Universidade Federal Fluminense
Instituto de Computação
Departamento de Ciência da Computação

Arthur Pitzer Caetano

BBLISS: UM SISTEMA DE STREAMING AO VIVO ENTRE NAVEGADORES WEB

Niterói-RJ
2017

ARTHUR PITZER CAETANO

BBLISS: UM SISTEMA DE STREAMING AO VIVO ENTRE NAVEGADORES WEB

Trabalho submetido ao Curso de Bacharelado em Ciência da Computação da Universidade Federal Fluminense como requisito parcial para a obtenção do título de Bacharel em Ciência da Computação.

Orientadora: Profa. Débora Christina Muchaluat Saade

Niterói-RJ

2017

ARTHUR PITZER CAETANO

BBLISS: UM SISTEMA DE STREAMING AO VIVO ENTRE NAVEGADORES WEB

Trabalho submetido ao Curso de Bacharelado em Ciência da Computação da Universidade Federal Fluminense como requisito parcial para a obtenção do título de Bacharel em Ciência da Computação.

Aprovado por:

Profa. Débora Christina Muchaluat Saade, D.Sc. - Orientadora
UFF

Prof. Antonio Augusto de Aragão Rocha, D.Sc.
UFF

Prof. Diego Gimenez Passos, D.Sc.
UFF

Niterói-RJ
2017

Dedico este trabalho a minha família, que sempre acreditou no valor da educação, e a todos os meus professores, que me ensinaram a aprender.

Agradecimentos

Agradeço à Professora Débora por me orientar durante a graduação. Obrigado não apenas pelas lições acadêmicas, mas também pelos conselhos e exemplos que levarei para toda a vida.

Agradeço aos amigos do MídiaCom Joel, Glauco, Douglas e Fábio, pelas discussões nas reuniões semanais, onde suas críticas construtivas ajudaram a tornar esse trabalho possível.

Agradeço ao amigo Gustavo pelas discussões que ajudaram a transformar uma ideia neste trabalho.

Agradeço à minha irmã Luiza e às amigas Stéfani e Verônica por me ajudarem durante os testes.

Resumo

Os constantes avanços nas técnicas de compressão e o aumento na velocidade de acesso à Internet tornaram possível a popularização das aplicações de *streaming* de áudio e vídeo. Além de entreter, esse tipo de aplicação permite que usuários compartilhem eventos instantâneos, sendo úteis também como veículo jornalístico.

Porém, existem diversos desafios que dificultam a implementação de sistemas de *streaming* de multimídia, em especial de streaming ao vivo. Entre eles, podemos destacar o problema de escalabilidade, que pode ser resolvido com o uso de CDNs (*Content Delivery Networks*). Apesar de eficientes, as CDNs têm custo elevado e restringem a operação de sistemas de streaming a grandes empresas.

Este trabalho apresenta o BBLiss (*Browser-to-Browser Live Interactive Streaming System*), um sistema de streaming de vídeo ao vivo entre navegadores web. Sua principal característica é a utilização de uma arquitetura *peer-to-peer* e de uma estrutura de *peers* em árvore para oferecer escalabilidade a custos mais baixos. Sua implementação é baseada no WebRTC, um software aberto para prover comunicação em tempo real entre navegadores web. Resultados de testes mostram que, apesar de certas limitações, foi possível implementar um sistema que permite ao usuário realizar *streaming* ao vivo a partir de seu navegador web de forma barata e escalável.

Palavras-chave: *BBLiss, Streaming ao vivo, Peer-to-Peer, WebRTC*

Abstract

Constant developments in compression techniques and the increasing Internet access speed have made streaming systems accessible to a larger audience. These systems can be used beyond mere entertainment. They are also valuable as journalistic tools, since they allow users to share instantaneous events.

But there are still many challenges to be overcome in order to implement such systems. Scalability is still one of the biggest concerns. Usually, large scale streaming systems use CDNs (Content Delivery Networks) to provide scalability. This kind of solution is very efficient, however, it has high costs and is affordable only by large companies.

This work proposes BBLiss, a Browser-to-Browser Live Interactive Streaming System. The main feature of this system is to deliver multimedia streams using a peer-to-peer tree-based architecture, capable of providing scalability at low costs. Its implementation is based on WebRTC, an open software that provides real-time communication between web browsers. Test results show that, even facing some limitations, BBLiss allows the user to live stream from web browsers in a cheap and scalable way.

Keywords: *BBLiss, Streaming ao vivo, Peer-to-Peer, WebRTC*

Sumário

Lista de Figuras

Capítulo 1

Introdução

Em 2013, o aumento das tarifas de transporte público foi o estopim para as chamadas Manifestações dos 20 centavos. Estima-se que esses foram os maiores protestos desde 1992, quando o Brasil se mobilizou pelo *impeachment* do então presidente Collor. "Não é só pelos 20 centavos", esse era um dos lemas usados pelos manifestantes, ficou claro que a insatisfação ia bem além de 20 centavos quando outros itens foram incluídos à pauta de reivindicações como: reforma política, melhorias na segurança pública, educação e saúde. Os protestos se estenderam das capitais para todo o Brasil e reuniram diferentes setores de toda a sociedade.

Um dos pontos mais interessantes sobre esse e outros movimentos populares contemporâneos, como a Primavera Árabe, foi o papel desempenhado pela tecnologia. A maior parte da mobilização e divulgação das manifestações foi feita através das redes sociais. A grande mídia geralmente estava horas atrasada em relação aos fatos que eram reportados em tempo real nas *timelines* e *feeds* dos manifestantes.

Além de demonstrar como a Internet pode ser usada pela sociedade na defesa de seus interesses, esses movimentos são marcos históricos de uma mudança mais sutil: mais que ter acesso a tecnologias web, passa-se a compreender seu poder. Graças às redes sociais e a serviços de *streaming*, cada manifestante pode compartilhar sua versão dos fatos por conta própria, ao invés de depender de um agente intermediário para veiculá-las ao público. Não são mais necessárias grandes emissoras ou jornais para captar e distribuir notícias. Agora os indivíduos estão interligados, permitindo que a informação flua diretamente entre eles. Desta forma, mais pontos de vista são considerados e torna-se mais difícil deturpar informação ou manipular a opinião pública. Dar voz a cada vez mais

pessoas é importante para fortalecer a democracia e para caminhar em direção a uma sociedade mais igualitária.

Por décadas, as mídias tradicionais como editoras e emissoras de rádio e TV se sustentaram sobre dois modelos de negócio. O primeiro é aquele onde o espectador paga para ter acesso à informação, seja comprando um livro ou pagando um canal por assinatura. O segundo é usado no rádio e na TV aberta, onde a emissora vende espaços na sua grade de programação para anunciantes. Nenhum desses modelos é compatível com a forma de consumir conteúdo que a web introduziu. Hoje pode-se ter acesso a praticamente qualquer tipo de conteúdo, a qualquer momento, e na maioria das vezes, gratuitamente.

A competição gerada pelas mídias digitais vem causando a chamada "Crise do Jornalismo" [1]. A queda na venda de impressos e no faturamento vindo de anúncios em rádio e TV abertos vem gerando desemprego e um crise de credibilidade na mídia tradicional. De fato, a crise não é do jornalismo como profissão, mas sim dos modelos de negócio vigentes. Hoje não há mais necessidade de que a produção e distribuição de conteúdo sejam mantidas por uma mesma entidade. Pelo contrário, as novas tecnologias favorecem modelos descentralizados onde produtores independentes distribuem seu conteúdo ao público diretamente.

A vantagem da Internet como meio de distribuição de informação sobre mídias impressas e radiodifusão reside principalmente no custo, tempo de disponibilidade e velocidade de propagação. Caminha-se para uma dissociação dos processos de produção e distribuição de conteúdo. A web permitiu que muitos jornalistas, diretores e músicos independentes tivessem acesso ao público de todo o mundo. Quem ganha é a sociedade que tem acesso a uma variedade maior de informação e entretenimento, e também aqueles que podem publicar matérias e lançar filmes sem intermediários.

Nesse cenário, o *streaming* ao vivo ganha força ao popularizar algo que antes era caro e de privilégio das emissoras de TV. Transmitir conteúdo audiovisual ao vivo de qualquer lugar é um passo importante na direção desse modelo de mídia mais democrática, ao permitir que eventos instantâneos possam ser transmitidos em tempo real por qualquer pessoa. No entanto, esse benefício vem acompanhado de uma série de desafios técnicos decorrentes da arquitetura da Internet.

1.1 Desafios

Em uma aplicação de *streaming* ao vivo, muitos clientes estão interessados no conteúdo que está disponível apenas em um *host*, a fonte que gera o fluxo. Além disso, não basta que o fluxo chegue a esses clientes, mas também é necessário que ele chegue a tempo de ser reproduzido. Esses são os dois maiores desafios na implementação de um sistema de *streaming* ao vivo. Desafios como prover escalabilidade e como oferecer níveis de Qualidade de Serviço (QoS) mínimos estão presentes nessas aplicações e continuam sendo investigados. Essas duas questões devem ser respondidas considerando as limitações de uma rede de melhor esforço onde a comunicação entre *hosts* é feita em *unicast*, como ocorre na Internet.

Idealmente, o problema de escalabilidade poderia ser resolvido usando *multicast*. Esse modelo de comunicação permite que um grupo de usuários receba informação de uma fonte sem sobrecarregá-la e sem congestionar a rede com cópias desnecessárias dessa informação. As principais propostas para implementar multicast na Internet são o IP multicast [2] e multicast na camada de aplicação [3].

O IP multicast é oferecido como um recurso na camada de rede e exige que todos os roteadores envolvidos implementem esse protocolo, por essa razão é viável apenas em domínios restritos e não está disponível em toda Internet.

Resta então utilizar multicast na camada de aplicação. Para isso é construída uma rede de sobreposição, onde os nós são ligados logicamente. O problema de escalabilidade pode ser atacado com essa estratégia, pois é possível limitar o número de requisições feitas a origem e utilizar outros nós que já receberam o conteúdo para atender parte das requisições. Por outro lado, não é possível garantir que cópias desnecessárias não trafeguem na rede. Um mesmo roteador pode pertencer ao caminho de dois nós até a fonte, ou seja, duas cópias do mesmo conteúdo irão passar pelo mesmo roteador.

Outra solução bastante eficiente e amplamente empregada em aplicações de streaming comerciais são as CDNs (*Content Delivery Networks*) [4]. Elas consistem em uma rede de servidores espalhados por *data centers* em todo mundo. CDNs permitem um balanceamento de carga entre os servidores que a compõem, além de outras otimizações como atender clientes com servidores geograficamente mais próximos e utilização eficiente de cache. Porém, tantos benefícios vêm a um preço bastante alto. Apenas grandes empresas podem arcar com tais custos, deixando fora do mercado empresas menores e

iniciativas independentes.

Outro obstáculo é a garantia de qualidade de serviço. Na radiodifusão de rádio e TV, o meio é usado unicamente pela emissora, que pode transmitir dados a taxa máxima a todo momento. Na Internet, o cenário é distinto. Ela utiliza comutação de pacotes e funciona sobre o modelo de melhor esforço, onde todos os pacotes compartilham o mesmo meio de transmissão e os roteadores fazem o melhor possível para entregá-los ao destino. Por essa razão, não é possível garantir uma determinada taxa de transmissão, já que existe um atraso não determinístico na entrega de pacotes.

Para otimizar a transmissão e tentar oferecer tempos de entrega menores, recorre-se a técnicas de compressão e QoS que minimizem a latência na comunicação na rede.

1.2 Objetivos

Técnicas de compressão e arquiteturas P2P *(Peer-to-Peer)* se destacam como formas de contornar os desafios existentes em aplicações de *streaming* ao vivo. Explorar redundâncias espaciais e temporais de vídeos permite representá-los de forma mais compacta e portanto, mais apropriada ao envio pela rede. Em um sistema P2P, os recursos e serviços oferecidos são replicados proporcionalmente à quantidade de usuários, tornando o sistema escalável a custos reduzidos. Portanto, aliar essas técnicas favorece a implementação de sistemas de *streaming* ao vivo. O WebRTC [5, 6] é uma plataforma *open source* oferecida em diversos navegadores web que é capaz de capturar, codificar e distribuir dados multimídia diretamente de navegador para navegador. Graças a essas características, o WebRTC se mostra uma opção bastante atraente para desenvolvedores de sistemas distribuídos e de tempo real. O objetivo principal deste trabalho de conclusão de curso é propor um sistema para *streaming* ao vivo P2P que funcione na web. Este sistema é chamado BBLiss (*Browser-to-Browser Live interactive streaming system*) e é construído sobre o WebRTC.

1.3 Estrutura do Texto

O Capítulo 2 discute alguns trabalhos relacionados. Cada um deles é descrito brevemente e comparado sob algum aspecto com o BBLiss proposto neste trabalho.

O Capítulo 3 apresenta o WebRTC, que foi usado na implementação do BBLiss.

São discutidas suas funcionalidades e as tecnologias que o compõem. Apresentam-se com especial atenção os mecanismos usados para estabelecimento de conexões *peer-to-peer*.

O Capítulo 4 é dedicado a descrever em detalhes a proposta deste trabalho, apresentando a arquitetura e o funcionamento do BBLiss.

O Capítulo 5 apresenta a metodologia de testes e os principais resultados.

O Capítulo 6 conclui o trabalho e traz sugestões de trabalhos futuros.

Capítulo 2

Trabalhos Relacionados

Este capítulo apresenta os trabalhos relacionados BemTV [7], *P2P Live Video Streaming in WebRTC* [8], [9] e MobileCast [10]. Ideias presentes nesses trabalhos inspiraram a proposta do sistema BBLiss.

2.1 BemTV

O trabalho BemTV [7] traz uma proposta de modelo híbrido de streaming CDN (*Content Delivery Network*) e P2P. O objetivo do BemTV é diminuir os custos com CDN usando recursos de *peers* vizinhos que estejam recebendo o mesmo vídeo. Cada *peer* requisita partes do vídeo a outros *peers* geograficamente próximos, caso nenhum dos seus vizinhos possua a parte de mídia desejada, o *peer* a requisita à CDN. Nesse sistema, apenas o canal de dados genérico do WebRTC é utilizado, a codificação e transmissão entre *peers* e CDNs é feita usando HLS [11] ou HTTP DASH [12]. Essa estratégia se mostrou eficiente na redução de requisições à CDN, porém acarreta uma explosão de troca de mensagens em grupos de *peers* muito grandes.

Este trabalho foi uma inspiração importante para o BBLiss por demonstrar como diferentes usos de redes *peer-to-peer* podem ajudar a reduzir custos com infraestruturas mais caras, como CDNs no caso.

No BemTV, o WebRTC é usado apenas para obter *chunks* de uma *playlist* HLS ou DASH, que são exibidos em um *player* desenvolvido por terceiros. No BBLiss, o WebRTC é usado na captura, codificação e transmissão do fluxo, que é exibido através do elemento de vídeo do HTML5 [13]. Isso resulta em uma aplicação mais compacta e que utiliza

melhor os recursos oferecidos nos navegadores Web.

2.2 *P2P Live Video Streaming in WebRTC*

O artigo [8] apresenta um sistema de *streaming* ao vivo entre navegadores que também utiliza o WebRTC. Esse sistema utiliza uma estrutura em malha que obedece à seguinte regra: cada par, exceto a fonte, recebe dados de no máximo dois vizinhos. Por outro lado, cada *peer* pode enviar dados para n outros *peers*. Foi demonstrado que essa regra garante que depois de um tempo suficientemente longo, *peers* mais próximos da origem e que possuem conexões mais estáveis enviam dados para mais vizinhos.

Por dar bastante enfoque ao processo de formação da rede de sobreposição, este trabalho serviu de inspiração para as regras usadas na formação e manutenção da árvore de distribuição utilizada no BBLiss.

Este trabalho implementou e testou um sistema de *streaming* de vídeo armazenado usando topologia em malha. Na época de sua publicação (Janeiro de 2014), conclui que o WebRTC ainda não estava maduro o suficiente para permitir a implementação de sistemas de *streaming* de grande porte. Apesar disso, o BBLiss demonstrou que é possível implementar um sistema de *streaming* ao vivo usando uma topologia de árvore com a versão atual do WebRTC. Espera-se que testes futuros demonstrem que o BBLiss é capaz de suportar uma carga considerável.

2.3 *Performance of DASH and WebRTC Video Services for Mobile Users*

O objetivo de [9] foi realizar uma análise quantitativa do desempenho de aplicações de *streaming* adaptativo em clientes móveis. Para isso, foram comparadas duas aplicações de videoconferência, uma baseada em HTTP DASH e outra em WebRTC. Essas aplicações foram testadas em um cenário rural e outro urbano, com largura de banda variável. O estudo concluiu que mesmo usando técnicas de *streaming* adaptativo, as aplicações não foram capazes de manter os requisitos de qualidade mínimos, às vezes com taxas de reprodução do vídeo abaixo de 12 quadros por segundo. Isso demonstra que lidar com dispositivos móveis ainda é um desafio para aplicações de *streaming*.

Apesar do BBLiss não considerar mobilidade de *peers*, este trabalho levantou questões como o uso de redes móveis em sistemas de *streaming*. Foi considerando esse tipo de caso de uso que surgiu a ideia de limitar o número de *peers* vizinhos em função da disponibilidade de recursos.

2.4 MobileCast

O trabalho [10] apresenta um protocolo de *streaming* ao vivo *peer-to-peer* chamado MobileCast. Esse protocolo busca combinar as topologias em árvore e em malha para amenizar suas desvantagens. O trabalho também traz o conceito de *peers* de *streaming* e *peers* de armazenamento. Os *peers* de *streaming* são geralmente dispositivos móveis que possuem poucos recursos e atuam apenas como reprodutores e retransmissores de um fluxo. Já os *peers* de armazenamento, possuem mais recursos e além de reproduzir e retransmitir um fluxo, eles também o armazenam, potencializando sua capacidade de contribuir para a rede.

Esse trabalho foi importante para o BBLiss por destacar as limitações e vantagens das topologias em malha e em árvore. A topologia em árvore foi escolhida no BBLiss, entre outras razões, por permitir um controle temporal mais natural do que a topologia em malha. *Peers* que ocupam o mesmo patamar de altura na árvore possuem valores próximos de atraso em relação à origem. Isso permite que *peers* em patamares superiores distribuam o fluxo para *peers* de patamares inferiores. Essa organização surge naturalmente da estrutura em árvore, fazendo com que cada patamar funcione como um *buffer* para os patamares inferiores. Mais detalhes sobre o funcionamento da árvore de distribuição estão no Capítulo 4.

Capítulo 3

WebRTC

O WebRTC [5,6] é um projeto *open source*, iniciado pela Google em 2011 que visa prover comunicação em tempo real entre browsers. Composto por diferentes protocolos e APIs, hoje o WebRTC passa pelo processo de padronização no W3C *(World Wide Web Consortium)* e no IETF *(Internet Engineering Task Force)*. Versões preliminares funcionais já estão disponíveis em navegadores como Chrome, Firefox e Opera. O WebRTC é oferecido ao desenvolvedor através de um API JavaScript. Por ainda não estar padronizado, o desenvolvedor que optar pelo uso do WebRTC ainda encontrará problemas de compatibilidade da API entre navegadores web. No entanto, esses problemas tendem a ser solucionados enquanto a padronização progride. Apesar de ainda incompleto, já é possível utilizar o WebRTC em aplicações que necessitem de comunicação P2P e *streaming*, sendo especialmente vantajoso por dispensar o uso de *plugins* e ser compatível com múltiplas plataformas. Este capítulo apresenta algumas características do WebRTC.

3.1 Codificação

Apesar de depender da implementação feita em cada navegador, o WebRTC prevê o uso de diversos codecs de áudio e vídeo. Entre os codecs de áudio podemos destacar o OPUS [14], que é de código aberto, livre de *royalties*, e que possui boas taxas de compressão e baixa latência.

Com relação aos codecs de vídeo, houve por muito tempo uma discussão intensa na comunidade sobre a adoção do VP8 [15] ou do H.264 [16] como codec de vídeo obrigatório. Comparações entre os dois codecs demonstram que o H.264 tem qualidade perceptual e

velocidade de codificação mais alta que o VP8, tornando-o mais adequado para aplicações de *streaming* [17]. Apesar dessa vantagem técnica, a causa da polêmica foi comercial. O H.264 é padrão ITU [16] e pode ser utilizado através de implementações gratuitas, como o OpenH.264 desenvolvido pela Cisco [18]. Apesar disso, seu termo de licença [19] diz que apenas aplicações gratuitas para o usuário final são livres de *royalties*, os demais usos do H.264 devem pagar *royalties* para a MPEG LA, instituição responsável por coletá-los e distribuí-los entre os proprietários das patentes. Essa discussão agora parece caminhar para uma resolução. O VP8 está disponível em todos os navegadores com suporte ao WebRTC. O H.264, que antes era suportado apenas pelo Firefox, passou a ser disponibilizado também nas últimas versões do Chrome [20].

3.2 Transmissão

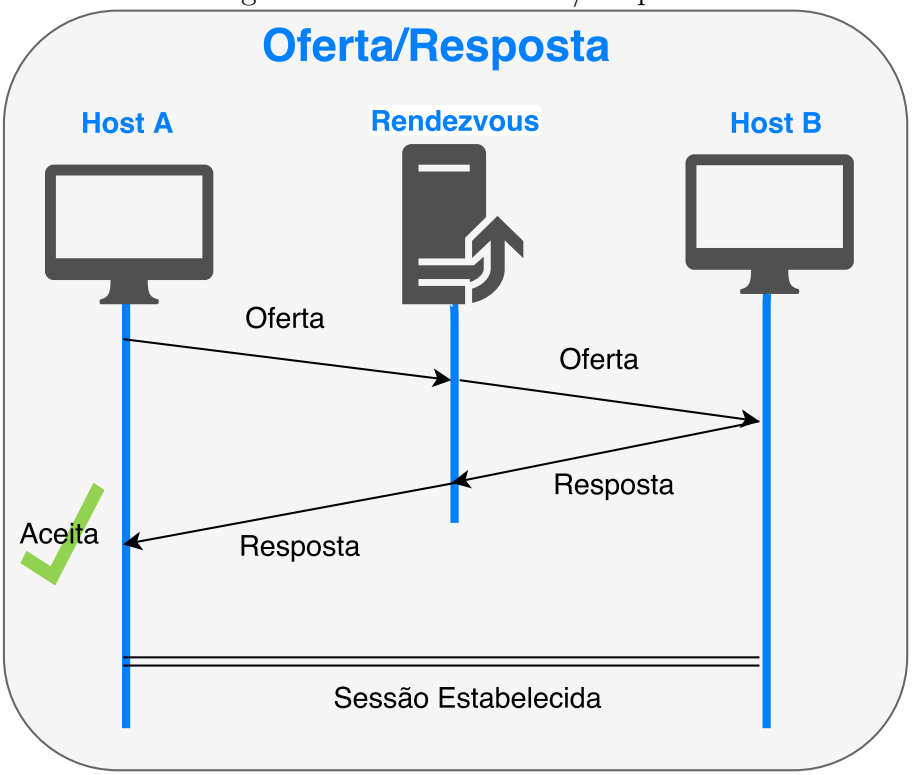
O WebRTC permite transferir tanto fluxos multimídia quanto dados genéricos. A transmissão de fluxos é feita usando o protocolo SRTP (*Secure Real-time Transport Protocol*) [21], que adiciona características de segurança ao RTP (*Real-time Transport Protocol*) [22] tradicional. Já o canal de dados genéricos utiliza o protocolo SCTP (*Stream Control Transmission Protocol*) [23]. O SCTP é um protocolo da camada de transporte orientado à mensagem assim como o UDP, porém inclui entrega confiável e controle de congestionamento, assim como o TCP.

A principal preocupação do SRTP é mitigar ataques de *replay*. Neles, o atacante se coloca entre o transmissor e o receptor e substitui novos pacotes por pacotes enviados anteriormente, causando a reprodução de trechos já exibidos. Para evitar isso, o SRTP inclui um *checksum* em cada pacote para garantir a integridade dos dados e também mantém índices de pacotes já recebidos no destino. Dessa forma, o atacante não é capaz de reenviar pacotes antigos e nem alterar o índice de pacotes sem ser detectado.

Mas antes de realizar qualquer comunicação *peer-to-peer*, é necessário que as partes envolvidas concordem em alguns pontos. Durante o processo de negociação, os *peers* chegam a um consenso sobre quais serão os codecs, portas e endereços utilizados. Estabelecer uma conexão *peer-to-peer* não é tarefa trivial considerando a arquitetura da Internet hoje. A ampla adoção de NATs (*Network Address Translator*), *firewalls* e políticas de segurança hostis a protocolos como o UDP, dificultam que dois *peers* estabeleçam uma co-

nexão direta. Para lidar com esses problemas o WebRTC utiliza um framework chamado ICE (*Interactive Connectivity Establishment*) [24], que combina técnicas para transpassar NATs e *firewalls* afim de conectar *peers*.

Figura 3.1: Modelo Oferta/Resposta



O objetivo do ICE é permitir o estabelecimento de sessões multimídia usando o mesmo modelo de oferta/resposta empregado no protocolo SIP (*Session Initiation Protocol*) [25] para trocar mensagens SDP (*Session Description Protocol*) [26] entre dois hosts. Como é possível acompanhar na figura 3.1, o modelo é assim chamado porque o *host* que inicia a sessão envia a outro *host* uma mensagem SDP chamada de oferta. Ao receber uma oferta, o outro *host* envia uma mensagem SDP chamada de resposta. Caso o *host* que iniciou a negociação aceite a resposta, uma conexão é estabelecida. Em um primeiro momento os *hosts* não são capazes de trocar mensagens diretamente. Por essa razão, um ponto de *rendezvous* é usado para intermediar a comunicação entre eles.

As mensagens SDP descrevem quais codecs estão disponíveis em ordem de preferência e uma lista de candidatos ICE; que são tuplas de IP, porta e protocolo de transporte. Existem três tipos de candidatos segundo a técnica de atravessamento de NAT e *firewall* que utilizam. Antes de apresentá-los, os próximos parágrafos descrevem brevemente como funciona o NAT.

Os NATs são utilizados para promover maior controle gerencial de redes locais e permitir que diversos hosts se comuniquem com uma rede externa usando um único endereço IP. Esse mecanismo é necessário por não haver endereços IPv4 suficientes para que todos os dispositivos existentes se conectem à Internet. NATs são bastante comuns nas redes de acesso domésticas.

A tradução de endereços é feita da seguinte forma: quando um host envia um pacote para fora da rede local, o NAT substitui o endereço de transporte (IP e porta) de origem por um endereço de transporte válido na rede externa. Essa troca é armazenada em uma tabela, que informa o endereço de origem, a tradução e endereço de destino. Essa tradução é feita para que o receptor do pacote possa responder o remetente. Quando um pacote enviado de uma rede externa chega ao NAT, ele avalia se o endereço de origem está presente na tabela de tradução como um endereço de destino usado anteriormente, em caso positivo, o NAT faz a tradução do endereço de destino para um endereço válido na rede interna e encaminha o pacote. Apesar de existirem diferentes tipos de NATs, esse é o processo básico adotado. A consequência negativa disso para as aplicações *peer-to-peer* é que não é possível iniciar uma conexão com um host que utilize NAT a partir de um host de uma rede externa.

Agora sabendo da influência dos NATs no processo de conexão de *peers*, é importante entender quais são os tipos de candidatos ICE que são trocados nas mensagens SDP.

O primeiro tipo é o candidato host. O IP presente nesse tipo de candidato é o endereço de alguma interface física ou lógica do host, que pode ser válido apenas na sua rede local. Esse tipo de candidato pode ser usado para estabelecer conexões quando ambos os hosts estão na mesma LAN ou caso o host tenha um IP público.

O segundo tipo é o candidato reflexivo. O endereço de transporte desses candidatos é um endereço necessariamente válido na Internet. Caso o host esteja atrás de um NAT, o endereço de transporte desse candidato será o resultado de uma tradução. É possível obter esses endereços usando serviços de STUN (*Simple Traversal of User Datagram Protocol Through Network Address Translators*) [27], que são capazes de descobrir um IP válido e criar uma entrada na tabela de tradução. Dessa forma, o host pode informar seu IP público para que outro host estabeleça uma conexão. 86% das conexões WebRTC são feitas através de candidatos reflexivos segundo [28].

Por último temos candidatos de retransmissão. Esses candidatos são obtidos por meio do protocolo TURN (*Traversal Using Relays around NAT*) [29] que utiliza uma entidade intermediária para encaminhar todo fluxo entre dois hosts. Esse tipo de candidato permite que conexões sejam estabelecidas mesmo em cenários onde os firewalls são mais hostis a protocolos como o UDP ou haja NATs simétricos. Caso não seja possível estabelecer uma conexão usando nenhum dos candidatos anteriores, os candidatos de retransmissão sempre irão funcionar. Porém o TURN é geralmente oferecido como um serviço pago, dado ao grande consumo de banda.

O ICE tenta estabelecer conexões priorizando candidatos host, em seguida reflexivos e por último candidatos de retransmissão. Essa ordem é estabelecida porque espera-se que candidatos host deem origem a conexões com latências e custos menores.

Capítulo 4

Proposta

Este trabalho propõe o BBLiss (*Browser-to-Browser Live Interactive Streaming System*), que tem como funcionalidade principal permitir *streaming* ao vivo de maneira distribuída e barata em múltiplas plataformas.

Por identificar o WebRTC como ferramenta promissora, ele foi utilizado na implementação deste trabalho. Através do BBLiss, pretende-se identificar as principais decisões de design envolvidas no desenvolvimento desse tipo de sistema. Ele também poderá ser utilizado como uma plataforma funcional para testar e explorar novos algoritmos. O BBLiss herda características do WebRTC como ser compatível com múltiplas plataformas e dispensar a utilização de plugins.

Como dito anteriormente, o BBLiss é focado em streaming ao vivo (*live streaming*). Nesse cenário, um *peers* captura e transmite o conteúdo para todos os espectadores. Portanto, muito do esforço de implementação é voltado para construir uma árvore multicast na camada de aplicação, de forma a não sobrecarregar a fonte do conteúdo e entregar o fluxo aos espectadores em tempo hábil. Para nortear as decisões de design do BBLiss, buscamos sempre priorizar a diminuição do atraso de reprodução em relação à origem e continuidade de reprodução no destino.

4.1 Árvore de Distribuição

A premissa central deste sistema é que, explorando a replicação de conteúdo entre *peers*, é possível prover escalabilidade de tamanho ao sistema. Quando muitos usuários realizam streaming de um mesmo conteúdo a partir de um mesmo ponto de distribuição,

surgem dois problemas: a sobrecarga na fonte e a duplicação de informação trafegada na rede. Com a utilização de multicast na camada de aplicação, é possível reduzir drasticamente o problema de sobrecarga na origem, no entanto, não há garantia de que as cópias desnecessárias na rede serão eliminadas [30]. Essa limitação é consequência do fato de que a topologia da rede de sobreposição não é necessariamente igual à topologia física. Portanto, um mesmo roteador pode pertencer a mais de um caminho na rede de sobreposição e receber pacotes duplicados.

Mesmo sendo ineficiente na redução de cópias trafegadas na rede, aplicações de streaming ainda podem se beneficiar do multicast na camada de aplicação para balanceamento de carga. Essa distribuição de carga pode ser feita organizando os *peers* em uma árvore. Cada fluxo é distribuído em uma árvore própria, onde o *peer* que gera o conteúdo ocupa a raiz. Cada nó da árvore suporta um determinado número de filhos, ao quais ele repassa uma cópia do fluxo recebido de seu pai. Dessa forma, o conteúdo é replicado entre os *peers* e a carga é distribuída entre nós pai. Níveis superiores da árvore funcionam como *buffers* para os níveis inferiores.

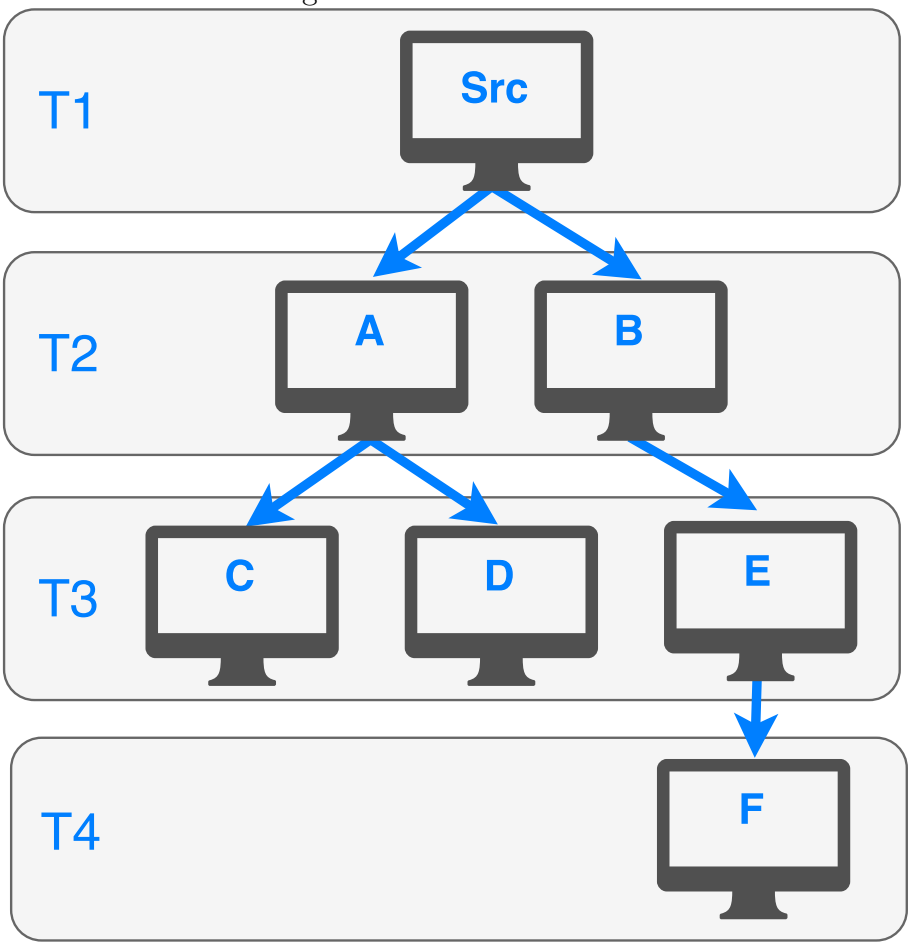
É interessante notar a relação que existe entre carga em um *peer* e atraso de reprodução em relação à origem. Um *peer* que é filho da origem tende a receber o conteúdo primeiro que um *peer* neto da origem, ou seja, o atraso de reprodução é proporcional à distância da origem. É natural que em uma aplicação de streaming ao vivo, o objetivo seja minimizar o atraso de reprodução em relação à origem. Para isso devemos fazer com que cada *peer* suporte um número grande de filhos, para que a árvore tenha uma altura pequena e consequentemente um atraso de reprodução menor em todos os *peers*.

Porém, se tentarmos maximizar esse critério, a árvore se degeneraria em uma topologia onde todos os *peers* estão conectados diretamente à origem, criando o problema de escalabilidade que tentávamos resolver em primeiro lugar. Por essa razão, devemos limitar o número de filhos de cada *peer*, de modo que exista um compromisso entre distribuição de carga e atraso de reprodução.

Na Figura 4.1 podemos observar que o *peer* A é capaz de suportar dois filhos, C e D, mantendo-os na zona de atraso T_2. Já o *peer* B só suporta um filho, obrigando o *peer* F a se conectar ao *peer* E, o que o deixa na zona de atraso T_3. Onde T_i representa o atraso em relação à origem de modo que $T_i < T_{i+1}$.

Apesar de utilizar o conceito de árvore, o BBLiss não mantém em momento algum

Figura 4.1: Zonas de atraso



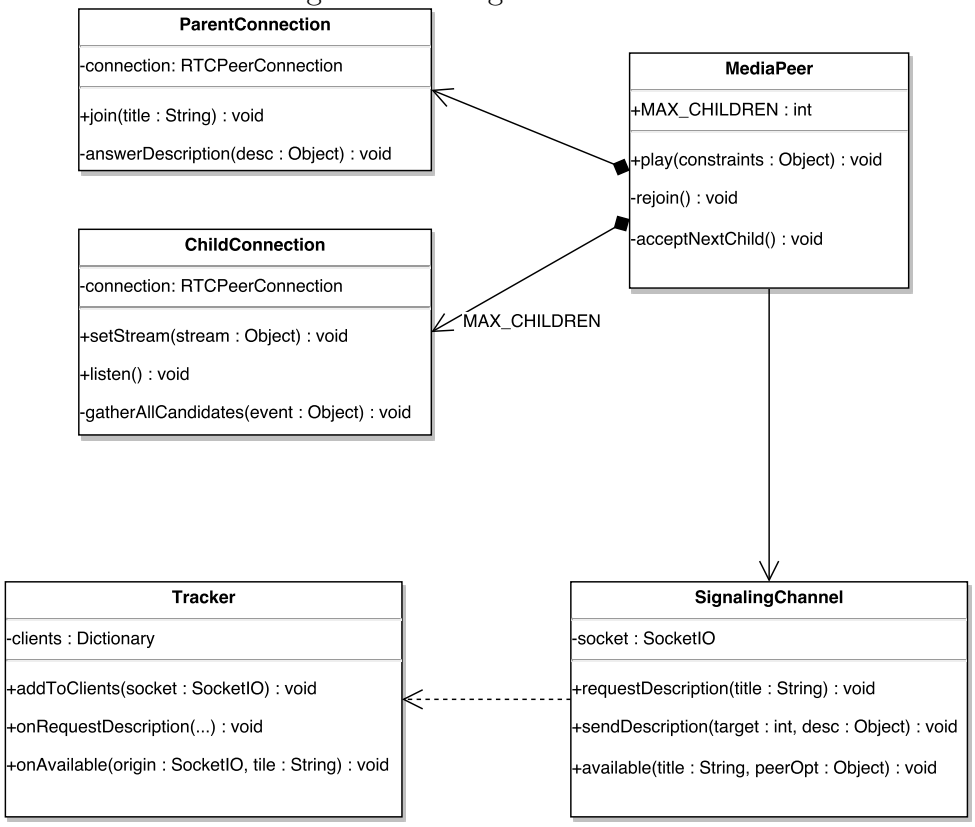
uma representação dessa estrutura de distribuição. O método de associação de *peers* dispensa a existência de tal representação e utiliza apenas uma fila com *peers* disponíveis. Como vantagens desta estratégia, temos uma aplicação mais simples, um protocolo com menos *overhead* de controle, e também transparência de topologia para os *peers*, que só se comunicam com seus pais e filhos. Uma desvantagem é que qualquer informação de topologia da árvore deve ser extraída da árvore de fluxos real.

4.2 A Estrutura

Vejamos agora as entidades que compõem o BBLiss e como elas se relacionam para formar a árvore de distribuição. A Figura 4.2 mostra uma versão concisa do diagrama de classes UML representando as entidades do BBLiss.

O *Tracker* é uma entidade que é executada em um servidor conhecido pela aplicação. Sua responsabilidade principal é permitir que *peers* se conectem para assistir a

Figura 4.2: Diagrama de Classe



um fluxo. Para isso, o Tracker mantém uma lista de fluxos ativos e de *peers* conectados. Como dito anteriormente, a árvore de distribuição não é mantida em memória. Tudo que o Tracker precisa para conectar *peers* em uma árvore é uma lista de *peers* capazes de receber novos filhos. O processo de associação de *peers* será abordado posteriormente.

Em cada *peer*, existe um objeto do tipo *MediaPeer*. Esse objeto funciona como um mediador, controlando o ciclo de vida de um *peer* e integrando diferentes objetos para gerar o comportamento desejado.

Cada *MediaPeer* possui um objeto *ParentConnection*. Esse é o objeto que de fato realiza o envio do stream. Os responsáveis por lidar com o recebimento desse mesmo fluxo são os objetos *ChildConnection*, que se conectam ao objeto *ParentConnection* de outro *peer*.

A comunicação entre um *MediaPeer* e o Tracker é feita por meio de um objeto *SignalingChannel*. Ele encapsula a negociação de sessão descrita na Seção 3.2 e se comunica usando o protocolo descrito na Subseção 4.2.1

Após o processo de negociação, todo fluxo de mídia é trocado diretamente entre *peers*, dispensando o *Tracker*. Caso o *Tracker* fique indisponível, nenhum dos fluxos ativos

é prejudicado; porém nenhum *peer* conseguirá se conectar ou reconectar.

4.2.1 Protocolo de Sinalização

No BBLiss, todo fluxo multimídia é trocado diretamente entre *peers*. Porém, uma fase de negociação utilizando uma entidade central chamada *Tracker* é necessária para as conexões *peer-to-peer* sejam estabelecidas. A seguir é apresentado o protocolo de sinalização utilizado pelo BBLiss para negociar as conexões entre *peers*. Os parâmetros de cada mensagem são nomeados em itálico.

- AVAILABLE *TITLE*: Enviada por um *peer* ao *Tracker* para que seja incluído na lista de *peers* disponíveis para a distribuição do fluxo com título *TITLE*.
- REQUEST_DESCRIPTION *TITLE*: Enviada por um *peer* para o *Tracker*, solicitando que seja incluído na árvore de distribuição do fluxo de título *TITLE*.
- SEND_DESCRIPTION *PEER_A SDP*: Um *peer* envia essa mensagem ao *Tracker* para encaminhar uma descrição *SDP* para *PEER_A*, a fim de estabelecer uma conexão.

Na Seção 4.3 é apresentada a utilização dessas mensagens no contexto de funcionamento do BBLiss.

4.3 O Funcionamento

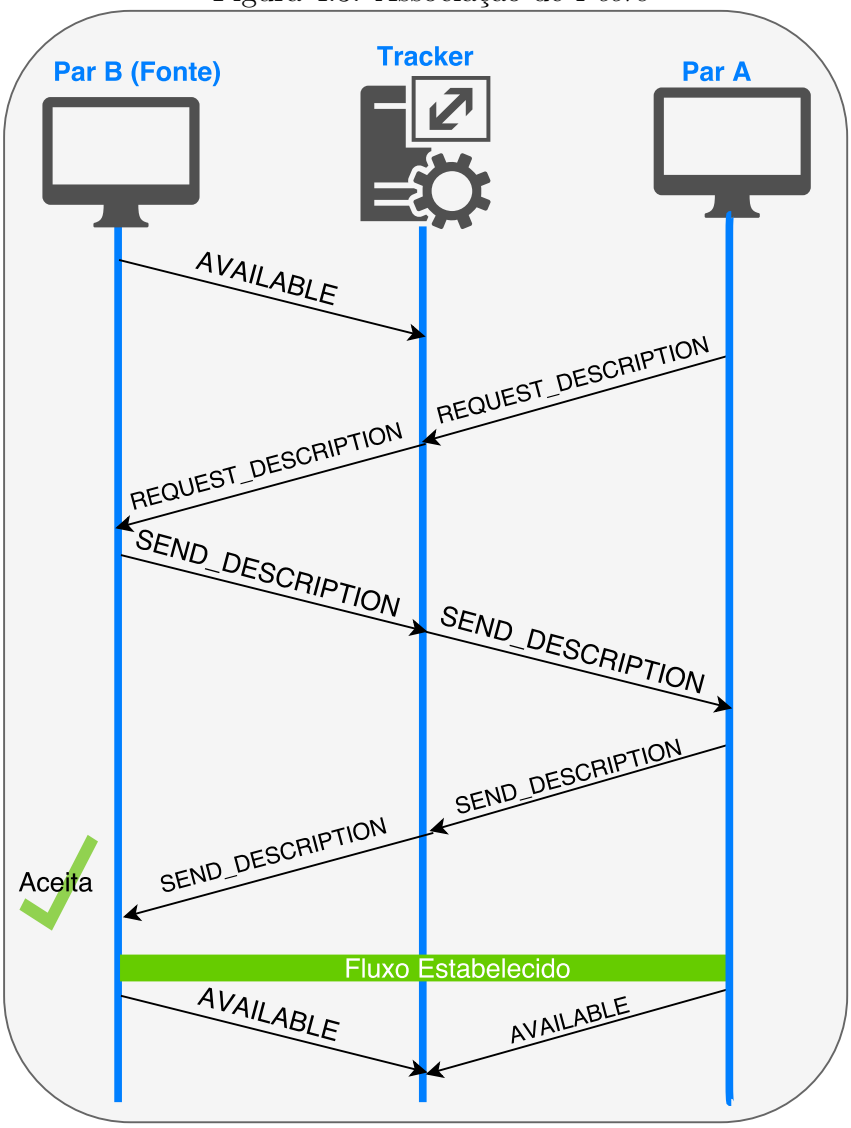
4.3.1 Inicialização

O streaming se inicia quando um usuário acessa a aplicação web e informa o título do fluxo que irá transmitir. A aplicação também irá pedir permissão para acessar a câmera e o microfone para iniciar a captura. Nesse momento, uma conexão Web Socket [31] é iniciada com o Tracker e o *peer* envia uma mensagem AVAILABLE. Ao receber essa mensagem o Tracker registra o fluxo em uma lista de fluxos ativos e inclui o *peer* raiz na lista de *peers* disponíveis para conexão.

4.3.2 Associação de *Peers*

Na Figura 4.3, podemos acompanhar o processo de associação entre dois *peers*, após um deles ter iniciado o fluxo como descrito na Subseção 4.3.1. Um novo *peer* (*peer* A) pode acessar a lista de fluxos ativos fornecida pelo tracker para escolher a qual deles deseja assistir. Após optar por um deles, o *peer* A envia uma mensagem REQUEST_DESCRIPTION seguida do título do fluxo escolhido (fluxo F). Logo após, será iniciado o processo de Oferta/Resposta descrito na Seção 3.2.

Figura 4.3: Associação de *Peers*



Ao receber essa mensagem, o Tracker irá acessar a lista de *peers* disponíveis do fluxo F e selecionar um possível pai para o *peer* A (*peer* B). Nessa implementação do BBLiss, o critério de seleção de pai é simplesmente escolher o primeiro *peer* disponível na lista, mas algoritmos que selecionem segundo algum critério de otimização podem

ser facilmente implementados. O Tracker então, encaminha ao *peer* B a mensagem REQUEST_DESCRIPTION seguida do identificador do *peer* A. Ao receber essa mensagem, o *peer* B envia ao tracker a mensagem SEND_DESCRIPTION seguida do identificador do *peer* A e da oferta gerada. O Tracker encaminha essa mensagem para o *peer* A, que por sua vez responde com uma mensagem SEND_DESCRIPION seguida do identificador do *peer* B e de uma resposta. O Tracker encaminha a mensagem de resposta e caso o *peer* B aceite-a, uma conexão é estabelecida entre os dois e o fluxo começa a ser transmitido de B para A.

Após se conectarem, ambos os *peers* avaliam seus recursos disponíveis e caso julguem possível suportar mais uma conexão, notificam o *Tracker* com uma mensagem AVAILABLE. Na implementação atual cada *peer* suporta no máximo dois filhos. Porém, será fácil para versões futuras alterarem esse limite dinamicamente em cada *peer*.

Caso o *peer* A não aceite a oferta do *peer* B, um temporizador determina que a negociação falhou. Isso faz com que o *peer* B envie novamente uma mensagem de AVAILABLE para que seja reinserido na lista de disponíveis. O *peer* A por sua vez repete o processo de associação.

Note que devido ao atraso não determinístico da rede, existe a possibilidade de que o *peer* B seja sempre reinserido na lista de *peers* disponíveis antes do *peer* A tentar se juntar à árvore, o que acarreta em múltiplas falhas de conexão.

4.3.3 Reconexão

Já que o sistema utiliza uma árvore de distribuição, cada *peer* depende de todos os seus ancestrais para receber um fluxo. Quando um *peer* é desconectado, todos os seus descendentes têm a reprodução interrompida. É importante que os *peers* detectem falhas e saibam se recuperar. Na implementação, foi adotada a seguinte estratégia para lidar com falha de *peers*.

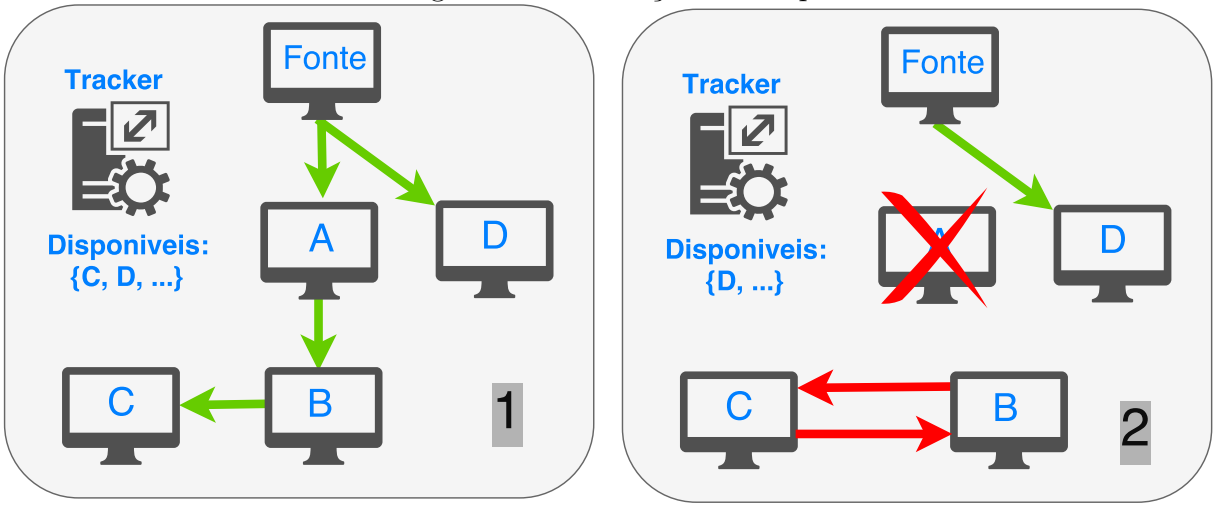
Assim que um *peer* detecta que sua conexão com o pai foi interrompida, ele também interrompe a conexão com seus filhos, e repete o mesmo processo de associação que um *peer* recém chegado, como já descrito na Subseção 4.3.2. Esse método gera reconexões graduais e automáticas. A principal vantagem é a simplicidade de implementação, já que este mecanismo já é utilizado no processo de associação. Por outro lado, *peers* de níveis mais baixos levam mais tempo para detectar uma falha e retomar a reprodução.

4.3.4 *Loops*

Ainda considerando o cenário de falha, é possível que um *peer* se conecte a um descendente que ainda não detectou a falha, formando um *loop*. Podemos acompanhar o processo de formação de um *loop* através da Figura 4.4. Nela, observamos que o *peer* A liga o ramo (B, C) ao *peer* Fonte. Naturalmente, os *peers* C e D estão disponíveis para retransmitir o fluxo, já que ocupam a posição de folhas na árvore de distribuição, por isso estão na lista de *peers* disponíveis do *Tracker*. No cenário 2 da mesma Figura, vemos que o *peer* A se desconecta, fazendo com que o fluxo pare de chegar aos *peers* C e B. O *peer* B irá detectar que seu pai se desconectou e irá iniciar o processo de reconexão conforme explicado na Subseção 4.3.3. É nesse momento que o problema ocorre. A atual política de seleção de pais ignora a topologia da árvore de distribuição e escolhe sempre o primeiro *peer* da lista de disponíveis. Com isso, o *Tracker* irá selecionar o *peer* C para se tornar pai do *peer* B, sendo que B já é pai de C na árvore de distribuição. Assim, um *loop* se forma.

Eventualmente o WebRTC irá detectar que nenhuma informação está sendo transmitida nas conexões entre C e B e irá considerar que ambos se desconectaram. Os *peers* C e B continuarão a tentar se reconectar à árvore de distribuição até que sejam bem sucedidos. No cenário ilustrado, a próxima reconexão será bem sucedida, já que o próximo *peer* da lista de disponíveis é o *peer* D, que não é descendente nem de B nem de C e ainda está conectado à fonte. *Loops* são problemáticos para o BBLiss porque aumentam muito o tempo de reconexão conforme será demonstrado nos testes apresentados no Capítulo 5.

Figura 4.4: Formação de Loop



4.3.5 Encerramento do Fluxo

Como já dito, o fluxo é gerado no *peer* que ocupa a raiz da árvore de distribuição. Portanto, o BBLiss é capaz de se recuperar da desconexão de qualquer *peer*, a menos que ele seja a raiz. Quando isso acontece o sistema considera que o fluxo foi encerrado, e se inicia o processo de desconexão da árvore de distribuição que era utilizada.

Para isso é necessário primeiro detectar que a raiz se desconectou. Seria possível fazer isso da mesma forma que o BBLiss detecta a desconexão de qualquer outro *peer*, fazendo com que os filhos detectem a desconexão dos pais. Bastaria informar aos filhos da raiz que eles ocupam tal posição, e que em caso de desconexão de seu pai, ao invés de se reconectarem, eles devem informar o encerramento do fluxo. Porém isso violaria a característica de transparência de topologia, que é desejável para que o sistema mantenha a escalabilidade. Por isso, outro método é utilizado.

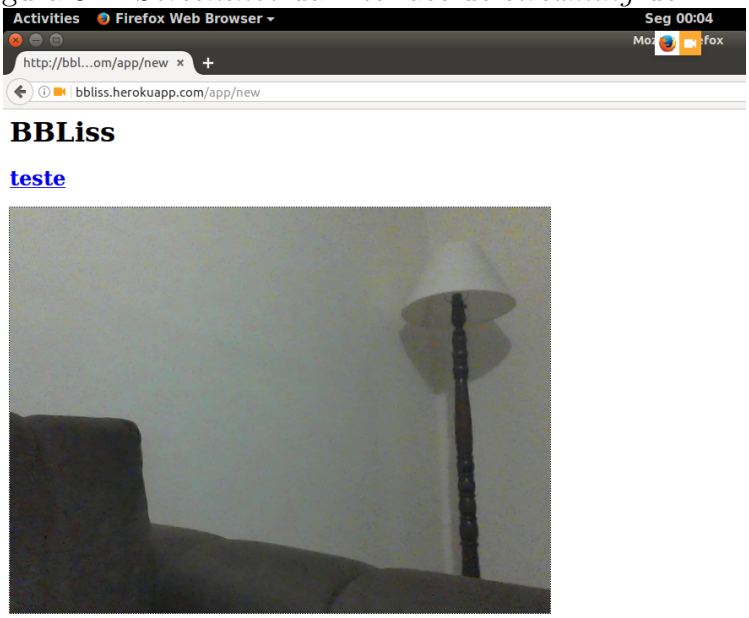
No BBLiss, cada *peer* mantém uma conexão Web Socket com o *Tracker* para troca de mensagens de controle. Essa conexão permite que o *Tracker* saiba se um *peer* ainda está conectado. Essa capacidade é usada para fins de *debug* e, em especial, para detectar que a raiz do fluxo se desconectou. Quando o *Tracker* detecta a desconexão de um *peer* raiz, ele remove o fluxo que era gerado por esse *peer* da lista de fluxos ativos. A partir desse momento nenhum *peer* consegue se conectar ou reconectar à árvore de distribuição desse fluxo, gerando a desconexão gradual das árvores que perderam suas raízes.

Capítulo 5

Testes

Os testes realizados com o sistema BBLiss tiveram como objetivo verificar seu funcionamento normal e também avaliá-lo em caso de falha de algum nó ancestral na árvore. Foram realizados dois tipos de teste. O primeiro teste realizado visa medir o tempo de início da reprodução. O segundo teste recria um cenário de falha onde os *peers* devem retomar a reprodução interrompida. Ambas as variáveis são analisadas considerando *peers* que ocupam diferentes alturas na árvore de distribuição. Foram considerados *peers* com alturas até 4.

Figura 5.1: *Screenshot* da interface de *streaming* do BBLiss



Não foram realizados testes com *peers* em redes diferentes devido às limitações relacionadas ao uso de STUN citadas na Seção 3.2. Portanto, todos os testes foram realizados instanciando *peers* manualmente na mesma máquina e executando o *Tracker*

em um servidor remoto. Dessa forma, o tráfego de mídia fica limitado à rede local e apenas o tráfego de controle passa pela Internet. Os cenários de teste foram montados limitando o número de filhos em cada *peer*, possibilitando assim, controlar a altura da árvore de distribuição em função da quantidade de *peers* que a compõem. Na Figura 5.1 podemos ver a tela onde o usuário transmite o *stream*. O navegador utilizado para os testes foi o Mozilla Firefox 50.1.0 para Linux.

5.1 Tempo de início da reprodução

No primeiro teste, um temporizador era iniciado quando o *peer* enviava a mensagem REQUEST_DESCRIPTION ao *Tracker*, sinalizando seu interesse em receber um *stream*. Após todo o processo de negociação e estabelecimento de conexão já explicado na Subseção 4.3.2, a reprodução do vídeo se iniciava e o temporizador era finalizado. Especialmente para este teste, o *Tracker* informa a cada *peer* qual é a altura que ele ocupa na árvore de distribuição. Apesar de não utilizar nenhuma informação de topologia da árvore de distribuição para operar, apenas para fins de depuração o *Tracker* armazena essa estrutura, o que permitiu informar a altura que um *peer* ocupa na árvore.

Foram coletadas 86 amostras do par tempo de início da reprodução e altura do *peer*. Após a remoção de *outliers*, restaram 72 itens. Na Tabela 5.1 é mostrado um resumo da variável "tempo de início da reprodução"e um intervalo de confiança de 95% para a média. A Figura 5.2 mostra o histograma da variável "tempo de início de reprodução". Na Figura 5.3 é exibido o *boxplot* do tempo de início de reprodução para cada altura dos *peers*. A Figura 5.4 mostra os valores médios do tempo de início de reprodução juntamente com seus respectivos intervalos de confiança a 95%.

Através da Figura 5.4 é perceptível que em 95% das vezes, o tempo médio de início de reprodução estará entre 800ms e 2s. Essa variação pode ser considerada pequena e parece não ser proporcional à altura dos *peers*. Isso era esperado e comprova uma das vantagens da topologia em árvore em relação à topologia em malha, o baixo tempo de início de reprodução. Isso ocorre porque basta que um novo *peer* se conecte a apenas um outro *peer* para começar a receber um fluxo.

Figura 5.2: Tempo de início de reprodução (ms)

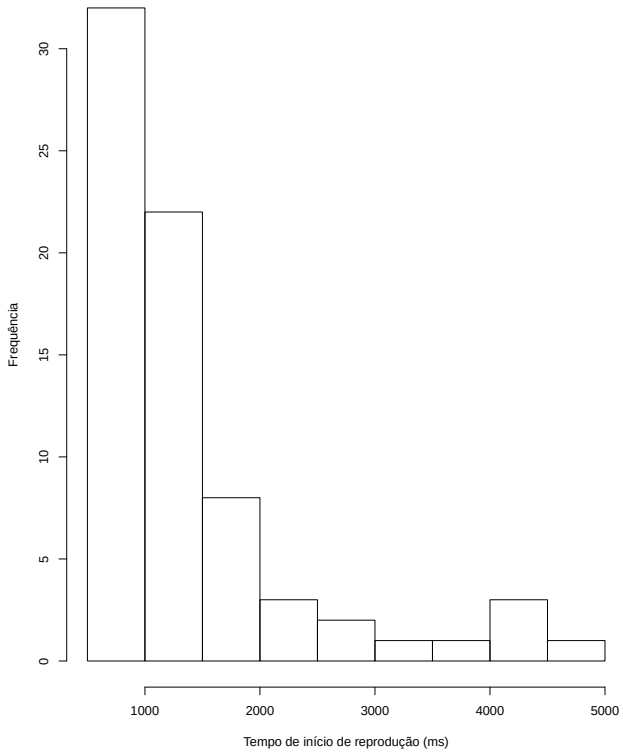


Tabela 5.1: Resumo "Tempo de início da reprodução"(ms)

Mínimo	509.2
1º Quartil	746.6
Mediana	1048.0
Média	1359.0
3º Quartil	1549.0
Máximo	4690.0
Desvio Padrão	944.4902
IC 95% para média	(1138.151, 1578.883)

Figura 5.3: Tempo de início de reprodução (ms) para diferentes alturas de *peers*

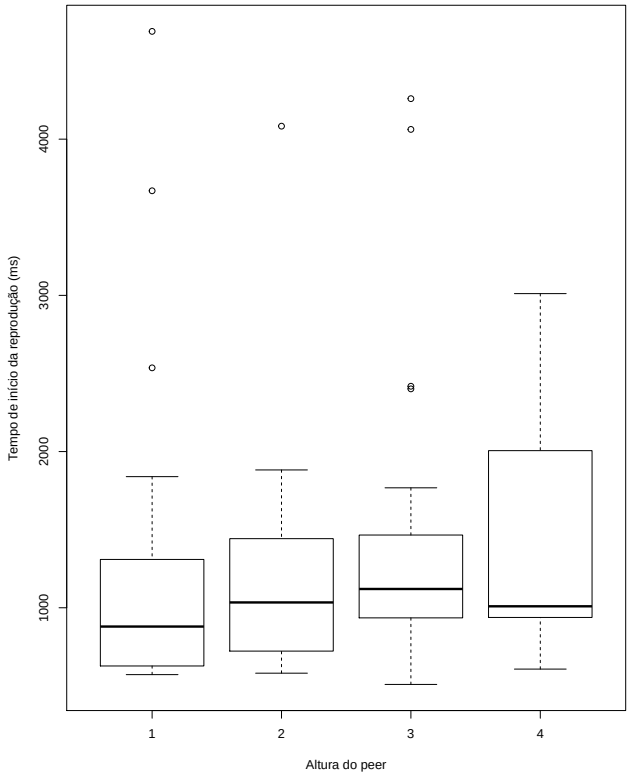


Figura 5.4: Média do Tempo de início de reprodução (ms) para diferentes alturas de *peers*

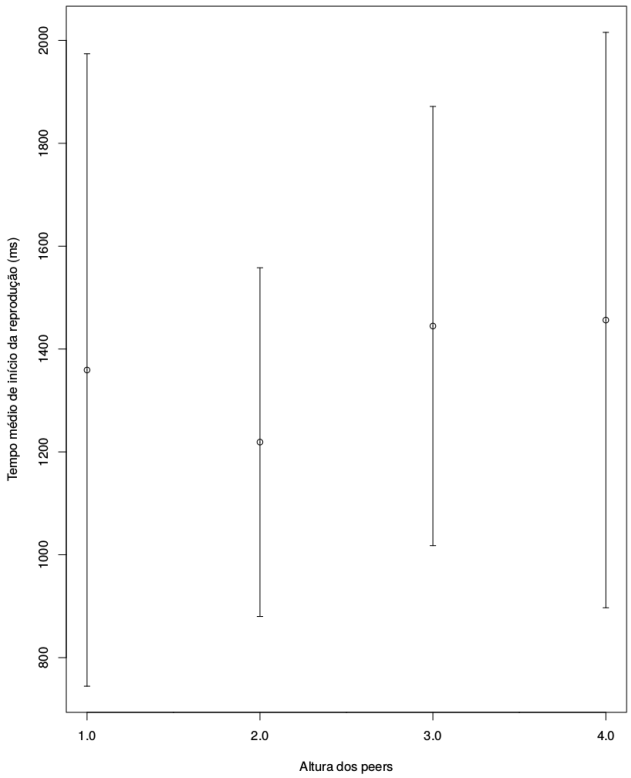


Tabela 5.2: Resumo da variável "Tempo de retomada da reprodução"(s)

Mínimo	25.81
1º Quartil	43.57
Mediana	71.89
Média	71.49
3º Quartil	88.15
Máximo	140.20
Desvio Padrão	34.40
IC 95% para média	(61.80, 81.16)

5.2 Tempo de retomada da reprodução

O segundo teste realizado mediu o tempo em segundos gasto por um *peer* para retomar a reprodução após um de seus ancestrais ser desconectado.

Cada sessão de teste consistiu em iniciar um fluxo e gerar uma árvore com determinada altura. Após montar a árvore, o *peer* que ligava um dos ramos à raiz era desconectado. Como consequência, todo o ramo parava de receber o fluxo e inciava o processo de reconexão, conforme apresentado na Seção 4.3.3. O tempo gasto por cada *peer* do momento da desconexão até a retomada da reprodução foi medido. As medições foram feitas através dos *logs* gerados no *Tracker* sempre que um *peer* requisitava uma reconexão.

Foram coletadas 51 amostras do par tempo de retomada da reprodução e altura do *peer*. Não foi detecado nenhum *outlier*. Na Tabela 5.2 é mostrado um resumo da variável "tempo de retomada da reprodução"e um intervalo de confiança de 95% para a média. A Figura 5.5 mostra o histograma da variável dessa variável. Na Figura 5.6 é exibido o *boxplot* do tempo de retomada da reprodução para cada altura dos *peers*. A Figura 5.7 mostra os valores médios do tempo de retomada de reprodução juntamente com seus respectivos intervalos de confiança a 95%.

Concluímos com esses testes que o tempo médio de retomada de reprodução após desconexão se mantém, com 95% de certeza, entre 1min 1seg e 1min 21seg, como podemos ver na Tabela 5.2. Esses valores podem ser considerados muito altos para a maior parte dos usuários. Atribuímos essa demora ao método utilizado pelo WebRTC para detectar

Figura 5.5: Tempo de retomada de reprodução (s)

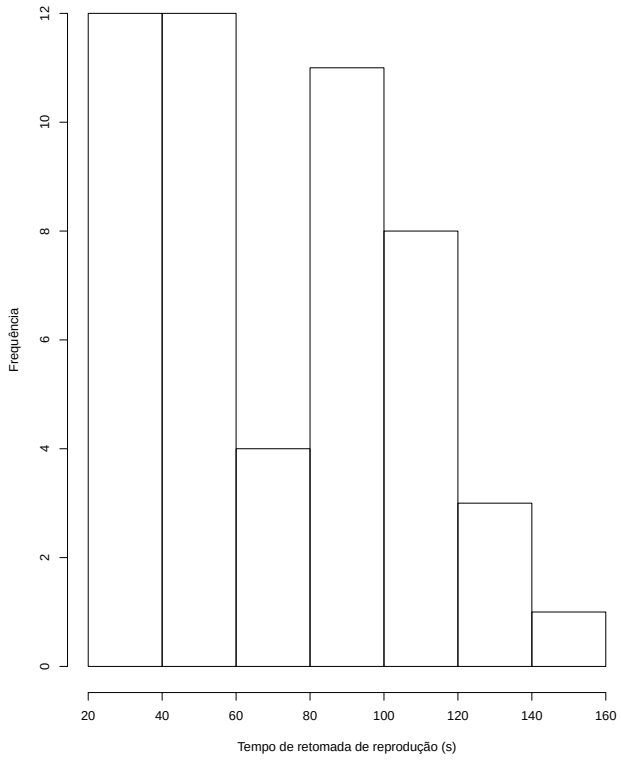


Figura 5.6: Tempo de retomada de reprodução (s) para diferentes alturas de *peers*

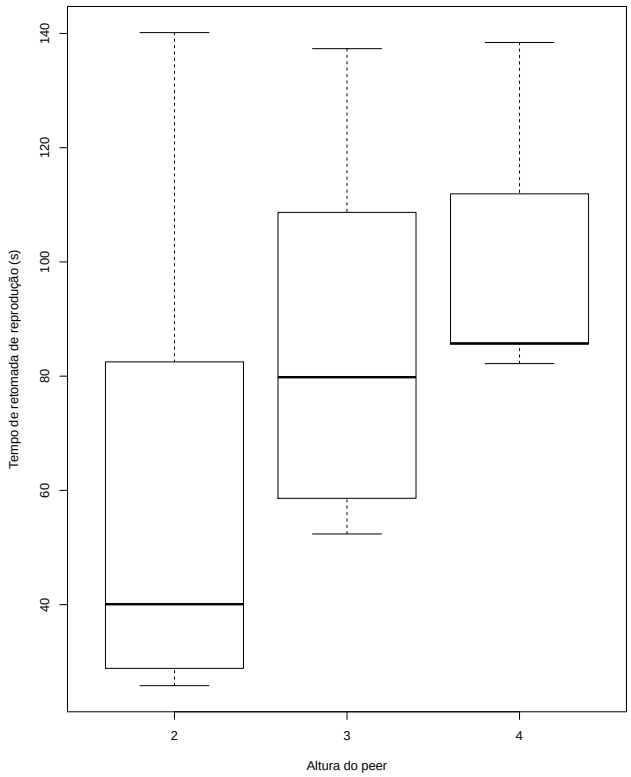
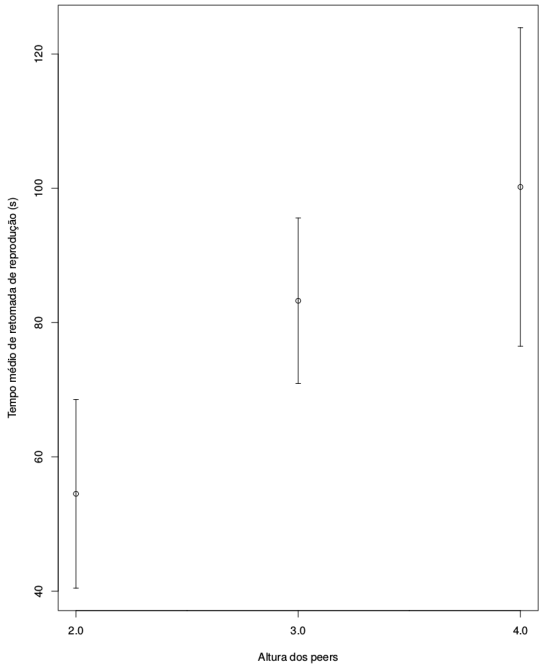


Figura 5.7: Média do Tempo de retomada de reprodução (s) para diferentes alturas de *peers*



desconexões. Incluir essa funcionalidade na aplicação poderia nos permitir detectar mais rapidamente a desconexão de um nó pai.

Durante os testes, também foi perceptível que a variância do tempo de retomada da reprodução em *peers* de altura 2 foi maior que a de *peers* nas demais alturas. A razão disso é que existem *peers* de altura 2 em árvores de alturas 2, 3 e 4; formadas por quantidades diferentes de *peers*. Quanto maior a quantidade de *peers* em uma árvore maior será a chance de se formarem *loops* durante o processo de reconexão, conforme discutido na Subseção 4.3.4, o que leva a um aumento no tempo de retomada de reprodução. Ou seja, *peers* de altura 2 estiveram expostos a probabilidades diferentes de formação de *loops*, daí a maior variância nos tempos de retomada de reprodução. A Figura 5.6 mostra os *boxplots* dos tempos de retomada de reprodução para cada altura de árvore.

5.3 Limitações

A primeira limitação identificada é a incompatibilidade da API WebRTC entre navegadores. Cada navegador implementa seu *working draft* de uma forma diferente, exigindo que qualquer aplicação que deva funcionar em múltiplos navegadores utilize uma

camada de abstração para acessar a API WebRTC. No BBLiss foi utilizada a biblioteca webrtc-adapter 2.0.8 [32].

Outra limitação encontrada foi a incapacidade de conectar *peers* que estão atrás de NATs simétricos sem utilizar TURN. NATs simétricos operam de uma forma que impossibilita o funcionamento do STUN, tornando inúteis os candidatos host e reflexivos. A única forma de conectar esse tipo de *peer* é através de TURN, um serviço que usa um servidor intermediário para encaminhar fluxo entre os *peers*. O problema é que em geral esse tipo de serviço é pago, dado o grande consumo de banda.

Como citado anteriormente o STUN analisa o endereço de transporte de origem de um pacote recebido para informar ao *peer* qual é o seu IP acessível publicamente. Essa estratégia não funciona em NATs simétricos, porque cada conexão que sai da rede local tem seu endereço de origem traduzido para uma porta aleatória. Portanto, o endereço e porta identificados pelo STUN não serão mais válidos para que um outro *peer* se conecte, já que as portas são atribuídas aleatoriamente a cada nova conexão de saída.

Para agravar o problema a maior parte das conexões domésticas utilizam NATs simétricos. Por essa razão, a atual implementação do BBLiss só permite que um fluxo seja distribuído entre *peers* na mesma rede. Para determinar o tipo de NAT em uso foi usada a ferramenta pystun [33].

Após implementação e teste do BBLiss, foi possível identificar uma série de desafios, limitações e potenciais trabalhos futuros. Ainda há bastante o que melhorar, mas o sistema já é capaz de construir uma árvore de distribuição e lidar com reconexões automaticamente.

Capítulo 6

Conclusão

O BBLiss obteve sucesso ao permitir que um usuário realize *streaming* ao vivo diretamente de seu navegador web para uma grande quantidade de espectadores. Isso foi possível utilizando uma infraestrutura bastante enxuta graças ao uso de uma rede *peer-to-peer* formada pelos usuários que assistem ao *streaming*. Com o BBLiss em funcionamento será possível testar novos algoritmos e funcionalidades em uma aplicação real e acessível.

Apesar das limitações descritas no Capítulo 5, o BBLiss cumpre sua proposta e também abre espaço para diversas melhorias e propostas de trabalhos futuros sugeridas na próxima Subeção 4.3.2.

A versão mais recente do BBLiss está disponível no link http://bbliss.heroku.com. O código fonte está disponível em http://github.com/pitzer42/bbliss sob a licença MIT. Os dados coletados durante os testes estão disponíveis no formato CSV no mesmo repositório, assim como os *scripts* utilizados para analisá-los e para gerar gráficos.

6.1 Trabalhos Futuros

Uma melhoria imediata no BBLiss é utilizar uma infraestrutura de TURN própria. Existem algumas implementações gratuitas [34] que podem ser executadas em algum serviço de hospedagem gratuita ou em servidores na universidade. Isso irá permitir que o BBLiss conecte *peers* de redes diferentes, ampliando sua aplicabilidade.

O sistema poderia lidar com *loops* e desconexões de forma mais inteligente. Uma solução mais adequada seria ao invés desconectar os filhos de um *peer* que deixa a árvore, mantê-los conectados e enviar uma mensagem de bloqueio a todos eles. Essa mensagem

iria impedir que esse *peers* aceitassem novas conexões até que o fluxo voltasse a ser retransmitido. Dessa forma, apenas a raiz da subárvore iria tentar se reconectar. Outra vantagem é que isso poderia dificultar a formação de *loops*, já que a raiz da subárvore não conseguiria se reconectar a uma das folhas. *Loops* ainda podem ser formados caso o atraso de propagação da mensagem na subárvore seja maior que o tempo gasto para a raiz desta subárvore contatar o *Tracker* e selecionar uma folha.

Outra forma de melhorar o desempenho do sistema é associar *peers* segundo algum critério de otimização. Os *peers* poderiam fornecer informações sobre seu contexto para que o Tracker possa conectá-los de maneira mais inteligente. Por exemplo, os *peers* poderiam informar uma geolocalização aproximada para que o Tracker possa conectar *peers* próximos, a fim de diminuir a latência entre eles.

A implementação atual oferece um parâmetro para enviar esse tipo de informação extra e também um ponto de extensão no *Tracker* para implementar a política de associação.

Graças a arquitetura de software utilizada no sistema, é possível incluir novas funcionalidades apenas criando *peers* mais especializados. Por exemplo, é possível incluir a funcionalidade de gravação do fluxo implementando um *peer* que, além de retransmitir, também armazene o fluxo recebido. Outra funcionalidade que pode ser facilmente incorporada é a criação de "super-peers". Esse tipo de *peer* pode ser uma máquina de alta disponibilidade e com recursos abundantes, que potencializaria a capacidade de transmissão de conteúdo. Super-peers podem ser implementados usando as versões nativas [35] do WebRTC e as interfaces fornecidas pelo BBLiss.

Outro trabalho futuro interessante é incluir suporte a aplicações interativas. Oferecer aplicações interativas juntamente com o stream pode melhor a experiência do usuário. Esse tipo de aplicação permite incluir informações e funcionalidades adicionais relacionadas ao conteúdo. Essa proposta já existe no Sistema de TV Digital Brasileiro, que utiliza a linguagem declarativa NCL *(Nested Context Language)* [36] para implementar essas aplicações. NCL é uma linguagem declarativa que usa abstrações apropriadas para sincronizar diferentes mídias e adicionar interatividade a elas.

É possível usar um conversor como o NCL4Web [37] para que o desenvolvedor tenha a sua disposição todo o ferramental da linguagem NCL e ao mesmo tempo a flexibilidade de executar sua aplicação em um navegador web. Dessa forma, o BBLiss pode ser usado para

transmitir as mídias que serão integradas na aplicação NCL, gerando mais engajamento do espectador e enriquecendo sua experiência.

Na Subseção 4.3.5 foi apresentado o mecanismo pelo qual um fluxo se encerra. O processo gira entorno de detectar a desconexão da raiz para concluir que o fluxo foi encerrado. Porém, essa desconexão pode ser momentânea e não quer dizer necessariamente que o fluxo foi encerrado. Seria interessante informar ao usuário que o fluxo parou de ser gerado e permitir que a raiz retome a transmissão caso consiga se reconectar.

Para o BBLiss a topologia completa da árvore de distribuição de um fluxo é transparente. Isso permite que otimizações sejam feitas não apenas pelo *Tracker* no momento de associar *peers*, mas também localmente pelos próprios *peers* conectados à árvore. Por exemplo, considerando que *peers* geograficamente próximos podem se comunicar com menor latência, um *peer* poderia detectar que está mais próximo da origem que seu pai e negociar uma troca de posição na árvore de distribuição. Isso gera o efeito global de aproximar *peers* geograficamente próximos na topologia da rede de sobreposição.

Referências Bibliográficas

[1] M. Luengo, "Constructing the crisis of journalism," *Journalism Studies*, vol. 15, no. 5, pp. 576–585, 2014. [Online]. Available: http://dx.doi.org/10.1080/1461670X.2014.891858

[2] S. Deering, "Host extensions for ip multicasting," Internet Requests for Comments, RFC Editor, STD 5, August 1989, http://www.rfc-editor.org/rfc/rfc1112.txt. [Online]. Available: http://www.rfc-editor.org/rfc/rfc1112.txt

[3] J. F. Kurose and K. W. Ross, *Computer Networking: A Top-Down Approach (6th Edition)*, 6th ed. Pearson, 2012.

[4] A. Vakali and G. Pallis, "Content delivery networks: status and trends," *IEEE Internet Computing*, vol. 7, no. 6, pp. 68–74, Nov 2003.

[5] (2016) Webrtc 1.0: Real-time communication between browsers. [Online]. Available: https://www.w3.org/TR/webrtc/

[6] H. Alvestrand, "Overview: Real time protocols for browser-based applications," Working Draft, IETF Secretariat, Internet-Draft draft-ietf-rtcweb-overview-16, November 2016, http://www.ietf.org/internet-drafts/draft-ietf-rtcweb-overview-16.txt. [Online]. Available: http://www.ietf.org/internet-drafts/draft-ietf-rtcweb-overview-16.txt

[7] L. F. G. S. Flávio Ribeiro Nogueira Barbosa, "Towards the application of webrtc peer-to-peer to scale live video streaming over the internet," *IX Workshop de Redes P2P, Dinâmicas, Sociais e Orientadas a Conteúdo - Florianópolis*, 2014. [Online]. Available: http://sbrc2014.ufsc.br/anais/files/wp2p/ST4-1.pdf

[8] F. Rhinow, P. P. Veloso, C. Puyelo, S. Barrett, and E. O. Nuallain, "P2p live video streaming in webrtc," in *2014 World Congress on Computer Applications and Information Systems (WCCAIS) - Hammamet, Tunisia*, Jan 2014, pp. 1–6.

[9] F. Fund, C. Wang, Y. Liu, T. Korakis, M. Zink, and S. S. Panwar, "Performance of dash and webrtc video services for mobile users," in *2013 20th International Packet Video Workshop - San Jose, USA*, Dec 2013, pp. 1–8.

[10] T. T. T. Ha, Y. Won, and J. Kim, "Topology and architecture design for peer to peer video live streaming system on mobile broadcasting social media," in *2014 International Conference on Information Science Applications (ICISA) - Seoul, Korea*, May 2014, pp. 1–4.

[11] R. Pantos and W. May, "Http live streaming," Working Draft, IETF Secretariat, Internet-Draft draft-pantos-http-live-streaming-20, September 2016, http://www.ietf.org/internet-drafts/draft-pantos-http-live-streaming-20.txt. [Online]. Available: http://www.ietf.org/internet-drafts/draft-pantos-http-live-streaming-20.txt

[12] *nformation technology – Dynamic adaptive streaming over HTTP (DASH) – Part 1: Media presentation description and segment formats*, ISO/IEC ISO/IEC 23 009-1:2014, 2014.

[13] *HTML5 - A vocabulary and associated APIs for HTML and XHTML*, Std., 2014. [Online]. Available: https://www.w3.org/TR/2014/REC-html5-20141028/

[14] J. Valin, K. Vos, and T. Terriberry, "Definition of the opus audio codec," Internet Requests for Comments, RFC Editor, RFC 6716, September 2012.

[15] J. Bankoski, J. Koleszar, L. Quillio, J. Salonen, P. Wilkins, and Y. Xu, "Vp8 data format and decoding guide," Internet Requests for Comments, RFC Editor, RFC 6386, November 2011, http://www.rfc-editor.org/rfc/rfc6386.txt. [Online]. Available: http://www.rfc-editor.org/rfc/rfc6386.txt

[16] M. P. E. Group, "Advanced video coding for generic audiovisual services," International Telecommunication Union, Tech. Rep., 2016. [Online]. Available: http://www.itu.int/rec/T-REC-H.264-201610-P/en

[17] Y. O. Sharrab and N. J. Sarhan, "Detailed comparative analysis of vp8 and h.264," in *2012 IEEE International Symposium on Multimedia - Irvine, USA*, Dec 2012, pp. 133–140.

[18] R. Trollope. (2016) Open-sourced h.264 removes barriers to webrtc. [Online]. Available: http://blogs.cisco.com/collaboration/open-source-h-264-removes-barriers-webrtc

[19] (2013, October) Avc patent portfolio license briefing. [Online]. Available: http://www.mpegla.com/main/programs/AVC/Documents/avcweb.pdf

[20] (2016, May) Enables webrtc h.264 by default in preparation for m52. [Online]. Available: https://chromium.googlesource.com/chromium/src.git/+/487e16d68766a496d1174e880036e789c927b32d

[21] M. Baugher, D. McGrew, M. Naslund, E. Carrara, and K. Norrman, "The secure real-time transport protocol (srtp)," Internet Requests for Comments, RFC Editor, RFC 3711, March 2004, http://www.rfc-editor.org/rfc/rfc3711.txt. [Online]. Available: http://www.rfc-editor.org/rfc/rfc3711.txt

[22] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson, "Rtp: A transport protocol for real-time applications," Internet Requests for Comments, RFC Editor, STD 64, July 2003, http://www.rfc-editor.org/rfc/rfc3550.txt. [Online]. Available: http://www.rfc-editor.org/rfc/rfc3550.txt

[23] R. Stewart, "Stream control transmission protocol," Internet Requests for Comments, RFC Editor, RFC 4960, September 2007, http://www.rfc-editor.org/rfc/rfc4960.txt. [Online]. Available: http://www.rfc-editor.org/rfc/rfc4960.txt

[24] J. Rosenberg, "Interactive connectivity establishment (ice): A protocol for network address translator (nat) traversal for offer/answer protocols," Internet Requests for Comments, RFC Editor, RFC 5245, April 2010, http://www.rfc-editor.org/rfc/rfc5245.txt. [Online]. Available: http://www.rfc-editor.org/rfc/rfc5245.txt

[25] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley, and E. Schooler, "Sip: Session initiation protocol," Internet Requests for Comments, RFC Editor, RFC 3261, June 2002,

http://www.rfc-editor.org/rfc/rfc3261.txt. [Online]. Available: http://www.rfc-editor.org/rfc/rfc3261.txt

[26] M. Handley, V. Jacobson, and C. Perkins, "Sdp: Session description protocol," Internet Requests for Comments, RFC Editor, RFC 4566, July 2006, http://www.rfc-editor.org/rfc/rfc4566.txt. [Online]. Available: http://www.rfc-editor.org/rfc/rfc4566.txt

[27] J. Rosenberg, R. Mahy, P. Matthews, and D. Wing, "Session traversal utilities for nat (stun)," Internet Requests for Comments, RFC Editor, RFC 5389, October 2008, http://www.rfc-editor.org/rfc/rfc5389.txt. [Online]. Available: http://www.rfc-editor.org/rfc/rfc5389.txt

[28] Webrtc statistics and metrics. [Online]. Available: http://webrtcstats.com/

[29] R. Mahy, P. Matthews, and J. Rosenberg, "Traversal using relays around nat (turn): Relay extensions to session traversal utilities for nat (stun)," Internet Requests for Comments, RFC Editor, RFC 5766, April 2010, http://www.rfc-editor.org/rfc/rfc5766.txt. [Online]. Available: http://www.rfc-editor.org/rfc/rfc5766.txt

[30] T. Chu and Z. Xiong, "Combined wavelet video coding and error control for internet streaming and multicast," in *Global Telecommunications Conference, 2001. GLOBECOM'01. IEEE - San Antonio, USA*, vol. 2. IEEE, 2001, pp. 1430–1434.

[31] I. Fette and A. Melnikov, "The websocket protocol," Internet Requests for Comments, RFC Editor, RFC 6455, December 2011, http://www.rfc-editor.org/rfc/rfc6455.txt. [Online]. Available: http://www.rfc-editor.org/rfc/rfc6455.txt

[32] Webrtc adapter. [Online]. Available: https://github.com/webrtc/adapter

[33] Pystun - a python stun client for getting nat type and external ip. [Online]. Available: https://github.com/jtriley/pystun

[34] Open-source turn server implementation. [Online]. Available: http://turnserver.sourceforge.net/

[35] Webrtc native apis. [Online]. Available: https://webrtc.org/native-code/native-apis/

[36] L. F. G. Soares and R. F. Rodrigues, "Nested context language 3.0 part 8–ncl digital tv profiles," *Monografias em Ciência da Computação do Departamento de Informática da PUC-Rio*, vol. 1200, no. 35, p. 06, 2006.

[37] E. C. O. Silva, J. A. dos Santos, and D. C. Muchaluat-Saade, "Ncl4web: translating ncl applications to html5 web pages," in *Proceedings of the 2013 ACM symposium on Document engineering - Florence, Italy.* ACM, 2013, pp. 253–262.