Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Cliente DASH: Estratégias de Melhoria na Compreensão e Análise de Desempenho de Algoritmos Adaptativos de Streaming

Gustavo Costa de Souza

Monografia apresentada como requisito parcial
para conclusão do Curso de Computação — Licenciatura

Orientador
Prof. Dr. Marcos Fagundes Caetano

Coorientador
Prof. Dr. Marcelo Antonio Marotta

Brasília
2021

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Cliente DASH: Estratégias de Melhoria na Compreensão e Análise de Desempenho de Algoritmos Adaptativos de Streaming

Gustavo Costa de Souza

Monografia apresentada como requisito parcial
para conclusão do Curso de Computação — Licenciatura

Prof. Dr. Marcos Fagundes Caetano (Orientador)
CIC/UnB

Prof. Dr. Marcelo Antonio Marotta
CIC/UnB

Prof. Dr. João José Costa Gondim
CIC/UnB

Prof. Dr. Wilson Henrique Veneziano
Coordenador do Curso de Computação — Licenciatura

Brasília, 05 de novembro de 2021

Dedicatória

A todos que contribuíram para a minha formação e, sobretudo, aos meus pais Edvar e Regilene, e as minhas irmãs Bárbara e Hallana.

Agradecimentos

Primeiramente agradeço a Deus, por ter me dado forças, superado o desânimo e me permitido concluir este trabalho diante das adversidades. Sem Ele, nada disso seria possível.

Aos meus pais Edvar e Regilene, por sempre terem investido na minha educação, cuidado de mim nos momentos difíceis e pelo amor e carinho. Vocês, meus pais, são a fonte de todo empenho empregado neste trabalho.

A minha irmã Barbara, por ter me oferecido conselhos e compartilhado experiência sobre a produção do Trabalho de Conclusão de Curso. Você, Bárbara, é uma ótima irmã.

A minha outra irmã Hallana, por me motivar a ser um irmão melhor e responsável. Você, Hallana, também é uma excelente irmã.

Ao professor Caetano, por ter sido meu orientador e pela dedicação, sabedoria, paciência, companheirismo e solicitude durante todo esse tempo em que estivemos juntos. Você, Caetano, me motivou a não desistir e a fazer o melhor de mim, sendo um grande exemplo de professor.

Ao professor Marotta, pelas correções e ensinamentos que me permitiram apresentar um melhor desempenho no processo de desenvolvimento do meu Trabalho de Conclusão de Curso. Suas contribuições foram valiosas a minha formação acadêmica.

A Banca Examinadora, pela disponibilidade e boa vontade em estar presente na apresentação do meu Trabalho de Conclusão de Curso.

Aos meus amigos, pelo apoio demonstrado ao longo de todo o período de tempo em que me dediquei a este trabalho. Vocês tornaram este processo mais leve.

A todos os professores que passaram pela minha formação acadêmica. Vocês contribuiram no meu desenvolvimento intelectual e a ser uma pessoa melhor.

Aos colegas de curso, por tornarem os anos de estudo mais divertidos, por compartilharem comigo tantos momentos de descobertas e aprendizado e por todo o companheirismo ao longo deste percurso.

A todos que participaram, direta ou indiretamente, do desenvolvimento deste trabalho de pesquisa, enriquecendo o meu processo de aprendizagem.

Resumo

O *Streaming* Adaptável sobre HTTP (HAS), juntamente com o padrão DASH, tornou-se a técnica dominante de entrega de vídeo pela Internet. Neste modelo, o algoritmo adapativo de taxa (ABR) é um componente central na determinação da qualidade de vídeo disponível que é transmitida pela rede. Por este motivo, múltiplas propostas de ambientes que reproduzem aplicações HAS para avaliação de lógicas ABR, tanto com fins de investigação quanto de produção, foram desenvolvidas. A plataforma *pyDash*[1] [1], entretanto, é uma ferramenta educacional para o desenvolvimento e estudo de algoritmos adaptativos de *streaming* de vídeo (ABR). Com base na análise do estado da arte de ferramentas já desenvolvidas, e com o propósito de otimizar o estudo e a compreensão de algoritmos ABR pelos estudantes, melhorias na ferramenta foram implementadas. Três foram as propostas de melhorias incorporadas à plataforma *pyDash*. A primeira envolve a habilitação de um decodificador e reprodutor de mídia MPEG-DASH usando o *framework* Gstreamer. A segunda está relacionada à proposta de uma aplicação que exibe estátisticas consideradas relevantes na avaliação de desempenho de algoritmos adaptativos, através de gráficos dinâmicos. E, a terceira compreende a inserção de computação de funções de Qualidade de Experiência (QoE) do usuário, usando métricas computadas durante a sessão de *streaming* da ferramenta. Como resultado destas melhorias, é demonstrado que o entendimento da dinâmica e do desempenho de soluções ABR pelos alunos serão potencializados.

Palavras-chave: Algoritmos Adapativos de Taxa (ABR), *Streaming* Adaptável Dinâmico sobre HTTP (DASH), Gstreamer, Qualidade de Experiência (QoE)

[1] https://github.com/mfcaetano/pydash

Abstract

HTTP Adaptive Streaming (HAS), along with the DASH standard, has become the dominant technique for delivering video over the Internet. In this model, the rate adaptive algorithm (ABR) is a central component in determining the available video quality that is transmitted over the network. For this reason, multiple proposals for environments that reproduce HAS applications for ABR's logic evaluation, both for research and production purposes, were developed. The platform *pyDash* [2] [1], however, is an educational tool for the development and study of adaptive video *streaming* algorithms (ABR). Based on the analysis of the state of the art of tools already developed, and with the purpose of optimizing the study and understanding of the algorithms ABR by the students, improvements in the tool were introduced. Three were the improvements implemented in the platform *pyDash*. The first involves enabling a MPEG-DASH media decoder and player using the *framework* Gstreamer. The second is related to the proposal of an application that provides relevant statistical data on the performance evaluation of adaptive algorithms, through dynamic graphics. And, the third comprises the insertion of computation of user Quality of Experience (QoE) functions, using metrics computed during a *streaming* session of the tool. As a result of these improvements, the understanding of the dynamics and performance of ABR solutions by the students will be enhanced.

Keywords: Adaptive Bitrate Algorithm (ABR), Dynamic Adaptive Streaming over HTTP (DASH), Gstreamer, Quality of Experience (QoE)

[2] https://github.com/mfcaetano/pydash

Sumário

1 Introdução **1**

1.1 Problema 3

1.2 Objetivos 4

1.2.1 Objetivo geral 4

1.2.2 Objetivos específicos 4

1.3 Metodologia 4

1.4 Estrutura do trabalho 5

2 Fundamentação Teórica **6**

2.1 Protocolos de Streaming de Vídeo 6

2.1.1 Streaming sobre UDP 6

2.1.2 Streaming sobre HTTP 8

2.1.3 Streaming Adaptável sobre HTTP e DASH 10

2.2 Algoritmo Adaptativo de Taxa (ABR) 13

2.3 Métricas de Qualidade de Experiência (QoE) 14

3 Trabalhos Relacionados **18**

3.1 Dash.js 18

3.2 TAPAS 20

3.3 AStream 21

3.4 *End-to-End DASH Platform* 21

3.5 Dashc 22

3.6 GoDASH 23

3.7 Framework Simulativo para Aplicações de Streaming Adaptável sobre HTTP 24

3.8 Sabre 24

3.9 PyDash 26

3.9.1 Arquitetura 26

3.9.2 Funcionamento 29

3.9.3 *Whiteboard* 33

3.9.4 Implementação de um Novo Algoritmo ABR com pyDash 34
3.10 Análise Comparativa 39

4 Melhorias implementadas na plataforma *pyDash* **41**
4.1 Reprodutor de Mídia DASH 41
4.1.1 Gstreamer 42
4.1.2 Estrutura do Reprodutor de Mídia DASH 45
4.1.3 Implementando o Reprodutor de Mídia DASH no *pyDash* 49
4.2 Aplicação de Exibição de Estatísticas em Tempo Real 53
4.2.1 Script Python 53
4.2.2 Funcionamento 55
4.3 Funções de Desempenho QoE 58
4.3.1 Função QoE 1 58
4.3.2 Função QoE 2 59
4.3.3 Implementando Funções QoE no *pyDash* 59

5 Resultados **61**
5.1 Configurações de Experimento 61
5.2 Demonstração do Reprodutor de Vídeo DASH 62
5.3 Demonstração da Aplicação de Exibição de Estatísticas em Tempo Real . . 64
5.4 Demonstração do Reprodutor de Mídia DASH com a Exibição de Estátisticas em Tempo Real 65
5.5 Demonstração das Funções QoE 67

6 Conclusão e Trabalhos Futuros **70**

Referências **73**

Lista de Figuras

1.1 Ilustração esquematizada do *Streaming* Adaptável sobre HTTP (HAS). . . . 2

2.1 Streaming tradicional com RTSP. 7
2.2 Arquitetura cliente-servidor DASH. 11
2.3 Estrutura da Descrição de Apresentação de Mídia (MPD). 13
2.4 Encapsulamento QoS e QoE. 15

3.1 Captura de tela da aplicação cliente DASH da ferramenta dash.js. 19
3.2 Fluxo de dados e módulos da ferramenta TAPAS. 20
3.3 Captura de tela do cliente DASH da plataforma. 22
3.4 Visão geral da arquitetura Sabre. 25
3.5 DASH *Dataset* utilizado na plataforma *pyDash*. 27
3.6 Vídeo *BigBuckBunny* dividido em segmentos de tamanhos fixos. 27
3.7 Composição do cliente DASH da plataforma *pyDash*. 28
3.8 Fluxo de mensagens do funcionamento do *pyDash*. 31
3.9 Área de transferência de estatística *Whiteboard*. 33

4.1 Exemplo de *pipeline*. 42
4.2 Elementos do GStreamer com seus *pads*. 43
4.3 Estrutura conceitual de uma conexão entre *pads no Gstreamer*. 44
4.4 Máquina de estados de um *pipeline* ou elemento. 44
4.5 Pipeline Gstreamer para o reprodutor de mídia DASH. 45
4.6 Relação de interação entre *Whiteboard* e o script Python 54
4.7 Funcionamento da aplicação de exibição de gráficos dinâmicos. 57

5.1 Exemplo 1 de captura de tela do *pyDash* executado com o reprodutor de mídia DASH habilitado. 62
5.2 Exemplo 2 de captura de tela do *pyDash* executado com o reprodutor de mídia DASH habilitado. 63
5.3 Exemplo 3 de captura de tela do *pyDash* executado com o reprodutor de mídia DASH habilitado. 63

5.4 Exemplo 1 de captura de tela do *pyDash* executado com a exibição de estátisticas em tempo real através de gráficos dinâmicos. 64
5.5 Comando de execução da aplicação que exibe gráficos dinâmicos. 65
5.6 Exemplo 1 de captura de tela do *pyDash* executado com o reprodutor de mídia DASH habilitado e a exibição de estátisticas em tempo real através de gráficos dinâmicos. 66
5.7 Exemplo 2 de captura de tela do *pyDash* executado com o reprodutor de mídia DASH habilitado e a exibição de estátisticas em tempo real através de gráficos dinâmicos. 67
5.8 Exemplo 1 de captura de tela do *pyDash* com o resultado das métricas e funções QoE do algoritmo *R2ADynamicSegmentSizeSelection*. 68
5.9 Exemplo 2 de captura de tela do *pyDash* com o resultado das métricas e funções QoE do algoritmo *r2a_fuzzy*. 68

Lista de Tabelas

2.1 Sumário de símbolos usados na computação de métricas de QoE. 16

3.1 Parâmetros do arquivo de configuração *dash_client.json*. 30
3.2 Comparação de recursos das ferramentas de análise de algoritmos ABR. . . 40

4.1 Elementos Gstreamer usados na criação do reprodutor de mídia DASH. . . . 46
4.2 Três níveis de desempenho do cliente DASH com base nas métricas de performance. 59

Lista de Abreviaturas e Siglas

3GPP 3rd Generation Partnership Project.

ABR Adaptive BitRate.

API Aplication Programming Interface.

CIC Departamento de Ciência da Computação.

CPU Central Process Unit.

CSV Comma-separated values.

DASH Dynamic Adaptive Streaming Over HTTP.

GUI Graphical User Interface.

HAS HTTP Adaptive Streaming.

HDS HTTP Dynamic Streaming.

HLS HTTP Live Streaming.

HTTP Hypertext Transfer Protocol.

IF Influence Factor.

IP Internet Protocol.

ISO International Organization for Standardization.

ISP Internet Service Provider.

JSON JavaScript Object Notation.

MPD Media Presentation Description.

MPEG Moving Picture Experts Group.

NAT Network Adress Translate.

PDS Progressive Download Streaming.

QoE Quality of Experience.

QoS Quality of Service.

QUIC Quick UDP Internet Protocol.

RAM Random Access Memory.

RFC Request for Comments.

RTCP Real-Time Control Protocol.

RTMP Real-Time Messaging Protocol.

RTP Real-Time Transport Protocol.

RTSP Real-Time Streaming Protocol.

TCP Transfer Control Protocol.

UDP User Datagram Protocol.

UnB Univerdade de Brasília.

URL Uniform Resource Locator.

VoD Video on Demand.

XML eXtensible Markup Language.

Capítulo 1

Introdução

Segundo relatórios da Cisco [2], o *streaming* de vídeo - distribuição de vídeo feita de ponta a ponta pela Internet - contou com cerca de 80% do tráfego da Internet em 2020. A Sandvine, empresa de equipamentos de rede, também destacou que o *streaming* de vídeo já supera 60% do volume total de tráfego transportado pela Internet em pesquisa realizada em 2019 [3].

A subtituição de assinatura de TV a cabo/Video sob Demanda (VoD) por provedoras de serviço de *streaming* online já é uma realidade [4]. Elas investem cada vez em mais em conteúdo próprio e em infraestrutura capaz de prover um serviço de *streaming* de qualidade para o usuário final. Empresas como Netflix, Amazon, Hulu, Apple TV+, entre outras, se destacam entre os consumidores finais. Além disso, a recente pandemia de coronavírus (COVID-19) aumentou consideravelmente a quantidade de *streaming* de vídeo em 2020 [5].

Atualmente, há dois métodos principais de entrega de mídia pela Internet (*streaming*): o serviço de *streaming* tradicional e o serviço de *streaming* adaptável. No primeiro método, o controle de entrega de mídia é executado no lado do servidor. Ou seja, o cliente atua passivamente no recebimento de mídia, cuja taxa de bits é moderada por um servidor. Neste serviço de entrega podem ser usados protocolos orientados a conexão, como o Protocolo de Mensagem em Tempo Real (RTMP) ou protocolos sem conexão, como o Protocolo de Transporte em Tempo Real (RTP). Por outro lado, no *streaming adaptável*, a lógica de adaptação da taxa de bits da mídia é implementada no lado do cliente. Ou seja, o cliente requisita os dados na taxa de bits determinada de acordo com sua lógica de seleção, enquanto o servidor simplesmente hospeda o conteúdo de mídia. Neste método de entrega, o cliente realiza requisições usando o Protocolo de Transferência de Hipertexto (HTTP) e o Protocolo de Controle de Transmissão (TCP) como protocolos da camada de aplicação e de transporte, respectivamente.

Desde sua difusão, o serviço de *streaming* adaptável, também conhecido como *Streaming* Adaptável sobre HTTP (HAS), rapidamente se tornou a abordagem dominante para *streaming* de vídeo [6]. A vantagem do HAS é a possibilidade do emprego de adaptação da taxa de bits da mídia em relação às condições variáveis de rede. Esta característica contribui para uma experiência de *streaming* contínua (ou pelo menos mais suave) em relação aos serviços de *streaming* tradicional.

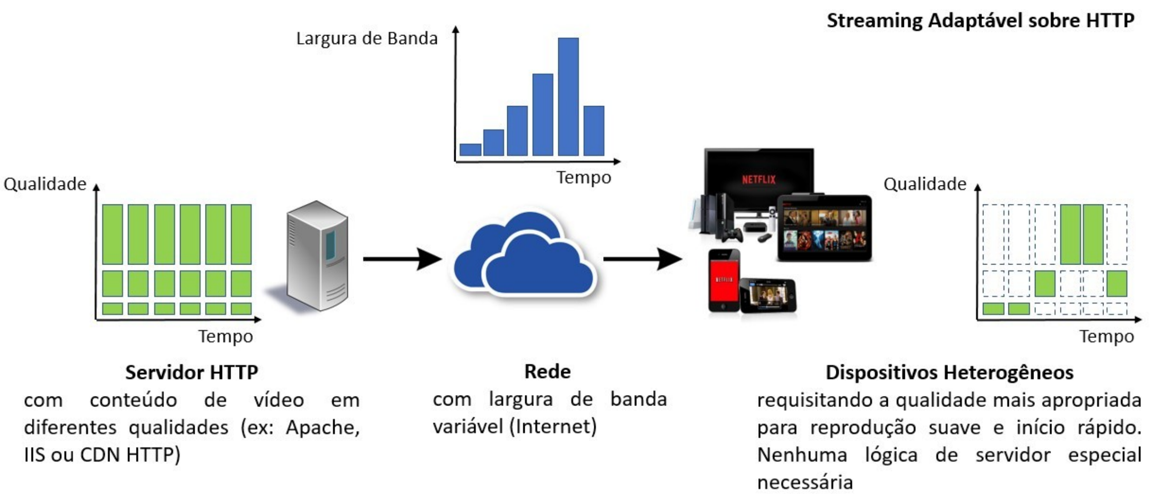

Figura 1.1: Ilustração esquematizada do *Streaming* Adaptável sobre HTTP (HAS) (Fonte: [7]).

A Figura 1.1 ilustra o processo de ponta a ponta do *streaming* de um vídeo sobre HTTP. Como apresentado, o servidor HTTP hospeda conteúdo de vídeo segmentado em diferentes qualidades. Conforme a largura de banda disponível, o dispositivo cliente (*Desktop*, *Smartphone*, por exemplo) requisita a qualidade mais adequada para o *streaming* contínuo/suave do vídeo. Por fim, à medida que os segmentos são baixados da Internet, eles são reproduzidos no cliente.

Dentre as formas de HAS disponíveis, o *Dynamic Adaptive Streaming Over HTTP (DASH)* é o padrão ISO adotado pelas principais provedores de serviços e conteúdo de *streaming* de vídeo [8]. A explicação está, sobretudo, na redução de custo de implantação e no melhor emprego dos recursos de rede, em relação ao Download de Streaming Progressivo (PDS).

Um aspecto chave em clientes de sistemas DASH é o design do algoritmo Adaptativo de Taxa (ABR). Esses algoritmos usam uma variedade de entradas diferentes (por exemplo, ocupação do *buffer* de reprodução, medições de largura de banda, etc.) para selecionar a taxa de bits dos próximos segmentos. Por este motivo, o comportamento do algoritmo ABR tem influência direta no desempenho do sistema e na percepção da Qualidade de

Experiência (QoE) [9]. A QoE leva em consideração a experiência e o nível de satisfação do usuário final e é de muito interesse para estudiosos acadêmicos e industriais na área de multimídia [10]. Embora tradicionalmente a Qualidade de Serviço (QoS) tenha sido usado para medir a eficácia de um serviço, ele não leva em consideração fatores relacionados ao usuário final (como expectativa do usuário). Além disso, a QoS está limitada a serviços de telecomunicações e depende apenas de medições técnicas. A QoE, por outro lado, engloba domínios além das telecomunicações.

1.1 Problema

Como os algoritmos adaptativos desempanham papel fundamental em sitemas de video de *streaming*, é desejável que as lógicas ABR entreguem índices satisfatórios de QoE ao destinatário final. Nesse contexto, ambientes que permitem projetar e implementar lógicas de adaptação a pesquisadores, acadêmicos ou qualquer estudioso sobre o assunto são fundamentais para testar seu desempenho.

Existem várias ferramentas para sistemas de HAS propostas na literatura. Dentre as mais conhecidas estão o dash.js [11], GPAC [12] e Exoplayer[13]. Há também, ferramentas de emuladores de clientes DASH. Incluem-se nesta categoria TAPAS [14], AStream [15], dashc[16] e goDASH [17]. De outro modo, em [18] e [19], encontram-se abordagens simulativas de aplicações de HAS. Todas estas propostas são destinadas a avaliação de novos algoritmos ABR. Entretanto, elas não têm foco educacional. Geralmente, ou elas têm propósitos acadêmicos, ou seja, para quem já é conhecedor da área, ou têm propósitos relacionados a ambientes de produção. Além disso, muitas destas ferramentas requerem prévio conhecimento técnico para instalá-las e operá-las, dificultando seu uso para o estudo de diferentes algoritmos adapatativos de taxa (ABR). Logo, apesar das propostas ensejarem a rápida implementação de novos algoritmos ABR, elas não têm propósitos educacionais.

Diferentemente, temos a plataforma *pyDash*[1] [1], um *framework* para o estudo de algoritmos adapativos de *streaming*. Esta ferramenta foi desenvolvida com objetivo educacional e, inclusive, utilizada em projetos de cursos de redes de computadores pelos alunos da Univerdade de Brasília (UnB). Em [1] é possível encontrar detalhes e resultados satisfatórios do uso da plataforma.

A ferramenta *pyDash*, entretanto, ainda está em desenvolvimento e possui pontos passíveis de melhorias no seu enfoque educacional. Dentre eles, podemos citar a incorporação de um decodificador e reprodutor de mídia e a computação de funções de QoE para a observação do desempenho do algoritmo ABR. Além disso, como estatísticas são regis-

[1] `https://github.com/mfcaetano/pydash`

tradas a todo momento durante a sessão de *streaming*, como ocupação do *buffer*, largura de banda, etc, uma aplicação que exibe estas informações de forma mais amigável pode impulsionar a melhor visualização dos dados de desempenho pelos alunos. Tendo em vista estas observações, se faz necessário o aprimoramento da plataforma *pyDash*, implementando propostas de melhoria e que estejam mais adequadas com objetivo educacional da plataforma.

1.2 Objetivos

1.2.1 Objetivo geral

O objetivo geral deste trabalho é demonstrar aprimoramentos para a ferramenta *pyDash*, visando contribuir no seu propósito educacional.

1.2.2 Objetivos específicos

O objetivos específicos deste trabalho consistem em:

- Melhorar o estudo e análise de desempenho dos algoritmos adaptativos de *streaming* (ABR) pelos alunos;
- Apresentar um projeto de decodificação e reprodução de segmentos de vídeo MPEG-DASH para o *pyDash*;
- Elaborar uma representação dinâmica e em tempo real de estatísticas, utilizando *software* para criação de gráficos e visualizações de dados;
- Validar o desempenho de algoritmos adaptativos (ABR) através da computação de funções de QoE.

1.3 Metodologia

A metodologia de pesquisa empregada neste trabalho se baseia em: reconhecer os esforços já realizados no desenvolvimento de plataformas/clientes DASH, através do levantamento do estado da arte; identificar elementos que auxiliam no estudo e a análise de algoritmos ABR; investigar a plataforma *pyDash*; implementar melhorias passíveis de serem incorporadas na ferramenta *pyDash*; demonstrar as melhorias propostas através de testes experimentais, com a utilização de algoritmos adaptativos desenvolvidos pelos alunos.

1.4 Estrutura do trabalho

O presente trabalho se organiza com a seguinte estrutura: o Capítulo 2 resume as técnicas de entrega de vídeo em redes de pacotes, como o DASH surgiu como escolha preferencial para entrega de conteúdo e examina os conceitos de algorimtos adaptativos de taxa (ABR) e de Qualidade de Experiência (QoE). No Capítulo 3, é apresentado uma revisão da literatura sobre ferramentas de avaliação de algoritmos adaptativos em sistemas HAS, o projeto de arquitetura, componentes e funcionamento da plataforma *pyDash*, bem como uma análise comparativa com as outras ferramentas. No Capítulo 4, são demonstradas as melhorias implementadas no *pyDash*. No Capítulo 5, é realizado a demonstração dos resultados obtidos. Por fim, o Capítulo 6 oferece comentários finais e ressalta questões de pesquisa para potenciais trabalhos futuros.

Capítulo 2

Fundamentação Teórica

Neste capítulo é apresentado a teoria fundamental necessária para se compreender o funcionamento do protocolo DASH, bem como as técnicas de algoritmos de adaptação de taxa e as medições de desempenho QoE. A Seção 2.1 introduz os protocolos de entrega de vídeo em redes de pacotes, apresentando suas características e seu funcionamento. A Seção 2.2 contextualiza o componente central de aplicações que implementam o protocolo DASH e descreve as técnicas de adapatação de taxa. Por fim, na Seção 2.3 é detalhado as principais métricas QoE usadas na mensuração do desemepenho de clientes DASH.

2.1 Protocolos de Streaming de Vídeo

Pessoas, em todas as partes do mundo, estão usando a Internet para assistir a vídeos e shows de televisão por demanda (VoD). A esta solução de entrega de contéudo de vídeo pela Internet chama-se *streaming* de vídeo. Em aplicações de *streaming* de vídeo, os vídeos pré-gravados são armazenados em servidores aos quais os usuários enviam solicitações para verem os vídeos sob demanda. Segundo [20], os sistemas de *streaming* de vídeo podem ser classificados em três categorias: *Streaming* sobre UDP, *Streaming* sobre HTTP e *Streaming* Adaptativo sobre HTTP (HAS).

2.1.1 Streaming sobre UDP

O *Streaming* sobre UDP, também abordado como sistema tradicional de *streaming*, é a técnica de entrega de vídeo sobre os protocolos RTP/UDP. Nesta técnica, a orgiem de fluxo de vídeo acontece do servidor ao cliente, como ilustrado na Figura 2.1. O cliente recebe os trechos de vídeo encapsulado dentro de pacotes projetados especialmente para suportar *streaming* em tempo real, como o Protocolo de Transporte em Tempo Real (RTP) ou um esquema semelhante (possivelmente patenteado) [20]. Enviado por um

servidor de mídia, a mensagem RTP utiliza o protocolo UDP (protocolo sem conexão) para o transporte do pacote.

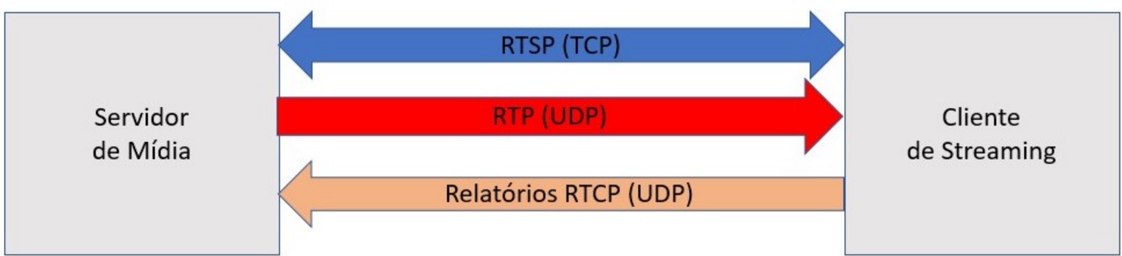

Figura 2.1: Streaming tradicional com RTSP (Fonte: [6]).

Além disso, no *Streaming* sobre UDP, cliente e servidor mantêm, em paralelo, uma conexão de controle separada sobre a qual o cliente envia comandos referentes a mudanças de estado de sessão (como pausar, retomar, reposicionar e assim por diante). Um protocolo comum para controlar os servidores de mídia em sistemas tradicionais de *streaming* é o Protocolo de *Streaming* em Tempo Real (RTSP). O RTSP é responsável por configurar uma sessão de *streaming* e manter as informações de estado durante essa sessão, mas não é responsável pela entrega de mídia real, a qual é tarefa do protocolo RTP.

Concomitantemente, o cliente envia pacotes de controle ao servidor usando o Protocolo de Controle de Transporte em Tempo Real (RTCP). A principal função do RTCP é fornecer um feedback da qualidade dos serviços oferecidos pelo RTP. Assim, com base nos relatórios do RTCP enviados pelo cliente, o servidor de mídia pode realizar adaptação de taxa e agendamento de entrega de dados.

Segundo [20] e [6], o *Streaming* sobre UDP possui algumas desvantagens:

- Por transmitir dados com taxa constante, o *Streaming* sobre UDP pode deixar de oferecer reprodução contínua devido à imprevisibilidade na variação de largura de banda disponível entre servidor e cliente.
- O custo geral e a complexidade da implantação de um sistema de vídeo por demanda em grande escala são altos. Além do servidor de mídia, o *Streaming* sobre UDP exige um servidor de controle de mídia, como um servidor RTSP. Ele é necessário para processar solicitações de interatividade cliente-servidor e acompanhar o estado da aplicação (por exemplo, o ponto de reprodução do cliente no vídeo) para cada sessão do cliente em andamento.

- Protocolos ou configurações adicionais são necessários durante o estabelecimento da sessão. Muitos dispositivos de Tradução de Endereço de Rede (NAT) e *firewalls* são configurados para bloquearem o controle ou tráfego UDP.

2.1.2 Streaming sobre HTTP

De acordo com Tanenbaum [21], o HTTP tem se tornado mais um protocolo de transporte, que oferece um meio para os processos comunicarem conteúdo entre os limites de diferentes redes. Nesse sentido, aplicações de usuários não precisam ser um navegador Web nem um servidor Web. Nos dias atuais, é possível, por exemplo, que um reprodutor de mídia solicite informações básicas sobre os arquivos armazenados no servidor usando o protocolo HTTP. Isto é feito através dos comandos HTTP GET, PUT, etc. Por este motivo, à técnica de entrega de vídeo sobre os protocolos HTTP/TCP denomina-se *Streaming* sobre HTTP.

HTTP

O Protocolo de Trânsferência de Hipertexto (HTTP) é um protocolo da camada de aplicação do tipo solicitação-resposta, que usa o TCP como seu protocolo de transporte. Especificado na RFC 2616, o HTTP define como os clientes requisitam páginas aos servidores e como eles as transferem aos clientes.

Uma típica mensagem de requisição HTTP apresenta duas seções principais na formatção da mensagem: **linha de requisição/solicitação** e **linhas de cabeçalho**. A linha de requisição é composta por três campos nesta ordem: o do método, o qual pode assumir valores diferentes como GET, POST, HEAD, PUT e DELETE; o do URL e o da versão HTTP. O método mais comum das mensagens de requisição HTTP emprega o método GET, o qual é usado quando o cliente quer requisitar um objeto e este é identificado no campo do URL. Nas linhas de cabeçalho, é possível especificar, por exemplo, o hospedeiro no qual o objeto reside, atráves do cabeçalho *Host*, se o cliente quer ou não usar conexões persistentes, com o cabeçalho *Connection*, e muitos outros.

Por outro lado, uma típica mensagem de resposta HTTP contém três seções: uma linha inicial ou **linha de estado**, **linhas de cabeçalho** e, em seguida, o **corpo da entidade**, parte principal da mensagem e que contém o objeto solicitado. A linha de estado tem três campos: o de versão do protocolo, um código de estado e uma mensagem de estado correspondente. Um código de estado 200, por exemplo, cuja mensagem de estado retorna OK é o mais comum de acontecer quando um objeto é entregue sem erros ao cliente. As linhas de cabeçalho seguem a mesma lógica das de mensagens de requisição,

exceto por conter cabeçalhos extras com dados sobre o objeto requisitado. O corpo da entidade comporta o objeto/dado propriamente dito.

Benefícios de usar HTTP

Ao usar o protocolo HTTP, o *Streaming* HTTP oferece alguns benefícios [22]:

- Diferentemente do *Streaming* sobre UDP, o cliente mantém o estado da sessão de reprodução, de forma que os servidores não precisam rastrear o estado da sessão.
- Os servidores HTTP são servidores Web comuns. Isto reduz significativamente os custos operacionais e permite a implantação de caches para melhorar o desempenho e reduzir a carga da rede.
- O tráfego HTTP pode atravessar NATs e *firewalls*, o que fornece alcance mais onipresente.

Funcionamento do Streaming sobre HTTP

Conforme Kurose esclarece em [20], neste paradigma o vídeo é tratado como um arquivo comum armazenado em um servidor HTTP. Cada arquivo possui uma qualidade específica e um Localizador Uniforme de Recursos (URL) correspondente. Uma vez que o usuário queira assistir a um vídeo, o cliente estabelece uma conexão TCP com o servidor e emite uma solicitação HTTP *GET* para esse URL. O servidor, então, entrega o arquivo de vídeo o mais rápido quanto as condições de tráfego permitirem, dentro de uma mensagem de resposta HTTP. Assim que a mensagem chega no lado do cliente, os bytes são armazenados em um *buffer* da aplicação. Ou seja, ao invés de iniciar a reprodução imediatamente quando o vídeo começa a chegar ao cliente, o cliente cria uma reserva de vídeo em um *buffer* de aplicativo. Desse modo, o cliente inicia a reprodução somente após um certo número de bytes atingir um limite pré determinado no *buffer*. Por fim, o aplicativo de *streaming* exibe o vídeo à medida que recebe e armazena em *buffer* os *frames* correspondentes às últimas partes do vídeo.

Aplicação

Abordagem aplicada pelo YouTube em 2005, o *Streaming* de *Download* Progressivo (PDS) foi uma das primeiras implementações da técnica de *Streaming* sobre HTTP[14]. Com a técnica PDS, o cliente baixa o arquivo de vídeo inteiro (de forma sequencial e de um mesmo servidor) com qualidade de vídeo constante, tão rápido quanto a conexão TCP permitir, e inicia a reprodução do vídeo antes que o download seja concluído através da mídia já armezanada em *buffer*.

Entretanto, o PDS apresenta uma desvantagem. Apesar do usário escolher a taxa de bits do vídeo a ser continuamente entregue, a qualidade de vídeo permanece fixa durante toda a transmissão. Isto conduzia a dois problemas. Quando a taxa de transmissão entre cliente e servidor aumentava, o uso de banda disponível era desperdiçada. Por outro lado, quando a taxa de transmissão reduzia, paralisações indesejadas na reprodução seriam causadas. Foi neste contexto e na tentativa de melhorar o uso dos recursos de rede disponíveis para o stream de vídeo que surgiu o *Streaming* Adaptativo sobre HTTP.

2.1.3 Streaming Adaptável sobre HTTP e DASH

Uma evolução do *Streaming* sobre HTTP foi a técnica de *Streaming* Adaptável sobre HTTP (HAS). Este paradigma de entrega de vídeo foi introduzido pela *Move Networks*, em meados de 2005. Naquela época, HAS se tornou popular devido aos custos de implantação mais baratos e por oferecer melhor utilização de recursos de rede em relação à técnica de PDS e outros métodos de *streaming* proprietários [6]. Este cenário viabilizou o desenvolvimento de soluções comerciais como *Microsoft's Smooth Streaming*, *Apple's HTTP Live Streaming* (HLS), *Adobe's HTTP Dynamic Streaming* (HDS) e muitas outras soluções de código aberto. Para evitar a fragmentação do mercado, o *Moving Picture Experts Group* (MPEG) junto com o *3rd Generation Partnership Project* (3GPP) começaram a trabalhar no *streaming* HTTP de mídia MPEG e HAS, respectivamente, cujos esforços acabaram resultando na padronização de *Streaming* Adaptável Dinâmico sobre HTTP (DASH) [23].

Características do MPEG-DASH

No protocolo DASH, a taxa de bits e a resolução do conteúdo de vídeo podem variar dinamicamente para corresponder à largura de banda disponível na rede e à resolução da tela do usuário [6]. Esta é uma característica particularmente relevante, pois na Internet naturalmente há variação de largura de banda (flutuação na taxa). Além disso, clientes usando diferentes tecnologias de acesso à Internet, que possuem taxa de transmissão diferentes, estão a realizar *stream* de vídeo. Logo, se a largura de banda ponta a ponta disponível oscilar durante a sessão de *streaming*, segmentos de video codificados em uma taxa mais próxima da realidade da rede do cliente podem ser escolhidos. Por exemplo, clientes com conexões de fibra óptica, que permitem a transferência de dados em alta velocidade, podem receber uma versão de alta taxa de bits (e de alta qualidade). Por outro lado, clientes com conexões 3G podem receber uma versão de baixa taxa de bits, pois não permitem a taxa de transferência de dados em alta velocidade comparado à de fibra óptica.

Podemos citar que as principais características do protocolo de *streaming* DASH são:

- Lógica de *streaming* implementada no lado do cliente, permitindo maior escalabilidade e flexibilidade.
- Solicitação de taxas de bits de vídeo com base nas condições de rede observadas.
- Adaptação continua à situação de largura de banda do cliente.
- Solicitação e recebimento de dados em termos de segundos/pedaços de vídeo em vez de um fluxo contínuo de pacotes de vídeo.
- Redução de atrasos de inicialização e *buffering* (paradas durante o vídeo).
- Não bloqueio do tráfego em dispositivos de NAT e *firewalls* pelo uso de HTTP.

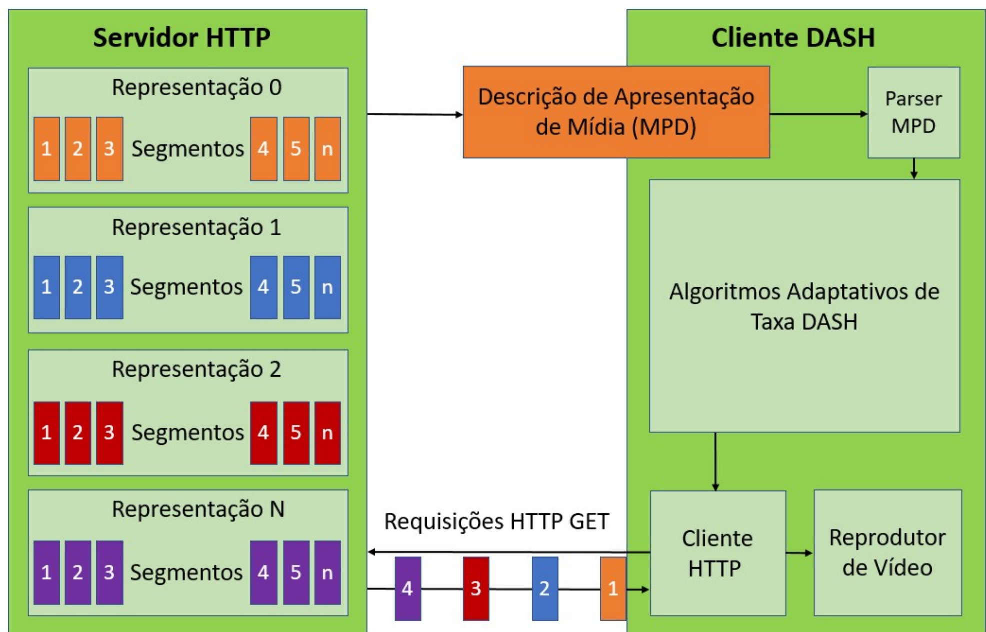

Figura 2.2: Arquitetura cliente-servidor DASH (Fonte: [22]).

A Figura 2.2 ilustra um esquema da arquitetura cliente-servidor DASH. Conforme o exemplo mostrado, o conteúdo de vídeo é codificado em diferentes resoluções ou taxas de bits (representações). Cada versão é fragmentada em pequenos segmentos ou *chunks*, contendo alguns segundos de vídeo cada (por exemplo, 2s, 4s, 6s, etc), e armazenada no servidor HTTP, com cada versão tendo uma URL de *download* específica. O servidor HTTP também tem um arquivo de manifesto denominado Descrição de Apresentação de Mídia (MPD), ilustrado na Figura 2.3. O arquivo MPD é um documento XML que fornece

um índice para os segmentos de mídia disponíveis no servidor. No lado do cliente, o DASH implementa a lógica de adaptação da taxa de bits que emite dinamicamente solicitações, uma de cada vez, e baixa pedaços de segmentos de vídeo de alguns segundos de duração, que são descritos no MPD do servidor usando mensagens HTTP GET [20]. Durante o *download*, o cliente DASH emprega uma lógica de adaptação de taxa ABR para selecionar uma taxa de bits adequada para o próximo segmento a ser buscado. Esse comportamento é chamado de troca de taxa de bits (do Inglês, *bitrate switching*), em que o objetivo do cliente é buscar os segmentos de maior taxa de bits, enquanto mantém dados suficientes no *buffer* de reprodução para evitar travamentos de vídeo e, assim, obter uma boa compensação de Qualidade de Experiência (QoE) [6]. Dessa forma, quando a quantidade de largura de banda disponível é alta, o cliente naturalmente seleciona segmentos de uma versão de maior taxa de bits (alta qualidade); e quando a largura de banda disponível é baixa, ele naturalmente seleciona a partir de uma versão de menor taxa de bits (baixa qualidade).

Diferentemente das soluções proprietárias, o protocolo DASH fornece uma especificação aberta para o cliente DASH. Nele é exigido apenas que a lógica de adaptação (ABR) seja implementada e use solicitações HTTP para buscar os segmentos de vídeo em servidores HTTP. No reprodutor do cliente, os segmentos podem ser decodificados e consumidos, independentemente dos outros segmentos durante a transmissão de dados.

Arquivo de Manifesto (MPD)

O arquivo MPD contém as informações de conteúdo como perfis de vídeo, metadados, codecs, endereços IP de servidor e URLs de *download*. Ou seja, é nele que estão contidas as informações necessárias que descrevem os relacionamentos entre os segmentos e as representações disponíveis.

A Figura 2.3 ilustra um exemplo de arquivo MPD. O MPD consiste em três componentes principais: Períodos, Representações e Segmentos. Os elementos do Período são a parte mais externa do MPD. Os Períodos são normalmente peças maiores de mídia que são reproduzidas sequencialmente. Dentro de um Período, podem ocorrer várias codificações diferentes do conteúdo. Cada alternativa é chamada de Representação. Essas Representações alternativas podem ter, por exemplo, diferentes taxas de bits, taxas de *frames* ou resoluções de vídeo. Finalmente, cada Representação descreve uma série de segmentos por URLs HTTP. Em cada Representação, há um único segmento de inicialização, que contém os dados de configuração, e muitos segmentos de mídia. Concatenando o segmento de inicialização e uma série de segmentos de mídia resulta em um fluxo contínuo de vídeo.

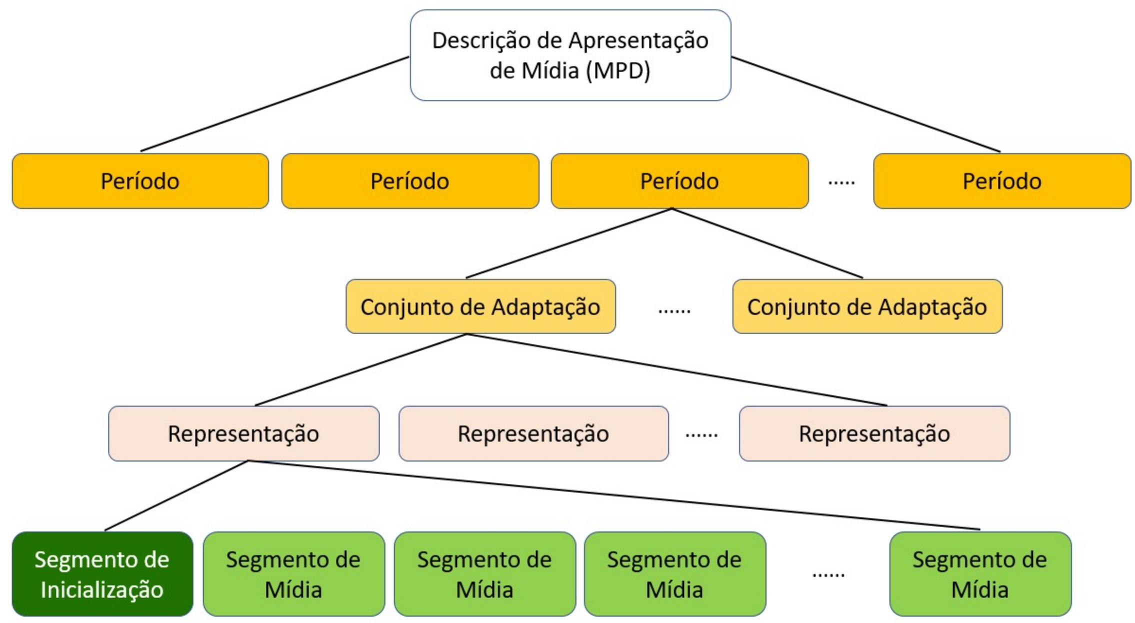

Figura 2.3: Estrutura da Descrição de Apresentação de Mídia (MPD) (Fonte: [6]).

2.2 Algoritmo Adaptativo de Taxa (ABR)

O Algoritmo Adaptativo de Taxa (ABR) é o componente chave de qualquer sistema de *streaming* de vídeo adaptável. Em protocolos que se utilizam do padrão MPEG-DASH, o algoritmo ABR é a solução responsável por escolher dinamicamente, dentro do conjunto discreto de níveis de vídeo disponíveis L no servidor, o nível de vídeo l_i a ser transmitido para cada segmento de vídeo s_i baixado no cliente, dada a largura de banda disponível, com a restrição de evitar interrupções de vídeo e de atingir a melhor qualidade possível, ou seja, com a maior taxa de bits.

De acordo com [22], os algoritmos ABR implementados no cliente podem ser amplamente classificados em três categorias com base nos sinais de *feedback* que usam:

1. *Throughput-based ABR*: A seleção dos segmentos de vídeo subseqüentes depende da largura de banda TCP estimada pela camada de aplicação como sinal de *feedback*. Algoritmos desta classe diferem-se em termos de como estimam a vazão e como eles usam as estimativas. Exemplos de algoritmos desta categoria incluem: [24], [25] e [26].

2. *Buffer-based ABR*: O algoritmo usa informações do *buffer* da aplicação cliente como um critério para selecionar a próxima taxa de bits do segmento durante a reprodução do vídeo. Referências desta categoria podem sem encontradas em [27], [28] e [29].

3. *Hybrid/Control theory based ABR*: Algortimos desta classe usam estimativas de largura de banda disponível, ocupação do *buffer*, tamanho do segmento e/ou duração para selecionar a taxa de representação que melhor satisfaça as preferências de QoE dos usuários. Têm-se como exemplos deste grupo [30] [31] e [32].

Em comum, algoritmos ABR tentam evitar problemas de *streaming* como instabilidade de vídeo, oscilações de qualidade e esvaziamenmto de buffer, enquanto buscam otimizar a experiência do usuário final. Conforme [22] e [6], os objetivos gerais de algoritmos ABR são:

- Maximizar a qualidade do vídeo;
- Evitar ou minimizar interrupções de reprodução de vídeo devido ao esvaziamento de buffer;
- Minimizar o número de mudanças na qualidade do vídeo;
- Minimizar o atraso de incicialização, entendido como o tempo decorrido para iniciar a reprodução do stream de vídeo.

O desafio principal no design de algoritmos ABR está em balancear esses objetivos. Para exemplificar, um algoritmo projetado para maximar a qualidade de vídeo selecionará segmentos com taxa de bits mais altas, porém a probabiliade de elevar o número de interrupções aumenta, uma vez que segmentos com alta taxa de bits levam mais tempo para serem transmitidadas pela rede, causando o esvaziamento de *buffer*. Da mesma forma, é sempre possível minimizar o número de interrupções transmitindo sempre na taxa mais baixa.

2.3 Métricas de Qualidade de Experiência (QoE)

Como outros aplicativos, a entrega de conteúdo multimídia em redes IP tem que lidar com as limitações inerentes de QoS da rede [22]. Limitações de fluxo de dados por recursos de rede podem acontecer em locais como na extremidade do servidor, na interconexão de redes ISP, no link de acesso, ou dentro da rede doméstica. Quando o tráfego de dados encontra alguma destas limitações de largura de banda, naturalmente ocorrem atrasos e perdas de pacotes, o que pode impactar a Qualidade de Experiência (QoE) do usuário.

A QoE, segundo segundo [33], é definida como “o grau de satisfação ou incômdo do usuário de um aplicativo ou serviço. Resulta do cumprimento de suas expectativas com relação à utilidade e/ou aproveitamento do aplicativo ou serviço, à luz da personalidade e do estado atual do usuário”. Portanto, a diferença entre QoS e QoE, é que esta última leva

em consideração, além dos atrasos e perdas de pacote, entre outros, fatores do usuário final, como a expectativa do usuário, grau de satisfação, ambiente em que o usuário está inserido, etc. A Figura 2.4 ilustra este entendimento.

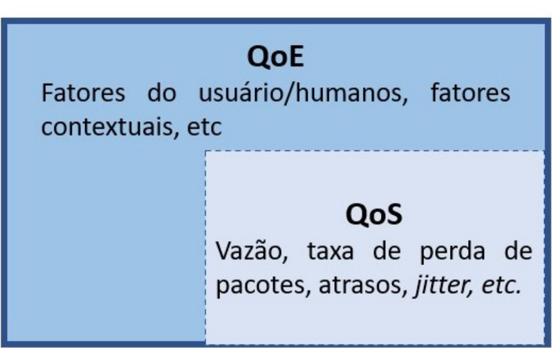

Figura 2.4: Encapsulamento QoS e QoE (Fonte: [22]).

A QoE é particularmente interessante em aplicações de *streaming*. No *Streaming* Adaptável sobre HTTP (HAS), por exemplo, temos diversos interessados em oferecer uma alta QoE aos usuários finais, desde provedores de serviço a desenvolvedores de soluções de algoritmos adaptativos de taxa (ABR).

No que tange à nomenclatura, as funções, ou modelos, de QoE são diferentes de métricas de QoE. A primeira refere-se a fórmulas de auferição do desempenho geral do cliente DASH, que tem como parâmetros de entrada justamente as métricas de QoE. As métricas QoE, por outro lado, são parâmetros "puros" que medem um parâmetro de desemepenho unicamente.

As principais métricas utilizadas na computação de funções de QoE, como taxa média de bits, frequência de *rebuffering*, etc, bem como a forma de obtê-las estão descritas nas próximas subseções. A Tabela 2.1 apresenta os símbolos usados na formulação de métricas de QoE, ou Fatores de Influência (IFs).

Taxa Média de Bits

A taxa média de bits é a codificação média das taxas de bits dos segmentos baixados. Indica a qualidade de áudio/video em bits por segundo. Altas taxas de bits indicam uma boa qualidade e vice-versa. Esta métrica é formulada como sendo:

$$\frac{1}{K}(\sum_{k=1}^{K} r_k) \tag{2.1}$$

Tabela 2.1: Sumário de símbolos usados na computação de métricas de QoE.

Símbolo	Descrição
k	Número de segmento
K	Número total de segmentos baixados
r_k	Taxa de bits do segmento k (bits/s)
sr_k	Tamanho do segmento k (bits)
V_k	Velocidade média de download estimada durante o processo de download do segmento k (bits/s)
$\frac{sr_k}{V_k}$	Tempo gasto para baixar o segmento k (s)
N_{vq}	Número de variações de qualidade
N_p	Número de pausas
l	Duração total do vídeo (s)
B_{init}	Nível de ocupação do *buffer* necessário para iniciar a reprodução de vídeo (s)
B_k	Nível de ocupação de *buffer* quando se começa a baixar o segmento k (s)
T_s	Tempo gasto para ocupar o B_{init} durante a primeira reprodução de vídeo (s)

Frequência da Variação de Qualidade

A frequência da variação de qualidade indica a frequência (número de vezes) que mudanças na qualidade ocorreram ao longo da sessão de *streaming*. Mudanças de qualidade muito frequentes são negativas ao usuário final. Esta métrica é formulada como sendo:

$$\frac{N_{vq}}{l} \tag{2.2}$$

Amplitude da Variação de Qualidade

A amplitude da variação de qualidade indica o "espaçamento" entre os níveis de mudança de qualidade. Em geral, variações para qualidades inferiores, com mudança de qualidade de magnitudes mais baixas, ou seja, em etapas graduais (alto para médio ou médio para baixo) é considerado menos incômodo do que o de altas magnitudes (alto para baixo abruptamente) [34].Esta métrica é formulada como sendo:

$$\sum_{k=1}^{K-1} |r_{k+1} - r_k| \tag{2.3}$$

Duração de *Rebuffering*

A duração de *rebuffering* indica o tempo em que o *buffer* de reprodução esteve vazio, ou seja, sem segmentos a serem tocados. É conhecido também por indicar a duração das

pausas. Normalmente, longas pausas levam ao incõmodo do usuário final e à degradação da QoE. Esta métrica é formulada como sendo:

$$\sum_{k=1}^{K}(\frac{sr_k}{V_k} - B_k) \tag{2.4}$$

Frequência de *Rebuffering*

A frequência de *rebuffering* indica a frequência (número de vezes) que ocorreu o esvaziamento do *buffer*, ou seja, a interrupção do vídeo. Interrupções muito frequentes são consideradas incômodas e podem resultar em uma experiência negativa para o usuário final. Esta métrica é formulada como sendo:

$$\frac{N_p}{l} \tag{2.5}$$

Atraso de Inicialização

O atraso de inicialização é o tempo usado pela aplicação para armazenar alguns bits de vídeo. Isto tem como propósito minimizar penalidades relacionados ao *rebuffering*. O atraso de inicialização, ou atraso de carregamento inicial está geralmente presente em todos os aplicativos de *streaming*. Esta métrica é formulada como sendo:

$$T_s >= B_{init} \tag{2.6}$$

Otimizar a QoE do DASH é uma meta desafiadora. Como explicado na seção anterior, essas métricas são frequentemente interdependentes, tendo relacionamentos complexos e contra-intuitivos [22].

Este capítulo introduziu a base teórica necessária para compreender a visão geral dos protocolos de *streaming* de vídeo, em especial o DASH. Também vimos que o principal componente de um cliente DASH são os algoritmos adapatativos de taxa (ABR), os quais são desenvolvidos com as mais variadas abordagens. Além disso, fornecemos a fundamentação teórica de métricas de QoE e sua importância na avaliação de desempenho em aplicações HAS. No próximo capítulo, são demonstrados trabalhos desenvolvidos pela comunidade acadêmica que tratam da avaliação de algoritmos ABR em protocolos de *streaming* HAS, com ou sem a computação de métricas de QoE.

Capítulo 3

Trabalhos Relacionados

Neste capítulo são apresentados algumas das implementações de aplicações HAS existentes na litetarura. Um dos principais objetivos de uma aplicação HAS projetado para avaliação de algoritmos adapatativos é permitir desenvolvê-los e/ou avaliá-los sob diversas configurações de vídeo, rede, etc. Novas soluções ABR capazes de selecionar dinamicamente segmentos de vídeo em uma taxa de bits que se ajuste à largura de banda disponível, evitando pausas de reprodução e maximizando as métricas de qualidade de experiência do usuário (QoE) são concebidas a todo momento. Nesse sentido, várias ferramentas foram propostas pela comunidade acadêmica para a avaliação de desemepenho utilizando diferentes abordagens. *Frameworks* que realizam uma implementação completa do protocolo DASH são classificados como efetivos DASH *players*, pois possuem todas as características de reprodutores desenvolvidos com a finalidade de realizarem o *streaming* de conteúdo DASH. Entre os principais DASH *players* está o dash.js [11], GPAC [12] e ExoPlayer [13], sendo estes dois últimos não tão explorados pela literatura; e por isso não fazem parte de objeto de estudo. Serão analisados, também, ferramentas de emuladores de clientes DASH, bem como abordagens simulativas de aplicações de HAS. A penúltima seção 3.9 aborda a ferramenta *pyDash* em maiores detalhes. Por fim, na seção 3.10 é feita uma análise comparativa entre os trabalhos citados neste capítulo e a plataforma *pyDash*, confrontando diferenças e semelhanças.

3.1 Dash.js

O dash.js [1] é o reprodutor de mídia referência do padrão MPEG-DASH mantido pelo DASH *Industry Forum* (DASH-IF) [11], um consórcio que inclui a maioria dos principais participantes da indústria de streaming de vídeo e que conta com mais de 60 membros. O *player* dash.js é escrito em JavaScript e executado dentro de um navegador web. Os

[1] https://github.com/Dash-Industry-Forum/dash.js?

principais objetivos deste projeto incluem: construir uma biblioteca JavaScript de código aberto para a reprodução de diversas mídias MPEG-DASH; dispor dos algoritmos de adaptação de melhor desempenho; ser gratuito para uso comercial e independente de codec e/ou navegador; etc. Por ser *open-source*, todo o código no projeto dash.js está disponível para criação de clientes DASH gratuitamente para fins pessoais ou comerciais. Além disso, o projeto também está disponível para desenvolvedores que queiram submeter melhorias ou estender o código e contribuir de volta para este projeto.

A Figura 3.1 apresenta a tela da aplicação cliente DASH[2]. No exemplo mostrado, a aplicação cliente está na versão 4.0.1. Para iniciar o *streaming* é preciso configurar o URL do servidor DASH, cujo arquivo MPD está hospedado. Depois de iniciado, o vídeo é exibido com informações de qualidade e tempo de *streaming*. Além disso, é possível configurar a exibição de gráficos dinâmicos através do painel lateral direito e acompanhar a evolução de parâmetros de vídeo como, por exemplo, o tamanho do *buffer* e a taxa de bits selecionada, dependendo de sua ativação. Nesse sentido, dash.js possui recursos tanto visuais quanto gráficos para avaliar o desempenho do algoritmo ABR. Para um cenário educacional e de estudo, são recursos que auxiliam observar a dinâmica do funcionamento de algoritmos adaptativos.

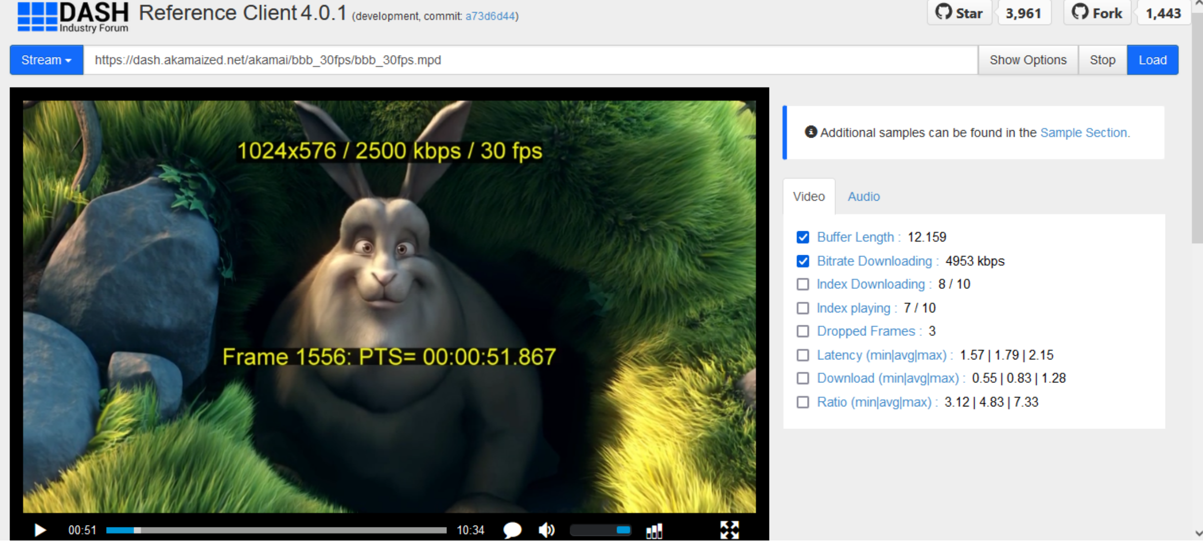

Figura 3.1: Captura de tela da aplicação cliente DASH da ferramenta dash.js.

[2]https://reference.dashif.org/dash.js/latest/samples/dash-if-reference-player/index.html

3.2 TAPAS

Consolidada na literatura sobre o estudo de desempenho e avaliação de algoritmos adaptativos de streaming de video (ABR), TAPAS [3] [14], *a Tool for rApid Prototyping Adaptive Streaming controllers*, é uma ferramenta uma flexível, escrita em Python e que facilita a implementação de novos algoritmos. Como objetivo principal, permite que pesquisadores se concetrem unicamente na elaboração de lógicas ABR e que sejam capazes de coduzir experimentos de avaliação de performance, sem a necessidade de desenvolver um ambiente de reprodução de vídeo completo. Outro ponto positivo é que durante os experimentos o usuário tem a opção de habilitar ou não o modo de *decode/display* do vídeo, recurso este implementado com o *framework GStreamer* [35].

Ainda, segundo os autores, a ferramenta tinha como finalidade minimizar o consumo de CPU e memória, possibilitando a execução de um amplo número de sessões de streaming concorrentes em uma mesma máquina. Na figura Figura 3.2, é possível observar o fluxo de dados e a arquitetura básica do *player*. TAPAS é implementado pela agregação de três componentes principais:

1. *Controller*: módulo onde é implementada a lógica de seleção do próximo segmento a ser baixado;

2. *Parser*: módulo de análise do arquivo de descrição da apresentação de mídia (MPD); e

3. *MediaEngine*: módulo implemetando com *GStreamer* e responsábel pelo armazenamento do segmento de vídeo baixado na fila de buffer e reproduzi-lo.

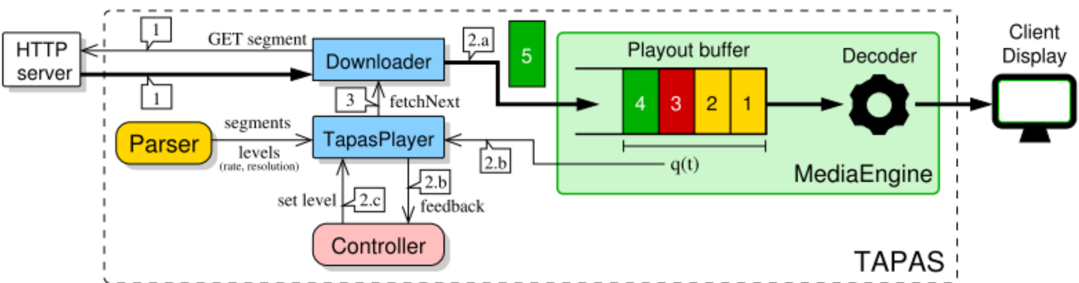

Figura 3.2: Fluxo de dados e módulos da ferramenta TAPAS (Fonte: [14]).

[3] https://github.com/ldecicco/tapas

3.3 AStream

A ferramenta AStream[4] é uma plataforma baseada em cliente DASH, a qual foi desenvolvida para avaliar a performance de algortimos ABR, especialmente do algoritmo adaptativo *Segment Aware Rate Adaptation* (SARA), proposto em [15]. Semelhante ao TAPAS, também é escrito em Python e implementado por agregação de módulos, os quais são:

1. *MPD parser*: módulo que realiza o *download* e análise do arquivo de manifesto (MPD) para obter informações do segmento (taxas de bits disponíveis, URLs de segmento e tamanhos de segmento)responsável pela seleção do nível de vídeo do próximo segmento a ser baixado;
2. *Segment download*: módulo que realiza o download de segmentos do servidor DASH;
3. *Video buffer*: módulo que emula o buffer do player de vídeo; e
4. *Rate adaption*: módulo usado para determinar a taxa de representação do download do próximo segmento.

Apesar de denominações diferentes, tanto os módulos da ferramenta TAPAS quanto do AStream desempenham funções idênticas. O módulo *rate adaption*, por exemplo, possui o mesmo papel do componete *controller* de [14]. Logo, é possível observar que estas ferramentas possuem módulos essenciais para a implementação de um emulador de cliente DASH.

3.4 *End-to-End DASH Platform*

O trabalho desenvolvido em [9] apresenta uma plataforma DASH de ponta a ponta, ou seja, desde o lado do servidor até o lado do cliente. O componente principal desta plataforma é o cliente DASH, implementado completamente com o *framework GStreamer* [35], que é escrito em C, e com a utilização de Python *bindings*, bibliotecas que unem duas linguagens de programação, de forma que uma biblioteca escrita para uma linguagem possa ser usada em outra. Além disso, a plataforma foi projetada especialmente para a validação de um algoritmo de adaptação de vídeo, também proposto no trabalho [9].

O grande destaque desse cliente DASH está na inclusão de Interfaces Gráficas de Usuário (GUI), conforme a Figura 3.3. Nela é possível visualizar a reprodução do segmento de vídeo baixado, inserir parâmetros de configuração e visualizar a dinâmica de seleção de segmentos atráves de gráficos dinâmicos. A plataforma utiliza componentes *open-source*, porém detalhes de seu código-fonte está restrita aos desenvolvedores do trabalho.

[4] https://github.com/pari685/AStream

Do ponto de vista educacional, está plataforma apresenta recursos interessantes como a exibição do vídeo e de estátisticas de desempenho em tempo real.

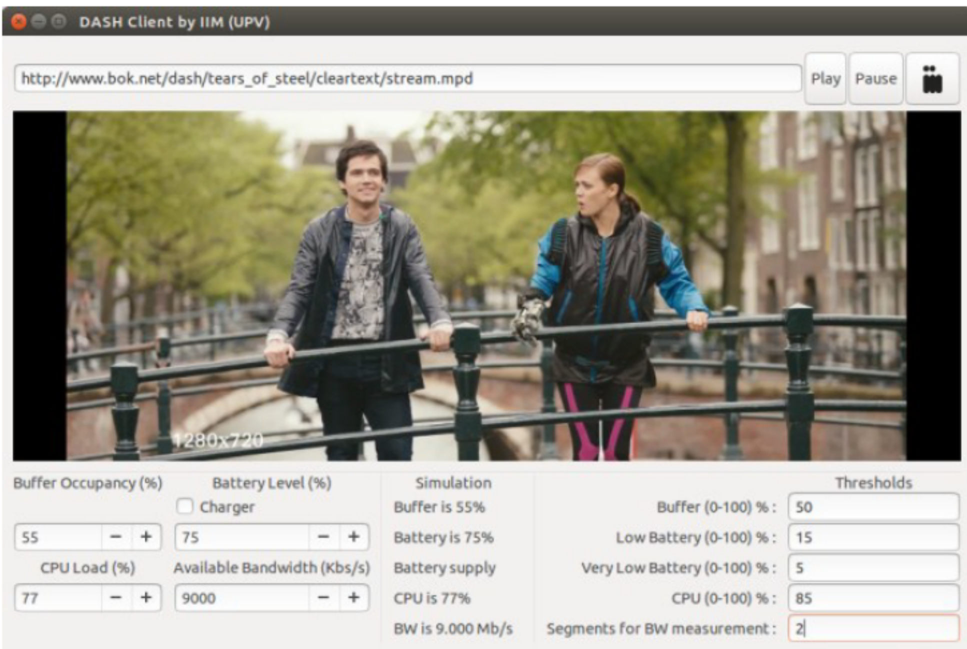

Figura 3.3: Captura de tela do cliente DASH da plataforma (Fonte: [9]).

3.5 Dashc

A ferramenta dashc [5] [16] é um cliente DASH com foco em fornecer um *player* de baixo consumo de memória (RAM) e CPU, além de ser ter sido projetado para possibilitar experimentos em larga escala, ou seja, com múltiplos clientes sendo executados em uma mesma máquina. Para suportar o aspecto de escalabilidade, o reprodutor foi escrito em OCaml [6], uma linguagem de programação de força industrial que suporta estilos funcionais, imperativos e orientados a objetos, o que explica em parte a baixa utilização de recursos computacionais - sendo esta, inclusive, inferior ao da ferramenta TAPAS [14]. Além disso, possui 3 partes lógicas principais:

1. *Buffer emulator*: acompanha a fila/reprodução de buffer disponível, diminuindo-o constantemente como no player de vídeo real e aumentando-o quando novos segmentos chegam;

2. *Adapation algorithm*: toma uma decisão sobre a qualidade do próximo segmento solicitado;

[5]https://github.com/uccmisl/dashc
[6]https://ocaml.org/

3. *Logging system*: grava informações sobre o desempenho de rede e do algoritmo de adaptação em um arquivo.

Um ponto de destaque nesta ferramenta é seu sistema de log, que foi implementado de forma a registrar o máximo de informações potencialmente úteis quanto possível para análise. Esta característica axilia na computação de métricas de QoE para posterior análise de desempenho do algoritmo adaptativo.

3.6 GoDASH

Com a proposta similar ao das ferramentas [14], [15] e [16], goDASH [7] [17] é uma plataforma para o desenvolvimento, avaliação e implementação de novos modelos e técnicas de algoritmos adaptativos de streaming de conteúdo de video adaptativo sobre HTTP (DASH). A plataforma é escrita na linguagem golang [8] do Google, com o objetivo de simplificar a implementação de novas estratégias de adaptação. Segundo os autores, uma das principais funcionalidades é capacidade de transmitir conteúdo DASH sem decodificar o vídeo real, o que resulta em baixo consumo de memória e na capacidade de executar múltiplos *players* em uma mesma máquina. Além disso, goDASH tem suporte para diferentes arquivos de manifesto (MPD), suporte para o protocolo QUIC e facilidade de implementar novas funcionlaidades, porém demanda o prévio conhecimento técnico para instalar e utilizar outras ferramentas incorporadas na plataforma, como por exemplo o Mininet [36], ferramenta de emulação de rede.

Apesar de não ser explicitado a implementação de sua arquitetura, goDASH é uma ferramenta modular, dinâmica e de simples configuração. A ferramenta provê uma descrição detalhada de como preparar o setup do player, e fornece um arquivo de configuração facilmente modificável no formato JavaScript Object Notation (JSON) para a inicialização do cliente DASH.

Também vale a pena destacar o sistema de log implementado no cliente. A partir do arquivo de configuração, é possível habilitar a impressão da saída de log no terminal e/ou em um arquivo *.txt* ou *.csv*. Algumas métricas de QoE também são computadas nesta ferramenta, dentre elas o tempo de recebimento, duração de pausa, nível de *buffer*, qualidade selecionada para cada segmento, muito influenciado pelo sistema de log do *dashc* [16],

[7] https://github.com/uccmisl/godash
[8] https://golang.org/

3.7 Framework Simulativo para Aplicações de Streaming Adaptável sobre HTTP

Ott, Miller e Wolisz [18] propuseram um modelo de simulação para uma aplicação HAS. Segundo os autores, o modelo foi projetado com dois objetivos principais em mente. O primeiro foi de desenvolver uma plataforma com estrutura modularizada com blocos funcionais separados logicamente. O segundo foi de manter a complexidade do modelo o mais baixo possível, implicando em Interfaces de Programação de Aplicativos (do Inglês, *Application Programming Interface* - APIs) limpas e simples. Além disso, foi projetado de forma que os usuários não tenham que modificar o código para integrar seus próprios algoritmos de adaptação. O *framework* consiste em três blocos funcionais principais:

1. *Client module*: coleta e mantém as informações requisitadas pelo algoritmo de adaptação, como o nível buffer, largura de banda disponível e metadados sobre o vídeo;
2. *Adaptation algorithm module*: implementa o algoritmo de adaptação de vídeo; e
3. *Server module*: gerencia os clientes conectados e atende às suas solicitações

Todos esses módulos são integrados ao simulador *ns-3* [37], o qual suporta muitas configurações de redes, tanto sem fio quanto com fio. Contudo, uma vez que nenhuma reprodução é realizada durante a simulação, nenhum *streaming* de conteúdo de vídeo real é realizada, o que, por um lado, ajuda a lidar com problemas de escalabilidade, por outro, inibe assistir às qualidades de vídeo selecionadas, pausas, etc. A característica de exibição de vídeo é algo que favorece a análise de desempenho e contribui para observar os impactos de mudança de qualidade, característica inerente de aplicações DASH.

3.8 Sabre

Sabre [19] é uma ferramenta de simulação com a proposta de facilitar a prototipação inicial de algoritmos ABR e rapidamente avaliá-los em um ambiente semelhante ao de *players* reais de produção. A ferramenta é escrita em Python, e foi concebida a abstrair os detalhes de implementação de reprodutores de vídeo, que em muitas vezes são complexas, possibilitando que pesquisadores da área foquem na concepção de lógicas HAS. A Figura 3.4 ilustra a arquitetura da ferramenta Sabre, a qual possui os seguintes componentes:

1. *Video Adapter*: lê um arquivo manifesto (do Inglês, *Manifest file* - MPD) simplificado para o vídeo que está sendo simulado;

2. *Schedule Controller*: gerencia a tarefa de download e escolha de *bitrate* do próximo segmento e possibilita a implementação de algoritmo de substituição de segmento [19];

3. *Network Model*: emula as condições de rede a partir de um arquivo de configuração de rede (do Inglês, *network trace file*); e

4. *Buffer Model*: emula o enfileiramento de segmentos na fila de buffer e sua reprodução.

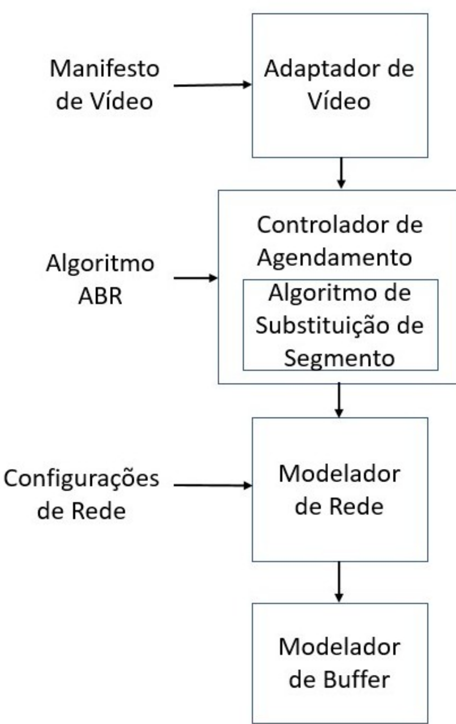

Figura 3.4: Visão geral da arquitetura Sabre (Fonte: [19]).

Uma peculiariedade, entretanto, é que esta ferramenta necessita como entrada uma especificação de vídeo fornecida como um arquivo no formato JavaScript Object Notation (JSON). Neste arquivo, está contemplado a duração do segmento, a lista de *bitrates* e o tamanho de cada segmento de vídeo disponível. Logo, ao contrário de *players* baseados em emulação, os quais utilizam *datasets* de vídeo projetados para experimentos DASH, Sabre simula os segmentos de vídeo. Sem o uso de vídeo reais, não é possível realizar a exibição de vídeo, recurso que pode melhorar o estudo e análise de desempenho de algorimtos adaptativos em ambientes educacionais.

3.9 PyDash

Desenvolvido no âmbito da Univerdade de Brasília (UnB) por professores do Departamento de Ciência da Computação (CIC), a plataforma *pyDash*[9] [1] é uma ferramenta educacional baseada em *framework* para o estudo de algoritmos adaptativos de *streaming* de vídeo. A ferramenta *pyDash* permite a implementação e a avaliação de algoritmos ABR. Para facilitar a aprendizagem e a rápida implementação de novos algoritmos, a arquitetura *pyDash* foi projetada de forma a abstrair os detalhes da comunicação da rede, permitindo que os alunos se concentrem exclusivamente no desenvolvimento e avaliação de lógicas ABR.

3.9.1 Arquitetura

A plataforma *pyDash* é implementada usando apenas dois hosts: a primeira máquina executa o cliente DASH (`DashClient`) e a segunda executa um servidor HTTP padrão. É importante ressaltar que o cliente da plataforma *pyDash* pode ser utilizado em qualquer outro cenário desde que os vídeos sejam disponibilizados através de um servidor HTTP seguindo o formato MPEG-DASH. Além disso um modelador de largura de banda é empregado no lado do cliente para emular cenários de restrição de taxa de transferência.

Lado Servidor

O servidor HTTP da plataforma *pyDash* consiste na hospedagem de vídeos seguindo o formato MPEG-DASH. A Figura 3.5 apresenta o *dataset* armazenado no servidor, que está hospedado no seguinte endereço:

- `http://45.171.101.167/DASHDataset/`

Todos os vídeos publicados no servidor HTTP são codificados no *codec* H.264, em diferentes tamanhos de segmentos e representações, que são encapsuladas no formato de mídia `.m4s`. É importante destacar que os arquivos de mídia são codificados sem áudio. No exemplo do filme *Big Buck Bunny*, conforme observado na Figura 3.6, o mesmo vídeo está segmentado em tamanhos fixos diferentes. Para cada tamanho de segmento de vídeo, verifica-se o arquivo MPD e 20 qualidades diferentes em que o vídeo é codificado, sendo a menor qualidade de 46980 bps e a maior 4726737 bps. Por fim, para cada representação, há um único segmento de inicialização que contém os dados de configuração e muitos segmentos de mídia. Para o video *BigBuckBunny*, com tamanhos de segmento de 1 segundo, por exemplo, temos 1 arquivo de inicialização e 596 arquivos que compõem o video original.

[9] `https://github.com/mfcaetano/pydash`

Index of /DASHDataset

Name	Last modified	Size	Description
Parent Directory		-	
BigBuckBunny/	2021-03-16 20:15	-	
ElephantsDream/	2021-03-16 20:15	-	
OfForestAndMen/	2021-03-16 20:15	-	
RedBullPlayStreets/	2021-03-16 20:15	-	
TearsOfSteel/	2021-03-16 20:15	-	
TheSwissAccount/	2021-03-16 20:15	-	
Valkaama/	2021-03-16 20:15	-	

Apache/2.4.41 (Ubuntu) Server at 45.171.101.167 Port 80

Figura 3.5: DASH *Dataset* utilizado na plataforma *pyDash*.

Index of /DASHDataset/BigBuckBunny

Name	Last modified	Size	Description
Parent Directory		-	
1sec/	2021-03-16 20:15	-	
2sec/	2021-03-16 20:15	-	
4sec/	2021-03-16 20:15	-	
6sec/	2021-03-16 20:15	-	
10sec/	2021-03-16 20:15	-	
15sec/	2021-03-16 20:15	-	

Apache/2.4.41 (Ubuntu) Server at 45.171.101.167 Port 80

Figura 3.6: Vídeo *BigBuckBunny* dividido em segmentos de tamanhos fixos.

Lado Cliente

O cliente da arquitetura *pyDash* é uma ferramenta *open-source* escrita em Python[10]. Sua estrutura foi concebida a simplificar o projeto e a experimentação de algoritmos de *streaming* adaptativos (ABR). A grande vantagem da ferramenta é sua flexibilidade: novos algoritmos ABR podem ser implementados no cliente através da herança de classe base, como será mostrado a seguir. Dessa forma, o aluno pode se concentrar apenas no projeto do algoritmo ABR, aproveitando todos os outros componentes implementados.

Conforme ilustrado na Figura 3.7, a classe `DashClient` é implementada pela agregação de três componentes principais de interação, organizados em uma estrutura de pilha: 1) `Player`, o qual implementa estruturas de armazenamento da representação do vídeo baixado no *buffer*, simula a reprodução do vídeo e gera estatísticas; 2) `IR2A` (*Interface of*

[10] https://www.python.org/

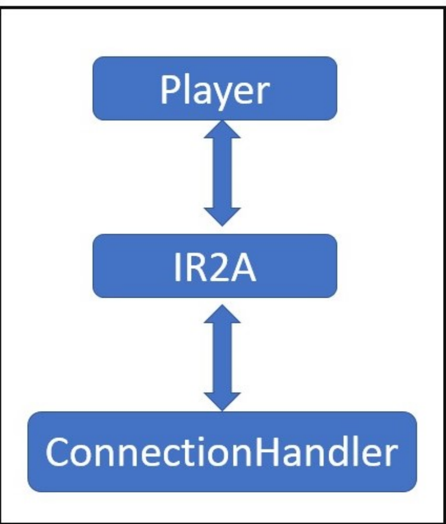

Figura 3.7: Composição do cliente DASH da plataforma *pyDash*.

Rate Adaptation Algorithms), o qual seleciona o nível de vídeo do próximo segmento a ser baixado; 3) `ConnectionHandler`, o qual realiza uma conexão no servidor HTTP e implementa o algoritmo de *traffic shaping* (modelador de largura de banda). Cada um desses três componentes são classes implementadas que operam na classe `DashClient`. Ademais, é importante mencionar que a ferramenta inclui outros componentes que auxiliam no correto funcionamento do *pyDash*.

Componentes do Cliente DASH

A seguir, são descritas as características essenciais dos três componentes do `DashClient`.

1. `Player` - A classe `Player` implementa o comportamento do *player* de vídeo. Nela está implementada uma estrutura de dados que emula o comportamento de enfileiramento e consumo do *buffer*. A classe `Player`, pois, monitora a dinâmica de ocupação e esvaziamento do *buffer* durante a execução do video. Esta classe monitora, também, diversas métricas de desempenho, como exemplo: o número de pausas, o tamanho das pausas, as qualidades selecionadas, etc. Além disso é nesta classe que dado início ao processo de requisição tanto do arquivo descritor MPD, quanto dos segmentos de vídeo. Ele monta requisições através de mensagens, meio pelo qual as classes se comunicam.
2. `IR2A` (*Interface of Rate Adaptation Algorithms*) - Classe base que deve ser herdada pela classe do novo algoritmo ABR a ser implementado. Além disso, define métodos

abstratos que devem ser obrigatoriamente implementados para que o algoritmo possa ser validado. Após ser carregado no arquivo *dash_client.json*, o algoritmo ABR se localizará nesta camada, fazendo a intermediação entre as mensagens que descem (*resquest*) e sobem (*response*) na pilha.

3. `ConnectionHandler` - A classe `ConnectionHandler` é responsável por montar requisições HTTP GET. Ou seja, é ela que faz a comunicação entre a plataforma *pyDash* e o servidor HTTP definido no arquivo *dash_client.json*. Esta classe, também, obtém as mensagens de resposta tanto do arquivo MPD quanto dos segmentos de vídeo. Além disso, é nela que está contido a implementação do algoritmo de *traffic shaping* (modelador de largura de banda) destinado ao controle da banda e consequente teste do protocolo ABR implementado.

Arquivo de configuração JSON

O cliente DASH permite a configuração de parâmetros através do arquivo *dash_client.json*. Na Listagem 3.1 é possível visualizar um exemplo do arquivo JSON de configuração da plataforma, enquanto na Tabela 3.1 é contido a descrição de cada um dos parâmetros fornecidos.

```
{
    "buffering_until": 5,
    "max_buffer_size": 60,
    "playbak_step": 1,
    "traffic_shaping_profile_interval": "5",
    "traffic_shaping_profile_sequence": "LMH",
    "traffic_shaping_seed": "1",
    "url_mpd": "http://45.171.101.167/DASHDataset/BigBuckBunny/1sec/
BigBuckBunny_1s_simple_2014_05_09.mpd",
    "r2a_algorithm": "R2AFixed"
}
```

Listing 3.1: Arquivo de configuração *dash_client.json*

3.9.2 Funcionamento

Ao iniciar uma sessão de *streaming*, o cliente solicita o arquivo MPD do servidor HTTP e, em seguida, começa a solicitar segmentos de vídeo (normalmente em ordem sequencial) o mais rápido possível para preencher o *buffer* de reprodução. Uma vez que este *buffer* está cheio, ou seja, alcança um limite pré configurado, o cliente simula a reprodução do vídeo e continua a fazer o *download* de novos segmentos periodicamente de acordo com seu algoritmo ABR escolhido. A Figura 3.7 apresenta em maiores detalhes os módulos

Tabela 3.1: Parâmetros do arquivo de configuração *dash_client.json*.

Parâmetro	**Descrição**
buffering_until	tamanho inicial do *buffer* (em segundos) antes do vídeo começar a ser reproduzido
max_buffer_size	tamanho máximo do *buffer* (em segundos)
traffic_shaping_profile _interval	tempo (em segundos) que o perfil corrente de *Traffic Shaping* (TF) será aplicado antes de ser modificado
traffic_shaping_profile _sequence	valor real de banda que será aplicado a restrição de TF. Este parâmetro aceita uma sequência de letras: "L", "M" e "H" para indicar perfil com menor, média e maior restrição de banda, respectivamente, podendo apresentar sequência e quantidade variável
traffic_shaping_seed	semente utilizada pela Distribuição de Probabilidade Exponencial na geração da sequência de larguras de banda aplicada
url_mpd	a URL completa para o arquivo MPD
r2a_algorithm	nome da classe referente ao algoritmo ABR que será carregada

que formam a classe `DashClient`, bem como o seu modo de operação (*workflow*). De modo geral, quando é iniciada a execução da plataforma *pyDash* (sessão de *streaming*), primeiramente é feito a requisição do arquivo de manifesto (MPD) e obtido as informações de vídeo através de um MPD *parser*. De posse das informações de vídeo, são feitas as requisições de segmentos de vídeo do servidor HTTP. Ao completar o *download* de todos os *chunks* de vídeo, os módulos são finalizados e a sessão de *streaming* termina.

O cliente DASH consiste em três camadas: `Player`, `IR2A` e `ConnectionHandler`. Como mostrado na Figura 3.7, no topo da pilha está a classe `Player`, em seguida a classe `IR2A` e, por último, a classe `ConnectionHandler`. Fazendo uma analogia, a arquitetura do cliente funciona de forma semelhante à pilha de protocolo da Internet (TCP/IP). Ou seja, as camadas operam de forma hierárquica e se comunicam através de mensagens. No *pyDash*, as mensagens de requisições fluem da camada mais alta (`Player`) até a camada mais baixa (`ConnectionHandler`), através da função `self.send_down`. Alternativamente, as mensagens de resposta fluem da camada mais baixa até a camada mais alta, através da função `self.send_up`.

As mensagens que fluem entre as camadas podem ser de dois tipos: `base.Message` ou `base.SSMessage`. Mensagens do tipo `base.Message` são usadas para requisições e respostas do arquivo MPD. Por outro lado, mensagens do tipo `base.SSMessage` são usadas para requisições e repostas de segmentos de vídeo.

A Figura 3.8 ilustra o fluxo de mensagens do funcionamento do *pyDash*. Ao iniciar a execução da plataforma *pyDash*, a classe `DashClient` instancia e inicializa os três móludos.

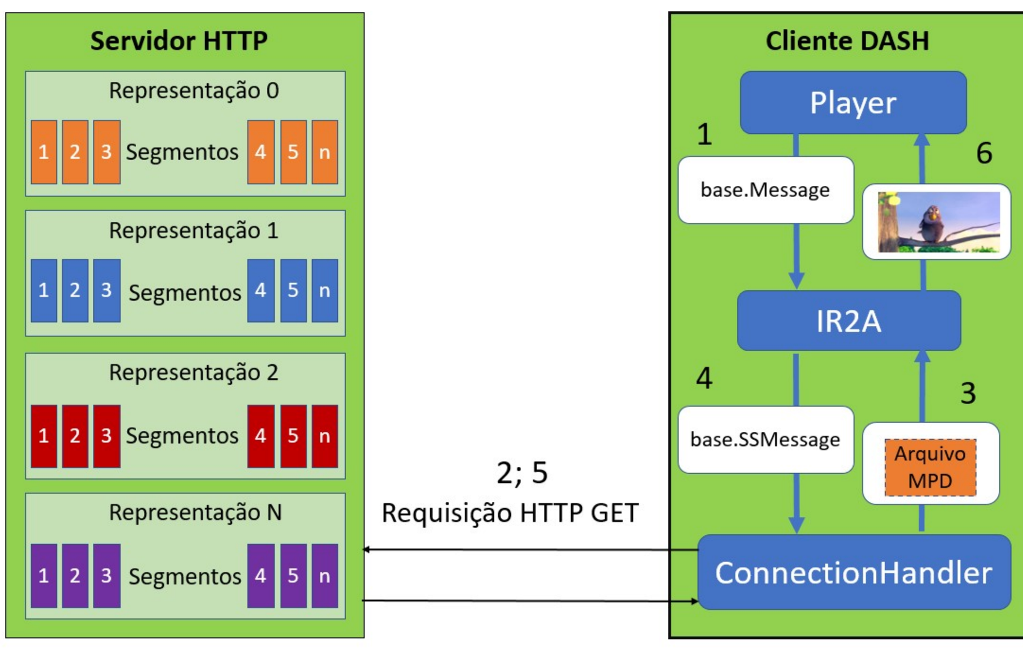

Figura 3.8: Fluxo de mensagens do funcionamento do *pyDash*.

Em seguida, é começado o processo de montagem da requisição do arquivo de manifesto MPD, indicado pelo momento de número um (1) da figura. A montagem de requisição tem início na classe `Player`, utilizando uma mensagem (`msg`) do tipo `base.Message`. O URL para a requisição do MPD é obtido do campo `url_mpd` inserido no arquivo de configuração *dash_client.json*. Completado sua montagem, a mensagem (`msg`) é encaminhada para a camada de baixo `IR2A`. Nesta camada, está carregado o algoritmo ABR, indicado no campo `r2a_algorithm` do arquivo de configuração JSON. A classe do algoritmo inserido, então, recebe essa requisição através do método sobrescrito `handle_xml_request`. Neste método, o algoritmo tem a oportunidade de alterar ou não a mensagem de requisição do arquivo MPD. Em seguida, a mensagem `msg` é encaminhada novamente para a camada de baixo `ConnectionHandler`. A partir deste ponto, a mensagem é lida e convertida em uma requisição HTTP, indicada pelo momento de número dois (2) da figura.

Ao obter sucesso na requisição do arquivo de manifesto (MPD) com o servidor HTTP, o MPD será efetivamente recebido na classe `ConnectionHandler`. Devidamente encapsulado em uma mensagem `msg` do tipo `base.Message`, indicado pelo momento três (3), a classe `ConnectionHandler`, então, encaminhará à camada superior o conteúdo do arquivo MPD. É importante destacar que, quando o cliente DASH faz o *donwload* do manifesto (MPD), uma lista `qi` (*quality index*) contendo todas as qualidades em que o vídeo foi codificado é populado. O `qi[0]`, por exemplo, indica a menor qualidade. Esta operação é possível devido ao *parser* MPD disponibilizado na plataforma, implementado pela classe

`pydash.player.mpd_node`. Na camada `IR2A`, a classe do algoritmo ABR recebe a mensagem através do método sobrescrito `handle_xml_response`. Neste método, o algoritmo tem, novamente, a oportuinidade de executar alguma intrução, como por exemplo, extrair a lista de qualidades possíveis do arquivo MPD. Por fim, continuando o processo de subida da mensagem `msg`, esta chega na camada superior `Player`.

A partir de então é iniciada o processo de requisição de segmentos do servidor HTTP na qualidade selecionada. De posse das informações do arquivo MPD, a classe `Player` dará início à montagem da mensagem de requisição `msg`. Esta mensagem conterá, inicialmente, informações sobre o próximo segmento a ser recuperado. O processo de requisição dos segmentos de vídeo (`ss`) segue da mesma forma como aconteceu com a requisição ao arquivo MPD. Dessa vez, entretanto, mensagens `msg` do tipo `base.SSMessage` são montadas, indicadas pelo momento quatro (4). Após montada a mensagem, esta será encaminhada à camada subsequente `IR2A`, onde será recebida pelo método sobrescrito `handle_segment_size_request`. Neste método, o algoritmo ABR definirá qual a taxa de bits do segmento que será recuperada na requisição HTTP. A escolha é feita através do método `msg.add_quality_id(x)`, onde `x` é um valor inteiro positivo que representa a qualidade em *bps*. Por fim, a mensagem `msg` será encaminhada à camada inferior. Na classe `ConnectionHandler`, a mensagem `msg` será convertida em uma requisição HTTP, indicada pelo momento cinco (5).

Quando o *download* do segmento `ss` é completado, a mídia é inicialmente recebida na classe `ConnectionHandler`. Novamente, a classe `ConnectionHandler` iniciará o processo de subida na pilha desta informação, de forma que o segmento `ss` chegue até a classe `Player`, indicado no momento seis (6). Na classe `Player`, o segmento é enfileirado no *buffer* de reprodução. No caso do *pyDash*, o *buffer* de reprodução não armazena os segmentos de vídeo, mas é implementado como uma estrutura de dados, que mantém o comprimento do *buffer* medido em segundos. Para obter a dinâmica do *buffer* de reprodução, duas ações são emuladas: o preenchimento e o consumo do *buffer*. O preenchimento do *buffer* acontece quando um novo segmento é baixado, enquanto o consumo do *buffer* é continuamente esvaziado durante a reprodução do vídeo. Na intenção de emular a ação de consumo do *buffer* de reprodução, é empregado uma *thread* (`handle_video_playback`) que a cada T segundos diminui o tamanho do *buffer* em T segundos. Por padrão, $T =$ 1s. A *thread* de consumo do *buffer* encerra quando todos os segmentos baixados sejam reproduzidos.

O processo de requisições de segmentos de vídeo (momentos 4, 5 e 6) é repetido até que todos os segmentos da *playlist*, indicado pelo arquivo MPD, sejam baixados. Ou seja, a classe `DashClient` encerra a comunicação com o servidor HTTP quando todos os segmentos de vídeo são baixados. Ao final da sessão de *streaming* (reprodução de todos os

segmentos de vídeo pela *thread* de consumo do *buffer* na classe `Player`), a plataforma gera resultados em forma de gráficos na pasta `results`. Esses resultados são obtidos através de estatísticas coletadas com a estrutura `base.whiteboard` (*Whiteboard*), explicado a seguir. Antes de finalizar, ainda, o `Player` imprime métricas de desempenho da sessão como: número de pausas, tamanho das pausas, qualidades selecionadas, etc.

3.9.3 *Whiteboard*

Durante a execução da plataforma *pyDash*, diversas estatísticas são coletadas. Tais estatísticas são geradas pela classe `Player` e escritas na estrutura de ***Whiteboard***. O ***Whiteboard*** consiste em uma classe *Singleton* que permite a outras classes terem acesso às estatísticas que estão sendo geradas no andamento da reprodução de *streaming*.

A principal classe que se aproveita da estrutura de *Whiteboard* é a `IR2A`, pois as estatísticas geradas podem ser consideradas na lógica de implementação de um novo algoritmo ABR. A Figura 3.9 apresenta a relação de troca de informação entre a classe `Player` e o algoritmo ABR implementado. Como o novo algoritmo ABR herda os atributos da classe abstrata `IR2A`, ele tem acesso ao objeto `self.whiteboard`, que permite acesso às estatísticas geradas pela plataforma ao longo da sessão de *streaming*.

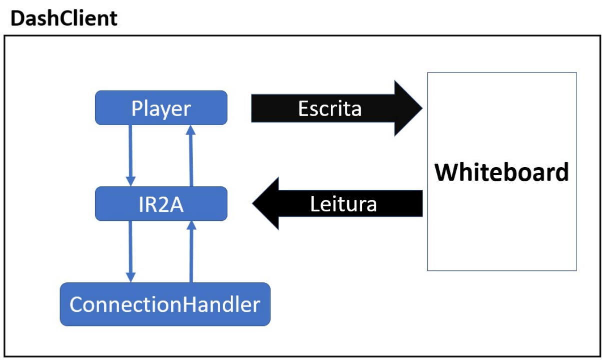

Figura 3.9: Área de transferência de estatística *Whiteboard*.

Através do objeto `self.whiteboard`, as seguintes estatísticas podem ser obtidas pelo novo algoritmo ABR implementado:

- `get_playback_qi()`: Método que obtém uma lista de tuplas. Cada tupla contém dois elementos: o momento em que o dado foi coletado (em segundos) e a qualidade **QI** observada durante a reprodução do trecho do vídeo, nesta ordem. A lista é ordenada de forma crescente pelo campo tempo.

- `get_playback_pauses()`: Método que obtém uma lista de tuplas. Cada tupla contém dois elementos: o momento em que o dado foi coletado (em segundos) e o tamanho da pausa de vídeo (em segundos) ocorrida, nesta ordem. A lista é ordenada de forma crescente pelo primeiro campo de tempo.

- `get_playback_buffer_size()`: Método que retorna uma lista de tuplas. Cada tupla contém dois elementos: o momento em que o dado foi coletado (em segundos) e o tamanho do *buffer* observado durante a reprodução do vídeo, nesta ordem. A lista está ordenada de forma crescente pelo campo tempo.

- `get_playback_history()` - Método que obtém uma lista de tuplas. Cada tupla contém dois elementos: o momento em que o dado foi coletado (em segundos) e o status de reprodução do vídeo. O status terá o valor um (1) quando foi possível reproduzir o vídeo e zero (0) quando não foi possível. Esta lista representa o histórico de reprodução do vídeo. A lista está ordenada de forma crescente pelo campo tempo.

- `get_playback_segment_size_time_at_buffer()` - Método que obtém uma lista de tuplas. Cada tupla contém um elemento: o tempo (em segundos) que cada segmento de vídeo gasta no *buffer* antes de ser reproduzido pela *thread* de consumo do *buffer*. A lista está ordenada do segmento mais antigo até o mais novo (do início até o final do vídeo reproduzido).

3.9.4 Implementação de um Novo Algoritmo ABR com pyDash

Esta seção fornece a descrição da classe abstrata `IR2A`, bem como um exemplo que de como um novo algoritmo ABR pode ser implementado. Conforme descrito anteriormente, a arquitetura da plataforma *pyDash* está organizada em três camadas. Na camada do meio, encontra-se a classe abstrata `IR2A`. Esta classe desempenha um papel de interface para os novos algoritmos ABR a serem incorporados a plataforma.

Primeiramene, para realizar a implementação de um novo algoritmo ABR com *pyDash*, deve-se criar um arquivo Python, dentro da pasta `r2a`, definindo a classe do algoritmo. Esta classe deve necessáriamente herdar a classe base `IR2A`, apresentada no código da Listagem 3.2. Ou seja, suponha que o nome da classe do algoritmo seja `R2ANewAlgorithm1`. Dessa forma, a classe de implementação deverá ficar `class R2ANewAlgorithm1(IR2A)`, indicando que a classe de implementação do seu algoritmo está herdando a classe `IR2A`.

```
1 class IR2A(SimpleModule):
2     ...
3 
4     @abstractmethod
5     def handle_xml_request(self, msg):
```

```
        pass

    @abstractmethod
    def handle_xml_response(self, msg):
        pass

    @abstractmethod
    def handle_segment_size_request(self, msg):
        pass

    @abstractmethod
    def handle_segment_size_response(self, msg):
        pass

    ...
```

Listing 3.2: Métodos abstratos da classe *IR2A*

Com a herança da classe base `IR2A`, os métodos abstratos, indicados no código 3.2 ficam visíveis pela classe do novo algoritmo. A implementação destes métodos é necessária pela classe do novo algoritmo.

Com relação aos métodos abstratos apresentados no código 3.2, seguem abaixo as suas respectivas descrições:

- `handle_xml_request(self, msg)`: Método que lida com a requisição do arquivo MPD. O parâmetro `msg` é uma mensagem do tipo `base.Message`. A mensagem, gerada na camada superior, deve ser encaminhada a camada inferior ao final da execução deste método com o comando `self.send_down(msg)`.

- `handle_xml_response(self, msg)`: Método que lida com a resposta da requisição do arquivo MPD. O parâmetro `msg` é uma mensagem do tipo `base.Message`. Nela, o atributo `payload` contém o conteúdo XML do arquivo MPD recuperado do servidor HTTP. Seu conteúdo pode ser obtido pelo método `msg.get_payload()`. Para se obter a lista de qualidades disponíveis (`self.qi`), é posssível utilizar as funcionalidades do *parser* MPD disponibilizado pela plataforma *pyDash*. Abaixo é apresentado um exemplo de código em que a lista de qualidades (em bps) é criada através do *parser* feito no conteúdo do arquivo MPD.

```
    from player.parser import
    ...
    self.parsed_mpd = parse_mpd(msg.get_payload())
    self.qi = self.parsed_mpd.get_qi()

```

Ao final da execução do método `handle_xml_response(self, msg)`, a mensagem deve ser encaminha a camada superior através da função `self.send_up(msg)`.

- `handle_segment_size_request(self, msg)`: Método que lida com a requisição de um segmento de vídeo. O parâmetro `msg` é uma mensagem do tipo `base.SSMessage`. A definição da qualidade do segmento é feita através do método `add_quality_id()` da mensagem `msg`, seguindo a política ABR implementada. Suponha, por exemplo, que `self.qi` contenha uma lista com 20 qualidades disponíveis. Se na política de escolha for determinado a requisição do segmento de maior qualidade, então a qualidade será definida como `msg.add_quality_id(self.qi[19])`. Por fim, a mensagem gerada na camada superior deve ser encaminhada a camada inferior ao final da execução deste método com o comando `self.send_down(msg)`.

- `handle_segment_size_response(self, msg)`: Método que lida com a resposta da requisição de um segmento de vídeo. O parâmetro `msg` é uma mensagem do tipo `base.SSMessage`. Nesta mensagem encontram-se informações necessárias para a representação do segmento de vídeo requisitado. É possível, por exemplo, obter o tamanho do segmento recebido através do método `msg.get_bit_length()` e calcular a diferença de tempo transcorrido entre o momento da requisição realizada e a resposta recebida. Com isso, o algoritmo ABR é capaz de calcular a taxa de transferência obtida para a mensagem `msg` recebida. Por fim, ao final da execução do método, a mensagem `msg` deve ser encaminhada à camada superior através da função `self.send_up(msg)`.

Cabe destacar que outros métodos também fazem parte da classe abstrata `IR2`, como `initialize(self)` e `finalization(self)`. Entretanto, estas não são precisam necessáriamente serem implementadas pela classe do novo algoritmo ABR, bastanto a declaração destes métodos.

Por fim, para que o novo algoritmo seja carregado dinâmicamente na plataforma *pyDash*, é preciso que o valor do campo `r2a_algorithm`, presente no arquivo de configuração JSON, seja o nome da classe do algoritmo implementado. Dessa forma, se a nova classe teve o nome definido como `R2ANewAlgorithm1`, então no valor do campo `r2a_algorithm` deve ser inserido "`R2ANewAlgorithm1`". Além disso, é uma boa prática que tanto o nome da classe quanto o nome do arquivo sejam os mesmos (classe `R2ANewAlgorithm1` contida no arquivo `r2a/r2anewalgorithm1.py`).

Exemplificação

O código 3.3 apresenta como um algoritmo adaptativo de *streaming* pode ser implementado com *pyDash*. Nesse exemplo, foi considerado a implementação do algoritmo 1

chamado `R2A_AverageThroughput`.

Algorithm 1 R2A_AverageThroughput

Pré: $listaQualidades, totalSegmentosBaixados, amostrasDeBandaEstimada$
Pós: $taxaBitsSelecionada$

$avg \leftarrow 0$
enquanto requisitando segmentos de vídeo **faça**
  $avg \leftarrow (amostrasDeBandaEstimada/totalSegmentosBaixados)/2$
  $taxaBitsSelecionada \leftarrow listaQualidades[0]$
  $i \leftarrow 0$
  enquanto $avg \geq listaQualidades[i]$ **faça**
    $taxaBitsSelecionada \leftarrow listaQualidades[i]$
    $i \leftarrow i + 1$
  fim enquanto
  Requisitar segmento na $taxaBitsSelecionada$
fim enquanto

Conforme o algoritmo 1, apresentado de forma simplificada em pseudocódigo, a taxa de bits do k-th segmento de vídeo a ser baixado *taxaBitsSelecionada* é igual a maior qualiadade q_i do conjunto de representações disponíveis em *listaQualidades* que não ultrapassa 50% da média aritmética das amostras de largura de banda estimadas. A k-th amostra de largura de banda $amostrasDeBandaEstimada_k$ é computada como $amostrasDeBandaEstimada_k = 8s_{k-1}/t_{k-1}$, onde s_{k-1} é o tamanho do segmento baixado em bytes, e t_{k-1} é o tempo gasto para baixar o último segmento em segundos. O tempo t_{k-1} é calculado através da diferença do tempo gasto desde um requisição até o recebimento da resposta. O responsável por determinar o limite máximo da taxa de bits da qualidade a ser selecionada avg_k é computada como $avg_k = \sum_{k=1}^{k}(amostrasDeBandaEstimada_k/k)/2$. As amostras de largura de banda são somadas, obitda sua média aritmética recupera a metade da média aritmética (avg). resultando em amostras de largura de banda filtradas . Por fim, a qualiadade q_i do próximo segmento de vídeo a ser baixado é a qualiadade q_i disponível que mais se aproxima de avg_k. Ou seja, a saída do algoritmo é um nível de qualidade inferior à metade da média das amostras de taxa de transmissão computadas.

```
1  from r2a.ir2a import IR2A
2  from player.parser import *
3  import time
4  from statistics import mean
5
6  class R2A_AverageThroughput(IR2A):
7      def __init__(self, id):
8          IR2A.__init__(self, id)
9          self.throughputs = []
10         self.request_time = 0
```

```
11          self.qi = []
12
13      def handle_xml_request(self, msg):
14          self.request_time = time.perf_counter()
15          self.send_down(msg)
16
17      def handle_xml_response(self, msg):
18          parsed_mpd = parse_mpd(msg.get_payload())
19          self.qi = parsed_mpd.get_qi()
20
21          t = time.perf_counter() - self.request_time
22          self.throughputs.append(msg.get_bit_length() / t)
23
24          self.send_up(msg)
25
26      def handle_segment_size_request(self, msg):
27          self.request_time = time.perf_counter()
28          avg = mean(self.throughputs) / 2
29
30          selected_qi = self.qi[0]
31          for i in self.qi:
32              if avg > i:
33                  selected_qi = i
34
35          msg.add_quality_id(selected_qi)
36          self.send_down(msg)
37
38      def handle_segment_size_response(self, msg):
39          t = time.perf_counter() - self.request_time
40          self.throughputs.append(msg.get_bit_length() / t)
41          self.send_up(msg)
42
43      def initialize(self):
44          pass
45
46      def finalization(self):
47          pass
```

Listing 3.3: Implementação do algoritmo R2A_AverageThroughput com pyDash

Conforme mostrado no código 3.3, as linhas 21 e 39 obtêm o tempo gasto (t_{k-1}) para baixar o último segmento . A função `msg.get_bit_length ()` recupera o tamanho do segmento baixado em bits ($8s_{k-1}$) e as amostras de larguras de banda ($throughput_k$) são coletadas tanto na linha 22 quanto na linha 40. Quando o algorimto vai selecionar a qualidade do segmenteo de vídeo k-th, o método `handle_segment_size_request` obtém

o resultado da função de filtro das larguras de banda (avg_k) na linha 28 e inicia sua política de seleção nas linhas 30-33. Finalmente, a função `msg.add_quality_id` adiciona a taxa de bits do vídeo a ser recuperado na mensagem de requisição `msg` na linha 35 e a encaminha à camada de baixo `IR2A` com a função `self.send_down` na linha 36.

3.10 Análise Comparativa

O estudo e a comparação dos trabalhos já desenvolvidos permite destacar quais características diferenciam as múltiplas ferramentas já desenvolvidas. Além disso, permite pensar o que pode ser incorporado ou evitado no desenvolvimento de uma proposta que auxiliará na análise e compreensão do comportamento de algoritmos adapativos de *streaming* (ABR). A Tabela 3.2 apresenta uma comparação entre as ferramentas discutidas.

Primeiramente, é importante destacar que a maioria das ferramentas são emuladores de clientes DASH. Elas oferecem uma alternativa ao uso de clientes DASH reais, pois permitem conduzir experimentos mais próximos de ambientes de produção, através de *streaming* de conteúdo de vídeo real de *datasets* focado nesta finalidade. Apesar disto, nem todas as ferramentas desta abordagem fazem uso de um decodificador de mídia, conforme a Tabela 3.2. A maioria delas tiveram como propóstio oferecer uma aplicação que reduzisse p uso de recursos computacionais. Entretanto, nada impede que a decodificação e o *display* do vídeo seja implementado e sua habilitação ou não fique por conta do interesse do usuário. A proposta do *display* do vídeo possibilita ao estudante perceber como está a qualidade do vídeo de fato. Dadas essas justificativas, a opção de habilitar um decodificador de mídia será incorporado ao *pyDash*.

Como observado na Tabela 3.2, são poucas as ferramentas que incorporam gráficos dinâmicos para visualização de parâmetros de desempenho como nível de *buffer*, qualidade selecionada, dentre outros. Para além de simples textos de *logs*, a observação da evolução de tais parâmetros são relevantes na análise de desempenho de algoritmos ABR em experimentos DASH. Por este motivo, as ferramentas [11] e [9] são referências em prover uma abordagem capaz de visualizar a evolução de desempenho em tempo real ao longo da sessão de *streaming*. Como já fora discutido, o projeto de um algoritmo ABR envolve muitos *trade-offs* e abordagens diferentes. Conseguir observar como seu algoritmo reage às variações de largura de banda disponível durante a sessão de *streaming* permite perceber se tal algoritmo está desempenhando conforme projetado. Por esta razão, a exibição de gráficos dinâmicos que auxiliem no estudo e compreensão da performance de algoritmos ABR será integrado ao *pyDash*, que tem propósitos educacionais.

Por fim, a computação de métricas QoE é de grande relevância no estudo acerca do desempenho de soluções HAS. Métricas calculadas ao final da reprodução de vídeo como

Tabela 3.2: Comparação de recursos das ferramentas de análise de algoritmos ABR.

Ferramenta	**Decoder/ *Display***	**Exibe Gráficos Dinâmicos**	**Computa Métricas QoE**	**Ambiente**
Dash.js [11]	Sim	Sim	Não	Produção
TAPAS [14]	Opcional	Não	Não	Acadêmico
AStream [15]	Não	Não	Não	Pesquisa
End-to-End DASH Platform [9]	Sim	Sim	Não	Pesquisa
Dashc [16]	Não	Não	Sim	Pesquisa
GoDASH [17]	Não	Não	Sim	Acadêmico
Framework HAS [18]	Não	Não	Sim	Acadêmico
Sabre [19]	Não	Não	Sim	Pesquisa
PyDash [1]	Não	Não	Sim	Educacional

qualidade média selecionada, quantidade de interrupções, etc, são utilizadas em modelos de QoE [10], as quais indicam estimativas gerais de desempenho. Dado a importância desta funcionalidade, inclusive para sistemas de *logs*, e a relativa ausência de funções QoE em uma proposta educacional como *pyDash*, será incluido funções QoE que revelem clareza no desempenho de algorimtos ABR.

Nesta capítulo, recursos presentes em ferramentas que buscaram realizar a tarefa de prover um ambiente de estudo de algoritmos ABR foram verificadas. Cada proposta traz consigo abordagens e funcionalidades distintas e, após fazer a análise dos pontos positivos e negativos, foram destacadas as estratégias que mais favorecem no estudo e aprendizagem do desempenho de algoritmos adaptativos de *streaming* em aplicações HAS. No próximo capítulo, será apresentado a integração e o resultado das propostas de implementação na plataforma *pyDash* assimiladas nesta seção.

Capítulo 4

Melhorias implementadas na plataforma *pyDash*

Este capítulo tem como objetivo descrever os aprimoramentos feitos na plataforma *pyDash* para atingir os objetivos deste trabalho. Acerca das propostas de melhoria, todas elas evidenciam, de uma forma ou de outra, a performance do algoritmo adaptativo. Exibindo o vídeo, o reprodutor de mídia permitirá ao aluno observar os impactos de mudança da qualidade selecionada, bem como as pausas no vídeo. Os gráficos dinâmicos irão mostrar dados estatísticos, com mais clareza, de desempenho de algoritmos adaptativos em aplicações HAS. E, por fim, as funções de desempenho de QoE fornecerá indicadores de desempenho do algoritmo em determinado cenário de *streaming*. São por estes motivos que estas melhorias na plataforma *pyDash* vão contribuir, ainda mais, para o estudo e análise de desempenho de algoritmos adaptativos pelos alunos. Cada seção apresenta uma melhoria, desde sua forma conceitual até sua inclusão prática na ferramenta. Assim, a seção 4.1 explica desde a concepção do reprodutor de mídia DASH até sua implementação e funcionamento com o *pyDash*. A seção 4.2 apresenta o que foi necessário para produzir uma proposta de aplicação que exibe estatísticas de dempenho, através de um *softtware* de criação e visualização de dados. Na seção 4.3, está especificado as duas funções de QoE incluidas para estimar o desempenho geral do cliente DASH.

4.1 Reprodutor de Mídia DASH

O objetivo desta seção é demonstrar a aplicação Gstreamer necessária para criar o reprodutor de mídia DASH. Antes de descrevê-lo e apresentar como ele foi implementado na plataforma *pyDash*, é preciso entender a aplicação GStreamer e seu modelo de computação *dataflow*, no qual o *framework* se baseia.

4.1.1 Gstreamer

O Gstreamer[1] é um *framework* de código livre/aberto para criação de aplicativos de mídia de *streaming* e que lidam com processamento de dados multimídia, como áudio e vídeo [38]. Além de robusto e flexível, o *framework* suporta diversos formatos de áudio e vídeo, e é amplamente utilizado na indústria e na academia [39]. Projetado para ser mutliplataforma, é possível utilizá-lo sobre Linux, Mac OS X e Microsoft Windows. O *framework* em si consiste de um conjunto de bibliotecas C, porém tem *bindings* - Interface de Programação de Aplicativo (do Inglês, *Application Progamming Interface* - API) que permite que uma linguagem de programação use uma biblioteca estrangeira - para linguagens de programação como Python, C++, Perl, GNU Guile (guile) e Ruby [40].

Visão geral do funcionamento

Conceitualmente, o Gstreamer se baseia no modelo de computação *dataflow*. Nesse modelo, uma aplicação multimídia estrutura-se como um grafo em que os nós representam elementos de processamento, ou atores, e as arestas representam conexões entre elementos por onde fluem amostras de áudio e vídeo e dados de controle. Os atores recebem dados através de suas portas de entrada e emitem dados através de suas porta de saída. Um *pipeline* é um *dataflow* em que os dados fluem através das arestas na mesma ordem em que foram produzidos.

A Figura 4.1 apresenta o leiaute de um *pipeline* típico para processamento multimídia. Os nós da figura (*source*, *filter* e *sink*) representam elementos e as arestas representam as conexões por onde fluem as amostras de áudio e vídeo e dados de controle.

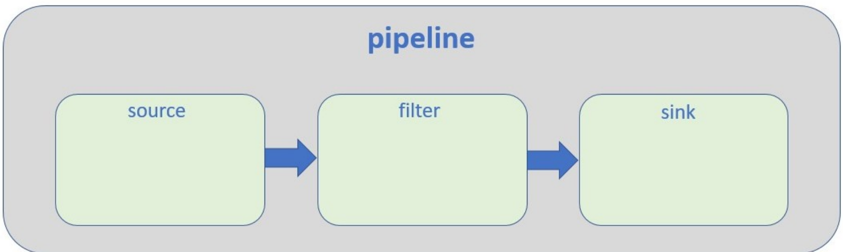

Figura 4.1: Exemplo de *pipeline*.

Na terminologia do GStreamer os atores são chamados de "elementos" e as portas de "*pads*". Os elementos são os blocos básicos de construção do GStreamer. Eles processam os dados à medida que fluem dos elementos de origem (produtores de dados) para os elementos coletores (consumidores de dados), passando pelos elementos de filtro. As

[1] https://gstreamer.freedesktop.org/

pads são portas através das quais os elementos do GStreamer se comunicam. Há dois tipos de *pads*: *sink pads* e *source pads*. As *sink pads* são as portas por meio das quais os dados entram em um elemento, e as *source pads* são as portas por meio das quais os dados saem de um elemento. Os elementos são classificados de acordo com o tipo de suas *pads*. A Figura 4.2 apresenta as classes de elementos. Elementos produtores (*sources*) possuem apenas *source pads*. Elementos processadores (*filter*) possuem ambos os tipos, *source pads* e *sink pads*. E, elementos consumidores (*sink*) possuem apenas *sink pads*. Consequentemente, produtores apenas produzem dados, processadores consomem e produzem dados, e consumidores apenas consomem dados.

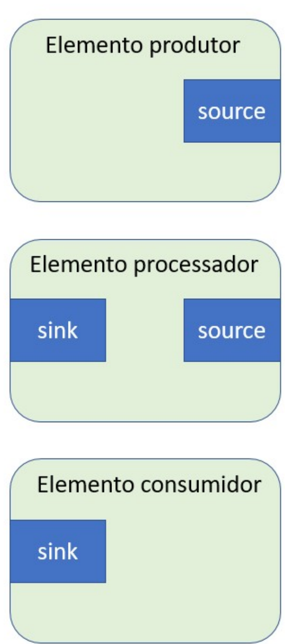

Figura 4.2: Elementos do GStreamer com seus *pads*.

Dois tipos de dados trafegam através das conexões (arestas) de um *pipeline* GStreamer: segmentos de dados (*buffers*) e eventos (*events*). Os *buffers* carregam segmentos do conteúdo processado entre *pads* (por exemplo, amostras de áudio e vídeo codificadas ou *raw*) e fluem exclusivamente na direção das conexões, ou seja, de *source pads* para *sink pads*. Já os eventos carregam informação de controle; eles também fluem entre conexões, mas podem percorrê-las em ambos os sentidos: fluxo abaixo (*downstream*), de *source pads* para *sink pads*, ou fluxo acima (*upstream*), de *sink pads* para *source pads*.

Buffers e eventos percorrem as conexões em paralelo. Ou seja, conceitualmente cada conexão entre as *pads* dos elementos pode ser entendida como consistindo de dois canais, um canal unidirecional para *buffers* e outro canal bidirecional para eventos. A Figura 4.3

ilustra a estrutura conceitual de uma conexão entre *pads*. Na figura,“B” é o canal de *buffers* e “E” é o canal de eventos.

Figura 4.3: Estrutura conceitual de uma conexão entre *pads no Gstreamer*.

No Gstreamer, todo elemento, incluindo o *pipeline*, possui um estado que pode ser nulo (identificado pela constante simbólica `NULL`), pronto (`READY`), pausado (`PAUSED`) ou tocando (`PLAYING`). No estado inicial, `NULL`, o elemento não possui recursos alocados. No estado `READY`, o elemento aloca recursos globais que não dependem do conteúdo a ser processado. No estado `PAUSED`, o elemento aloca recursos que dependem do conteúdo a ser processado e se prepara para processá-lo. Finalmente, no estado `PLAYING`, o elemento inicia o processamento.

Para chegar do estado `NULL` (inicial) ao estado `PLAYING` (tocando) todo elemento tem que passar primeiro pelo estados `READY` e `PAUSED`, nessa ordem. De forma análoga, para sair do estado `PLAYING` e voltar ao estado inicial `NULL`, o elemento tem que passar pelos estados `PAUSED` e `READY`. A Figura 4.4 apresenta a máquina de estados de um elemento GStreamer.

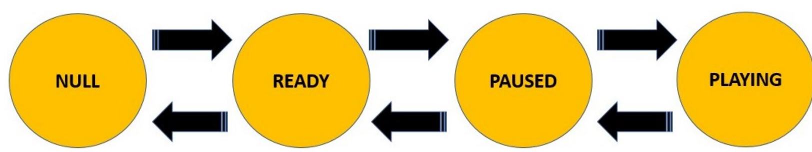

Figura 4.4: Máquina de estados de um *pipeline* ou elemento.

A discussão do modelo *dataflow* e da sua instanciação no GStreamer se encerra aqui. Na maior parte do tempo, programar no GStreamer consiste em montar *pipelines* e controlar seu funcionamento. Até agora foi discutido de maneira abstrata os conceitos envolvidos nessa montagem e controle. A partir da próxima subseção, será visto como foram utilizados esses conceitos na prática para a construção do reprodutor de mídia DASH.

4.1.2 Estrutura do Reprodutor de Mídia DASH

A Figura 4.5 apresenta o *pipeline* implementado no reprodutor de mídia DASH, enquanto um breve resumo da finalidade de cada elemento é fornecido na Tabela 4.1.

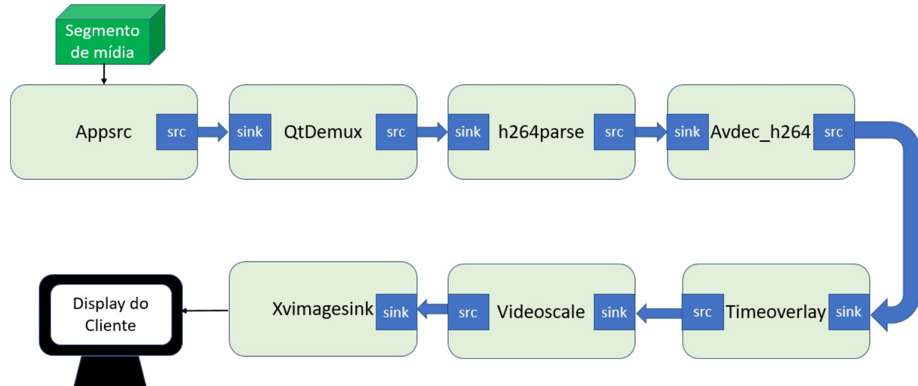

Figura 4.5: Pipeline Gstreamer para o reprodutor de mídia DASH.

Conforme a Figura 4.5, há um elemento produtor ("appsrc"), cinco processadores ("qtdemux", "h264parse", "avdec_h264", "timeoverlay" e "videoscale") e um consumidor ("xvimagesink"). O elemento "appsrc" possui uma única *source pad* que está conectada à *sink pad* do elemento subsequente, "qtdemux". Ou seja, os dados produzidos pelo elemento "appsrc" fluem através da sua *source pad* para a *sink pad* do elemento "qtdemux". No diagrama, essa conexão entre as *pads* é representada pela aresta entre os elementos "appsrc" e "qtdemux". Similarmente, as demais arestas denotam conexões entre *source pads* e *sink pads*.

O processo de desenvolvimento de uma aplicação GStreamer com *pipeline* estático (cuja topologia não muda em tempo de execução) é relativamente simples. Basta instanciar os elementos necessários, interconectar suas *pads* e iniciar o *pipeline* resultante. O *pipeline* da Figura 4.5, por exemplo, após iniciado opera da seguinte forma:

1. O elemento "appsrc" lê *buffers* inseridos pela aplicação e escreve o fluxo de bytes resultante na sua *source pad*.

2. O elemento "qtdemux" lê da sua *sink pad* um fluxo de bytes codificado no formato MPEG-4, demultiplexa-o e escreve o fluxo de vídeo resultante na sua *source pad* correspondente.

Tabela 4.1: Elementos Gstreamer usados na criação do reprodutor de mídia DASH.

Elemento	**Descrição**
Appsrc	Permite que a aplicação alimente/insira *buffers*/dados em um *pipeline*
QtDemux	Demultiplexador para extrair (QuickTime) H264 e componentes AAC de um fluxo de entrada MP4
H264parse	Analisador de fluxo de vídeo H264
Avdec_h264	Decodificador de vídeo H264
Timeoverlay	Sobrepõe registros de tempo do *buffer* inserido em *stream* de vídeo
Videoscale	Redimensionador de *frames* de vídeo
Xvimagesink	Renderizador de *frames* de vídeo descomprimido (*raw*) em uma janela de saída

3. O elemento "h264parse" lê da sua *sink pad* um fluxo de bytes codificado no formato H.264, analisa-o e escreve o fluxo de vídeo resultante na sua *source pad*.

4. O elemento "avdec_h264" lê da sua *sink pad* um fluxo de bytes codificado no formato H.264, decodifica-o e escreve o fluxo de vídeo descomprimido (*raw*) resultante na sua *source pad*. Um fluxo de vídeo descomprimido é uma sequência de amostras (quadros) de vídeo em que cada quadro é uma matriz de *pixels* codificados em algum modelo de cor.

5. O elemento "timeoverlay" lê da sua *sink pad* um fluxo de bytes descomprimido, sobrepõe as marcações de tempo do *buffer* de um fluxo de vídeo sobre si mesmo e escreve o fluxo de vídeo descomprimido (*raw*) resultante na sua *source pad*. É possível posicionar o texto e configurar os detalhes da fonte usando suas propriedades.

6. O elemento "videoscale" lê da sua *sink pad* um fluxo de bytes descomprimido, redimensiona os quadros de vídeo e escreve o fluxo de vídeo resultante (*raw*) na sua *source pad*. Por padrão, o elemento tentará negociar para o mesmo tamanho na *source* e *sinkpad*, de forma que nenhum ajuste seja necessário. Portanto, é seguro inserir este elemento em um *pipeline* para obter um comportamento mais robusto sem nenhum custo se nenhum dimensionamento for necessário.

7. O elemento "xvimagesink" lê um fluxo de vídeo descomprimido da sua *sink pad* e utiliza a biblioteca X11 [41] para reproduzir os quadros do fluxo de vídeo na tela.

A função de um *pipeline*, isto é, o que ele faz ou computa, é o resultado da combinação da função dos seus elementos que, conceitualmente, operam em paralelo. O *pipeline* anterior, portanto, (1) lê um *buffer* inserido, (2) demultiplexa-o, (3) analisa-o, (4) decodifica o fluxo vídeo resultante, (5) insere marcações de tempo do *buffe*, (6) redimensiona os qua-

dros de vídeo e (7) reproduz o fluxo decodificado no dispositivo de saída correspondente (tela). Conceitualmente, tudo isso acontece em paralelo, ou seja, podemos assumir que enquanto o *sink* “xvimagesink” está exibindo amostras, o *source* “appsrc” está lendo bytes que chegam da aplicação, o demultiplexador “qtdemux” está demultiplexando dados MPEG-4, o decodificador “avdec_h264” está decodificando dados H.264 e assim sucessivamente.

Criando o reprodutor de mídia DASH

A Listagem 4.1 apresenta o programa Python que implementa a classe do reprodutor de mídia DASH.

```python
...
class GstPlayer:
    ...
    def __init__(self):
        ...

    def push(self, data):
        ...

    def get_queued_bytes(self):
        ...

    def _get_state(self):
        ...

    def play_segment(self):
        ...

    def play(self):
        ...

    def pause(self):
        ...

    def stop(self):
        ...

    def quit_main_loop(self):
        ...
```

Listing 4.1: Métodos da classe GstPlayer em Python

Com relação aos métodos apresentados no código 4.1, seguem abaixo as suas respectivas descrições:

- `__init__(self)` - Método que tem a responsabilidade de criar o objeto da classe `GstPlayer`. Nela está contida todas as informações principais do objeto como o *pipeline* a ser criado. Abaixo é apresentado detalhes deste método.

```python
    def __init__(self):
        self.pipeline = Gst.parse_launch(
            "appsrc name=source is-live=true max-bytes=0 ! "
            "identity single-segment=true ! qtdemux ! "
            "h264parse ! avdec_h264 ! "
            "timeoverlay halignment=center valignment=top \
            text='Stream time:' shaded-background=true \
            font-desc='Sans, 24'!"
            "videoscale ! xvimagesink"
        )

        self.appsrc = self.pipeline.get_by_name("source")

        self.main_loop = GLib.MainLoop()
        self.thread = Thread(target=self.main_loop.run)
        self.thread.start()

```

  Nas linhas 2-10, está definido o *pipeline*. O comando `Gst.parse_launch` cria o *pipeline* com os elementos da Tabela 4.1 conectados em série. O elemento "appsrc" possui três propriedades configuradas. A propriedade `name` indica o identificador deste elemento. A segunda `is-live` instrui se a *source* deve se comportar como uma *live source*, isto é, só irá "empurrar" *buffers* no estado `PLAYING`. E, `max-bytes` indica a quantidade máxima de bytes que podem ser enfileirados internamente. O valor 0 para esta proprieadade indica quantidade ilimitada. Para o elemento "timeoverlay", foi configurado a exibição dos registros de tempo (*timestamps*) com um texto inicial alocado no meio do alto da imagem do vídeo, com o fundo do texto sendo sombreado para torná-lo mais legível em cima do fundo de vídeo.

  Na linha 12, o elemento produtor "appsrc" é obtido pelo seu identificador "source". Por fim, da linha 14-16, o loop principal responsável por tratar dos eventos que ocorrem durante a execução do *pipeline* é instaciado pela estrutura `GLib.MainLoop()` e iniciado em uma nova *thread*.

- `push(self, data)` - Método que insere *buffers*/dados no elemento "appsrc".

- `get_queued_bytes(self)` - Método que obtém o número de bytes atualmente enfileirados no elemento "appsrc".
- `play_segment(self)` - Método que inicia o *pipeline* e começa a transmitir os segmentos enfileirados na sua *source pad*. O estado do *pipeline* passa de `NULL` para `PLAYING`.
- `play(self)` - Método que retoma a exibição do vídeo, depois de pausado. Ou seja, o estado do *pipeline* passa de `PAUSED` para `PLAYING`.
- `pause(self)` - Método que pausa a reprodução do vídeo. O estado do *pipeline* passa de `PLAYING` para `PAUSED`.
- `stop(self)` - Método que encerra a reprodução do vídeo e libera recursos do *pipeline*. Ou seja, O estado do *pipeline* passa de `PLAYING` para `NULL`.
- `quit_main_loop(self)` - Método que encerra a *thread* do objeto `main_loop`.

Com todo esse histórico em mente, o código abaixo apresenta as blibliotecas e funções necessárias para o correto funcionamento da aplicação GStreamer.

```
1  import gi
2  gi.require_version("Gst", "1.0")
3  gi.require_version("GLib", "2.0")
4  from gi.repository import Gst, GLib
5  ...
6
7  class GstPlayer:
8      Gst.init()
9      ...
10
```

Como dito anteriormente, a biblioteca GStreamer é escrita em C e, para que seja possível desenvolver em outra linguagem é necessário usar vinculação (*binding*). A biblioteca de vinculação Python do GStreamer é chamada de PyGObject, e é importada na linha 1. Depois é preciso dizer ao PyGObject a versão mínima do GStreamer e GLib que o programa requer, descritas nas linhas 2 e 3, respectivamente. Feito isso, é possível importar o módulo "`Gst`", bem como o módulo "`GLib`" que são usados no programa. Por fim, antes de fazer qualquer outra coisa, é chamado `Gst.init()` para inicializar o GStreamer na linha 8.

4.1.3 Implementando o Reprodutor de Mídia DASH no *pyDash*

Na subseção anterior, foi introduzido a visão geral do *pipeline* GStreamer (ou seja, cadeia de elementos) necessária para criar o reprodutor de mídia DASH. Nesta subseção,

o objetivo é demonstrar a implementação do reprodutor de mídia DASH na plataforma *pyDash*.

Inicialmente foi criado um diretório chamado `dash_media_player` na composição do *pyDash*. Nesta pasta, o arquivo `gst_player.py` incorpora a classe `GstPlayer`. Este procedimento é importante para que a referida classe possa ser importada e utilizada em outra classe. Em seguida, foi adicionado no arquivo de configuração JSON *dash_client.json* a chave "`dash_media_player`", que tem a função de habilitar ou não o reprodutor de mídia DASH. Nesta opção, é possível atribuir a esta chave os valores "`on`", para habilitar o reprodutor de mídia DASH ou "`off`", caso contrário. A listagem 4.2 apresenta o exemplo do arquivo de configuração JSON com o reprodutor de mídia DASH habilitado.

```
1 {
2     "buffering_until": 5,
3     ...
4     "dash_media_player": "on"
5 }
```

Listing 4.2: Arquivo de configuração *dash_client.json* modificado

Ao configurar o valor "`on`" para o parâmetro "`dash_media_player`", um objeto da classe `GstPlayer`, mencionado na Listagem 4.1, é instanciado dentro da classe `Player`. Além disso, a opção "`on`" irá servir de parâmetro para demais ações do reprodutor de mídia. Ademais, variáveis de controle do reprodutor de mídia são declarados. O código abaixo apresenta o que foi mencionado.

```
1  ...
2  from dash_media_player.gst_player import GstPlayer
3  ...
4  class Player(SimpleModule):
5      def __init__(self, id):
6          ...
7          if self.dash_media_player_option.lower() == 'on':
8              self.dash_media_player = GstPlayer()
9              self.started_dash_media_player = False
10             self.dash_media_player_paused = False
11
```

Com o reprodutor de mídia DASH habilitado, é preciso que segmentos de vídeo real sejam recuperados da rede e introduzidos no tocador de mída. No contexto do *pyDash*, quando uma resposta HTTP de um segmento de vídeo é obtida, ela é entregue à camada `ConnectionHandler`. Para que o conteúdo do segmento de vídeo `ss` pudesse ser recuperado e adicionado na mensagem `msg`, em sua propriedade `payload`, foi incorporado a função `msg.add_payload(ss)` com esta finalidade. A partir desta implementação, o

conteúdo do segmento de vídeo fica encapsulado na mensagem `msg` e chega até a classe `Player`, onde o reprodutor de mída DASH foi escolhido para ser inserido.

O motivo pela escolha da classe `Player` para a inserção do reprodutor de mídia está no *modus operandi* do *pyDash*. Com explicado no Capítulo **??**, o funcionamento opera em uma estrutura de pilha. Cada uma das três classes principais que compõem o `DashClient` desempenha tarefas bem definidas. O `Player` é a classe responsável pela simulação da estrutura de *buffer* e "reprodução" do vídeo. Com isto em mente, ficou claro que o comportamento do reprodutor de mídia DASH deveria seguir o comportamento da classe `Player`, como as operações de inserção do segmento de vídeo no *buffer*, o consumo/reprodução do vídeo e seu encerramento. Dessa forma, foram implementados novos métodos na classe `Player` que pudussem manipular estas ações no reprodutor de mídia. A Listagem 4.3 apresenta os métodos inseridos.

```
...
class Player(SimpleModule):
    ...
    def push_video_to_dash_media_player(self, msg):
        ...
    def dash_media_player_playback(self):
        ...
    def close_dash_media_player_playback(self):
        ...
```

Listing 4.3: Métodos inseridos na classe *Player* para controlar o reprodutor de mídia DASH

Com relação aos métodos inseridos na classe `Player` e apresentados no código 4.3, seguem abaixo as suas respectivas descrições:

- `push_video_to_dash_media_player(self, msg)` - Método que recupera o conteúdo do segmento de vídeo e alimenta/insere o fluxo de vídeo no reprodutor de mídia DASH. Antes de inserir o segmento de vídeo recuperado, monta-o em um fluxo de vídeo, concatenando com o segmento de inicialização. O código abaixo demonstra sua descrição.

```
     ...
    class Player(SimpleModule):
        ...
        def push_video_to_dash_media_player(self, msg):
            video_segment = msg.get_payload()
            video_data = video_init + video_segment

            # adding the segment in the buffer
            self.dash_media_player.push(video_data)
```

```
10
```

Encapsulado na mensagem `msg`, o conteúdo do segmento de mídia pode ser recuperado com a função `msg.get_payload()`. De posse do segmento de vídeo baixado, é necessário concatená-lo com o segmento de inicialização para obter um fluxo contínuo de vídeo reproduzível. Em seguida, basta utilizar a função `push` do objeto `GstPlayer` para alimentar o reprodutor de mídia DASH.

- `dash_media_player_playback(self)` - Método que lida com o controle da reprodução do vídeo. Ou seja, inicia o tocador de mídia DASH, bemo como retoma a reprodução depois de ter sofrido uma pausa.
- `close_dash_media_player_playback(self)` - Método que encerra a *thread* do tocador de mídia DASH depois que todos os segmentos de vídeo baixados são reproduzidos.

Funcionamento

Antes de tudo, para que o reprodutor de mídia funcione, é preciso habilitá-lo, como explicado anteriormente.

No *streaming* de vídeo, quando um segmento é recebido pela rede, ele é destinado ao espaço alocado de *buffer* do cliente DASH. Na plataforma *pyDash*, o segmento de vídeo `ss` é recebido inicialmente pela classe `ConnectionHandler` e depois encapsulado na mensagem `msg`, por meio da chamada `msg.add_payload(ss)`. Vale ressaltar que o conteúdo do segmento de vídeo, propriamente dito, só é encapsulado quando o reprodutor de mídia DASH está habilitado. Em seguida, como explicado no Capítulo **??**, na seção de seu funcionamento, a mensagem `msg` é encaminhada a camada superior (`IR2A`), até chegar na classe `Player`. Nela, o método `handle_segment_size_response` confere se há um segmento na mensagem `msg` recebida para ser inserido no *buffer* emulado. Quando há, o método `self.buffering_video_segment(msg)` da classe `Player` é responsável por alimentar o *buffer* do cliente DASH. Analogamente, esta mesma operação é realizada, simultaneamente e no mesmo método, pela chamada do novo método `push_video_to_dash_media_player(msg)`.

Ao atingir o limite de preenchimento do *buffer* necessário para começar a consumír os dados de vídeo, a *thread* `handle_video_playback` é acordada e um segmento do *buffer* é consumido. Nessa mesma *thread*, o método `dash_media_player_playback()` é acionado, e um segmento de vídeo inserido no reprodutor de mídia DASH é tocado, estando em sincronia com a consumação do *buffer* emulado. Por outro lado, quando não há mais segmentos no *buffer* emulado, ou seja, uma pausa acontece, o reprodutor de mídia também

é pausado através do método `pause` da classe `GstPlayer`, interropendo a reprodução do vídeo, em simultâneo.

Por fim, quando não há mais segmentos a serem consumidos e a reprodução do vídeo original termina, o reprodutor de mídia DASH é encerrado através do método `close_dash_media_player_playback`. Esta chamada acontece assim que a *thread* de consumo `handle_video_playback` é encerrada e antes da finalização.

4.2 Aplicação de Exibição de Estatísticas em Tempo Real

Conforme explicado no Capítulo **??** seção 3.9.3, alguns gráficos são gerados na pasta `results`, ao final da execução da plataforma *pyDash*. Os valores utilizados para gerar os gráficos podem ser acessados através do objeto `self.whiteboard`. No geral, os gráficos se mostram convenientes, uma vez que permitem visualizar o desempenho do algoritmo ABR naquela sessão de *streaming*. A visualização, por exemplo, das decisões de qualidade do vídeo auxilia a entender como o algoritmo ABR se comportou em relação às variações de largura de banda na rede e/ou no tamanho de ocupação do *buffer*. No *pyDash*, porém, os gráficos são gerados apenas quando a plataforma finaliza. Ou seja, não é possível acompanhar a evolução da tomada de decisões de qualidade do vídeo no decorrer da sessão de *streaming*. Logo, uma solução que fosse capaz de demonstrar o comportamento e as decisões tomadas pelo algoritmo ABR em tempo real demonstra-se vantajosa na compreensão do desempenho do algoritmo ABR. Desse modo, esta seção apresenta a implementação de uma aplicação que exibe gráficos dinâmicos com os dados estátisticos fornecidos pelo *pyDash*.

4.2.1 Script Python

O *script* Python é uma classe que recupera dados de um objeto `self.whiteboard` e os escreve em um arquivo no formarto Valores Separados por Vírgulas (do Inglês, *Comma-separated values* - CSV). O *script*, chamado `log_writer.py`, encontra-se na pasta `utils`, incorporado ao *pyDash*. Nesta pasta, também, os arquivos CSV gerados são armazenados. O *script* foi implementado com a classe chamada `LogWriter`, a qual possui um método `__init__`, que tem por finalidade alocar os recursos necessários ao seu funcionamento, e o método `run`, o qual realiza a tarefa principal de ler os dados de um objeto `self.whiteboard` e escrevê-los em um arquivo CSV. A Figura 4.6 demonstra a interação do *Whiteboard* e o *script* Python.

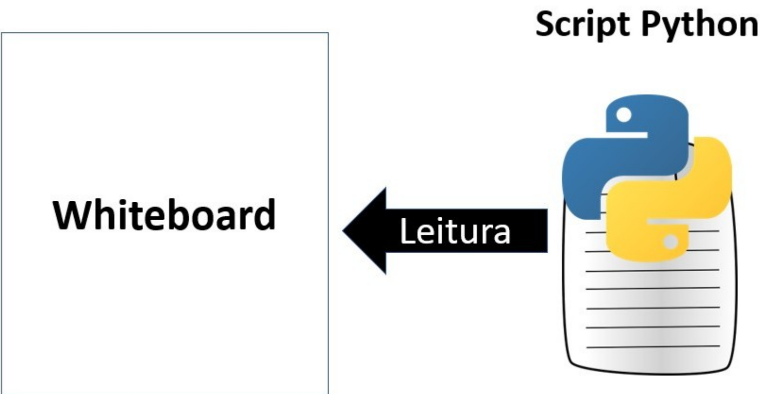

Figura 4.6: Relação de interação entre *Whiteboard* e o script Python .

O *script* Python faz uso da biblioteca `csv`[2], nativa do Python. O módulo `csv` implementa classes para ler e gravar dados tabulares em formato CSV. Neste formato, os campos de dados indicados normalmente são separados ou delimitados por uma vírgula. No *script*, o módulo `csv` é usado para armazenar dados obtidos do objeto `self.whiteboard` e gerar dois arquivos CSV, independentes e em paralelo: um arquivo com o registro do *index* da qualidade selecionada de cada segmento, chamada `qi_log`, e outro com os registros de ocupação do *buffer* ao longo do tempo, chamado `buffer_log`. Os dados utilizados na escrita do arquivo `qi_log` são obitdos da função `get_playback_qi` e estão separados em "qi", "segment" e "time_played", nesta ordem. E, os dados utilizados na escrita do arquivo `buffer_log` são obtidos da função `get_playback_buffer_size` e estão separados em "buffer_size" e "time", nesta ordem. Ambas as funções são do objeto `self.whiteboard`, detalhadas na seção 3.9.3.

Implementando o Script Python no pyDash

Para a fase de implementação do *script* no *pyDash*, a classe `Player` foi mais uma vez escolhida para comportar seu funcionamento. A Listagem 4.4 demonstra a implementação de um objeto da classe `LogWriter` na classe `Player` e sua incialização em uma *thread*.

```
1 ...
2 import threading
3 from utils.log_writer import LogWriter
4 ...
5 class Player(SimpleModule):
6     def __init__(self, id):
```

[2] https://docs.python.org/3/library/csv.html

```
7         ...
8         self.log_writer = LogWriter()
9         self.log_writer_thread = threading.Thread(target=self.log_writer
    .run)
10        self.log_writer_thread.start()
11        self.all_segments_played = False
12        ...
```

Listing 4.4: Código Python que instancia um objeto da classe *LogWriter* e a inicia em uma nova *thread*

O uso de uma *thread* para a execução da função `run` do objeto `log_writer`, mostrado no código 4.4, justifica-se devido ao *script* desempenhar uma função não prioritária. Ou seja, o *script* executa funções em segundo plano que não necessitam de grande prioridade para o funcionamento do cliente DASH. Além disso, o processo de leitura e escrita de dados da função `run` é de longa duração contida em uma estrutura de repetição, que se executada na *main thread*, ou *thread* principal (onde o programa principal está rodando), vai parecer que seu programa parou de responder. Isso ocorre pois o processo `run` aguarda pela leitura dos dados gravados no *Whiteboard* de todos os segmentos antes de encerrar. Tais dados são gerados/escritos apenas na *thread* principal.

O encerramento da nova *thread* acontece no método `finalization` da classe `Player`. Nesse ponto, a *flag* `self.all_segments_played`, que indica se todos os segmentos já foram tocados e condição de parada do loop da função `run`, já está como `True`. Ou seja, indica que a sessão de *streaming* encerrou e não há mais dados sendo gerados no *Whiteboard*. Dessa forma, *thread* do *script* é encerrada com os dados de registro de todos os segmentos ao longo da execução do programa cliente DASH.

4.2.2 Funcionamento

Atualmente, é possível encontrar bibliotecas de *software* para criação de gráficos e visualizações de dados em geral, como Matplotlib, MatLab, gnuplot, dentre outros. Em comum, eles demandam a utilização de uma estrutura de Interface Gráfica de Usuário (do Inglês, *Graphical User Interface* - GUI) para a exibição dos gráficos na tela. Dentres as opções disponíveis, foi escolhida a biblioteca Matplotlib[3] no projeto de aplicação que produzisse gráficos dinâmicos, pois ela é escrita na linguagem de programação Python e já estava sendo utilizada na plataforma *pyDash*.

O projeto de exibição de gráficos dinâmicos é relativamente simples. Primeiramente, é preciso que dados sejam gerados dinâmicamente, ou seja, em tempo real. Depois, é necessário que uma aplicação para criação de gráficos faça a leitura desses dados e atualize,

[3] https://matplotlib.org/

de tempos em tempos, a exibição do gráfico a cada novo dado lido. Na plataforma *pyDash*, o processo que gera dados já é implementado através da estrutura de *Whiteboard*. Dessa forma, foi pensado inicialmente em incorporar no *pyDash*, assim como foi com o reprodutor de mídia DASH, uma aplicação que lesse os dados do *Whiteboard* e que realizasse a exibição de gráficos dinâmicos em uma nova *thread*, ou seja, em segundo plano.

Entretanto, a proposta de incorporar a aplicação em uma nova *thread* e que fosse executada juntamente com a aplicação cliente DASH se mostrou inviável, uma vez que a maioria das estruturas de GUI exige que todas as atualizações na tela e, portanto, seu loop de evento principal, sejam executadas na *main thread* (programa principal). Essa limitação torna impossível enviar atualizações periódicas de um gráfico para uma *thread* de segundo plano. Dessa forma, foi cogitado inserir a aplicação exibidor de gráficos dinâmicos na *main thread* da plataforma *pyDash*, porém a complexidade de inserí-la era grande. Logo, optou-se por preservar a estrutura do funcionamento da mesma.

Pelos motivos apresentados anteriormente, foi proposta uma solução alternativa. A aplicação de gráficos dinâmicos desenvolvida passaria a ser uma aplicação independente, ou seja, seu processo teria que ser executado fora da execução do processo principal, porém ainda como um recurso da plataforma *pyDash*. Nesse sentido, a aplicação precisaria ler os dados a partir de uma fonte externa, pois não conseguiria acessar os dados do *Whiteboard* diretamente. Por este motivo, foi criado o *script* Python, descrito na subseção anterior, capaz de ler os dados gerados no *Whiteboard* e escrevê-los em um arquivo no formato CSV. O motivo de usar o formato CSV é devido a compatibilidade da biblioteca `pandas` [4], utilizado na aplicação, em realizar leituras de arquivos neste formato. A Figura 4.7 demonstra a forma de funcionamento da aplicação de exibição de estatísticas em tempo real.

Na programação, a diferença de um gráfico estático para um gráfico dinâmico está na taxa de atualização do gráfico que está sendo exibido. É como interpretar cada exbição de gráfico como um *frame*. Logo, o que confere a sensação do gráfico ser dinâmico, ou seja, comportar-se como uma animação, é o intervalo em que determinado gráfico exibido é atualizado.

Resumidamente, a aplicação foi idealizada de forma a exibir uma única figura com dois *plots* (espaços para a exibição do gráfico) que são atualizados constantemente. Um *plot* é responsável por exibir o *index* da qualidade selecionada *versus* segmento. E, o outro *plot* por exibir o tamanho da ocupação do *buffer* (segundos) *versus* tempo (segundos). O código 4.5 demonstra a estrutura da implementação da aplicação criada.

```
1 import pandas as pd
2 import matplotlib.pyplot as plt
```

[4] https://pandas.pydata.org/

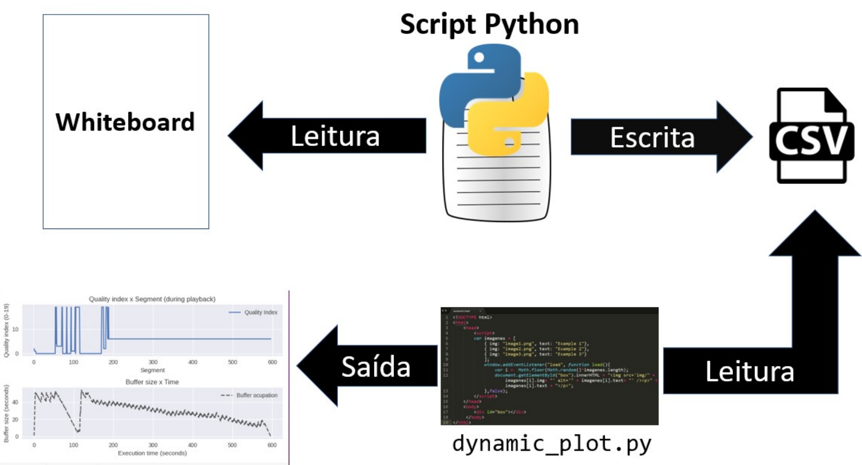

Figura 4.7: Funcionamento da aplicação de exibição de gráficos dinâmicos.

```
3 from matplotlib.animation import FuncAnimation
4
5 CSV_FILE_DIRECTORY = "./utils/"
6 plt.style.use('seaborn')
7 fig, (ax1, ax2) = plt.subplots(nrows=2, ncols=1)
8
9 def animate(i):
10     qi_data = pd.read_csv(CSV_FILE_DIRECTORY + 'qi_log.csv')
11     ...
12     ax1.plot(segment, qi, label='Quality Index')
13
14     buffer_data = pd.read_csv(CSV_FILE_DIRECTORY + 'buffer_log.csv')
15     ...
16     ax2.plot(time_buffer, buffer_size, label='Buffer ocupation', color=
       '#444444', linestyle='--')
17     ...
18
19 ani = FuncAnimation(fig, animate, interval=1)
20 plt.tight_layout()
21 plt.show()
```

Listing 4.5: Código Python da aplicação exibidor de gráficos dinâmicos

Conforme o código 4.5, os módulos utilizados na aplicação estão nas linhas 1-3. O módulo `pandas` é util na leitura dos dados dos arquivos CSV. É possível observar a operação de leitura dos arquivos nas linhas 10 e 14. O módulo `matplotlib.pyplot` é a interface de *design* dos gráficos, ou seja, ela é responsável por manipular como o gráfico

deve ser exibido. As linhas 12 e 16 demonstram o *layout* do gráfico criado. Os principais argumentos da função `plot` são os três primeiros, os quais indicam o eixo x, o eixo y e a legenda do gráfico, respectivamente.

Ainda sobre os módulos da aplicação, foi utilizada a API `FuncAnimation` do módulo `matplotlib.animation`. Na linha 15, a API é responsável por fazer a chamada da função de animação `animate(i)` dos *plots* inseridos na figura `fig`, referencidada na linha 7, com intervalo de 1 ms (indicado por `interval=1`). É na função `animate(i)` que está as operações de leitura dos dados dos arquivos CSV e o *design* dos gráficos. Logo, como novos dados são escritos no CSV durante a sessão de *streaming*, a todo momento um novo gráfico pode ser gerado com a função `animate(i)`, atualizado com a API `FuncAnimation` e exibida com as funções `tight_layout` e `show`. Este procedimento pode ser observado nas linhas 19-21 da Listagem 4.5.

4.3 Funções de Desempenho QoE

Para avaliar ainda mais o o desempenho geral e o impacto do algoritmo ABR, foram escolhidas duas funções QoE existentes. A primeira aqui denominada de **Função QoE 1** foi desenvolvida por Mok *et al.* [42]. A segunda, denominada de **Função QoE 2**, foi desenvolvida por Yin *et al.* [43]. A escolha das funções QoE foram determinadas de acordo com seu grau de relevância na literatura e de assimilação no desempenho do algoritmo ABR projetado. Logo, o objetivo nesta seção é descrever as funções de desempenho QoE escolhidas para implementação na plataforma *pyDash*, bem como sua incorporação.

4.3.1 Função QoE 1

A **Função QoE 1**, proposta por Mok *et al.* [42], é definida por:

$$Q_1 = 4.23 - 0.0672L_{ti} - 0.742L_{fr} - 0.106L_{tr} \quad (4.1)$$

Aqui, L_{ti}, L_{fr} e L_{tr} são, respectivamente, os níveis de atraso de inicialização (T_{init}), frequência de *rebuffering* (f_{rebuf}) e a média da duração de *rebuffering* (T_{rebuf}). Os níveis das pontuações usados são 1, 2 e 3, os quais representam "baixo", "médio" e "alto", respectivamente, e são mapeadas de acordo com a Tabela 4.2.

Esta função é considerada relevante na literatura por ser uma das primeiras funções QoE propostas. De acordo com [10], a frequência de *rebuffering* é considerada a principal métrica. Apesar disso, esta função deixa de fora um das principais métricas do HAS: a variação de qualidade. Por este motivo, a prómixa função QoE considerada para o trabalho pondera esta e outras métricas QoE.

Tabela 4.2: Três níveis de desempenho do cliente DASH com base nas métricas de performance (Fonte: [42]).

Nível	T_{init}	f_{rebuf}	T_{rebuf}
Baixo	0 - 1 segundos	0 - 0.02	0 - 5 segundos
Médio	1 - 5 segundos	0.02 - 0.15	5 - 10 segundos
Alto	$>$ 5 segundos	$>$ 0.15	$>$ 10 segundos

4.3.2 Função QoE 2

A **Função QoE 2**, proposta por Yin *et al.* [43], é definida por:

$$Q_2 = \frac{1}{K}\left(\sum_{k=1}^{K} r_k - \lambda \sum_{k=1}^{K-1} |r_{k+1} - r_k| - \mu Z_p - \mu_\theta Z_\theta\right) \tag{4.2}$$

Aqui, K é o número total de segmentos, r_k é taxa de bits selecionada para o segmento k, Z_p é a duração total do *rebuffering* - definida como o tempo total em que a reprodução é suspensa devido ao esvaziamento do *buffer* do cliente e Z_θ é o atraso de inicialização. Ainda, λ, μ e μ_θ são parâmetros de ponderação não negativos que correspondem a penalidades de variações de qualidade de vídeo, tempo de rebuffering e atraso de inicialização, respectivamente. Um valor relativamente pequeno de λ indica que o usuário não está essencialmente preocupado com a variabilidade da qualidade do vídeo. Quanto maior for λ, mais esforço é feito para alcançar mudanças mais suaves na qualidade do vídeo. Um valor grande de μ, em relação aos outros parâmetros, indica que um usuário está mais preocupado com o *rebuffering*. Nos casos em que os usuários preferem baixo atraso na inicialização, emprega-se um alto valor de μ_θ.

Usualmente, os valores de λ, μ e μ_θ seguem a configuração "Balanceado" [43]. Nesta configuração, o λ recebe 1. O μ é definida, por padrão, igual ao valor máximo da taxa de bits do vídeo disponibilizado. Ou seja, se a maior taxa de bits de um vídeo for 4.3 Mbps, então μ receberá o valor 4.3. E, o peso de $\mu_\theta = \mu$. É importante ressaltar que existem outras configurações, que variam de acordo as preferências do usuário, como "Evitar Instabilidade" e "Evitar *Rebuffering*" [43].

Algoritmos ABR conhecidos como MPC [43], Pensieve [44], Oboe [45], dentre outros foram projetados e avaliados de acordo esta função. Por este motivo e sua relevância na literatura, foi escolhida para implementação no trabalho.

4.3.3 Implementando Funções QoE no *pyDash*

Para a implementação das funções QoE citadas, foram utilizada as métricas já obtidas na finalização da classe `Player`. Entretanto, algumas métricas de desempenho estavam faltando para o cálculo das funções. Por este motivo, foi introduzido no *pyDash* a com-

putação de métricas QoE como: frequência de *rebuffering* e duração média e total de *rebuffering*. A métrica atraso de inicialização foi obtida a partir das estatísticas já computadas pela ferramenta.

De posse das métricas necessárias para o computo das funções, estas foram introduzidas na finalização da classe `Player`, junto com o resultado de outras métricas. A Listagem 4.6 apresenta as funções QoE incorporadas na linguagem Python.

```
1  ...
2  class Player(SimpleModule):
3      ...
4      def finalization(self):
5          ...
6          qoe_function_1 = 4.23 - (0.0672 * lti) - (0.742 * lfr) - (0.106
       * ltr)
7          ...
8          qoe_function_2 = average_video_quality - \
9                           (video_qual_vari_penalty *
10                          average_quality_variations) - \
11                          (rebuff_time_penalty *
12                          total_rebuffering_duration) - \
13                          (init_load_delay_penalty *
14                          initial_loading_delay)
```

Listing 4.6: Funções QoE implementadas na finalização da classe *Player* do *pyDash*

Nas linhas 5 e 8, estão definidas as duas funções QoE dentro do método `finalization` da classe `Player`. Os parâmetros necessários para o computo das funções também são obtidas no mesmo método, usando os conceitos de métricas QoE descritas na seção do 2.3 Capítulo 2.

Neste capítulo, apresentamos as propostas de melhoria para a ferramenta *pyDash*. Um projeto de decodificador e reprodutor de vídeo DASH com a finalidade de incorporá-lo na ferramenta, utilizando o *framework* Gstreamer, foi demonstrado. Vimos também a elaboração de uma aplicação que exibe estatísticas em tempo real e sua utilização com *pyDash*. E, por fim, a computação de funções de desempenho de QoE foram inseridas na finalização do cliente DASH. O próximo capítulo apresenta dos resultados das melhorias implementadas na ferramenta.

Capítulo 5

Resultados

Este capítulo demonstra as três melhorias descritas no Capítulo 4. A plataforma *pyDash* com as melhorias implementadas está disponível em `https://github.com/GuCosta/pydash_improvements`. Em particular, queremos mostrar que a dinâmica do reprodutor de mídia DASH segue idêntica à emulação do enfileiramento e consumo do *buffer* de reprodução, bem como a possibilidade da exibição do vídeo durante a sessão de *streaming*. Além disso, será mostrado também a aplicação de exibição de estatísticas em tempo real juntamente com a execução do *pyDash*. Por fim, será visto o resultado das funções QoE incorporadas na ferramenta. Os resultados são obtidos utilizando dois algoritmos ABR implementados por estudantes do curso de Transmissão de Dados da UnB, ofertado em 2021.

5.1 Configurações de Experimento

As demonstrações foram conduzidos usando o *dataset* indicado na Figura 3.5. O vídeo usado no trabalho é um trecho de 596 segundos do filme *Big Buck Bunny*. Ele contém 596 segmentos (1 segundo por segmento) e é codificado em vinte representações, cujas taxas de bits variam de 0.47 Mbps a 4.7 Mbps. O *pyDash* é executado em um notebook DELL Inspiron N7110 com 8GB de RAM e um processador Intel® Core™ i5-2410M 2.30GHz de 4 núcleos. Além disso, o arquivo *dash_client.json* da plataforma *pyDash* está configurado conforme apresentado na Listagem 5.1, exceto pelo campo `r2a_algorithm`. Os resultados são reportados utilizando dois algoritmos ABR no experimento: `r2a_fuzzy` [46] e `R2ADynamicSegmentSizeSelection` [47], individualmente.

```
{
    "buffering_until": 5,
    "max_buffer_size": 60,
    "playbak_step": 1,
```

```
5    "traffic_shaping_profile_interval": "5",
6    "traffic_shaping_profile_sequence": "LMH",
7    "traffic_shaping_seed": "1",
8    "url_mpd": "http://45.171.101.167/DASHDataset/BigBuckBunny/1sec/
   BigBuckBunny_1s_simple_2014_05_09.mpd",
9    "r2a_algorithm": ...,
10   "dash_media_player": "on",
11   "qoe_function_1_mok": "on",
12   "qoe_function_2_yin": "on"
13 }
```

Listing 5.1: Arquivo de configuração *dash_client.json* modificado

5.2 Demonstração do Reprodutor de Vídeo DASH

As Figuras 5.1 a 5.3 demonstram três momentos diferentes durante a excecução do *pyDash* com o reprodutor de mídia DASH habilitado. O cliente DASH das capturas de tela foram obtidas usando-se o algoritmo `R2ADynamicSegmentSizeSelection`. Nas figuras apresentadas, a tela de terminal de execução do *pyDash* segue à esquerda, enquanto a tela de exibição do vídeo segue a sua direita.

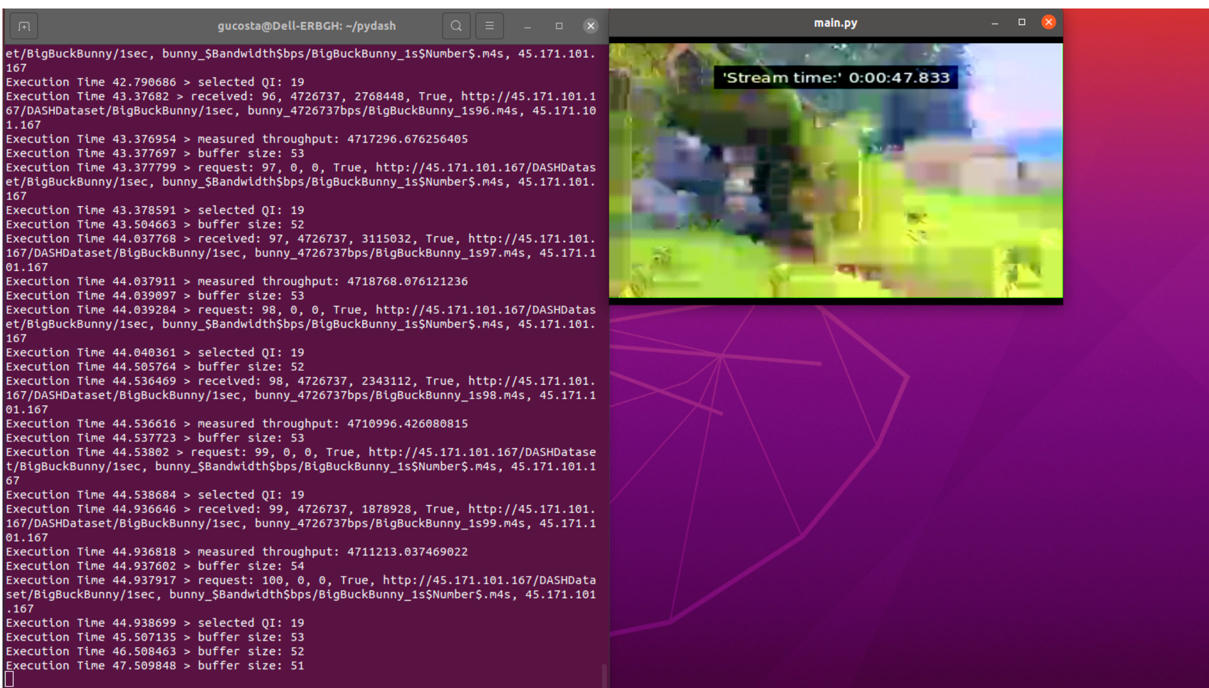

Figura 5.1: Exemplo 1 de captura de tela do *pyDash* executado com o reprodutor de mídia DASH habilitado.

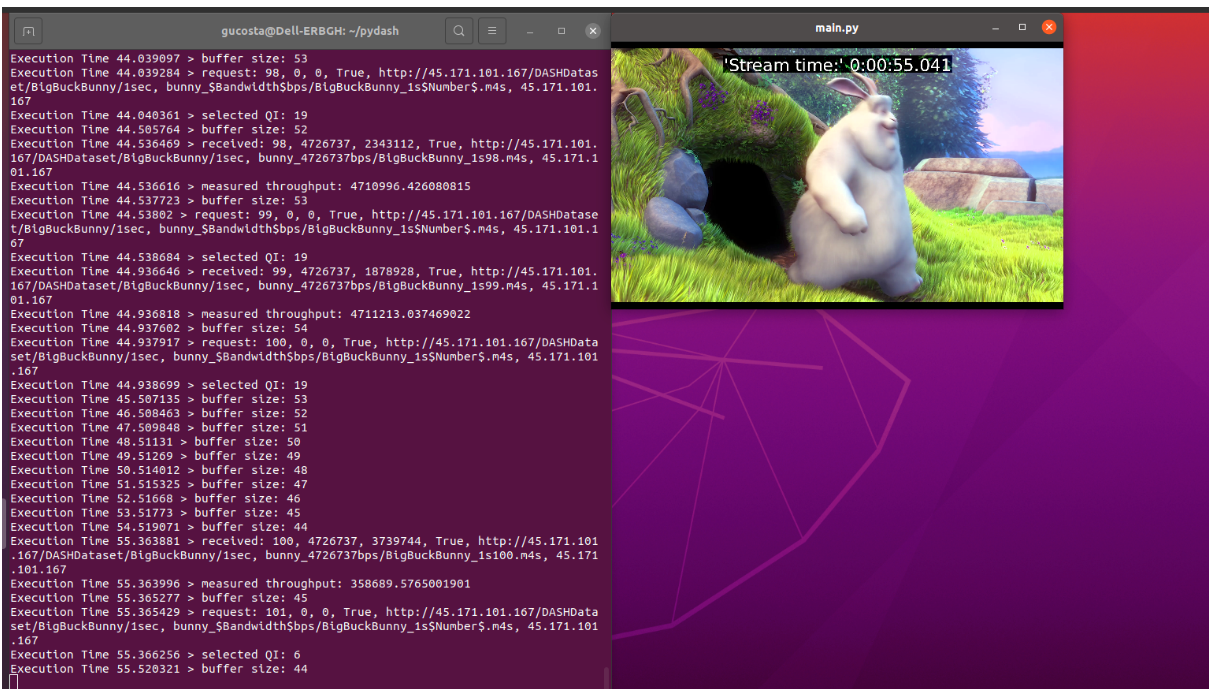

Figura 5.2: Exemplo 2 de captura de tela do *pyDash* executado com o reprodutor de mídia DASH habilitado.

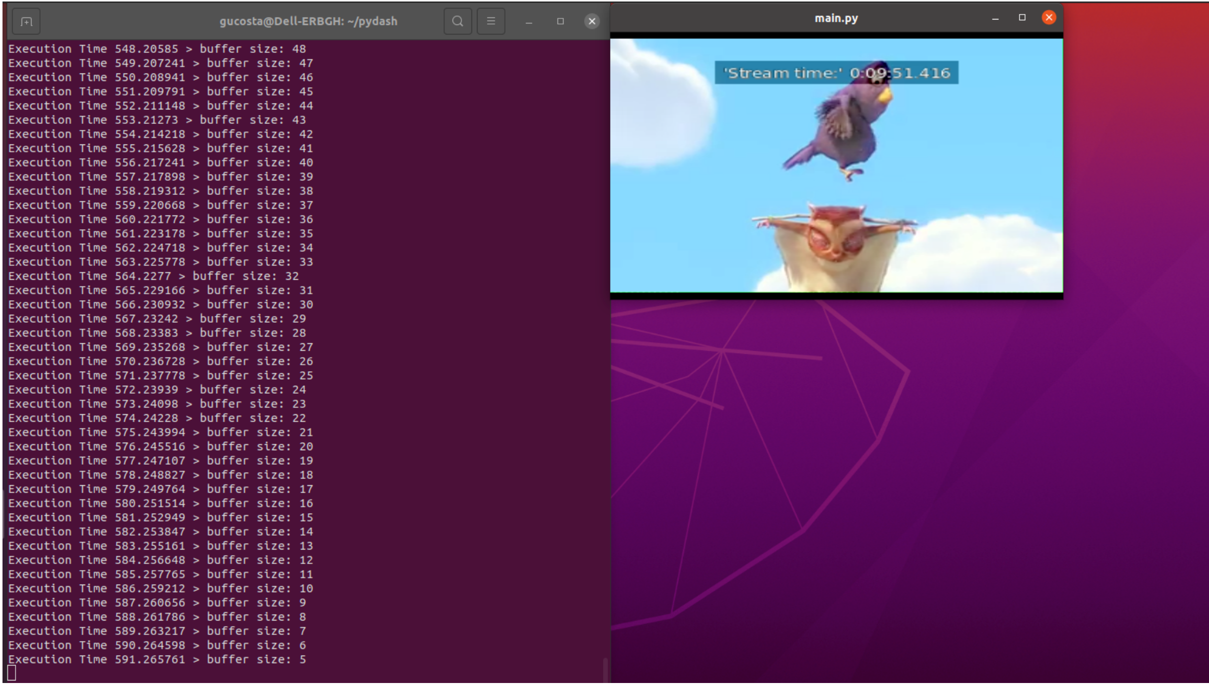

Figura 5.3: Exemplo 3 de captura de tela do *pyDash* executado com o reprodutor de mídia DASH habilitado.

Na Figura 5.1, nitidamente é possível observar que a qualidade selecionada para o segmento 47 foi baixa. Por outro lado, na Figura 5.2 podemos perceber que, momentos depois, o algoritmo fez a seleção de uma qualidade alta para o segmento 55 do vídeo. Logo, observamos que é possível visualizar mudanças de qualidade ocorridas durante a sessão de *streaming* apenas acompanhando a exibição do vídeo. A Figura 5.3, por fim, demonstra instantes finais da reprodução do vídeo. É possível perceber que, neste momento, não há mais requisições de segmentos, mas apenas a reprodução dos vídeos que estão no *buffer*.

Do início ao fim do vídeo, o reprodutor de mídia DASH não apresentou problemas de incopatibilidade com o formato de vídeo. Ou seja, todos os segmentos de vídeo obtidos da rede são recebidos e exibidos pelo reprodutor de mídia. Além disso, tanto a exibição quanto o encerramento do vídeo seguem sincronizados com a *thread* de consumo de *buffer* `handle_video_playback`.

5.3 Demonstração da Aplicação de Exibição de Estatísticas em Tempo Real

Na Figura 5.4, observamos um exemplo da execução do *pyDash* com a exibição de estatísticas em tempo real através de gráficos dinâmicos.

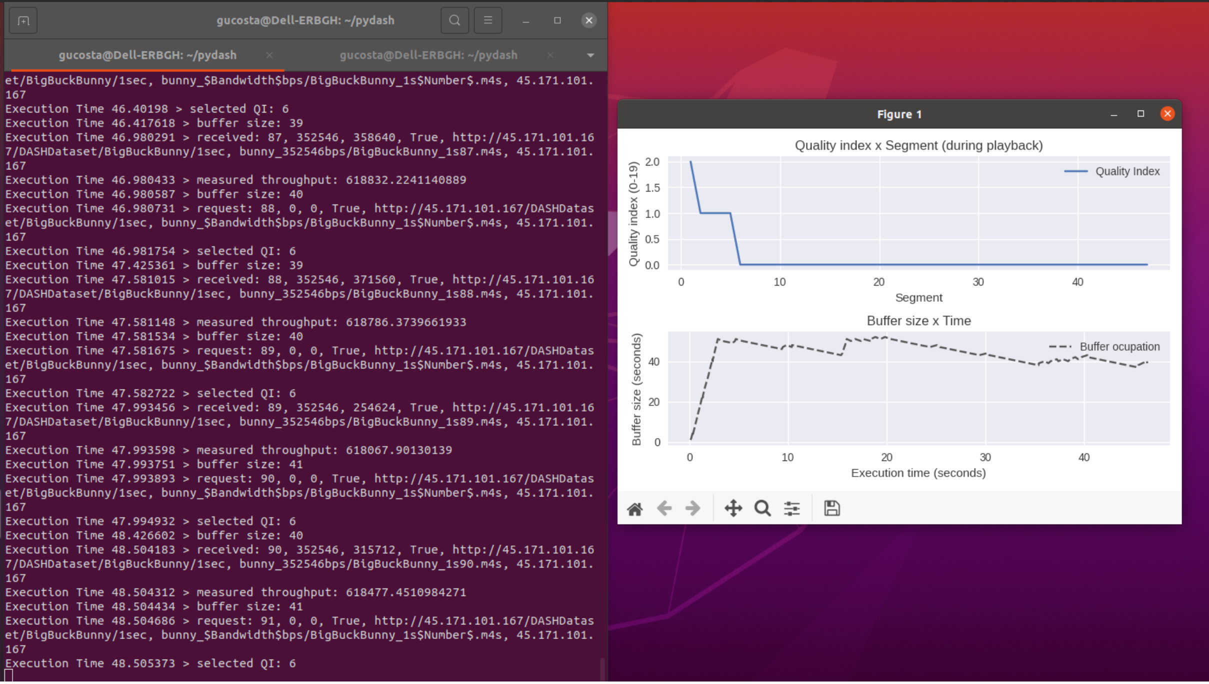

Figura 5.4: Exemplo 1 de captura de tela do *pyDash* executado com a exibição de estatísticas em tempo real através de gráficos dinâmicos.

Na figura apresentada, o terminal de execução está a esquerda, enquanto a aplicação que exibre os gráficos está a direita. Na exibição das estatísticas em tempo real, podemos perceber que são exibidos dois gráficos. No gráfico do canto superior, há a exibição do *index* da qualidade selecionada *versus* segmento. Ou seja, é demonstrado a evolução das qualidades selecionadas para cada segmento. Já no gráfico do canto inferior, é exibido a evolução da ocupação do *buffer* (em segundos) ao longo do tempo de exucução do *pyDash*.

Neste exemplo da Figura 5.4, foi usado o algoritmo `r2a_fuzzy`. Podemos percerber pela figura que o *buffer* foi preenchido com aproximadamente 45 segundos entre 0 e 4 segundos de execução. Entretanto, quando visualizamos as qualidades de vídeo reproduzidas de 0 a 45 segundos, aproximadamenmte, no gráfico superior, percebemos que o *index* de qualidade selecionada ficou em sua maioria com *index* = 0, ou seja, na pior qualidade. Este fenômeno ocorre, pois segmentos de vídeo codificados em baixas taxas de bits são transmitidas mais rapidamente pela rede cuja largura de banda disponível as supera, causando o preenchimento do *buffer* em poucos segundos.

Para que a aplicação seja executada simultaneamente com o *pyDash*, porém, é necessário abrir um novo *prompt* de comando, ou outra guia no mesmo terminal e iniciar um novo processo. A explicação deste motivo está descrito na seção 4.2 do Capítulo 4. Conforme a Figura 5.5 demonstra, o arquivo de execução denominado `dynamic_plot.py` encontra-se no mesmo diretório do aplicação *pyDash*.

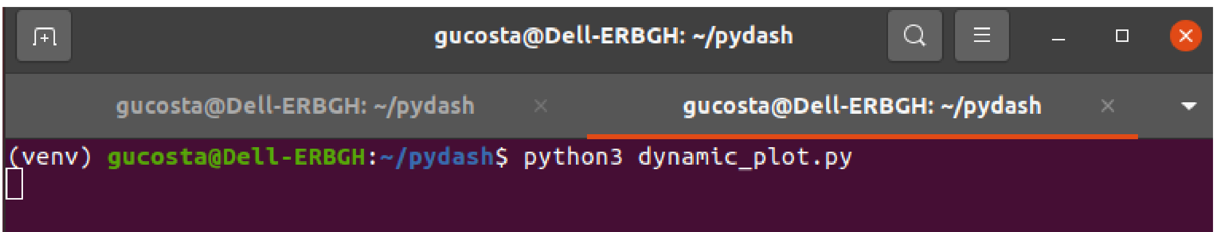

Figura 5.5: Comando de execução da aplicação que exibe gráficos dinâmicos.

5.4 Demonstração do Reprodutor de Mídia DASH com a Exibição de Estátisticas em Tempo Real

As Figuras 5.6 a 5.7 demonstram dois momentos diferentes durante a excecução do *pyDash* com o reprodutor de mídia DASH habilitado e a exibição de estátisticas em tempo real através de gráficos dinâmicos. Dessa vez, as figuras apresentam capturas de tela com o terminal de execução do *pyDash* à esquerda, a exibição do vídeo no canto superior direito e a aplicação que exibe estatísticas de monitoramento no canto inferior

direito. Aqui, as capturas de tela do cliente DASH foram obtidas usando-se o algoritmo `R2ADynamicSegmentSizeSelection`.

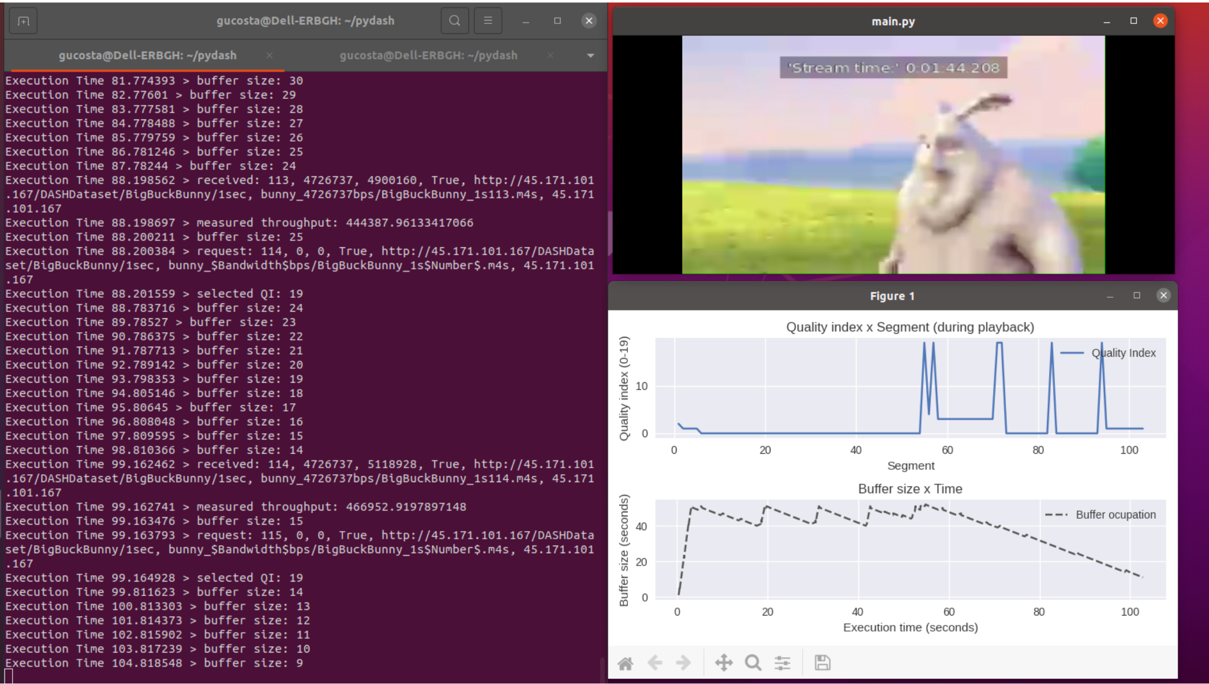

Figura 5.6: Exemplo 1 de captura de tela do *pyDash* executado com o reprodutor de mídia DASH habilitado e a exibição de estátisticas em tempo real através de gráficos dinâmicos.

Na Figura 5.6, conseguimos observar a qualidade de vídeo selecionada de aproximadamente 100 segmentos com cerca de 100 segundos de execução do *pyDash*. Primeiramente, é possível deduzir desta figura que a taxa de bits do vídeo exibida no reprodutor de mídia está em baixa qualidade, pois, além da distorção da imagem do segmento exibida, o gráfico superior indica o *index* da qualidade que está sendo mostrada.

O gráfico superior, quando exibido com reprodutor de mídia, acompanha os segmentos que estão sendo reproduzidos. Ou seja, podemos assumir que a qualidade que está sendo exibida na Figura 5.6 está entre *index* igual a 1 ou 2. Já no gráfico inferior da mesma figura, podemos visualizar a diminuição do tamanho do *buffer* entre 80 e 100 segundos de execução. Vale notar que este gráfico exibe estátisticas de *buffer* muito próximas com a do terminal. Nas linhas de execução do terminal, por exemplo, é possível notar que o último registro de tempo de execução está em 104 segundos. Da mesma forma no gráfico inferior, é possível reparar que o tempo de execução segue muito próximo de 104 segundos, indicando taxa de atualização aproximado com os registros do terminal.

A Figura 5.7 apresenta a captura de tela poucos segundos depois da captura de tela da Figura 5.6. Na Figura 5.7 podemos observar que a qualidade do vídeo melhorou. Isto pode ser constatado de duas formas. A primeira constatação está na melhoria da

imagem com relação ao da exibida na captura de tela da Figura 5.6. A segunda constatação está na indicação alta do *quality index*, que deve estar possívelmente no *index* 19, a mais alta qualidade. Além disso, podemos notar que o algoritmo ABR preencheu rapidadmente uma boa quantidade de segundos de vídeo em seu *buffer* durante seu início, porém, em compensação, selecionou segmentos de vídeo de baixímissima qualidade para os 50 primeiros segmentos de vídeo.

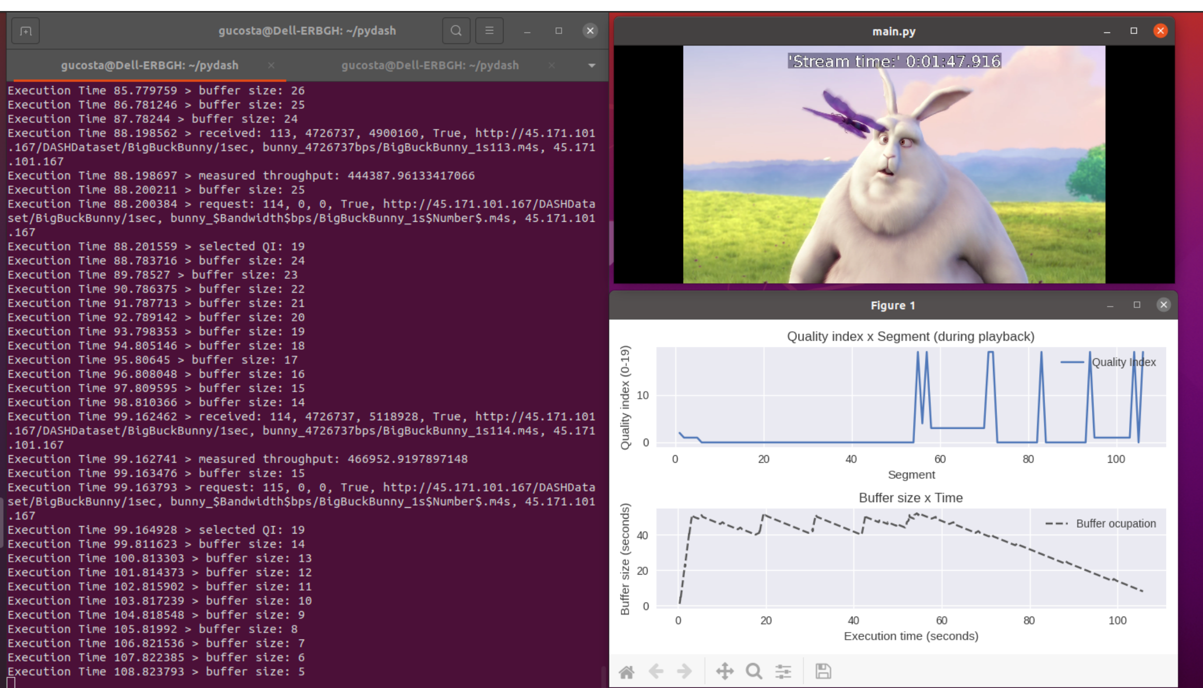

Figura 5.7: Exemplo 2 de captura de tela do *pyDash* executado com o reprodutor de mídia DASH habilitado e a exibição de estátisticas em tempo real através de gráficos dinâmicos.

5.5 Demonstração das Funções QoE

As Figuras 5.8 a 5.9 apresentam a finalização da sessão de *streaming* do *pyDash* com o algoritmos `R2ADynamicSegmentSizeSelection` e `r2a_fuzzy`, respectivamente, juntamente com indicadores de métricas e funções QoE incorporadas.

Se pararmos para analisar o resultado da **função QoE 1** dos dois algoritmos, perceberemos que foi o mesmo (3.31). Isto ocorreu, pois os índices de atraso de inicialização, a frequência de *rebuffering* e a duração média de *rebuffering* dos dois algoritmos estão no mesmo intervalo de níveis referentes da Tabela 4.2. Logo, apenas com esta função, não seria possível determinar qual algoritmo se sobressaiu em relação ao outro.

De outro modo, logo abaixo do resultado da **função QoE 1**, temos o resultado da **função QoE 2** nas duas figuras. Com esta função, percebemos que o algoritmo `r2a_fuzzy`

```
Execution Time 596.648429 > buffer size: 0
Execution Time 596.648429  thread 140560679622400 will be killed.
Finalization modules phase.
> Finalization module Player
Pauses number: 0
Average QI: 5.11
  >> Standard deviation: 3.92
  >> Variance: 15.37
Average QI distance: 0.58
  >> Standard deviation: 3.11
  >> Variance: 9.68
Initial Loading Delay: 0.61 seconds
Rebuffering Frequency: 0.0 pauses per second
Total Rebuffering Duration: 0.0 seconds
QoE 1 (Mok): 3.31
QoE 2 (Yin): 320051.61
> Finalization module R2ADynamicSegmentSizeSelection
> Finalization module ConnectionHandler
(venv) gucosta@Dell-ERBGH:~/pydash$
```

Figura 5.8: Exemplo 1 de captura de tela do *pyDash* com o resultado das métricas e funções QoE do algoritmo *R2ADynamicSegmentSizeSelection*.

teve desempenho levemente maior que o algoritmo `R2ADynamicSegmentSizeSelection`, pois quanto maior o resultado melhor é o desempenho. Agora, é fácil constatar esta diferença de desempenho, pois se observamos, por exemplo, a qualidade média de `r2a_fuzzy`, esta também foi superior a de `R2ADynamicSegmentSizeSelection`. Além disso, as variações de qualidade de vídeo de `R2ADynamicSegmentSizeSelection` foram maiores que a de `r2a_fuzzy`, o que deve ter diminuido seu índice.

```
Execution Time 596.743125 > buffer size: 0
Execution Time 596.743125  thread 139783930042112 will be killed.
Finalization modules phase.
> Finalization module Player
Pauses number: 0
Average QI: 5.65
  >> Standard deviation: 1.11
  >> Variance: 1.22
Average QI distance: 0.05
  >> Standard deviation: 0.28
  >> Variance: 0.08
Initial Loading Delay: 0.69 seconds
Rebuffering Frequency: 0.0 pauses per second
Total Rebuffering Duration: 0.0 seconds
QoE 1 (Mok): 3.31
QoE 2 (Yin): 329041.35
> Finalization module r2a_fuzzy
> Finalization module ConnectionHandler
(venv) gucosta@Dell-ERBGH:~/pydash$
```

Figura 5.9: Exemplo 2 de captura de tela do *pyDash* com o resultado das métricas e funções QoE do algoritmo *r2a_fuzzy*.

Os aprimoramentos feitos na ferramenta *pyDash* evidenciam que a análise do desempenho do algoritmo ABR durante sessões de *streaming* foram melhoradas e que elas auxiliam no estudo do funcionamento dinâmico de seleção dos segmentos de vídeo. Assim, este capítulo demonstrou os resultados das melhorias implementadas no *pyDash* e como elas podem impulsionar o entendimento de soluções ABR e seu desempenho. No próximo capítulo, são apresentadas as conclusões relativas ao desenvolvimento das melhorias e deste trabalho, bem como sugestões de trabalhos futuros.

Capítulo 6

Conclusão e Trabalhos Futuros

O problema descrito na seção 1.1 do Capítulo 1 foi resolvido como demonstrado nas sessões 4.1 a 4.3 do Capítulo 4, em que foi desenvolvido três contribuições na plataforma *pyDash* para melhorar o entendimento de algoritmos adapatativos de *streaming* pelos estudantes.

A ferramenta *pyDash*, com seu ojetivo educacional, apresentava pontos passíveis de melhoria na área de estudo e compreensão de algoritmos ABR. A incorporação de um reprodutor de mídia MPEG-DASH na ferramenta, a partir de agora, possibilita aos alunos assistirem aos vídeos e observarem o impacto na reprodução da qualidade durante as decisões de seleção do segmento do protocolo ABR implementado. Permite, também, ao estudante habilitar ou não a exibição do vídeo durante o estudo do desempenho do algoritmo ABR desenvolvido. Isto será um atrativo da ferramenta, pois quando falamos em *streaming* de vídeo, queremos saber se a qualidade do vídeo está ou não satisfatória, ainda mais quando se é o responsável pela lógica de seleção da qualidade de vídeo. O reprodutor de mídia DASH, entretanto, tem limitações. Não é possível habilitá-lo em sistemas operacionais de ambiente Windows, sendo possível apenas em máquinas que rodem Linux ou Mac OS. Além disso, o reprodutor está implementado sem um decodificador de áudio e/ou legenda. Desse modo, em *streams* de aúdio e vídeo, o reprodutor de mídia exibirá apenas o vídeo sem aúdio.

Como visto, outras duas melhorias foram implementadas no *pyDash*. Com relação ao desenvolvimento de uma aplicação de exibição de estatísticas em tempo real, o acompanhamento das escolhas de qualidade de vídeo para cada segmento com relação ao nível de ocupação do *buffer* permite visualizar o desempenho do algoritmo ABR de forma mais intuitiva e com uma interface mais amigável do que linhas no terminal de comando. Gráficos com estatísticas atualizadas dinâmicamente podem ser acompanhadas juntamente com a sessão de *streaming* do *pyDash*. Dessa forma, a proposta da representação dinâmica e em tempo real de estatísticas auxiliará no compreensão da dinâmica do algorimto adaptativo pelo estudante. Porém, esta utilidade está limitada à intenção do usuário em visualizar os

dados exibidos em gráficos dinâmcios. Ou seja, a funcionalidade não é automáticamente carregada assim que o *pyDash* é inicializado. O estudante precisa necessáriamente ter conhecimento desta utilidade e executá-lo em outra linha de comando do terminal para que a exibição de gráficos aconteça. É possível executá-lo tanto em ambiente Windows quanto Linux e Mac OS. Além disso, sua execução juntamente com o reprodutor de vídeo pode causar pequenos travamentos durante sua exibição, pois há o aumento significativo de consumo de CPU e memória RAM, devido á duas interfaces gráficas GUI estarem sendo exibidads.

Por fim, a terceira melhoria foi a implementação de computação de funções de desempenho QoE. Estas funções possiblitam indicar com mais clareza o desempenho do algoritmo ABR. Vimos que nem sempre uma função é suficiente para determinar qual solução se sobressaiu naquelas condições. Por este motivo, uma função mais completa foi adiconada para comparar as diferenças de desemepenho. Entretanto, estas funções só fazem sentido ao estudante, se ele conhecer sobre os parâmetros que as funções QoE levam em consideração, além de saberem sobre este domínio do conhecimento.

Ao longo de todo o trabalho, lições foram aprendidas. A principal delas foi o planejamento de produção dos capítulos e implementações. Sem um claro cronograma para a realização das atividades, este trabalho não teria sido concluido a tempo, visto as várias implementações e estudo sobre o tema.

Tendo em vista a incorporação das melhorias citadas, abre-se novas possibilidades de trabalhos futuros. Com o reprodutor de mídia DASH habilitado, é possível conduzir experimentos com o uso do *dataset* do *EmuStream* [4]. Com mais de 700 títulos, vídeos virtuais derivados de conteúdos de vídeo do mundo real podem ser gerados e utilizados para a avaliação de desempenho de algoritmos ABR em cenários de *streaming* mais realísticos. Além disso, o reprodutor de mídia é capaz de ser aprimorado para decodificar *streams* de vídeo compatíveis com H.265 e/ou que contenham *streams* de áudio.

Um caso particularmente relevante para a ferramenta *pyDash* seria a integração da aplicação de exibição de estatísticas em tempo real. Para não compremeter o funcionamento do programa principal do *pyDash*, a aplicação não foi testada em sua *main thread*. Isto possiblitaria a execução do cliente DASH sem a necessidade de abrir um novo processo em nova aba ou terminal. Ademais, as estatísticas de largura de banda aferidas pelo cliente DASH poderiam ser inseridas nos gráficos, a fim de oferecer uma comparação da taxa de bits selecionada pelo algoritmo ABR em relação à vazão estimada.

Além do que já fora mencionado, a ferramenta *pyDash* ainda está em fase de amadurecimento e ainda há diversas outras melhorias que podem ser desenvolvidas. Uma parte ainda que pode ser melhor explorada na ferramenta é o modelador de largura de banda. Para avaliar algoritmos ABR em cenários mais realísticos, seria de grande rele-

vância emular taxas de transferência de comunicação como, por exemplo, 5G através de arquivos de rastreamento disponível publicamente para esta finalidade. A ferramenta, também, é passível de ser aprimorada para possiblitar o projeto e avaliação de algoritmos de *streaming* de vídeo omnidirecional, tema que tem recebido grande relevância na área de *streaming* de vídeos imersivos e que pode ter como fonte de inspiração o trabalhado desenvolvido na ferramenta TAPAS-360º [1] [48].

[1] https://github.com/c3lab/tapas360

Referências

[1] Marcelo A. Marotta, Gustavo C. Souza, Maristela Holanda e Marcos F. Caetano: *PyDash – A Framework Based Educational Tool for Adaptive Streaming Video Algorithms Study*. IEEE Frontiers in Education Conference, páginas 1–8, 2021.

[2] Cisco, U.: *Cisco Annual Internet Report (2018–2023) White Paper*, 2020. https://www.cisco.com/c/en/us/solutions/collateral/executive-perspectives/annual-internet-report/white-paper-c11-741490.html, Acessado em 12 de junho de 2021.

[3] Sandvine: *The Global Internet Phenomena Report*, 2019. https://www.sandvine.com/press-releases/sandvine-releases-2019-global-internet-phenomena-report, Acessado em 12 de junho 2021.

[4] Zhang, Guanghui, Rudolf K. H. Ngan e Jack Y. B. Lee: *EmuStream—An End-to-End Platform for Streaming Video Performance Measurement*. IEEE Access, 8:669–680, 2020.

[5] Conviva: *Conviva's state of streaming q1 2020*, 2020. https://www.conviva.com/state-of-streaming/convivas-state-of-streaming-q1-2020/, Acessado em 13 de junho de 2021.

[6] Bentaleb, Abdelhak, Bayan Taani, Ali C. Begen, Christian Timmerer e Roger Zimmermann: *A Survey on Bitrate Adaptation Schemes for Streaming Media Over HTTP*. IEEE Communications Surveys Tutorials, 21(1):562–585, 2019.

[7] Bitmovin: *Mpeg-dash (dynamic adaptive streaming over http, iso/iec 23009-1)*, 2015. https://bitmovin.com/dynamic-adaptive-streaming-http-mpeg-dash, Acessado em 15 de junho de 2021.

[8] Stockhammer, Thomas: *Dynamic Adaptive Streaming over HTTP –: Standards and Design Principles*. Em *Proceedings of the Second Annual ACM Conference on Multimedia Systems*, MMSys '11, página 133–144, New York, NY, USA, 2011. Association for Computing Machinery, ISBN 9781450305181. https://doi.org/10.1145/1943552.1943572.

[9] Gómez, David, Fernando Boronat, Mario Montagud e Clara Luzón: *End-to-End DASH Platform Including a Network-Based and Client-Based Adaptive Quality*

Switching Module. Em *Proceedings of the 7th International Conference on Multimedia Systems*, MMSys '16, New York, NY, USA, 2016. Association for Computing Machinery, ISBN 9781450342971. https://doi.org/10.1145/2910017.2910638.

[10] Barman, Nabajeet e Maria G. Martini: *Qoe modeling for http adaptive video streaming–a survey and open challenges*. IEEE Access, 7:30831–30859, 2019.

[11] *dash.js*. https://github.com/Dash-Industry-Forum/dash.js. Acessado em 28 de abril de 2021.

[12] *Gpac*. https://gpac.wp.imt.fr/. Acessado em 28 de abril de 2021.

[13] *Exoplayer*. https://github.com/google/ExoPlayer. Acessado em 28 de abril de 2021.

[14] De Cicco, Luca, Vito Caldaralo, Vittorio Palmisano e Saverio Mascolo: *TAPAS: A Tool for RApid Prototyping of Adaptive Streaming Algorithms*. Em *Proceedings of the 2014 Workshop on Design, Quality and Deployment of Adaptive Video Streaming*, VideoNext '14, página 1–6, New York, NY, USA, 2014. Association for Computing Machinery, ISBN 9781450332811. https://doi-org.ez54.periodicos.capes.gov.br/10.1145/2676652.2676654.

[15] Juluri, Parikshit, Venkatesh Tamarapalli e Deep Medhi: *SARA: Segment aware rate adaptation algorithm for dynamic adaptive streaming over HTTP*. Em *2015 IEEE International Conference on Communication Workshop (ICCW)*, páginas 1765–1770, 2015.

[16] Reviakin, Aleksandr, Ahmed H. Zahran e Cormac J. Sreenan: *Dashc: A Highly Scalable Client Emulator for DASH Video*. Em *Proceedings of the 9th ACM Multimedia Systems Conference*, MMSys '18, página 409–414, New York, NY, USA, 2018. Association for Computing Machinery, ISBN 9781450351928. https://doi.org/10.1145/3204949.3208135.

[17] Raca, Darijo, Maëlle Manifacier e Jason J. Quinlan: *goDASH — GO Accelerated HAS Framework for Rapid Prototyping*. Em *2020 Twelfth International Conference on Quality of Multimedia Experience (QoMEX)*, páginas 1–4, 2020.

[18] Ott, Harald, Konstantin Miller e Adam Wolisz: *Simulation Framework for HTTP-Based Adaptive Streaming Applications*. Em *Proceedings of the Workshop on Ns-3*, página 95–102, New York, NY, USA, 2017. Association for Computing Machinery, ISBN 9781450352192. https://doi.org/10.1145/3067665.3067675.

[19] Spiteri, Kevin, Ramesh Sitaraman e Daniel Sparacio: *From Theory to Practice: Improving Bitrate Adaptation in the DASH Reference Player*. 15(2s), 2019, ISSN 1551-6857. https://doi.org/10.1145/3336497.

[20] KUROSE, James F.; ROSS, Keith W.: *Computer Networking: a top-down approach*. Pearson, 8ª edição, 2020.

[21] TANENBAUM, Andrew S.; WETHERALL David: *Redes de computadores*. São Paulo: Pearson Prentice Hall, 5ª edição, 2011.

[22] Kua, Jonathan, Grenville Armitage e Philip Branch: *A Survey of Rate Adaptation Techniques for Dynamic Adaptive Streaming Over HTTP*. IEEE Communications Surveys Tutorials, 19(3):1842–1866, 2017.

[23] Timmerer, C.: *HTTP Streaming of MPEG Media*, 2012. https://multimediacommunication.blogspot.com/2010/05/http-streaming-of-mpeg-media.html, Acessado em: 25/07/2021.

[24] Liu, Chenghao, Imed Bouazizi e Moncef Gabbouj: *Rate Adaptation for Adaptive HTTP Streaming*, página 169–174. Association for Computing Machinery, New York, NY, USA, 2011, ISBN 9781450305181. https://doi.org/10.1145/1943552.1943575.

[25] Rainer, Benjamin, Stefan Lederer, Christopher Müller e Christian Timmerer: *A seamless Web integration of adaptive HTTP streaming*. Em *2012 Proceedings of the 20th European Signal Processing Conference (EUSIPCO)*, páginas 1519–1523, 2012.

[26] Jiang, Junchen, Vyas Sekar e Hui Zhang: *Improving Fairness, Efficiency, and Stability in HTTP-Based Adaptive Video Streaming with FESTIVE*. Em *Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies*, CoNEXT '12, página 97–108, New York, NY, USA, 2012. Association for Computing Machinery, ISBN 9781450317757. https://doi.org/10.1145/2413176.2413189.

[27] Mueller, Christopher, Stefan Lederer, Reinhard Grandl e Christian Timmerer: *Oscillation compensating Dynamic Adaptive Streaming over HTTP*. Em *2015 IEEE International Conference on Multimedia and Expo (ICME)*, páginas 1–6, 2015.

[28] Huang, Te Yuan, Ramesh Johari, Nick McKeown, Matthew Trunnell e Mark Watson: *A Buffer-Based Approach to Rate Adaptation: Evidence from a Large Video Streaming Service*. SIGCOMM Comput. Commun. Rev., 44(4):187–198, agosto 2014, ISSN 0146-4833. https://doi.org/10.1145/2740070.2626296.

[29] Spiteri, Kevin, Rahul Urgaonkar e Ramesh K. Sitaraman: *BOLA: Near-Optimal Bitrate Adaptation for Online Videos*. IEEE/ACM Transactions on Networking, 28(4):1698–1711, 2020.

[30] Zhou, Chao, Chia Wen Lin, Xinggong Zhang e Zongming Guo: *Buffer-based smooth rate adaptation for dynamic HTTP streaming*. Em *2013 Asia-Pacific Signal and Information Processing Association Annual Summit and Conference*, páginas 1–9, 2013.

[31] Li, Zhi, Xiaoqing Zhu, Joshua Gahm, Rong Pan, Hao Hu, Ali C. Begen e David Oran: *Probe and Adapt: Rate Adaptation for HTTP Video Streaming At Scale*. IEEE Journal on Selected Areas in Communications, 32(4):719–733, 2014.

[32] Wang, Cong, Amr Rizk e Michael Zink: *SQUAD: A Spectrum-Based Quality Adaptation for Dynamic Adaptive Streaming over HTTP*. Association for Computing Machinery, New York, NY, USA, 2016, ISBN 9781450342971. https://doi.org/10.1145/2910017.2910593.

[33] Kjell Brunnström, Sergio Ariel Beker, et al.: *Qualinet White Paper on Definitions of Quality of Experience*. European Network on Quality of Experience in Multimedia Systems and Services (COST Action IC 1003), 2013.

[34] Tavakoli, Samira, Sebastian Egger, Michael Seufert, Raimund Schatz, Kjell Brunnström e Narciso García: *Perceptual Quality of HTTP Adaptive Streaming Strategies: Cross-Experimental Analysis of Multi-Laboratory and Crowdsourced Subjective Studies*. IEEE Journal on Selected Areas in Communications, 34(8):2141–2153, 2016.

[35] *GStreamer*. https://gstreamer.freedesktop.org/. Acessado em 28 de junho 2021.

[36] *Mininet*. http://mininet.org/. Acessado em 28 de junho 2021.

[37] *Ns-3 simulator*. https://www.nsnam.org/. Acessado em: 29/06/2021.

[38] Lima, Guilherme F., Rodrigo C.M. Santos e Roberto Gerson de Albuquerque Azevedo: *Programming Multimedia Applications in GStreamer*. Em *Proceedings of the 22nd Brazilian Symposium on Multimedia and the Web*, Webmedia '16, página 19–20, New York, NY, USA, 2016. Association for Computing Machinery, ISBN 9781450345125. https://doi-org.ez54.periodicos.capes.gov.br/10.1145/2976796.2988193.

[39] GStreamer: *GStreamer apps*, 2016. https://gstreamer.freedesktop.org/apps/, Acessado em 07 de outubro de 2021.

[40] ArchWiki: *GStreamer*, 2021. https://wiki.archlinux.org/title/GStreamer, Acessado em 07 de outubro de 2021.

[41] Foundation, X.Org: *X.Org*, 2020. https://www.x.org/wiki/, Acessado em 07 de outubro de 2021.

[42] Mok, Ricky K. P., Edmond W. W. Chan e Rocky K. C. Chang: *Measuring the quality of experience of HTTP video streaming*. Em *12th IFIP/IEEE International Symposium on Integrated Network Management (IM 2011) and Workshops*, páginas 485–492, 2011.

[43] Yin, Xiaoqi, Abhishek Jindal, Vyas Sekar e Bruno Sinopoli: *A Control-Theoretic Approach for Dynamic Adaptive Video Streaming over HTTP*. SIGCOMM Comput. Commun. Rev., 45(4):325–338, agosto 2015, ISSN 0146-4833. https://doi.org/10.1145/2829988.2787486.

[44] Mao, Hongzi, Ravi Netravali e Mohammad Alizadeh: *Neural Adaptive Video Streaming with Pensieve*. Em *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, página 197–210, New York, NY, USA, 2017. Association for Computing Machinery, ISBN 9781450346535. https://doi.org/10.1145/3098822.3098843.

[45] Akhtar, Zahaib, Yun Seong Nam, Ramesh Govindan, Sanjay Rao, Jessica Chen, Ethan Katz-Bassett, Bruno Ribeiro, Jibin Zhan e Hui Zhang: *Oboe: Auto-Tuning Video ABR Algorithms to Network Conditions*. Em *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, página 44–58, New York, NY, USA, 2018. Association for Computing Machinery, ISBN 9781450355674. https://doi.org/10.1145/3230543.3230558.

[46] Marina P. Garcia, Rafael F. Barbosa e Guilherme M. Camargo: *Algoritmo Fuzzy de Adaptação MPEG/DASH*. Universidade de Brasília, páginas 1–7, 2021.

[47] Mariana M. Cruz, Alexandre Ferreira e Yudi Yamane: *Transmissao de Dados DASH por Seleção Dinâmica de Tamanho de Segmento*. Universidade de Brasília, páginas 1–11, 2021.

[48] Ribezzo, Giuseppe, Luca De Cicco, Vittorio Palmisano e Saverio Mascolo: *TAPAS-360°: A Tool for the Design and Experimental Evaluation of 360° Video Streaming Systems*. Em *Proceedings of the 28th ACM International Conference on Multimedia*, MM '20, página 4477–4480, New York, NY, USA, 2020. Association for Computing Machinery, ISBN 9781450379885. https://doi.org/10.1145/3394171.3414541.