. Deixar um serviço TURN aberto publicamente acaba se tornando inviável em muitos casos. Por isso, o *software* desenvolvido utiliza um serviço de TURN próprio como mostrado na Figura 17.

Os servidores STUN e TURN de hospedagem própria mostrados acima estão disponíveis no Fylor através de um único *software*: o Coturn. Ele atua como servindo STUN na porta TCP 3478 e TURN na porta UDP 3478, além de mapear as portas UDP 49152 até a 49200 para uso do *relay* TURN, toda a operação roda dentro de um *container* Docker disponível na nuvem Fylor. O Coturn é um *software* de código aberto muito utilizado em implementações WebRTC, VoIP e outros casos de uso. O *software* já é bem testado em várias aplicações que estão em produção no mercado e é fundamentado na própria RFC 5766 do *Internet Engineering Task Force* (IETF) que especifica o serviço de TURN (MATTHEWS; ROSENBERG; MAHY, 2010).

### 4.1.5 *Trickle ICE*

Consultas a servidores STUN e TURN podem demorar um certo tempo para executar a depender da disponibilidade de cada servidor e da latência da conexão. Por isso, o Fylor implementa o conceito de *Trickle ICE*. Esta prática permite que cada candidato descoberto seja comunicado à outra parte assim que possível, sem aguardar a descoberta completa de candidatos padrão do WebRTC.

**Figura 18 – Implementação de *Trickle ICE***

```
pc.onicecandidate = (event) => {
  if (event.candidate) {
    socket.sendMessage({
      type: 'candidate',
      label: event.candidate.sdpMLineIndex,
      id: event.candidate.sdpMid,
      candidate: event.candidate.candidate,
    });
  } else {
    console.log('End of candidates');
  }
};
```

Fonte: Autor

A Figura 18 mostra a recepção do evento de novo candidato ICE e como o código do Fylor logo envia a informação deste candidato novo para o *peer* remoto. Isso permite que assim que o número de candidatos for suficiente será aberta uma conexão de canal de dados. Em casos onde há possibilidade de conexão direta em rede local isso é crucial

para a velocidade de negociação do canal de dados, visto que não é necessário aguardar resolução de STUN e TURN.

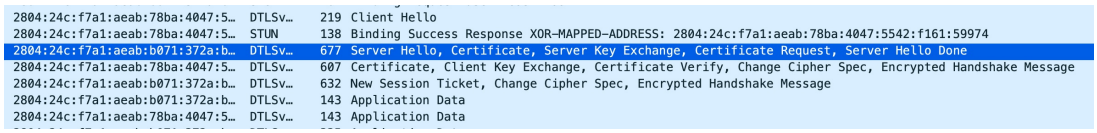
## 4.2 PROTOCOLO

O protocolo utilizado pelo Fylor é bem simples e consiste em trocas de mensagens de dois tipos: textuais em JSON, que são enviadas em um primeiro momento via camada de *signaling* e depois via canal de dados, ou do tipo binário, que trafega somente pelo canal de dados. As mensagens em JSON servem para vários fins, inicialmente carregam os dados na negociação de abertura de canal de dados entre os pares e depois orienta os dispositivos sobre quais arquivos serão enviados, o tamanho dos mesmos e em quantas partes serão divididos os arquivos. Já as mensagens binárias são usadas para enviar as partes de cada arquivo, até que sua transmissão esteja concluída.

Como o aplicativo também lida com *sockets* nativos na sua variante instalada, foi preciso desenvolver um método para garantir que as mensagens em JSON chegassem corretamente no destino, uma vez que nem sempre elas cabem em sua totalidade em um único pacote TCP. Para resolver isso, a técnica JsonSocket foi aplicada. Este tipo de mensagem encapsula cada JSON entre limitadores de início e fim, assim, o aplicativo aguarda até cada JSON recebido estar completo antes de fazer a interpretação do mesmo.

A camada de *signaling* no Fylor atua em alto nível, estando sobre TCP, na versão nativa, ou Web Sockets na versão PWA, então não é necessário ter controle adicional sobre retransmissão de mensagens, uma vez que os protocolos de camadas inferiores já lidam com isso. A única exceção são os eventos de desconexão, que exigem uma renegociação e retomada do ponto onde a conexão foi perdida.

**Figura 19 – Inspeção do início de conexão da camada de dados via Wireshark**




Fonte: Autor

Já a camada de dados atua sobre as tecnologias presentes no WebRTC e já inclui criptografia padrão, controle de fluxo de dados e garantir de entrega de mensagens ordenadas e completas de forma nativa através do DTLS. Para abrir o canal de dados é necessário efetuar alguns passos, neste momento é feita a seleção de qual candidato à conexão será utilizado, são trocados certificados e então o canal é liberado para uso com dados do usuário. Um resumo deste processo é visível na Figura 19, onde é utilizado STUN para definir o endereço e porta destinos, em seguida o *handshake* que estabelece a criptografia acontece, por fim os dados de aplicação são trafegados. A captura destes pacotes

de exemplo foi feita com Wireshark em um dos testes preliminares do Fylor. Também nota-se que o WebRTC preferiu o uso de endereço IPv6 neste caso.

Depois que a negociação via troca de candidatos é realizada, o aplicativo inicia o envio do arquivo selecionado. Neste momento somente o canal de dados é utilizado. Antes de enviar o arquivo, o remetente do arquivo envia um resumo do que será enviado ao destinatário, esta mensagem JsonSocket contém informações como o nome do arquivo em questão, seu tamanho total em bytes, o tipo do arquivo e em quantos pedaços ele estará dividido. Ao aceitar a transferência o destinatário envia uma mensagem do tipo *ACK* de volta ao remetente, a partir disso, o remetente inicia o envio dos pedaços de arquivo. Ao final da transferência, quando tudo estiver salvo do lado do destinatário, ele envia uma mensagem do tipo *END* ao remetente e a conexão é encerrada. Este fluxo vale para cada arquivo envolvido na transferência.

Quando a aplicação já está em funcionamento e transferindo dados de arquivos compartilhados, o controle do fluxo de mensagens é feito pelo Fylor. Cada arquivo é dividido em pedaços de 65536 bytes. Para evitar carregar arquivos grandes na memória, o Fylor também utiliza técnicas de *streaming* para a leitura dos arquivos em disco. Os arquivos são lidos em *buffers* de tamanho igual ao de cada pedaço, assim, é lido somente o que será enviado em seguida e logo após a memória consumida por este segmento é liberada. Dentro do aplicativo cada pedaço é chamado de *chunk*, e o envio dos *chunks* ocorre de forma controlada através da variável *bufferedAmount* do WebRTC, esta variável indica quantos bytes em tráfego ou em espera existem no canal de dados e quando vazia indica que todos os dados foram transferidos para o dispositivo remoto.

O tamanho dos *chunks*, neste caso de 65536 bytes, foi definido através da análise de como a leitura de arquivos do disco opera nos *browsers* modernos. Tanto no Chrome quanto em browsers baseados em WebKit como o Safari, a leitura dos arquivos via API do FileReader devolve *buffers* de 65536 bytes para arquivos maiores que 64 kibibytes, então optou-se pode utilizar este valor para sincronizar o número de chamadas à API do FileReader com os envios dos *chunks*.

## 4.3 RECURSOS

Dentro do Fylor é possível enviar um ou mais arquivos de qualquer tipo. Os arquivos podem ser selecionados através de um *file input* do navegador, quando sendo utilizado via PWA, ou via seletor de arquivos nativo do Android e iOS, quando usado em variante instalada. Uma vez indicados os arquivos, o usuário fica livre para selecionar para qual dispositivo deseja enviar. Os dispositivos identificados em rede local já aparecerão na lista de aparelhos disponíveis para receber arquivos. Para receber ou enviar arquivos pela *internet* basta compartilhar o *link* da sala randômica aberta ao acessar o aplicativo para o segundo aparelho, dispositivos na mesma sala virtual passam a se enxergar. Toda a

coordenação da sala é feita pelo servidor de sinalização na nuvem, porém nenhum dado sobre os arquivos propriamente ditos passa por este servidor.

Uma vez iniciado, todo o processo da transferência é feito de forma síncrona, com ambos os clientes conectados entre si a todo o momento. Em caso de queda de conexão entre os clientes, a transmissão é interrompida e pode ser retomada assim que a conexão for reestabelecida. Os arquivos não se encontram em nenhum local temporário que não sejam os armazenamentos de origem e destino. Não há servidor segurando estes arquivos para acesso posterior como tradicional no modelo cliente-servidor ou em um compartilhamento do tipo NFS.

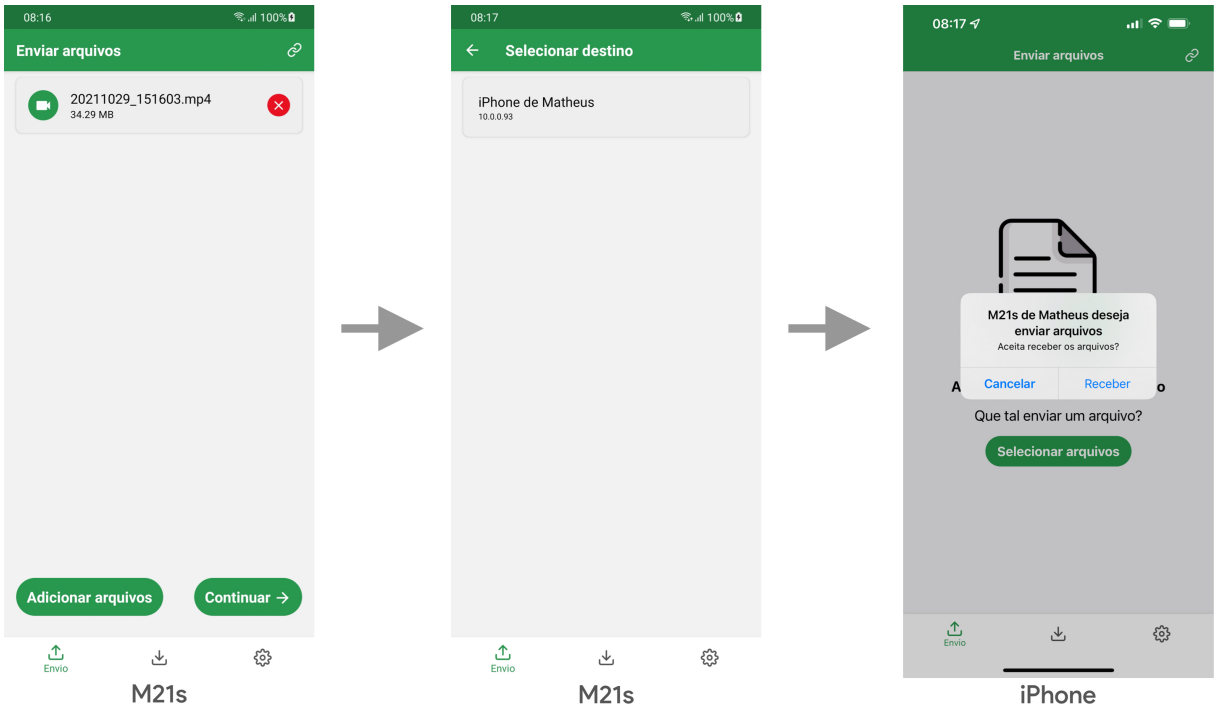
Além de enviar arquivos, o Fylor também guarda uma lista dos arquivos recebidos, para que possam ser transferidos à alguma pasta destino no sistema operacional do cliente final. Isto é importante, pois o *browser* não tem acesso direto ao sistema de arquivos em diversos sistemas. Um dos casos onde se observa este tipo de limitação é quando a PWA é usada no Safari dos sistemas iOS e iPadOS que rodam em iPhones e iPads respectivamente.

**Figura 20 – Tela inicial do aplicativo Fylor**


Fonte: Autor

Na Figura 20 é possível ver a tela principal do aplicativo quando usada em um dispositivo móvel, como um *smartphone*. O menu inferior permite alterar entre o modo de envio, visualização dos arquivos recebidos e configurações. Ao acessar o aplicativo sem o uso de *link* especial o usuário é levado à tela de envio, onde pode selecionar arquivos a enviar e qual o destino. O objetivo do Fylor é possuir uma interface de simples operação que possa funcionar tanto em dispositivos móveis de tela menor quanto em *desktops* ou *tablets*.

**Figura 21 – Fluxo das telas de envio local entre Android e iOS**

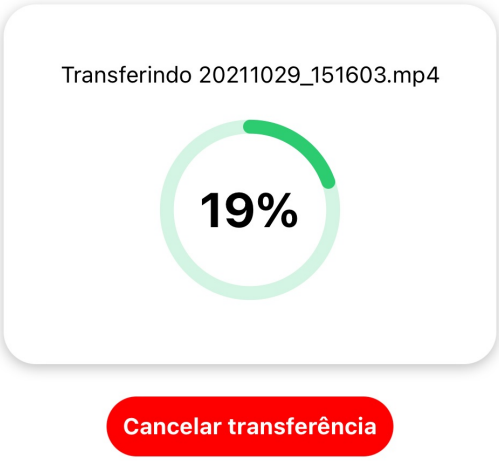



Fonte: Autor

A Figura 21 mostra o fluxo das telas de envio quando utilizada a transferência local. No exemplo, o remetente é um dispositivo Samsung M21s rodando Android e o destinatário é um iPhone 11. A primeira tela representada é apresentada logo depois da seleção do arquivo a ser enviado, neste caso um arquivo de vídeo do tipo MP4 com tamanho de 34,29 MB. A tela seguinte pede ao usuário que selecione o destino do arquivo, nesta tela são listados os dispositivos encontrados via mDNS ou que estão na mesma sala virtual se o aplicativo estiver conectado à *internet*, neste caso eles apenas estão em rede local via Wi-Fi. A tela final representa o lado do iPhone, que apenas está com o aplicativo aberto na tela inicial quando recebe a notificação que o M21s deseja enviar arquivos.

A notificação de que há arquivos a serem recebidos chegou ao iPhone via camada de *signaling* estabelecida pela conexão TCP direta entre Android e iPhone. Caso o iPhone aceite a requisição de receber os arquivos, o canal de dados é estabelecido e a transferência tem início.

**Figura 22 – Progresso da transferência de arquivo exemplo**




Fonte: Autor

Atualmente não há o recurso de transferência em segundo plano, portanto o aplicativo toma a tela inteira do dispositivo com uma janela modal contendo uma barra de progresso e um botão possibilitando o cancelamento da transferência. A Figura 22 mostra um recorte da tela do iPhone durante a transferência do arquivo de vídeo.

## 4.4 DESAFIOS

Fazer o Fylor funcionar nos mais diversos ambientes se mostrou o principal desafio durante o desenvolvimento. Por mais que as APIs do WebRTC sejam padronizadas entre os diferentes navegadores, algumas peculiaridades de cada *browser* tem de ser levadas em consideração, seja limites de uso ou comportamentos divergentes em casos específicos.

Logo no início do projeto foi identificada a limitação dos *browsers* atuais em lidar com *sockets* nativos, o que impossibilita a transferência local sem o auxílio de um serviço de *signaling* remoto. Este fato resultou na ideia de utilizar o *framework* Expo para desenvolver um aplicativo híbrido, que atuasse tanto no navegador quanto de forma instalada em Android e iOS. A versão instalada possui algumas vantagens como acesso à APIs do sistema operacional de cada plataforma, o que possibilita os testes com transferências sem conexão à *internet*.

Outro desafio percebido durante o desenvolvimento do aplicativo foi como lidar com a captura e recebimento dos arquivos utilizando o sistema de arquivos no navegador. As APIs de leitura e escrita do conjunto de APIs do *File System* na web ainda são limitadas e sua adesão ainda é baixa, no momento da escrita deste trabalho somente Chrome, Edge e Opera implementam as APIs necessárias para fazer um *stream* de escrita de arquivo direto à alguma pasta do usuário. Para contornar esse problema, o uso de

espaço temporário do *browser* se faz necessário, onde os arquivos ficam em uma *sandbox* com acesso exclusivo ao aplicativo e seu operador até que o usuário tomar alguma ação sobre eles.

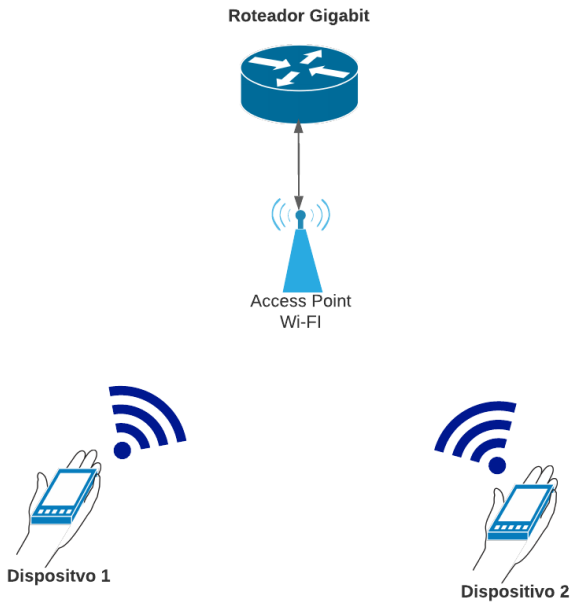
Aliado a tudo isso, existem os diferentes tipos de NAT, a presença ou não de *firewalls* e suas diferentes regras. Apesar destes problemas não serem exclusivos do caso de uso do Fylor, como pode-se observar em textos como o de Patel et al. (2015), o *software* desenvolvido como proposta busca evitar que estes problemas afetem o usuário, utilizando-se de diferentes serviços como STUN e TURN para contornar cada situação de forma transparente. Como não foi possível encontrar um serviço viável de TURN publicamente, a implementação de serviço próprio foi feita, e configurar todo o ambiente para isso também foi um desafio devido às várias configurações necessárias para que o WebRTC aceite o serviço como válido. No entanto, mesmo com todas as práticas aplicadas, em cenários com *firewalls* muito restritivos a comunicação ainda pode ser impossível.

# 5 ANÁLISE COMPARATIVA

Seguindo o objetivo deste trabalho, esta seção busca entender se o uso de WebRTC para transferência de arquivos pode ser viável em um ou mais cenários. Para isso, primeiro é preciso estabelecer os métodos de comparação com outras ferramentas e métricas a serem utilizadas. Como o aplicativo desenvolvido consegue operar em diferente dispositivos e situações de rede diferentes, optou-se por realizar um conjunto de testes com diferentes finalidades.

## 5.1 LABORATÓRIO DE TESTES

**Figura 23 – Rede Wi-Fi do laboratório para testes locais**




Fonte: Autor

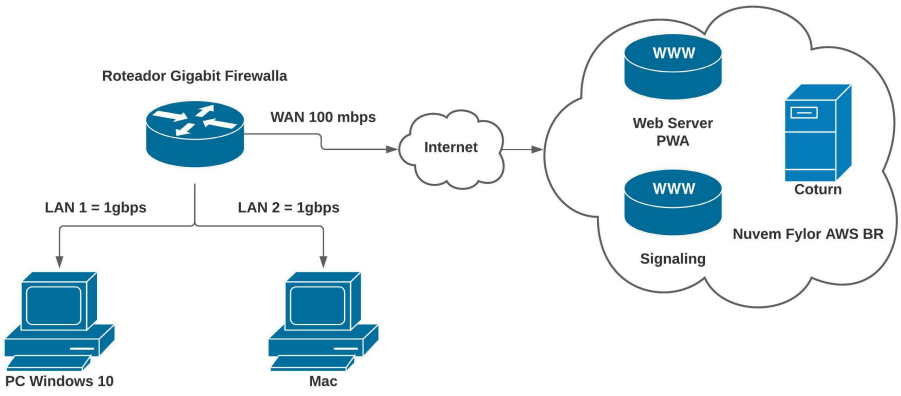
Para reforçar a isenção dos testes realizados, dois ambientes de rede foram criados localmente em laboratório. O primeiro para uso nos testes com dispositivos móveis. Sem conexão à internet, esta rede conta somente com um roteador e um *access point* Wi-Fi conectado ao roteador como mostra a Figura 23.

Este primeiro ambiente foi feito para os testes da variante instalada do Fylor em comparação com o AirDrop e com o Google Files, que utilizam conexão sem fio para sua operação. Os testes em comparação com AirDrop são feitos com dois dispositivos Apple: um iPhone 11 e um iPad 8, rodando iOS e iPadOS respectivamente. Os testes em comparação com Google Files utilizam dois dispositivos Android: um Samsung M21s e

um Xiaomi Mi A2 Lite. Cada aparelho está a um metro de distância do *access point*, sem obstáculos, assim como a há um metro de distância de um dispositivo para o outro.

O roteador utilizado opera em velocidades de até 3 gigabits e o *access point* Wi-Fi é do tipo 802.11ac com banda dupla de 2,4 Ghz e 5 Ghz. Este *access point* pode chegar a até 1300 mbps de velocidade nominal utilizando a banda de 5Ghz. Antes de cada teste, a velocidade do Wi-Fi em cada um dos dispositivos envolvidos é verificada, como forma de informar o limite teórico da conexão naquele momento.

**Figura 24 – Rede cabeada do laboratório para testes usando a PWA do Fylor**




Fonte: Autor

O segundo ambiente conta com uma rede mais complexa, como mostra a Figura 24, este ambiente é utilizado nos testes com a variante PWA do Fylor e podem ou não utilizar a conexão à *internet*. Por isso todos os elementos necessários para servir o serviço do Fylor devem estar presentes. Isso inclui o servidor de *signaling*, o servidor de TURN e o servidor que entrega a PWA aos *browsers*. Além disso o roteador responsável por encaminhar os pacotes trafegados é altamente configurável e permite criar diferentes cenários na rede que serão usados nos testes a seguir.

O roteador em questão é do modelo Firewalla Gold e possui 4GB de memória com CPU *quad-core* que garante até 3 gbps de velocidade na transmissão dos dados que passam pelo roteador. O roteador possui quatro portas de até 1 gbps cada, na configuração atual existem duas portas servindo como LAN e uma porta como WAN, com acesso à *internet* limitado a 100 mbps tanto de *upload* quanto *download*. Este roteador é muito útil para os testes pois seu sistema operacional é uma versão de Linux Ubuntu e isto o torna altamente programável, permitindo instalar e rodar programas como *tc* e *tshark* que serão usados em alguns testes a seguir.

A infraestrutura da nuvem Fylor montada para os testes está hospedada na AWS na região *sa-east-1* que fica em São Paulo. A região foi escolhida visando otimizar a latência de conexão, que é útil principalmente para o serviço TURN. Existem três componentes

principais rodando na nuvem, que são o servidor da aplicação PWA, o servidor de *signaling* que responde conexões Web Sockets e o servidor Coturn que serve solicitações de STUN e TURN.

Localmente existem dois computadores que serão utilizados nos testes, cada um conectado diretamente a uma porta LAN do roteador, o primeiro é um PC com Windows 10 com CPU AMD Ryzen 2600 de seis núcleos rodando a 3,4Ghz , 16GB de memória e placa de rede de 1 gbps que usa o Chrome 95.0.4638.69 como navegador. O segundo é um iMac com CPU Intel Core i5 4590 de quatro núcleos rodando a 3,3Ghz, 16GB de memória e placa de rede com velocidade de até 1 gbps, este computador está rodando a mesma versão do Chrome, porém para macOS.

## 5.2 METODOLOGIA UTILIZADA E RESULTADOS OBTIDOS

Seis tipos de comparação foram feitos, os cinco primeiros tipos são baseados em métricas de desempenho e comparam o funcionamento do Fylor em relação a algumas ferramentas em cinco situações de rede. Para mitigar resultados influenciados por variações de rede nos testes de desempenho, cada rodada foi realizada cinco vezes e os tempos apresentados são médias destas iterações de teste. Além disso, os testes foram sempre executados do início, isto é, ao final de cada iteração os aplicativos envolvidos, sejam *browsers* ou outros *softwares* eram totalmente fechados e reabertos.

Cada subseção a seguir mostra o cenário de testes realizado, as configurações de rede de cada experimento e explica o motivo da realização dos mesmos. Depois, em cada uma das subseções há a apresentação dos resultados de cada comparação com números obtidos dos testes em laboratório.

O sexto tipo de comparação é do tipo qualitativo e compara as características do Fylor e seu uso de WebRTC baseando-se em pesquisa das abordagens e características presentes em algumas ferramentas conhecidas no mercado, esta análise está na subseção 5.2.5.1.

Como o Fylor possibilita uso local sem a necessidade de internet através de sua variante instalada, ele pode ser comparado com *softwares* que fazem transferências em cenários similares como AirDrop e Google Files. Já a variante que funciona no *browser* de forma *online* pode ser comparada com outras ferramentas que usam rede convencional como Instant.io, que é baseado em WebTorrent e a já conhecida ferramenta SCP, que permite compartilhamento de arquivos via SSH.

Para realizar os testes foram selecionados três arquivos de tamanhos e características diferentes:

- **teste.txt**, um arquivo textual contendo 82.840 bytes (aproximadamente 83 kiloby-

tes). É o arquivo mais leve dos testes;

- **foto.png**, um arquivo de imagem do tipo PNG contendo 3.211.850 bytes (aproximadamente 3,2 megabytes);
- **grande.jpg**, um arquivo de imagem do tipo JPG contendo 14.282.246 bytes (aproximadamente 14,3 megabytes).

### 5.2.1 Fylor contra AirDrop

O AirDrop é um meio simples de trocar arquivos entre dois dispositivos Apple, ele é compatível com aparelhos modernos e que rodem iOS, iPadOS ou macOS. O grande diferencial do AirDrop é sua facilidade de uso, pois ele não depende de uma rede pré-estabelecida, ele monta uma rede *ad-hoc* entre dois aparelhos sob demanda antes de iniciar uma transferência de arquivos.

O protocolo base utilizado pelo AirDrop, como visto no trabalho de (HEINRICH et al., 2021), é o AWDL. Este protocolo é privado e foi desenvolvido pela Apple para uso em redes espontâneas. Além disso, o AirDrop troca metadados em HTTPS em uma funcionalidade semelhante ao uso do *signaling* pelo Fylor.

O Fylor também permite troca de arquivos sem a necessidade de acesso à internet, porém ele não utiliza rede *ad-hoc*, é necessário o uso de uma rede local, que pode ser uma rede Wi-Fi ou cabeada. Neste ponto o AirDrop possui vantagem, pois não depende de nenhuma rede existente e pode funcionar somente com a presença dos dois dispositivos envolvidos.

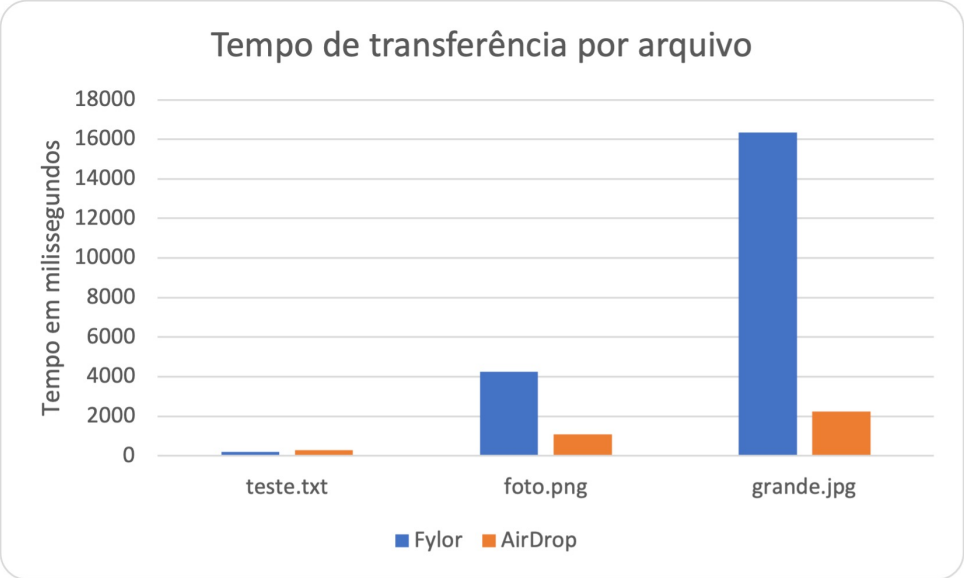
Como o AirDrop funciona em uma rede própria, não foi possível inspecionar o conteúdo do que foi trafegado, o foco do teste foi comparar o tempo de transferência de cada um dos três arquivos de teste estabelecidos. O teste se deu com um dispositivo iPhone 11 enviando arquivos para um iPad 8. Para medir o tempo de transferência no AirDrop foi utilizado um cronômetro digital disparado ao início e término da transferência, já no caso do Fylor, fez-se uso da ferramenta de depuração interna que indica o início e o fim de cada transmissão.

Para comparar as duas soluções foi utilizada a rede proposta na Figura 23. Porém como explicado o AirDrop cria a sua própria rede, então para manter a comparação justa os dispositivos estão a um metro de distância de um ao outro e também a um metro de distância do *access point*, formando um triângulo equilátero e sem obstáculos no caminho.

#### 5.2.1.1 Resultados

Ao realizar as iterações de teste com os três arquivos propostos, foram obtidos os seguintes resultados: no arquivo teste.txt o Fylor teve leve vantagem, transferindo o

**Figura 25 – Comparação de tempo de transferência entre Fylor e AirDrop**




Fonte: Autor

arquivo em 212 ms contra 300 ms quando usado o AirDrop. No segundo arquivo, o foto.png o AirDrop transferiu mais rapidamente que o Fylor, completando a tarefa em 1100 ms contra os 4250 ms levados pelo Fylor. Já no terceiro arquivo a diferença foi maior, o AirDrop levou apenas 2230 ms contra 16332 ms gastos pelo Fylor. Os resultados podem ser visualizados de forma gráfica na Figura 25.

É possível ver que o AirDrop leva um certo tempo para abrir a rede espontânea entre os dois dispositivos e negociando a transferência, mas depois de aberta a rede é eficiente em trocar arquivos rapidamente. Por isso o AirDrop tem resultados melhores nos últimos dois arquivos, mas acaba perdendo o comparativo no primeiro arquivo. O AirDrop se dá bem com arquivos grandes, pois o tempo de negociação acaba sendo irrelevante quando comparado com a velocidade de transferência dos dados.

O AirDrop inicia a negociação para a criação da rede própria via *bluetooth* e isto limita a distância que os aparelhos podem responder a uma requisição de compartilhamento na tecnologia. Além disso, como o AirDrop usa uma rede direta criada entre os dois dispositivos, ele depende da proximidade física entre os envolvidos e seu raio de alcance é menor do que o disponível em uma rede Wi-Fi convencional.

### 5.2.2 Fylor contra Google Files

O Google Files é um aplicativo de gerenciamento de arquivos que pode ser baixado para Android através da Play Store, em alguns dispositivos ele já vem instalado. Além de permitir manipular o sistema de arquivos de forma geral, ele serve como uma solução de transferência de arquivos análoga ao que faz o AirDrop, porém compatível somente com

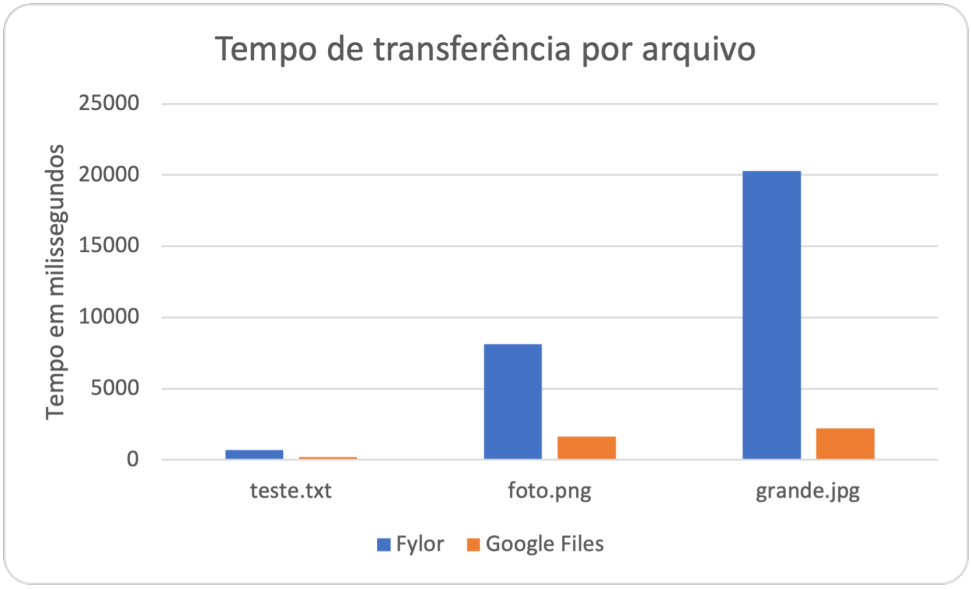
transferências entre dispositivos Android.

Assim como o AirDrop, o Google Files também cria uma rede própria após executar uma negociação inicial via *bluetooth*, ele utiliza um protocolo próprio operando em cima de Wi-Fi Direct para realizar a comunicação entre as partes (GOOGLE, 2021).

Para comparar as duas soluções foi utilizada a mesma rede presente no teste com o AirDrop. Para este teste dois dispositivos Android foram usados, um Samsung M21s e um Xiaomi Mi A2 Lite. Nos testes com cada arquivo o M21s é o remetente e o Mi A2 Lite é o destinatário. Os procedimentos de teste e cuidados foram os mesmos seguidos anteriormente, com distância de um metro entre roteador e dispositivos envolvidos, sem obstáculos no caminho. Os tempos foram capturados utilizando um cronômetro digital no caso do Google Files, para o Fylor fez-se uso da ferramenta de depuração interna que captura o início e fim de cada transferência.

#### 5.2.2.1 Resultados

**Figura 26 – Comparação de tempo de transferência entre Fylor e Google Files**




Fonte: Autor

A Figura 26 mostra os resultados dos testes realizados com os três arquivos. Em todos os testes o Google Files superou o Fylor, obtendo tempos de transferência mais baixos. No primeiro arquivo o Fylor levou 700 ms para completar a transferência enquanto o Google Files levou apenas 200 ms. No segundo arquivo o Fylor completou a tarefa em 8132 ms e o Google Files em 1650 ms. No terceiro e último arquivo a diferença foi maior, com o Fylor levando 20280 ms e o Google Files somente 2230 ms.

Nestes testes o comportamento foi semelhante ao teste contra o AirDrop, a criação da rede direta utilizando Wi-Fi Direct entre os dois dispositivos mostra-se eficiente para

transferência de arquivos de todos os tamanhos e supera o uso de rede local Wi-Fi do Fylor em tempo de transferência, pelo menos na implementação atual.

### 5.2.3 Fylor, SCP e Instant.io em rede local

Iniciando os testes com o segundo tipo de rede, proposto na Figura 24, há o cenário do uso de aplicativos para transferência de arquivos entre dois computadores em rede local, conectados através de cabos ethernet a um roteador central, este que provém acesso ao mundo externo através de sua interface WAN, através desta interface que os *browsers* usados nos testes chegam até os serviços a serem testados.

As ferramentas escolhidas para o teste entre os dois computadores foram: o próprio Fylor em sua versão PWA, a ferramenta SCP que permite a transferência pontual de arquivos via SSH de forma segura e a ferramenta *online* Instant.io, que opera de forma similar ao Fylor, porém permite a distribuição dos arquivos para N membros remotos devido à sua implementação base em Web Torrent. Estas ferramentas foram escolhidas pois cumprem o papel de enviar arquivos de ponta a ponta de forma direta, sem passar por um servidor de arquivos ou armazenar dados temporariamente em algum local.

O objetivo deste teste é estabelecer e medir a eficiência de uma transferência de arquivos via conexão local em cada uma das ferramentas, onde os dados dos arquivos trafegados passam somente pelo roteador Firewalla Gold até chegarem ao destinatário. Os testes foram sempre disparados do PC com Windows 10 enviando arquivos para o iMac.

O *software* para envio de arquivos via SCP utilizado no Windows é o pscp.exe e foi produzido pelos criadores do Putty como um *software* auxiliar, já o iMac está rodando um servidor SSH comum baseado em OpenSSH. As outras ferramentas testadas são aplicativos *web* e rodam através dos navegadores Chrome de cada máquina.

Duas métricas são comparadas neste teste, a primeira é o tempo de transferência e a segunda é a quantidade de bytes trafegados. O objetivo de medir o tempo de transferência é verificar qual solução consegue transferir os arquivos mais rapidamente, enquanto verificar a quantidade de bytes transmitidos ajudar a identificar qual solução apresenta menos *overhead* na rede. Como todas as transferências deste teste passam pelo roteador Firewalla Gold, foi possível registrar com precisão tudo o que foi trafegado em cada iteração através do *software tshark* instalado no próprio roteador e que capturou todos os pacotes trafegados.

Como Fylor e Instant.io são baseados em WebRTC, foi possível usar a ferramenta do próprio Chrome para depuração de aplicações WebRTC. Conhecida como *webrtc-internals* e disponível na URL especial *chrome://webrtc-internals/*. Esta ferramenta também consegue contar os bytes trafegados, bem como mostrar o estado dos canais de co-

municação ativos. Os dados de tráfego do Fylor e do Instant.io foram capturados usando este método.

**Figura 27 – Método *Measure-Command* usado para medir os tempos do SCP**

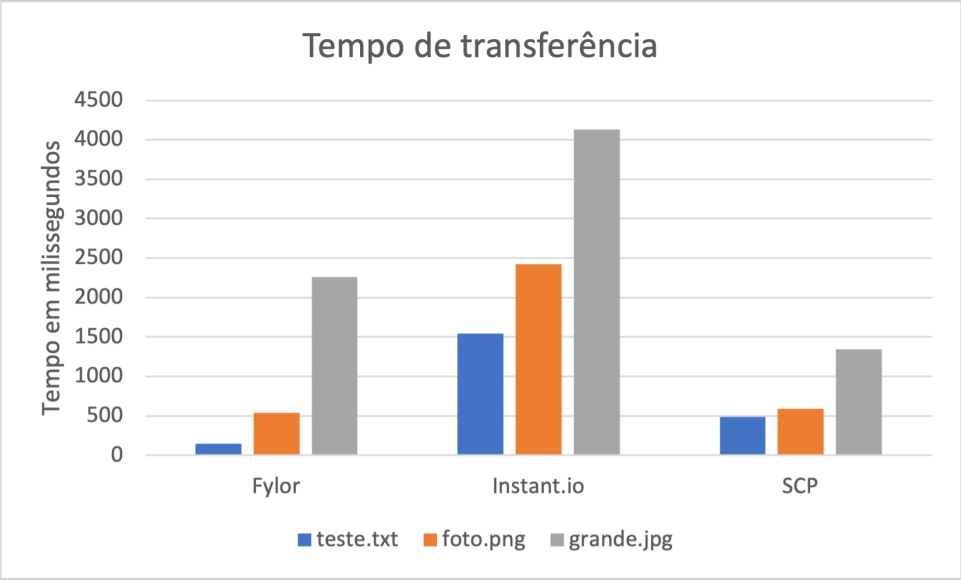
```
Measure-Command { .\pscp.exe -l matheus -pw '[redacted]' ./teste.txt matheus@10.0.0.137:/tmp/teste.txt }
```

Fonte: Autor

Para medir os tempos dispendidos em cada transferência no SCP foi utilizado o método *Measure-Command* do Windows PowerShell ao disparar o comando de cada transferência, este método devolve o tempo que o comando vinculado demora para executar e permitiu que se auferisse os tempos de cada iteração de forma automática, o comando completo utilizado é demonstrado na Figura 27. As medições de tempo no Fylor e Instant.io foram realizadas utilizando eventos disparados pelos próprios depuradores de cada aplicação.

#### 5.2.3.1 Resultados

**Figura 28 – Comparação de tempos de transferência entre Fylor, Instant.io e SCP**

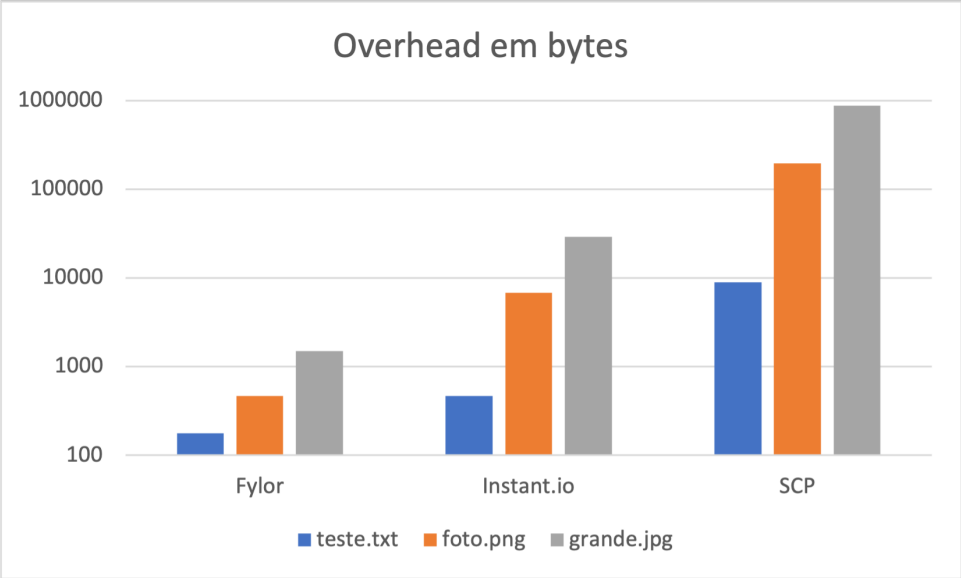



Fonte: Autor

A Figura 28 traz os resultados dos tempos de transferência de cada *software* para cada arquivo testado. O SCP teve a melhor média de resultados com os tempos de 486 ms para o primeiro arquivo, 594 ms para o segundo arquivo e 1341 ms para o terceiro arquivo. O Fylor foi o segundo colocado obtendo o melhor tempo no primeiro arquivo com apenas 150 ms, 540 ms no segundo arquivo e 2259 ms no terceiro. Por fim, o Instant.io obteve 1540 ms no primeiro arquivo, 2423 ms no segundo arquivo e 4131 ms no terceiro arquivo.

A quantidade de bytes trafegados também foi medida, neste teste o Fylor possui a menor quantidade de bytes trafegados em geral, com 83.014 bytes recebidos no canal de dados no primeiro arquivo, 3.212.318 bytes recebidos no segundo arquivo e 14.283.731 bytes recebidos no terceiro arquivo. O Instant.io ficou em segundo lugar com 83.307 bytes recebidos no primeiro arquivo, 3.218.665 bytes recebidos no segundo arquivo e 14.311.469 bytes recebidos no terceiro arquivo. Por fim, o iMac receptor da transferência SCP recebeu 91.722 bytes no primeiro arquivo, 3.405.910 bytes no segundo arquivo e 15.156.662 bytes no terceiro arquivo.

**Figura 29 – Comparação de *overhead* de bytes trafegados entre Fylor, Instant.io e SCP**




Fonte: Autor

A Figura 29 mostra os bytes de *overhead* enviados a cada transferência, onde o Fylor se sai melhor com 176 bytes de *overhead* no primeiro arquivo, 468 bytes no segundo arquivo e 1485 bytes no terceiro arquivo. Depois vem o Instant.io com 467 bytes no primeiro arquivo, 6805 bytes no segundo arquivo e 29223 bytes no terceiro arquivo. Por fim, há o SCP com 8882 bytes no primeiro arquivo, 194060 bytes no segundo arquivo e 874416 bytes de *overhead* no terceiro arquivo.

É possível ver o benefício do protocolo baseado em UDP do WebRTC fornecendo a diminuição do overhead em relação ao protocolo SSH que roda em cima em TCP utilizado nas transferências SCP. No entanto, o SCP se deu melhor nos tempos de transferência nos testes realizados.

### 5.2.4 Fylor, SCP e Instant.io em rede com perdas

Nem todos os tipos de rede do mundo real funcionarão tão bem para transferências de arquivos como a proposta no laboratório. Problemas podem aparecer, pacotes serem

perdidos e ocorrerem instabilidades na rede. Devido a isso, testes com redes instáveis são pertinentes. Para realizar os testes foi utilizada a mesma rede do cenário da Figura 24, porém o comando *tc* foi executado no roteador antes dos testes.

O *tc* é um comando especial do Linux que tem o objetivo de controlar o tráfego em uma rede, ele atua em uma interface de rede específica a ser definida pelo usuário, a partir disso é possível configurar uma série de comportamentos que afetam o encaminhamento de pacotes.

**Figura 30 – Comando *tc* executado no Firewalla Gold antes do teste com perdas**

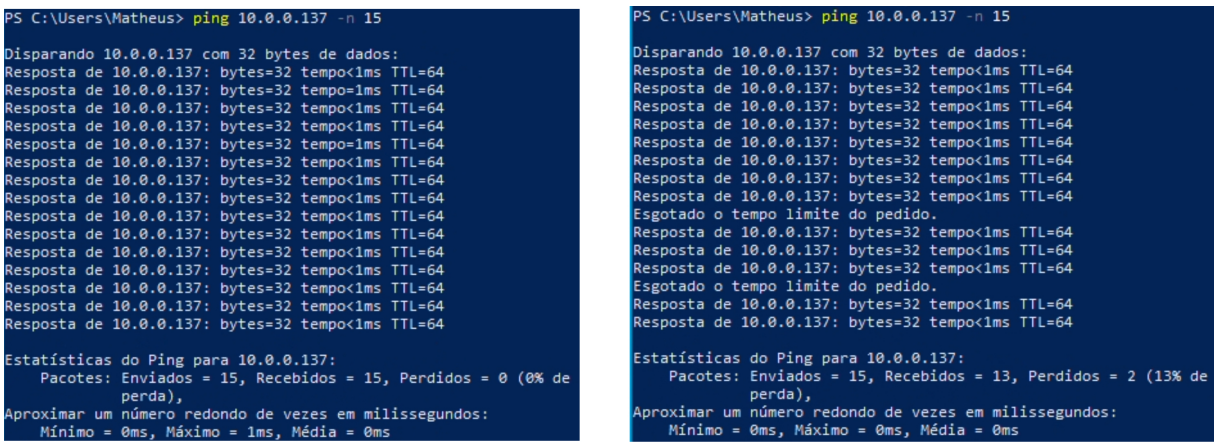
```
pi@firewalla:~ (Firewalla) $
pi@firewalla:~ (Firewalla) $ sudo tc qdisc add dev eth3 root netem loss 10%
```

Fonte: Autor

A Figura 30 mostra o comando exato que foi executado no roteador antes do início desta seção de testes. Este comando direciona o roteador a perder, ou ignorar, 10% dos pacotes que passarem pela interface *eth3*, esta interface corresponde ao local onde o iMac está conectado. Por tanto, para os testes a seguir o iMac perderá em média 10% dos pacotes que lhe forem direcionados. Como os testes sempre são realizados a partir do PC com Windows, todas as transferências serão afetadas por esta alteração na rede.

A hipótese deste teste é que as transferências utilizando Fylor e Instant.io possam ter vantagem devido ao seu número menos de pacotes trafegados, o que favorece estas duas ferramentas em situações de rede instável. Além disso, teoricamente o WebRTC deve tomar conta de qualquer retransmissão necessária para atuar no cenário novo com perdas.

**Figura 31 – Comando *ping* disparado contra o iMac antes e depois do *tc***




Fonte: Autor

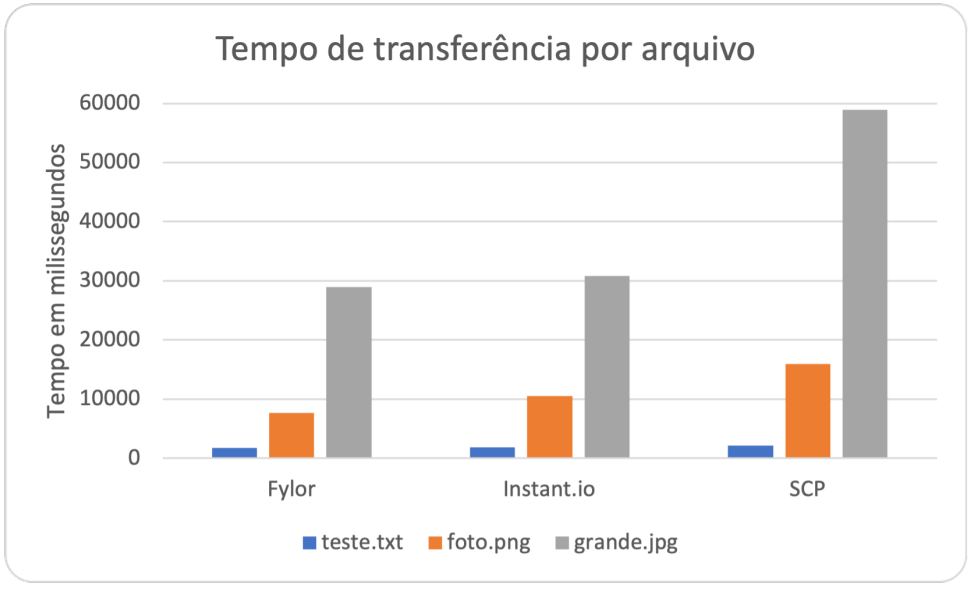
Neste teste não há como garantir uma condição de rede exatamente igual para todos e isto faz parte da natureza do teste, por isso a regra das cinco iterações se mantém. O objetivo do teste é capturar os tempos de transferência nesta rede com perda de pacotes.

Antes de realizar os testes é preciso avaliar o que de fato ocorreu na rede após do comando *tc* ter sido executado. Para comprovar que o ambiente está de fato diferente, o comando *ping* para o iMac a partir do PC Windows foi rodado antes e depois da execução do comando *tc*. Antes da mudança, nenhum pacote foi perdido num conjunto de 15 pacotes, depois da alteração, 2 pacotes foram perdidos, correspondendo a 13% do montante enviado como é visível na Figura 31.

Para obter os resultados comparando a métrica de tempo de transferência os mesmos métodos realizados no teste anterior foram aplicados, obtendo os tempo total de transferência do SCP, Fylor e Instant.io.

#### 5.2.4.1 Resultados

**Figura 32 – Comparação do tempo de transferência entre SCP, Fylor e Instant.io em rede com perdas**




Fonte: Autor

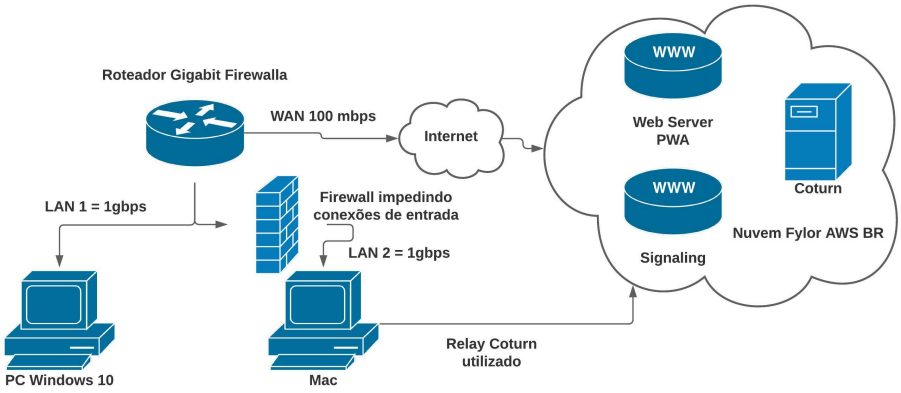
A Figura 32 é uma representação gráfica dos resultados obtidos neste teste. É possível ver que de fato tanto Fylor quanto Instant.io tem resultados melhores que SCP, principalmente para o terceiro e maior arquivo. Como o SCP trafega mais pacotes, a probabilidade da quantidade de pacotes perdidos enviados pelo protocolo é maior que a presente no Fylor e Instant.io.

Todas as ferramentas, no entanto, tiveram tempos consideravelmente maiores que no teste anterior, como já era esperado. Também vale ressaltar que todas as transferências

ocorreram sem problemas até o final, nenhum dos programas teve algum tipo de exceção ou erro durante a execução.

### 5.2.5 Fylor contra Instant.io forçando a utilização de TURN

**Figura 33 – Rede do segundo ambiente de testes modificada para forçar o uso de TURN**




Fonte: Autor

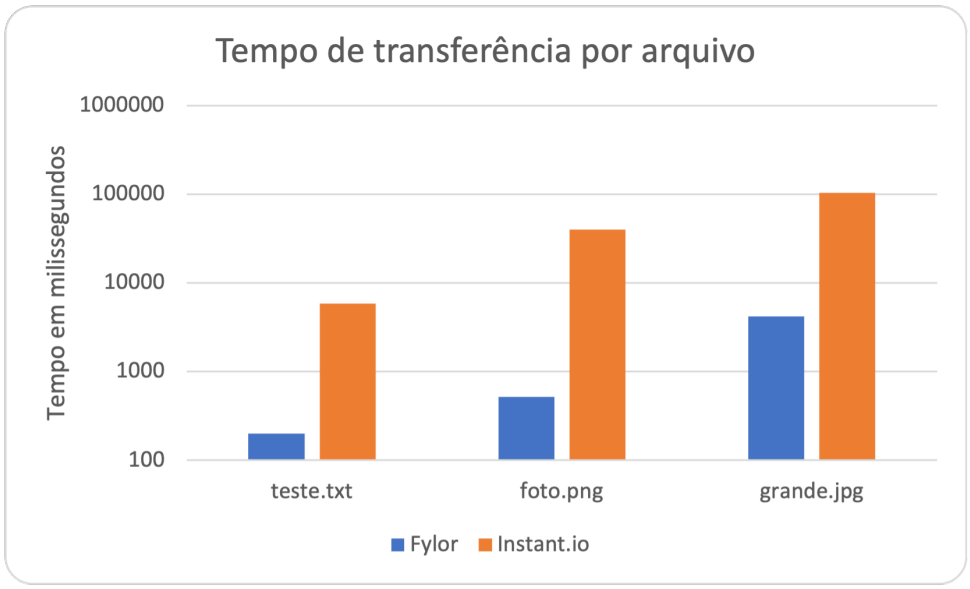
Existem casos nos quais uma conexão direta ponto a ponto não é possível. Para simular isso em laboratório, o cenário da rede proposta na Figura 24 foi modificado. Para este teste foi adicionada ao roteador uma configuração que barra conexões de entrada ao iMac. Isto faz com que aplicações que dependam de conexão direta falhem e tenho erros de *timeout*.

A Figura 33 exemplifica como a rede ficou estabelecida para este teste. O objetivo do teste é forçar o uso de TURN e ver como as aplicações se comportam em termos de tempos de transferência. O SCP não está presente neste teste, pois como esperado ao tentar conectar ao iMac foi obtivo o erro de *timeout*, pois o *firewall* barra todas as requisições de entrada. A navegação de *internet* através do iMac continuou funcionando normalmente e foi possível chegar aos serviços na nuvem Fylor e Instant.io.

#### 5.2.5.1 Resultados

Os tempos obtidos pelo Fylor não ficaram muito longe daqueles já obtidos na rede local, considerando que a WAN utilizada é rápida e conta com 100 mbps de capacidade, e ainda que a infraestrutura na AWS está próxima fisicamente e com boa rede de acesso, pode se dizer que o uso de TURN neste caso não afetou muito o desempenho. A figura 34 mostra os resultados obtidos neste teste. O Fylor obteve tempos de 200 ms, 521 ms e 4160 ms no primeiro, segundo e terceiro arquivo, respectivamente. Já o Instant.io obteve tempos piores, com 5865 ms no primeiro arquivo, 39764 ms no segundo e 103987 ms no terceiro arquivo.

**Figura 34 – Comparação do tempo de transferência entre Fylor e Instant.io usando TURN**




Fonte: Autor

**Figura 35 – Localização do servidor TURN utilizado pelo Instant.io**

Input interpretation

relay.socket.dev (domain)

Web hosting information

| | |
|---|---|
| name | Linode |
| location | San Jose, California, United States |
| host coordinates | 37° 17′ 49″N, 121° 49′ 8″W |

Fonte: Autor

Analisando os logs do *webrtc-internals* foi possível ver que o Fylor selecionou o servidor TURN corretamente e conectou ao Coturn após tentar comunicação direta e com uso de STUN. Já no caso do Instant.io, houve demora nas respostas do servidor TURN utilizado pela ferramenta, que, segundo o analisado é o servidor disponível em relay.socket.dev e que se encontra nos EUA como mostra a Figura 35.

É notável o Instant.io possui números ruins quando aliado ao uso de TURN. Percebe-se que o servidor não fornece tanta banda de dados aos seus usuários, segundo analisado durante a transferência dos testes. É possível que os desenvolvedores prefiram garantir o acesso ao serviço para o maior número de pessoas possível limitando a banda para cada dispositivo remoto.

### 5.2.6 Comparação de usos e características de soluções de compartilhamento

Após todos os testes em laboratório pode-se perceber que aplicações baseadas em WebRTC como o Fylor ou Instant.io possibilitam trocas de arquivos em vários cenários de rede. Podendo até mesmo operar sem um servidor de *signaling* dedicado como no caso do Fylor em sua variante nativa.

A fim de comparar qualitativamente alguns dos tipos mais usados de compartilhamento de arquivos hoje, os *softwares* de compartilhamento foram separados em três categorias:

- P2P na web, que abrange desde o Fylor, Instant.io até o BitTorrent e Web Torrent e muitos outros;
- *Drives* na nuvem, que representa os mais variados discos virtuais disponíveis hoje como Google Drive, OneDrive e iCloud Drive;
- Direto, esta categoria representa os meios diretos de transferência de arquivos, que não dependem de rede existente e inclui, por exemplo, o AirDrop e Google Files.

**Tabela 7 – Comparação qualitativa de soluções de compartilhamento de arquivos**

| **Característica** | **P2P na web** | ***Drives* na nuvem** | **Direto** |
|---|---|---|---|
| **Ambiente** | descentralizado | centralizado | descentralizado |
| **Usos** | envio pontual | armazenamento | compartilhamento |
| **Foco** | privacidade | disponibilidade | facilidade |
| **Meio** | *web* | *web* | conexão física direta |
| **Perda** | praticidade | privacidade | flexibilidade |

Fonte: Autor

A tabela 7 resume um conjunto de características presentes em cada uma das categorias, percebe-se que não há meio perfeito de compartilhamento de arquivos, que sirva para todos os cenários possíveis. Ao contrário, cada tipo possui uma série de vantagens e desvantagens e algumas das características que fortalecem cada categoria em um determinado cenário de uso pode ser ruim em outro, resultando em *tradeoffs* que o usuário deve levar em conta ao escolher um meio ou outro.

Ao compartilhar um arquivo em um disco virtual na nuvem, o usuário ganha praticidade e disponibilidade, porém perde em privacidade, ele precisa confiar na entidade centralizada que opera o serviço. Ao mesmo tempo que ao enviar um arquivo pelo Fylor não é prático em diversos casos, o usuário precisa estar sincronizado com a outra parte, ambos os aplicativos devem estar funcionando ao mesmo tempo, pois todo o processo é síncrono.

O compartilhamento direto é rápido e fácil, não depende de nenhuma rede estabelecida ou configurações prévias, somente precisa de dois aparelhos próximos um ao outro. No entanto não é tão flexível, não consigo enviar a longas distâncias, por exemplo.

Com base na análise feita até o momento pode-se perceber que o Fylor e outros aplicativos de transferência via WebRTC tem espaço e podem crescer no futuro, ainda existem obstáculos a serem superados como maior disponibilidade das APIs de acesso direto aos recursos do sistema operacional por parte do *browser*, que podem ampliar os horizontes das aplicações neste formato.

# 6 CONCLUSÃO

Por mais que o tema transferência de arquivos em rede não seja novo e esteja presente desde o início da computação distribuída, ainda há o que ser explorado. Novos ambientes de desenvolvimento surgem a todo o momento e sempre há espaço para refinamento e novas maneiras de se executar tarefas comuns. Isto agora acontece com a popularização do WebRTC nos navegadores, esta tecnologia é uma excelente porta de entrada para o compartilhamento de arquivos entre navegador sem a necessidade de instalação de *plugins* ou uso de outros *softwares*. Sendo necessário apenas que o código da aplicação rica escrita em JavaScript fazendo uso das APIs nativas do *browser* seja entregue por um servidor *web*. Graças às APIs modernas disponíveis atualmente, depois da primeira instalação (uso no navegador), a aplicação pode até mesmo ser usada sem conexão à *internet*.

Para fundamentar este trabalho, foi feita uma análise de algumas das diferentes alternativas para transferência de arquivos disponíveis no meio da ciência da computação. Trabalhos recentes relacionados a área de estudo foram levantados e analisados a fim de entender o que está sendo estudado e quais são propostas abordando o problema no meio acadêmico nos dias de hoje. Alguns filtros foram estabelecidos para assegurar a obtenção de trabalhos relevantes e que possam ser comparados ao uso de WebRTC, atendendo tanto o modelo tradicional de cliente-servidor quando o modelo P2P que é o foco desta pesquisa.

Os textos de cada autor escolhido foram lidos e explorados buscando trazer a contribuição de cada um para o tema. Como adição própria ao estudo, foi desenvolvido um aplicativo, o Fylor, que fez uso da suíte de tecnologias WebRTC e que busca atender diversos dos casos de uso de transferência de arquivos, tanto pela *internet* quanto localmente.

Desenvolver o Fylor se mostrou um dos principais desafios encontrados no decorrer do trabalho, pois diversas características do WebRTC tiveram de ser estudadas e entendidas para que o funcionamento da aplicação se desse de forma correta entre os diversos ambientes. Como o código de uma aplicação *web* sempre é interpretado pelos diferentes navegadores, há de se garantir o funcionamento pleno nas plataformas suportadas. Além disso, também foi crucial garantir que a aplicação funcionasse entre os modelos de rede encontrados com facilidade no dia a dia, como uso de alguns tipos de NAT e *firewalls* muitas vezes encontrados no ambiente corporativo.

Seis tipos de comparação foram feitas, cinco focando em desempenho de transferência de arquivos entre diferentes cenários de rede e uma comparação qualitativa entre

diferentes categorias de compartilhamento de arquivos. Através do desenvolvimento e testes com o Fylor, este trabalho conseguiu comprovar que o WebRTC pode ser utilizado como forma de envio de arquivos. Foram realizadas comparações tanto com ferramentas estabelecidas no mercado como com métodos mais recentes, como o Instant.io que também utiliza WebRTC como base.

Com base nos resultados obtidos, é possível verificar que o Fylor não é a melhor alternativa em todos os cenários, porém é viável em todos os cenários propostos, conseguindo concluir o envio e recebimento de arquivos seguindo o que foi proposto neste trabalho. Em alguns casos o Fylor consegue superar alternativas como SCP e Instant.io para transferências locais, porém seu foco é justamente em ser uma solução simples que serve para demonstrar a flexibilidade do uso do WebRTC.

Mesmo que em termos de desempenho absoluto o WebRTC não seja a melhor solução em muitos casos, a sua portabilidade e facilidade de implementação em diferentes plataformas sem a necessidade de grandes modificações de código é uma das principais vantagens da tecnologia. No caso de uso via navegadores modernos, o código pode ser simplesmente interpretado pelo navegador, que lidará com a comunicação com o sistema operacional e seus componentes. Como grande parte dos sistemas operacionais já contam com navegador de internet incluso, ou de fácil instalação, o WebRTC acaba estando presente de forma direta para uso de aplicações *web*, que apenas precisam fazer uso da tecnologia. Quando a portabilidade via navegador é levada em conta, o WebRTC acaba se tornando uma alternativa ainda mais interessante, principalmente em cenários onde não se pode instalar nenhum programa novo.

Ainda que o WebRTC já esteja em um estado maduro de desenvolvimento e implementação, há o que ser feito para melhorar o desempenho e funcionamento da suíte e proporcionar uma experiência de uso ainda melhor para casos de uso com rede local. Algumas APIs como o uso direto de *sockets* no *browser* podem ser úteis e podem dispensar o uso da variante nativa do aplicativo Fylor, por exemplo. No entanto, a especificação de Raw Sockets ainda é apenas um rascunho proposta pelo *Web Incubator Community Group* (WICG) no momento da escrita deste trabalho (WILLIGERS, 2021).

Estudos utilizando compressão de dados podem ser feitos tanto em cima da transferência de arquivos utilizando WebRTC, uma vez que muitas soluções de mercado não enviam os arquivos sem antes comprimir na origem quando necessário e fazem o processo reverso quando o arquivo chega no destino. Essa prática pode ganhar tempo de transferência, principalmente para arquivos grandes e que possuem padrões nos seus dados. Tecnologias novas e emergentes como o WebAssembly, que permitem execução de código de baixo nível no *browser* (como C++ e C), podem auxiliar em uma futura implementação de compressão rápida de arquivos no cliente.

Outra área de estudo interessante para continuar a pesquisa seria entender os

sistemas de arquivos distribuídos que funcionam de forma descentralizada. Tais práticas devem se popularizar nos próximos anos, como o IPFS abordado no trabalho de Benet (2014), StorJ e vários outros. Diversas pesquisas já estão sendo feitas nesse sentido a fim de buscar alternativas que possam se popularizar frente à centralização hoje proposta por poucas grandes empresas de tecnologia.

A partir dos itens vistos neste trabalho é possível concluir que o WebRTC é viável como tecnologia base para um aplicativo de trocas de arquivos. Dependendo do cenário encontrado, talvez até possa ser a melhor opção, ainda mais quando há a necessidade de usar apenas navegadores de *internet* e não se deseja instalar nenhum tipo de *software* extra do lado do cliente.

# REFERÊNCIAS

ABOUKHADIJEH, F. *WebTorrent*. 2021. Disponível em: <https://github.com-/webtorrent/webtorrent>. Citado na página 33.

ALBRIGHT, S. et al. *Simple Service Discovery Protocol/1.0*. [S.l.], 1999. Work in Progress. Disponível em: <https://datatracker.ietf.org/doc/html/draft-cai-ssdp-v1-03>. Citado na página 28.

BANERJEE, U. et al. eedtls: Energy-efficient datagram transport layer security for the internet of things. In: . [S.l.: s.n.], 2017. p. 1–6. Citado na página 18.

BENET, J. IPFS - content addressed, versioned, P2P file system. *CoRR*, abs/1407.3561, 2014. Disponível em: <http://arxiv.org/abs/1407.3561>. Citado 2 vezes nas páginas 29 e 69.

BERGKVIST, A. et al. W3C, 2011. Disponível em: <https://www.w3.org/TR/2011-/WD-webrtc-20111027/>. Citado na página 17.

CHESHIRE, S.; KROCHMAL, M. *Multicast DNS*. RFC Editor, 2013. RFC 6762. (Request for Comments, 6762). Disponível em: <https://rfc-editor.org/rfc/rfc6762.txt>. Citado na página 40.

CHESHIRE, S.; STEINBERG, D. H. *Zero configuration networking: the definitive guide*. [S.l.]: OReilly, 2006. Citado na página 40.

COHEN, B. *BitTorrent.org*. 2017. Disponível em: <http://www.bittorrent.org/beps-/bep_0003.html>. Citado na página 32.

DRNASIN, I.; GRGIC, M.; GLEDEC, G. Exploring WebRTC Potential for DICOM File Sharing. *Journal of Digital Imaging*, v. 33, n. 3, p. 697–707, jun. 2020. ISSN 0897-1889, 1618-727X. Disponível em: <http://link.springer.com/10.1007/s10278-019-00305-0>. Citado 3 vezes nas páginas 35, 36 e 37.

ERMOSHINA, K.; MUSIANI, F.; HALPIN, H. End-to-end encrypted messaging protocols: An overview. In: . [S.l.: s.n.], 2016. v. 9934, p. 244–254. ISBN 978-3-319-45981-3. Citado na página 13.

GOOGLE. *Visão geral do Wi-Fi Direct (ponto a ponto ou P2P)*. 2021. Disponível em: <https://developer.android.com/guide/topics/connectivity/wifip2p>. Citado na página 57.

GRIGORIK, I. *High-performance browser networking*. [S.l.]: OReilly, 2016. Citado 4 vezes nas páginas 19, 20, 21 e 23.

HEINRICH, A. et al. PrivateDrop: Practical Privacy-Preserving Authentication for Apple AirDrop. p. 18, 2021. Citado 5 vezes nas páginas 13, 27, 30, 31 e 55.

KHATAL, S. et al. FileShare: A Blockchain and IPFS Framework for Secure File Sharing and Data Provenance. In: PATNAIK, S.; YANG, X.-S.; SETHI, I. K. (Ed.). *Advances in Machine Learning and Computational Intelligence*. Singapore: Springer Singapore, 2021.

p. 825–833. ISBN 9789811552427 9789811552434. Series Title: Algorithms for Intelligent Systems. Disponível em: <http://link.springer.com/10.1007/978-981-15-5243-4_79>. Citado na página 29.

LORETO, S. *Real-time communication with WebRTC*. Sebastopol, CA: O'Reilly Media, 2014. ISBN 978-1449371876. Citado na página 14.

MATTHEWS, P.; ROSENBERG, J.; MAHY, R. *Traversal Using Relays around NAT (TURN): Relay Extensions to Session Traversal Utilities for NAT (STUN)*. RFC Editor, 2010. RFC 5766. (Request for Comments, 5766). Disponível em: <https://rfc-editor.org/rfc/rfc5766.txt>. Citado na página 45.

MOZILLA. *WebRTC API - APIs da Web: MDN*. 2021. Disponível em: <https://-/developer.mozilla.org/pt-BR/docs/Web/API/WebRTC_API>. Citado na página 14.

PATEL, P. et al. 2015. Disponível em: <https://tools.ietf.org/id/draft-jennings-behave-rtcweb-firewall-03.html>. Citado na página 51.

RESCORLA, E.; MODADUGU, N. *Datagram Transport Layer Security*. RFC Editor, 2006. RFC 4347. (Request for Comments, 4347). Disponível em: <https://rfc-editor.org-/rfc/rfc4347.txt>. Citado na página 19.

ROSENBERG, J. et al. *STUN - Simple Traversal of User Datagram Protocol (UDP) Through Network Address Translators (NATs)*. [S.l.], 2003. `http://www.rfc-editor.org/rfc/rfc3489.txt`. Disponível em: <http://www.rfc-editor.org/rfc/rfc3489.txt>. Citado na página 23.

SUMRAK, J. *How to Get Started With WebRTC: Intro to Browser APIs*. Twilio, 2021. Disponível em: <https://www.twilio.com/blog/get-started-webrtc>. Citado na página 15.

TANENBAUM, A. S.; WETHERALL, D. J. *Computer Networks*. 5th. ed. USA: Prentice Hall Press, 2010. ISBN 0132126958. Citado 2 vezes nas páginas 13 e 18.

VASS, J. *How Discord Handles Two and Half Million Concurrent Voice Users using WebRTC*. Discord Blog, 2018. Disponível em: <https://blog.discord.com/how-discord-handles-two-and-half-million-concurrent-voice-users-using-webrtc-ce01c3187429>. Citado na página 15.

WANG, F.; WU, Y.; HUANG, F. Rio: a personal storage system in multi-device and cloud. *The Journal of Supercomputing*, v. 76, n. 4, p. 2315–2338, abr. 2020. ISSN 0920-8542, 1573-0484. Disponível em: <http://link.springer.com/10.1007/s11227-018-2501-8>. Citado 2 vezes nas páginas 27 e 28.

WILLIGERS, E. 2021. Disponível em: <https://wicg.github.io/raw-sockets/>. Citado na página 68.

Özkan Canay; Halil Arslan. Comparison of Data Transfer Performance of Bit-Torrent Transmission Protocols. *Cumhuriyet Science Journal*, v. 40, n. 3, p. 762–762–767, set. 2019. ISSN 2587-2680. Disponível em: <https://doaj.org/article-/476b4321365e4d46a7ccf3baaeed6064>. Citado 3 vezes nas páginas 32, 33 e 34.