Departamento de Ciência da Computação (CIC/UnB)

Disciplina: Teleinformática e Redes 2

Semestre: 2021-1

FDASH: A Fuzzy-Based MPEG/DASH Adaptation Algorithm

Autores:

Matheus Augusto Silva Pinho 18/0024906

Felipe Oliveira Magno Neves 18/0016296

Luís Vinicius Capelletto 16/0134544

Prof. Lucas Costa

Turma A

Brasília, October 25, 2021

1. Introdução

O suntuoso crescimento da Internet e das tecnologias wireless ocasionaram o surgimento e a popularização dos serviços de streaming multimídia.

Em 2019, a Sandvine, empresa de inteligência de aplicativos e rede com sede em Waterloo, Ontário, Canadá, lançou um relatório que mostrou que cerca de 60 por cento de todo o tráfico na internet é proveniente do streaming de vídeos nas mais diversas plataformas. A tendência é que, ao longo dos anos, os acessos continuem a crescer.

A fim de acompanhar este crescimento, criou-se uma gama de protocolos que permitem o streaming adaptável de vídeo por HTTP, como o HTTP live streaming, HTTP dynamic streaming, MPEG dynamic streaming (MPEG DASH), entre outros.

1.1. Objetivo

Neste trabalho, busca-se implementar o algoritmo FDASH (baseado no protocolo MPEG-DASH), aplicando a Lógica Fuzzy para controlar o tempo de buffer e a resolução do vídeo entregue ao cliente, proporcionando assim, a distribuição de segmentos de alta qualidade e a reprodução de vídeos sem pausas, evitando a variação contante de resolução .

1.2. ABR

Ao passo que no passado, grande parte das tecnologias de streaming de vídeo eram baseadas em RTP com RTSP, hoje em dia, a principal técnica usada é a ABR (Adaptive Bitrate Streaming).

Os protocolos que adotam o ABR utilizam-se da ideia de um vídeo dividido em diversas partes denominadas segmentos, que são basicamente arquivos contendo metades discretas do conteúdo original. Dessa forma, no cliente, estes arquivos podem ser alinhados e remontados visando chegar novamente no arquivo completo que será reproduzido.

A sequência de segmentos é denominada profile e caracterizada pelos diversos atributos de compressão de arquivos de mídia existente como codec de vídeo, áudio, resolução, dentre outros. A partir destas características será possível definir o bitrate (fluxo de transferência de bits) do profile, que impactará diretamente na transmissão do conteúdo.

O grande diferencial do ABR é a possibilidade do cliente realizar a adaptação da chegada dos segmentos de acordo com as condições de rede existetes no local de acesso, selecionando o melhor profile disponível para download.

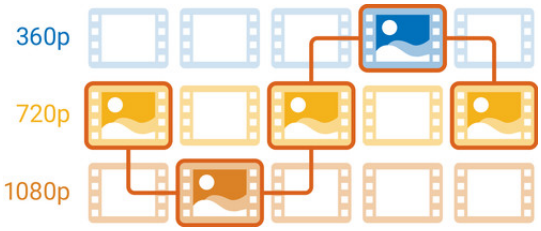

Figura 1. Adaptive Bitrate Streaming

A figura acima exemplifica a adaptação da qualidade de um vídeo baseada nas condições da rede local.

1.3. MPEG-DASH

Um pouco mais de uma década atrás, alguns protocolos proprietários estavam batalhando para chegar à larga adoção, aproveitando do declínio do RTMP para reprodução. Foi então que a MPEG (Moving Picture Experts Group) decidiu investir em um padrão aberto para competir com os protocoloa proprietários. Assim surgiu o MPEG-DASH (developed dynamic adaptive streaming over HTTP), adotado por grandes empresas de streaming como Netflix e Youtube.

Pode-se exemlificar os principais passos do processo de streaming gerado pelo MPEG-DASH da seguinte forma:

1. Codificação e segmanetação : O servidor de origem divide o vídeo em pequenos segmentos com poucos segundos de duração. Os segmentos são codificados e formatados de uma forma que possibilita a interpretação por variados dispositivos.

2. Entrega: Assim que os usuários começam a assistir algum conteúdo em uma plataforma de streaming, os segmentos são baixados para nossos aparelhos através da internet.

3. Decodificação e reprodução: A partir do momento que os aparelhos dos usuários recebem os conteúdos, eles são decodificados. A qualidade do vídeo varia para ajustar às condições de rede.

1.4. Lógica Fuzzy

A lógica fuzzy é uma abordagem computacional baseada em ”graus de verdade”, ao contrário da usual forma Booleana do ”verdadeiro ou falso” (0 ou 1) na qual se baseiam os computadores modernos.

2. Algoritmo

2.1. Descrição FDASH

2.1.1 Fuzzy Logic Controller

Uma das implementações mais usadas da Lógica Fuzzy é voltada ao Fuzzy Logic Controller, que pode ser visto na figura abaixo.

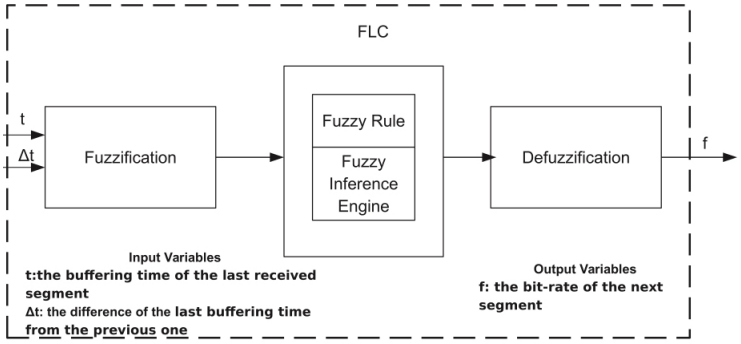

Figura 2. Estrutura do Fuzzy Logic Controller

O processo Fuzzy pode ser dividido em:

Fuzificação: Processo de associação de dados de entrada brutos ou numéricos a termos linguísticos correspondentes de acordo com a função de filiação escolhida. Por exemplo, um fuzzy controller pode receber informações de altura de pessoas e caracteiza-las com os termos linguísticos ”alta”, ”média” e ”baixa”, seguindo a lógica da função selecionada.

Implementação de uma estratégia linguística de controle: Após a fuzificação dos valores de entrada, o fuzzy controller utiliza da lógica de implementação estabelecida pela função correspondente para determinar os termos linguísticos resultantes na saída.

Defuzificação: Processo contrário à fuzzificação, após o processamento da etapa anterior, o controller irá transformar as variáveis linguísticas dadas na saída em valores numéricos brutos. Para isso, ele utilizará um dos diversos métodos matemáticos existentes, de acordo com as necessidades da aplicação.

2.1.2 Fuzificação

No nosso controller, as variáveis de entrada serão:

1 - O tempo de buffering t_i: tempo que o último segmento recebido i foi fuzificado. A partir da comparação dos valores encontrados com o tempo alvo T (35s) é possível realizar a associação com as seguintes variáveis linguística:

- **Short (S)**: Tempo de buffering corrente está abaixo do tempo alvo T
- **Close (S)**: Tempo de buffering corrente está próximo do tempo alvo T
- **Long (L)**: Tempo de buffering corrente está acima do tempo alvo T

2 - Variação tempo de buffer $\Delta t_i - t_{i-1}$: A diferença entre o tempo de buffer atual e o anterior também é analisada e associada com as seguintes variáveis linguística:

- **Falling (F)**: Diferença entre o último tempo de buffering e seu anterior está decaindo
- **Steady (S)**: Diferença entre o último tempo de buffering e seu anterior está estável
- **Rising (R)**: Diferença entre o último tempo de buffering e seu anterior está crescendo

2.1.3 Implementação de uma estratégia linguística de controle (Lógica Output)

A fim de definir as variáveis linguísticas da saída do programa (output), são usadas as seguintes regras base:

- **Regra 1**: se (short) e (falling), então (R)
- **Regra 2**: se (close) e (falling), então (SR)
- **Regra 3**: se (long) e (falling), então (NC)
- **Regra 4**: se (short) e (steady), então (SR)
- **Regra 5**: se (close) e (steady), então (NC)
- **Regra 6**: se (long) e (steady), então (SI)
- **Regra 7**: se (short) e (rising), então (NC)
- **Regra 8**: se (close) e (rising), então (SI)
- **Regra 9**: se (long) e (rising), então (I)

  Em que, R = reduce, SR = small reduce, NC = no change, SI = small increase e I = increase.

2.1.4 Defuzificação

Finalmente, o último passo do FLC é a defuzificação, que no nosso caso será feita partindo da lógica centróide.

$$f = \frac{N_2.R + N_1.SR + Z.NC + P_1.SI + P_2.I}{SR + R + NC + SI + I} \tag{1}$$

Em que:

$$I = \sqrt{{r_9}^2} \tag{2}$$

$$SI = \sqrt{{r_6}^2 + {r_8}^2} \tag{3}$$

$$NC = \sqrt{{r_3}^2 + {r_5}^2 + {r_7}^2} \tag{4}$$

$$SR = \sqrt{{r_2}^2 + {r_4}^2} \tag{5}$$

$$R = \sqrt{{r_1}^2} \tag{6}$$

Os valores casualmente usados para as variáveis são: $N_1 = 0.5$, $N_2 = 0.25$, $Z = 1$,$P_1 = 1.5$ e $P_2 = 2$

2.1.5 FDASH

Para a implementação da FDASH neste trabalho, definimos que o vídeo a ser transmitido é dividido em vários segmentos n de duração t, cada um com sua resolução de qualidade.

A taxa de transferência de um segmento i é estimada pelo cálculo:

$$r_i = \frac{b_i.t}{d_i - r_i} \tag{7}$$

Em que b_i = bitrate do segmento i, d_i = hora que o segmento i começa o download e r_i = tempo que todo o segmento foi recebido no cliente.

Através do FLC, conseguimos definir a resolução do próximo segmento de vídeo a ser determinado. Para tanto, usamos o output do controller f com a seguinte equação:

$$b_{i+1} = f.r_d \tag{8}$$

Em que r_d denota a taxa de transferência disponível na conexão, que é estimada pela média da taxa de transferência dos últimos k segmentos baixados em um período de tempo d definido.

$$r_d = \frac{1}{k}.\sum_{k}^{i=1}.r_1 \tag{9}$$

2.2. Implementação

2.2.1 FDASH sem modificações

A implementação do FDASH sem modificações é definida pela classe *R2A_FDASH*, no arquivo *r2a_fdash.py*.

```
def __init__(self, id):
    IR2A.__init__(self, id)
    self.qi = []
    self.throughputs = []
    self.request_time = 0
    self.current_qi_index = 0

    # Tempo de estimativa do throughput da conexão
    self.d = 60
    # Tempo de buffering Alvo
    self.T = 35
    # Distancia do tempo de buffering atual para o alvo
    self.set_buffering_time_membership()
    # Diferença entre os ultimos 2 tempos de buffering
    self.set_buffering_time_diff_membership()
    # Diferença entre qualidades
    self.set_quality_diff_membership()
    # Configura controlador FLC
    self.set_controller_rules()
    self.FDASHControl = ctrl.ControlSystem(self.rules)
    self.FDASH = ctrl.ControlSystemSimulation(self.FDASHControl)
```

Figura 3. Construtor da classe *R2A_FDASH*

Ilustado pela Figura 3, temos o construtor da classe *R2A_FDASH*, em que são definidos os seguintes atributos:

- **qi**: armazena a lista de qualidades disponíveis no arquivo XML
- **throughputs**: lista de tuplas das taxas de transferência (*bps*) dos segmentos de vídeo e os tempos(s) em que ocorreram
- **request_time**: armazena o tempo(s) em que a requisição de um segmento é feita
- **current_qi_index**: indice atual da lista de qualidades, com valor inicial 0
- **d**: Período de tempo em que estima-se a média dos últimos throughputs medidos, com valor fixo de 60s
- **T**: Tempo de buffering alvo utilizado como parâmetro para a variável de entrada *buff_time* do controlador FLC, possuindo o valor fixo de 35s

Além desses atributos, configura-se também no construtor o *Fuzzy Logic Controller* (FLC) através da biblioteca *scikit-fuzzy* [3] e dos métodos a seguir que a utilizam:

- **set_buffering_time_membership**: Método ilustrado pela Figura 4, o qual configura as funções de associação da variável antecedente (ou de entrada) do FLC que representa o comportamento do tempo de buffering atual em relação com tempo de buffering alvo **T** de 35s.

  Utilizando a biblioteca *scikit-fuzzy* [3], definimos uma variável de entrada do controlador FLC, com a label *buff_time* e um universo indo de 0 a 5T (175s), após isso geramos suas funções de associação do tipo trapezoidal e triangular, com quatro e três valores respectivamente, para cada parâmetro de comportamento: *Short* (S), *Close* (C) e *Long* (L).

  Para cada parâmetro de comportamento, adotou-se os mesmos valores de simulação descritos no artigo *FDASH* [1], denotando portanto as mesmas funções de associação. Por fim, armazenamos essa variável antecedente do FLC como um atributo da classe *R2A_FDASH*.

```
def set_buffering_time_membership(self):
    T = self.T
    buff_time = ctrl.Antecedent(np.arange(0, 5*T+0.01, 0.01), 'buff_time')

    # Diferença entre tempo de buffering atual com um valor alvo T
    buff_time['S'] = fuzz.trapmf(buff_time.universe, [0, 0, (2*T/3), T])
    buff_time['C'] = fuzz.trimf(buff_time.universe, [(2*T/3), T, 4*T])
    buff_time['L'] = fuzz.trapmf(buff_time.universe, [T, 4*T, np.inf,np.inf])
    self.buff_time = buff_time
```

Figura 4. Método que cria a variável antecedente *buff_time* do controlador FLC e suas funções de associação

- **set_buffering_time_diff_membership**: Método ilustrado pela Figura 5, o qual configura as funções de associação da variável de entrada do FLC que representa a diferença entre o último tempo de buffering e seu anterior, em que o tempo de buffering é definido pelo tempo em que um segmento recebido pelo cliente fica no buffer até começar a ser reproduzido pelo *player*.

  Utilizando a biblioteca *scikit-fuzzy* [3], definimos uma variável antecedente (ou de entrada) do controlador FLC, com um a label *buffering_time_diff* e um universo indo de 0 a 5T (175s), após isso geramos as funções de associação do tipo trapezoidal e triangular, para cada parâmetro de comportamento: *Falling* (F), *Steady* (S) e *Rising* (R).

  Para cada parâmetro de comportamento, adotou-se os mesmos valores de simulação descritos no artigo *FDASH* [1], denotando portanto as mesmas funções de associação. Por fim, armazenamos essa variável antecedente do FLC como um atributo da classe *R2A_FDASH*.

```
def set_buffering_time_diff_membership(self):
    T = self.T
    buff_time_diff = ctrl.Antecedent(np.arange(-T, 5*T+0.01, 0.01), 'buff_time_diff')

    # Diferencial da taxa de transferência entre os 2 ultimos tempos de buffering
    buff_time_diff['F'] = fuzz.trapmf(buff_time_diff.universe, [-T, -T, (-2*T/3), 0])
    buff_time_diff['S'] = fuzz.trimf(buff_time_diff.universe, [(-2*T/3), 0, 4*T])
    buff_time_diff['R'] = fuzz.trapmf(buff_time_diff.universe, [0, 4*T, np.inf,np.inf])
    self.buff_time_diff = buff_time_diff
```

Figura 5. Método que cria a variável antecedente *buff_time_diff* do controlador FLC e suas funções de associação

- **set_quality_diff_membership**: Método ilustrado pela Figura 6, o qual configura as funções de associação da variável de saída do FLC que representa o fator de incremento/decremento da qualidade do próximo segmento a ser requisitado pelo cliente.

  Utilizando a biblioteca *scikit-fuzzy* [3], definimos uma variável consequente (ou de saída) do controlador FLC, com um universo indo de 0 a 2.5, após isso geramos as funções de associação do tipo trapezoidal e triangular, para cada parâmetro de comportamento: *Reduce* (R), *Small Reduce* (SR), *No Change* (NC), *Small Increase* (SI) e *Increase* (I).

  Os valores utilizados como conjunto de parâmetros para cada função trapezoidal ou triangular de associação são os mesmos utilizados na simulação apresentada no artigo *FDASH* [1], denotando portanto as mesmas funções de associação, em que temos: $N2 = 0.25$, $N1 = 0.5$, $Z = 1$, $P1 = 1.5$, $P2 = 2$. Por fim, armazenamos essa variável consequente do FLC como um atributo da classe *R2A_FDASH*.

```python
def set_quality_diff_membership(self):
    N2 = 0.25
    N1 = 0.5
    Z = 1
    P1 = 1.5
    P2 = 2

    # Fator de qualidade varia de 0 a 2.5
    quality_diff = ctrl.Consequent(np.arange(0, P2+0.5, 0.01), 'quality_diff')

    # Fator de incremento/decremento da qualidade do próximo segmento
    quality_diff['R'] = fuzz.trapmf(quality_diff.universe, [0, 0, N2, N1])
    quality_diff['SR'] = fuzz.trimf(quality_diff.universe, [N2, N1, Z])
    quality_diff['NC'] = fuzz.trimf(quality_diff.universe, [N1, Z, P1])
    quality_diff['SI'] = fuzz.trimf(quality_diff.universe, [Z, P1, P2])
    quality_diff['I'] = fuzz.trapmf(quality_diff.universe, [P1, P2, np.inf,np.inf])
    self.quality_diff = quality_diff
```

Figura 6. Método que cria a variável consequente *quality_diff* do controlador FLC e suas funções de associação

- **set_controller_rules**: Método ilustrado pela Figura 7, o qual configura o conjunto de regras a ser aplicado no controlador FLC, utilizando a biblioteca *scikit-fuzzy* [3]. Utilizamos portanto os dois atributos das variáveis antecedentes *buff_time* e *buff_time_diff* e o atributo da variável consequente *quality_diff*, em que para cada combinação de parâmetros de comportamento das duas variáveis antecedentes, geramos os mesmos parâmetros de saída para a variável consequente do controlador FLC, definidos pelo artigo *FDASH* [1]. Por fim, armazenamos a lista de todas as regras do FLC como um atributo da classe *R2A_FDASH*.

```python
def set_controller_rules(self):
    rule1 = ctrl.Rule(self.buff_time['S'] & self.buff_time_diff['F'], self.quality_diff['R'])
    rule2 = ctrl.Rule(self.buff_time['C'] & self.buff_time_diff['F'], self.quality_diff['SR'])
    rule3 = ctrl.Rule(self.buff_time['L'] & self.buff_time_diff['F'], self.quality_diff['NC'])

    rule4 = ctrl.Rule(self.buff_time['S'] & self.buff_time_diff['S'], self.quality_diff['SR'])
    rule5 = ctrl.Rule(self.buff_time['C'] & self.buff_time_diff['S'], self.quality_diff['NC'])
    rule6 = ctrl.Rule(self.buff_time['L'] & self.buff_time_diff['S'], self.quality_diff['SI'])

    rule7 = ctrl.Rule(self.buff_time['S'] & self.buff_time_diff['R'], self.quality_diff['NC'])
    rule8 = ctrl.Rule(self.buff_time['C'] & self.buff_time_diff['R'], self.quality_diff['SI'])
    rule9 = ctrl.Rule(self.buff_time['L'] & self.buff_time_diff['R'], self.quality_diff['I'])
    self.rules = [rule1, rule2, rule3, rule4, rule5, rule6, rule7, rule8, rule9]
```

Figura 7. Método que cria o conjunto de regras a ser utilizado pelo controlador FLC

Após a criação das regras e das variáveis antecedentes e consequente do controlador FLC, utilizando também a biblioteca *scikit-fuzzy* [3], cria-se no construtor (3), o Sistema de Controle Fuzzy (ou controlador FLC) com o conjunto de regras criado, e logo em seguida, o Simulador do Sistema de Controle Fuzzy, que vai computar e calcular os resultados do controlador FLC, ambos definidos como um atributo da classe *R2A_FDASH*.

Tendo o simulador do controlador FLC configurado, pode-se então utiliza-lo nos demais métodos de requisição e resposta do arquivo XML e dos segmentos de vídeo presentes no servidor HTTP, presentes na interface *IR2A*. Com relação à requisição e resposta do arquivo XML, utiliza-se apenas dois métodos simples:

- **handle_xml_request**: Apenas envia a mensagem recebida pelo *player* da requisição do arquivo descritor *mpd*, enviando para a camada de baixo do *ConnectionHandler*, através do método *send_down(msg)*. Ilustrado na Figura 8.

```python
def handle_xml_request(self, msg):
    self.send_down(msg)
```

Figura 8. Método que faz a requisição do arquivo descritor *mpd*

- **handle_xml_response**: Recebe do *ConnectionHandler* o arquivo *mpd* encapsulado na mensagem, realiza a análise (*parse*) do seu conteúdo (*payload*), e em seguida armazena no atributo *qi* a lista das diferentes qualidades disponíveis no servidor HTTP, através de métodos disponibilizados pelo próprio *pydash*: ***parse_mpd***, ***get_payload*** e ***get_qi***. Ilustrado na Figura 9.

```
def handle_xml_response(self, msg):
    parsed_mpd = parse_mpd(msg.get_payload())
    self.qi = parsed_mpd.get_qi()
    self.send_up(msg)
```

Figura 9. Método que recebe a mensagem com o conteúdo do arquivo *mpd*

Com a lista de qualidades armazenada e o controlador FLC configurado, começa então o processo de requisições e respostas dos 596 do segmentos de 1s de vídeo presentes no servidor, os dois métodos que implementam a requisição de cada segmento está descrito abaixo:

- **handle_segment_size_request**: Método ilustrado pela Figura 10, no qual definimios qual será a qualidade selecionada para o próximo segmento a ser requisitado. Primeiramente utilizamos o método disponibilizado pela classe *Whiteboard*, que retorna uma lista dos tempos que cada segmento fica no buffer antes de ser reproduzido pelo *player*, visto que esses valores são utilizados como valores de entrada para o controlador FLC, e o armazenamos em um atributo chamado *pbt*.

```
def handle_segment_size_request(self, msg):
    self.pbt = self.whiteboard.get_playback_segment_size_time_at_buffer()

    if(len(self.pbt) > 1):
        self.update_troughputs()
        avg_throughput = mean(t[0] for t in self.throughputs)

        # Pega o tempo de buffering atual
        buffering_time = self.pbt[-1]
        # Calcula a diferença entre os 2 ultimos tempos de buffering
        buffering_time_diff = buffering_time - self.pbt[-2]

        # Insere as variaveis de entrada no simulador FLC e computa o resultado
        self.FDASH.input['buff_time'] = buffering_time
        self.FDASH.input['buff_time_diff'] = buffering_time_diff
        self.FDASH.compute()

        # Armazena o resultado calculado pelo simulador FLC
        factor = self.FDASH.output['quality_diff']

        # Media dos k ultimos throughtputs multiplicada por fator
        desired_quality_id = avg_throughput * factor

        # Descobrir indice da maior qualidade mais proximo da qualidade desejada
        selected_qi_index = np.searchsorted(self.qi, desired_quality_id, side='right') - 1
        self.current_qi_index = selected_qi_index if selected_qi_index > 0 else 0

    # Nos primeiros segmentos, escolher a menor qualidade possível
    msg.add_quality_id(self.qi[self.current_qi_index])
    self.request_time = time.perf_counter()
    self.send_down(msg)
```

Figura 10. Método que faz a requisição e define a qualidade do segmento de vídeo

Verificamos então se o tamanho de *pbt* é maior do que 1, já que precisamos de ao menos 2 valores para poder calcular o diferencial entre os 2 últimos tempos de buffering, que será utilizado como valor para a variável de entrada *buff_time_diff* do controlador FLC.

Portanto, nos primeiros segmentos, o id da qualidade definida na mensagem é a menor possível, selecionada através do atributo contendo o índice atual do id da qualidade *current_qi_index*, definido no construtor (3) com o valor inicial 0.

Antes de cada mensagem ser enviada para a camada de baixo, armazenamos no atributo *request_time* o tempo em que cada requisição de segmento *i* é feita, para ser utilizada posteriormente no cálculo de cada throughput *i*.

Quando o tamanho de *pbt* se torna maior que 2, começamos então a utilizar o simulador do controlador FLC para definir a qualidade do próximo segmento a ser requisitado.

Inicialmente, atualizamos a lista de throughputs, através do método *update_throughputs*, ilustrado pela Figura 11, que encarrega-se de remover os throughputs que foram medidos a mais de 60s atrás, verificando se o tempo atual *current_time* menos o tempo em que o throughput mais antigo foi medido, é maior do que o atributo do período *d* de 60s, setado no construtor (3), removendo através do método de listas *pop*, com parâmetro 0 (remove do início). Com a lista de throughputs atualizada, calculamos a média dessa lista, usando o método *mean* da biblioteca *statistics*, armazenando na variável *avg_throughput*.

```
def update_troughputs(self):
    current_time = time.perf_counter()
    while (current_time - self.throughputs[0][1] > self.d):
        self.throughputs.pop(0)
```

Figura 11. Método que atualiza a lista de throughputs para manter apenas os throughputs medidos dentro de um período (*d*) de 60s

Em seguida, através do atributo *pbt*, contendo a lista de tempos de buffering, pegamos o tempo de buffering atual (ou o último medido) e a diferença entre os 2 últimos tempos de buffering, esses valores são utilizados respectivamente como valores das variáveis antecedentes *buff_time* e *buff_time_diff* do simulador do controlador FLC.

Logo após, é calculado o resultado do FLC pelo simulador, em que a saída da variável consequente *quality_diff* representa o fator de incremento/decremento, sendo então esse fator multiplicado pela média dos throughputs *avg_throughput*, definindo então a qualidade desejada para o próximo segmento a ser requisitado, sendo armazenada em *desired_quality_id*.

Por fim, atualizamos o valor do índice atual do id da qualidadde *current_qi_index*, utilizando o método *searchsorted*, da biblioteca *numpy*, que identifica qual o índice que um elemento pode ser inserido em uma lista ordenada, mantendo a ordem da lista, com a opção de inserir a direita, caso o valor a ser inserido seja igual a um valor da lista passada como parâmetro.

Como queremos o indice da lista de qualidades *qi* que possui um id de qualidade que seja mais próximo e igual ou menor que o valor da qualidade desejada calculada *desired_quality_id*, pega-se o valor retornado pelo método *searchsorted* decrescido por 1, para pegar o índice anterior, atentando para o fato de retornar 0 caso o índice em que *desired_quality_id* seria inserido na lista *qi* seja 0.

Com o *current_qi_index* atualizado, o programa sai do bloco *if* e segue seu fluxo, atualizando o id da qualidade da mensagem do próximo segmento, salvando o tempo da requisição do segmento e enviando para a camada de baixo do *ConnectionHandler*

- **handle_segment_size_response**: Método ilustrado pela Figura 12, no qual calculamos o tempo decorrido desde que ocorreu a requisição do segmento até obter sua resposta (*round trip time*), depois armazenamos em uma tupla o tamanho em bits da mensagem dividido por esse tempo, denotando o throughput (de certa forma), e o tempo atual. Em seguida armazenamos essa tupla no atributo *throughputs* contendo a lista de throughputs, e enviamos então a mensagem para a camada superior do *player*.

```
def handle_segment_size_response(self, msg):
    t = time.perf_counter() - self.request_time
    throughput_tuple = (msg.get_bit_length() / t, time.perf_counter())
    self.throughputs.append(throughput_tuple)
    self.send_up(msg)
```

Figura 12. Método que recebe a mensagem com o segmento de vídeo

2.2.2 FDASH modificado

Observando os resultados não muito satisfatórios apresentados na seção 3.1, gerados pela implementação do algoritmo do artigo FDASH [1], foi proposta uma simples modificação de alterar o valor do período (*d*) em que a média dos troughputs é

calculada, de 60s para 5s.

O novo valor de período foi escolhido com o objetivo de fazer com o que o algoritmo consiga se adaptar melhor em relação às diferentes sequências TF de teste, em que cada perfil é aplicado durante 5s no decorrer da reprodução do vídeo. Os resultados obtidos com o algoritmo FDASH com essa modificação estão dispostos na seção 3.2.

2.2.3 FDASH alternativo

Alternativamente foi implementado um algoritmo ABR alternativo, também baseado na lógica Fuzzy, apresentado detalhadamente no artigo *Fuzzy-Based Quality Adaption Algorithm for improving QoE from MPEG/DASH Video* [2]. Ele é definido pela classe *R2A_FDASH_2* e encontra-se no arquivo *r2a_fdash_2.py*.

A implementação feita possui alguns parâmetros e valores modificados em relação ao apresentado no artigo citado, a fim de obter melhores resultados para os nossos cenários de teste, porém mantendo a mesma lógica Fuzzy.

Nesta implementação, as variáveis de entrada utilizadas para o controlador FLC são outras, constituindo do tamanho corrente do buffer, da diferenças entre os últimos 2 tamanhos do buffer e da proporção entre o *troughput* corrente e a qualidade selecionada anterior. Como esse algoritmo alternativo possui 3 variáveis de entrada, há um conjunto de 27 regras em sua lógica Fuzzy.

Semelhante ao apresentado pelo algoritmo FDASH sem modificações, ele multiplica o fator gerado na saída da Fuzzificação com a média dos throughputs, só que neste caso é utilizado uma média que foi suavizada denominada de *smooth throughput* T^s , através da equação:

$$T^S(i+1) = (1-\delta) * T^S(i) + \delta * T^M \tag{10}$$

Em que δ é igual a 0.8, e T^M é a média dos throughputs, medidos dentro de um período (d) dos últimos 5s decorridos.

Além disso, é utilizado um valor de tamanho de buffer, definido com o valor 15 (ou $1/4$ do tamanho máximo do buffer), usado de referência para indicar que o tamanho do buffer está "perigoso", chamado de *dangerous buffer size*, o qual é utilizado como parâmetro da variável de entrada que descreve o tamanho de buffer corrente e para impedir que a qualidade aumente caso o tamanho do buffer esteja inferior ao seu valor e portanto na zona perigosa.

Este novo algoritmo é apenas alternativo, portanto os detalhes e trechos de código de sua implementação foram omitidos neste relatório, visando também não deixar o relatório demasiadamente grande e seus resultados estão dispostos de forma breve na seção 3.3.

2.3. Exemplo de funcionamento

Iremos, agora, percorrer uma execução do algoritmo passo a passo em um caso hipotético, de forma a promover uma melhor intuição quanto ao funcionamento do algoritmo. A título de conveniência, as funções de pertinência a serem usadas neste exemplo serão as propostas pelo artigo FDASH [1].

Suponhamos que, inicialmente, o algoritmo receba como entrada 2 segundos para o tempo de buffering e -1 segundo para o tempo diferencial de buffering (termos explicados em Fuzzy Logic Controller). O algoritmo irá, então, executar a etapa de fuzzificação/difusão das variáveis, descrevendo-as como graus de pertencimento para cada função de pertinência.

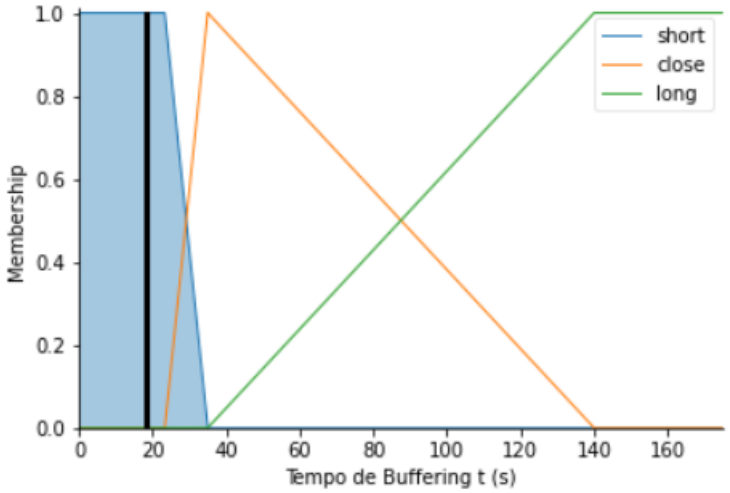

Figura 13. Fuzzificação do tempo de buffering

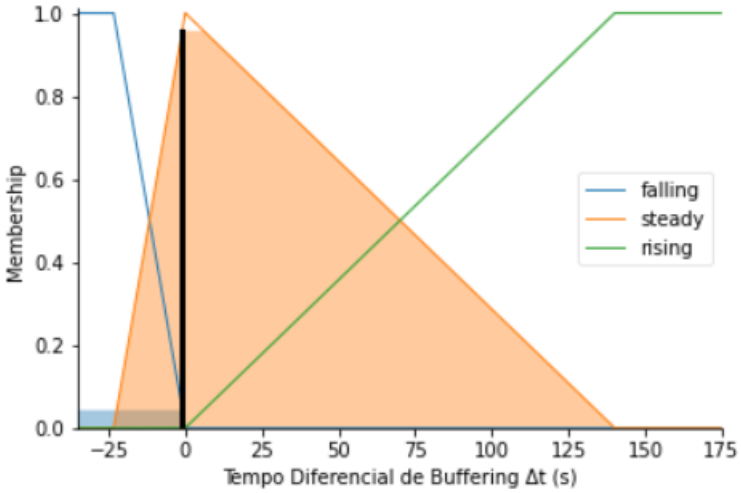

Figura 14. Fuzzificação do tempo diferencial de buffering

Observa-se na Figura 13 que o tempo de buffering recebido foi codificado como 100% short, 0% close e 0% long, e na Figura 14 que o tempo diferencial de buffering foi codificado como 4% falling, 96% steady e 0% rising.

As operações fuzzy são, então, aplicadas. Iremos, para esse exemplo, ignorar quaisquer regras não pertinentes às codificações das entrada. As operações aplicadas serão, então, a regra 1 e a regra 4:

Regra 1: if short and falling then reduce

Regra 4: if short and steady then small reduce

Substituindo as os nomes das entradas pelas entradas já codificadas e calculando os resultados, temos:

Regra 1: short = 100% and falling = 4% then reduce = 4%

Regra 4: short = 100% and steady = 96% then small reduce = 96%

Temos, então, a seguinte saída: reduce = 4%, small reduce = 96%, no change = 0%, small increase = 0% e increase = 0%. Agora o resultado sofre a defuzzificação. Existem vários modos de fazer essa etapa, podendo-se considerar apenas a classe

de saída com maior valor, ou fazendo a média ponderada dos centróides de cada classe de pertinência, ou misturas dessas abordagens. O resultado difuso da saída, assim como a saída defuzzificada podem ser vistos na Figura 15.

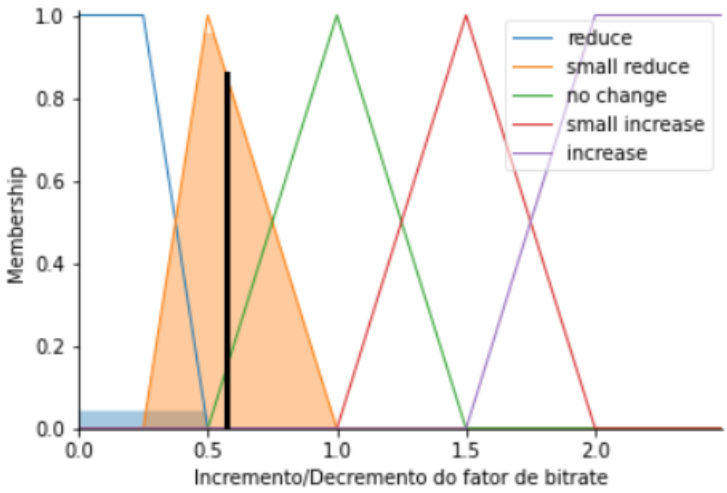

Figura 15. Saída difusa e resultado crisp interpretado

Neste caso o método da média ponderada das centróides foi usado, gerando a saída 0.57. Por fim, é aplicado o processo explicado em FDASH sem modificações, que utiliza a média móvel de troughputs e este valor de saída para definir a nova qualidade desejada. Este valor é então arredondado para a qualidade disponível (inferior) mais próxima, e a mesma é selecionada.

3. Resultados

Os dados apresentados nesta seção foram disponibilizados pela plataforma *pyDash* [5], em que estão dipostos os dados dos resultados obtidos dos testes realizados para as 4 sequências de tráfego (*traffic shaping* - TF) distintas pedidas.

3.1. FDASH sem modificações

Os dados obtidos dos testes realizados para cada sequência TF distinta no algoritmo FDASH sem modificações, com sua implementação descrita na seção 2.2.1, estão dispostos na tabela da Figura 16 abaixo.

TF Sequence	LMH	LLLLH	HHHHL	LH
Número de Pausas	2	0	48	157
Tempo médio de Pausa	2.02	0	11.86	3.88
Desvio Padrão	1.43	0	6.47	1.25
Variância	2.04	0	41.89	1.57
QI Médio	6.28	15.51	5.75	9.01
Desvio Padrão	2.7	3.77	4.97	3.8
Variância	7.27	14.18	24.7	14.46
Distância Média de QI	0.13	0.16	0.48	0.61
Desvio Padrão	0.58	0.54	1.55	1.64
Variância	0.34	0.29	2.39	2.7

Figura 16. Dados obtidos dos testes das 4 sequências de *traffic shaping* para o algoritmo FDASH sem modificações

Observando a tabela da Figura 16, pode-se inferir que o algoritmo e seus parâmetros definidos no artigo *FDASH* [1], em geral não se adaptam da melhor forma diante das diferentes sequências de TF. Supõe-se que isso ocorre em grande por conta do alto valor do período *d* de 60s, utilizado para calcular a média dos throughputs.

Como cada perfil de TF é aplicado apenas durante 5s, então a média de throughputs obtida não consegue se adaptar da melhor maneira, dado que em certos momentos da execução, pode ter um valor muito elevado ou baixo, aumentando ou diminuindo mais do que deveria a qualidade desejada calculada, causando portanto pausas ao esgotar o buffer, por selecionar uma qualidade muito alta, ou selecionando uma qualidade abaixo do que poderia.

Nos testes realizados, a sequência **LH** apresenta o maior número de pausas, igual a 157, denotando que quanto mais variar a restrição de banda, mais inclinado a gerar pausas é o algoritmo implementado.

Também pode-se concluir que o algoritmo não lida muito bem com restrições de banda mais elevadas com curtos períodos de baixa restrição, já que na sequência **HHHHL**, também há um número grande de pausas, igual a 48. Além disso, o tempo médio de pause é significantemente maior do que os ocorridos na sequência LH, isso ocorre pois o período em que essa sequência possui uma baixa restrição de banda (L), faz com que a média de troughputs fique com um valor maior e consequentemente aumentando mais do que deveria o id da qualidade selecionado para cada segmento, e como a restrição de banda é elevado, demora um tempo maior para a taxa de bits selecionada se ajustar, causando as pausas longas observadas.

Analisando a sequência **LMH**, o algoritmo quase consegue adaptar a taxa de bits para cada segmento durante a reprodução do vídeo, causando mesmo assim algumas poucas pausas, pelos motivos já citados anteriormente.

Já na sequência **LLLH**, como a restrição de banda é menor em grande parte, o algoritmo consegue adaptar a taxa de bits com relativo sucesso, sem causar pausas na reprodução do vídeo, porém pode também não conseguir a melhor média de taxa de bits possível durante toda a reprodução de vídeo.

3.1.1 LMH

A seguir encontram-se os gráficos da sequência **LMH** do FDASH sem modificações, ilustrados pela Figura 17.

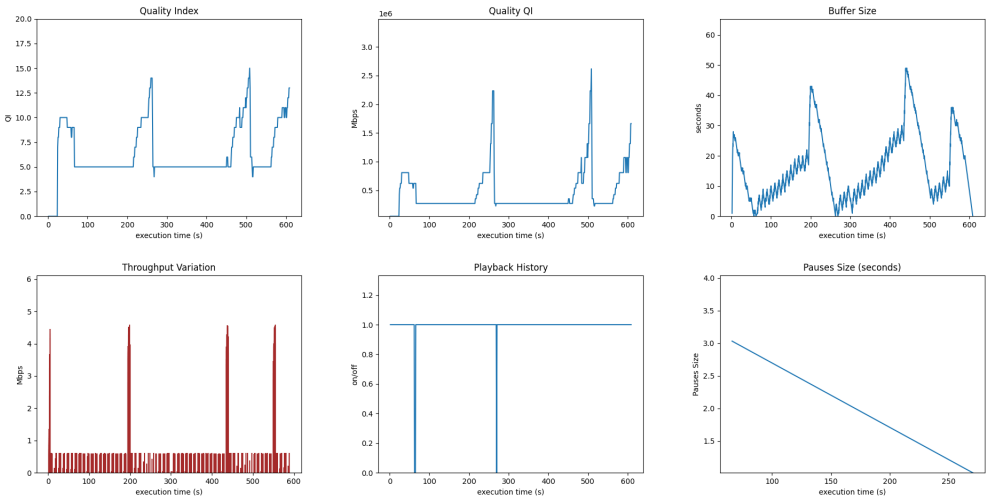

Figura 17. Gráficos da sequência LMH

Observando os gráficos da sequência LMH, ilustrados pela Figura 17, pode-se inferir que em certos momentos a qualidade escolhida pelo algoritmo é muito alta, podendo gerar algumas pausas pequenas ao esgotar o buffer, que varia bastante, mas em geral o algoritmo consegue manter uma qualidade estável, relativamente elevada e sem longas pausas durante a maior parte da reprodução do vídeo.

3.1.2 LLLLH

A seguir encontram-se os gráficos da sequência **LLLLH** do FDASH sem modificações, ilustrados pela Figura 18.

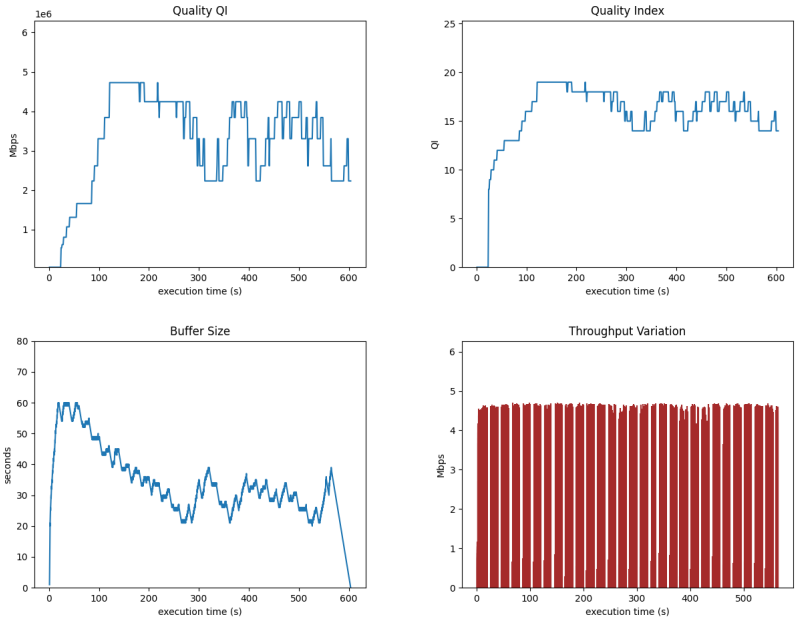

Figura 18. Gráficos da sequência LLLLH

Observando os gráficos da sequência LLLLH, ilustrados pela Figura 18, pode-ser inferir que, por se tratar a sequência com menor restrição de banda, não apresenta pausas, além de uma elavada e relativamente estável qualidade durante a reprodução do vídeo.

3.1.3 HHHHL

A seguir encontram-se os gráficos da sequência **HHHHL** do FDASH sem modificações, ilustrados pela Figura 19.

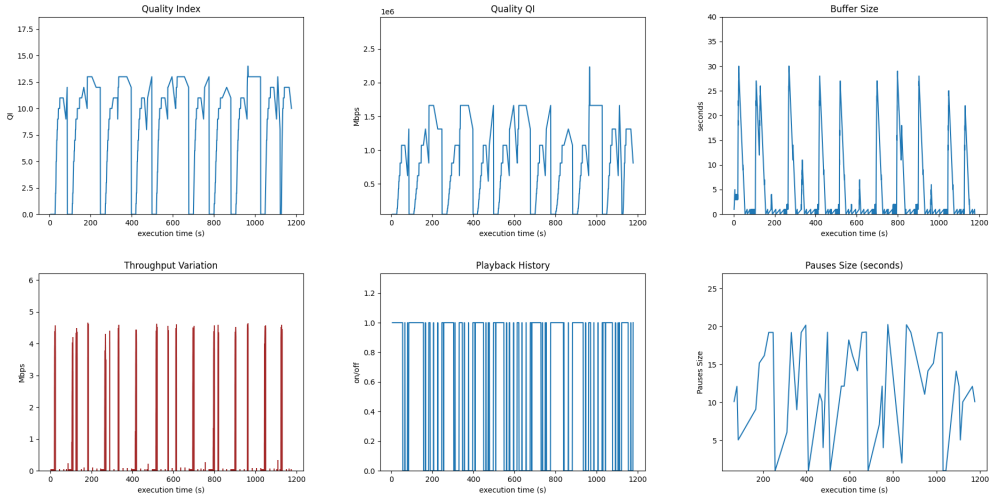

Figura 19. Gráficos da sequência HHHHL

Observando os gráficos da sequência HHHHL, ilustrados pela Figura 19, pode-se notar um número grande pausas (48 pausas), pois o período em que ele fica com uma pequena restrição de banda (L), faz com que a qualidade selecionada pelo algoritmo seja muito elevada, consequentemente causando muitas variações de qualidade entre os segmentos e muitas pausas com longa duração, pois com a alta restrição de banda durante a maior parte do tempo, há uma maior demora até o buffer encher novamente.

3.1.4 LH

A seguir encontram-se os gráficos da sequência **LH** do FDASH sem modificações, ilustrados pela Figura 20.

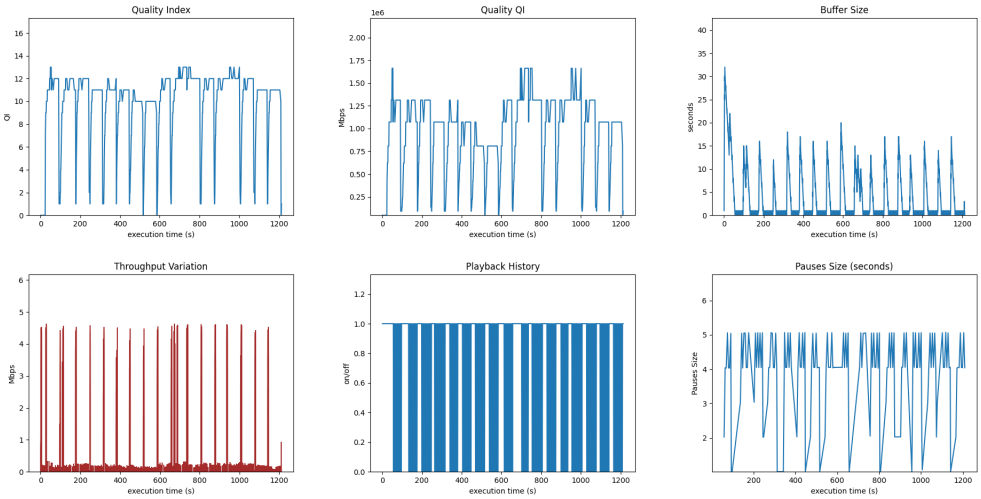

Figura 20. Gráficos da sequência LH

Observando os gráficos da sequência LH, ilustrados pela Figura 20, é notável que essa se trata da sequência em que esse algoritmo apresenta a pior performance, com o maior número de pausas (157), que apesar de possuir um tempo médio de pausa menor que o da sequência HHHHL, ainda assim apresenta pausas consideradas longas, com um tempo médio de pausa em torno de 4s. Além disso, a variação de qualidade durante a reprodução do vídeo é bem alta, devido à grande variação de restrição banda presente na sequência.

3.2. FDASH modificado

Os dados obtidos dos testes realizados para cada sequência TF distinta no algoritmo FDASH modificado, descrito na seção 2.2.2, estão dispostos na tabela da Figura 21 abaixo.

TF Sequence	LMH	LLLLH	HHHHL	LH
Número de Pausas	0	0	21	0
Tempo médio de Pausa	0	0	4.47	0
Desvio Padrão	0	0	2.54	0
Variância	0	0	6.47	0
QI Médio	5.68	15.46	4.45	8.47
Desvio Padrão	3.08	4.29	5.47	5.64
Variância	9.46	18.43	29.92	31.78
Distância Média de QI	1.36	1.03	0.66	1.58
Desvio Padrão	2.18	2.31	2.11	2.56
Variância	4.74	5.33	4.46	6.54

Figura 21. Dados obtidos dos testes das 4 sequências de *traffic shaping* para o algoritmo FDASH modificado

Observando a tabela da Figura 21, pode-se inferir que a alteração do período (d) de 60s para 5s, utilizado para se calcular a média do *throughputs*, causou durante a reprodução do vídeo, assim como esperado, uma expressiva melhoria da adaptação da taxa de bits realizada pelo algoritmo FDASH proposto, em relação aos resultados referentes ao algoritmo FDASH sem modificações, dispostos na tabela da Figura 16. Visto que a média dos *throughputs* agora consegue captar melhor as diferentes mudanças de restrição de banda na conexão.

Nota-se que apesar da média de qualidade ter diminuito levemente e as variações de qualidade terem aumentado em geral, isso é compensado pelo fato do algoritmo agora conseguir se adaptar melhor para as diferentes sequências TF, causando muito menos pausas e de menor duração.

Sendo que para a sequência **LH**, em que o número de pausas era muito elevado no algoritmo FDASH sem modificações, a pequena alteração feita causou com que não houvesse mais pausas durante a reprodução do vídeo, mantendo ainda uma boa média de qualidade. Ja ná sequência **HHHHL**, as pausas diminuiram consideravelmente (mais de 50%), além de seu tempo médio de pausa, que também decresceu significantemente.

Para as outras duas sequências **LMH** e **LLLLH**, a alteração não causou grandes modificações, apenas não gerando mais pausas na sequência LMH, que ocorriam no algoritmo FDASH sem modificações, e aumentando um pouco a variação de qualidade de ambos por uma certa margem.

3.2.1 LMH

A seguir encontram-se os gráficos da sequência **LMH** do FDASH modificado, ilustrados pela Figura 22.

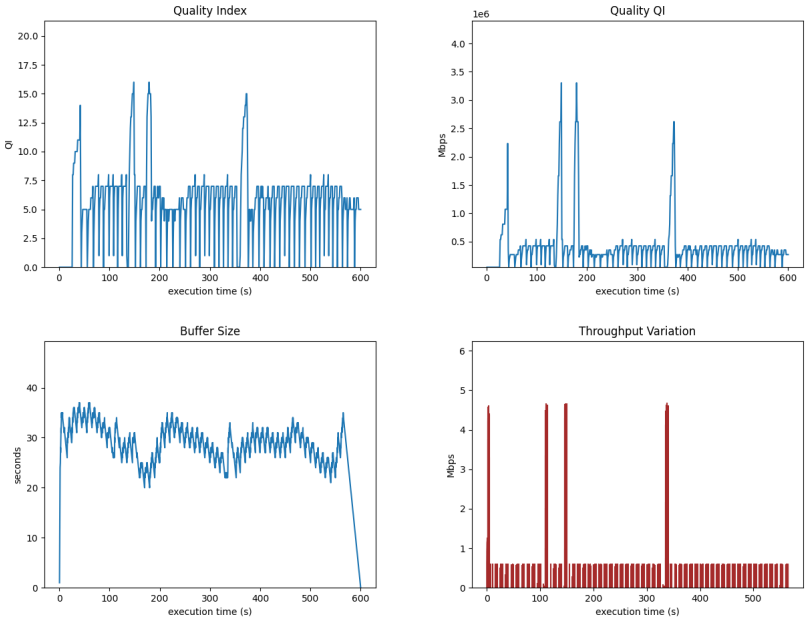

Figura 22. Gráficos da sequência LMH

Observando os gráficos da sequência LMH, ilustrados pela Figura 22, conclui-se que a variação das qualidade dos segmentos foi maior do que a dos gráficos da Figura 17 do algoritmo FDASH sem modificações, porém observando o gráfico do tamanho do buffer, nota-se que neste caso o buffer fica bem longe de se exaurir, mantendo-se estável e com um tamanho em torno de 30 segmentos, evitando assim as pausas na reprodução do vídeo, diferente do que ocorria anteriormente no algoritmo FDASH sem modificações.

3.2.2 LLLLH

A seguir encontram-se os gráficos da sequência **LLLLH** do FDASH modificado, ilustrados pela Figura 23.

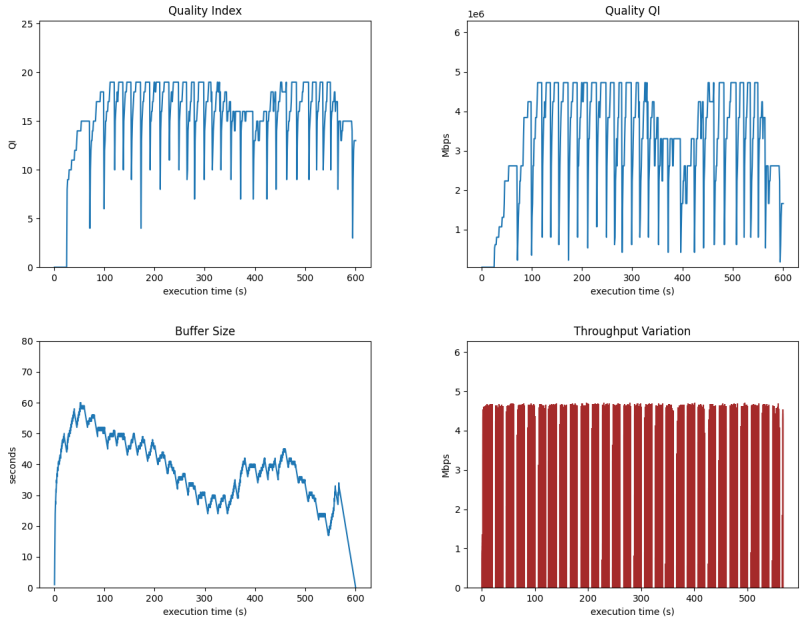

Figura 23. Gráficos da sequência LLLLH

Observando os gráficos da sequência LLLLH, ilustrados pela Figura 23, pode-se inferir que manteve-se quase o mesmo comportamento dos gráficos da Figura 18, obtidos do algoritmo FDASH sem modificações, havendo apenas uma maior variação de qualidade durante a reprodução de vídeo, o que configura uma certa queda na qualidade da experiência do usuário.

3.2.3 HHHHL

A seguir encontram-se os gráficos da sequência **HHHHL** do FDASH modificado, ilustrados pela Figura 24.

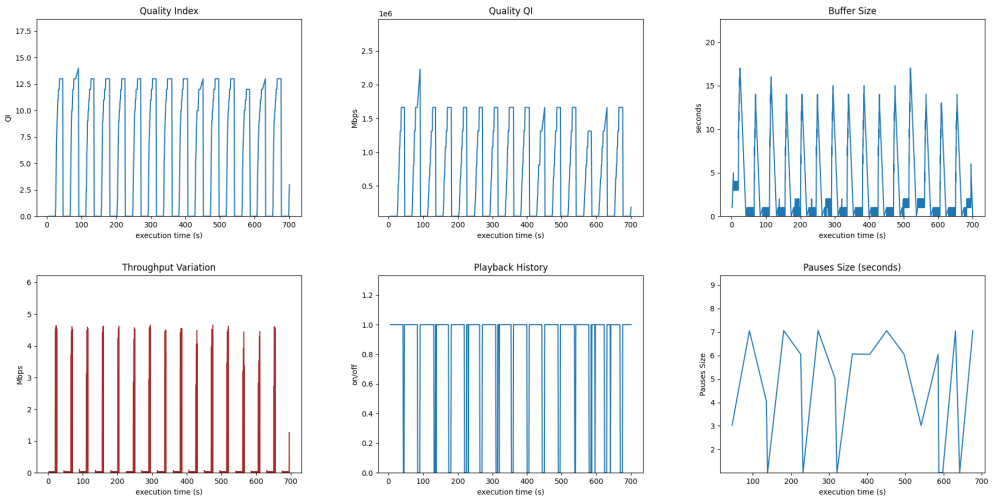

Figura 24. Gráficos da sequência HHHHL

Observando os gráficos da sequência HHHHL, ilustrados pela Figura 24, pode-se inferir que em relação ao comportamento observado nos gráficos da Figura 19, obtidos do algoritmo FDASH sem modificações, que apesar de apresentarem um valor próximo de qualidade média, há um aumento neste caso na estabilidade das variações de qualidade, sendo mais padronizadas.

Também nota-se que ocorrem muito menos pausas e que o tempo médio de pausa é muito inferior ao observado anteriormente no algoritmo FDASH sem modificações.

3.2.4 LH

A seguir encontram-se os gráficos da sequência **LH** do FDASH modificado, ilustrados pela Figura 25.

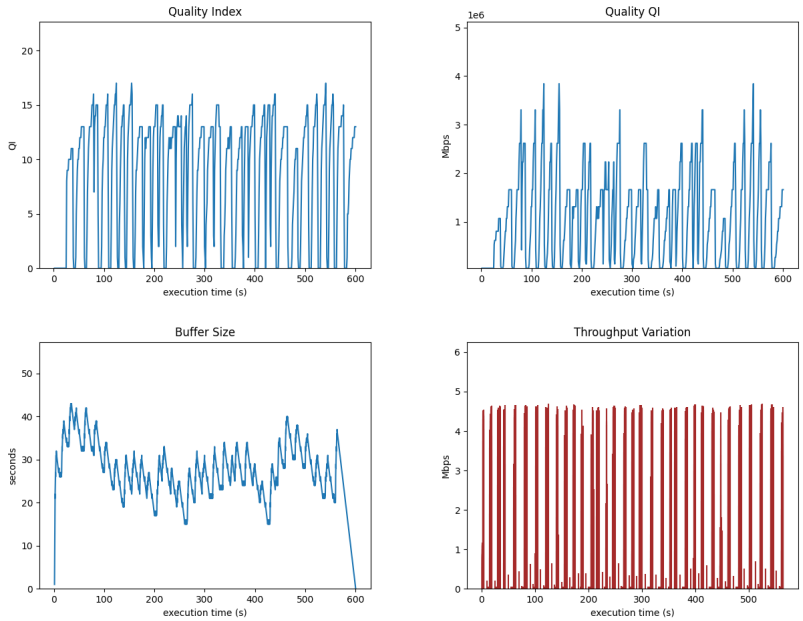

Figura 25. Gráficos da sequência LH

Observando os gráficos da sequência LH, ilustrados pela Figura 25, pode-se inferir que na sequência LH houve a maior melhoria de performance em relação ao comportamento observado nos gráficos da Figura 20, obtidos do algoritmo FDASH sem modificações. Isso pode ser notado pelo tamanho do buffer, que se mantém muito mais estável e distante de se exaurir, o que consequentemente faz com que não ocorra mais as inúmeras pausas durante a reprodução do vídeo. Além disso, a qualidade média também não é muito inferior ao observado anteriormente no algoritmo FDASH sem modificações.

3.3. FDASH alternativo

Os dados obtidos dos testes realizados para cada sequência TF distinta no algoritmo FDASH alternativo, descrito na seção 2.2.3, estão dispostos na tabela da Figura 26 abaixo.

TF Sequence	LMH	LLLLH	HHHHL	LH
Número de Pausas	0	0	0	0
Tempo médio de Pausa	0	0	0	0
Desvio Padrão	0	0	0	0
Variância	0	0	0	0
QI Médio	5.65	15.64	3.27	7.8
Desvio Padrão	3.78	3,73	5.43	6.33
Variância	14.26	13.89	29.46	40.07
Distância Média de QI	1.6	2.1	0.63	1.55
Desvio Padrão	2.49	2.0	2.33	2.45
Variância	6.18	4.02	5.41	6.02

Figura 26. Dados obtidos dos testes das 4 sequências de *traffic shaping* para o algoritmo FDASH alternativo

Observando a tabela da Figura 26, pode-se inferir que este algoritmo consegue evitar de forma satifatória, que pausas occorram em todas as 4 sequências TF dos cenários de teste, isso ocorre por conta de política de buffer, que não o deixa se exaurir muito, mesmo para altas restrições de banda. Nota-se também, que em relação à tabela da Figura 21, do algoritmo FDASH modificado, a qualidade se mantém aproximadamente a mesma. Tendo em vista o que foi apresentado nos resultados apresentados, este é o algoritmo ABR que mais consegue se adaptar de forma bem sucedida sob condições adversas de tráfego de rede e restrição de banda.

4. Conclusão

Neste projeto foi feita uma implementação do algoritmo FDASH, algoritmo ABR para um cliente MPEG-DASH. Após um breve estudo quanto ao funcionamento de lógica fuzzy e as motivações por trás do uso da mesma, ela destacou-se como boa alternativa para resolver a situação encontrada por clientes MPEG-DASH. Essa conclusão foi atingida tendo em vista que o uso de lógica fuzzy provê uma forma intuitiva de regulação suave da saída, dados parâmetros em intervalo especificado.

No decorrer da resolução do problema apresentado, diversos aspectos relativos ao conteúdo lecionado durante o semestre foram revisitados, aprofundados e consolidados. Ao elaborar uma implementação de algoritmo ABR no contexto do MPEG-DASH, foram necessários tanto conceitos relativos a protocolos de rede - como o processo de comunicação cliente-servidor, largura de banda do enlace de comunicação e round trip time (RTT) - quanto relativos ao funcionamento interno da implementação do FDASH - fuzzificação, defuzzificação e funções de pertinência.

Além disso, o problema estudado envolve diversas áreas de conhecimento, e permitiu que fossem adquiridos tanto experiência prática na área de redes, quanto entendimento em relação à interação que ocorre entre esta e outros campos - como aprendizado de máquina e controle dinâmico.

Após a implementação do algoritmo, percebeu-se que ele é extremamente sensível a alguns hiperparâmetros como ao número de membros inclusos em cada média móvel de bitrate tomada, ao tempo alvo de buffering e ao formato das funções de pertinência. A decisão tomada foi fazer duas versões do algoritmo: uma seguindo o artigo exatamente, e outra otimizando os parâmetros para os testes feitos pela biblioteca pydash. Embora não tenha sido possível - dentro do limite de tempo disponível - encontrar os parâmetros que gerem a melhor performance atingível, foram obtidos resultados substancialmente melhores ao ajustá-los levemente. Verificou-se, também, que as decisões tomadas pelo modelo, em ambos os casos, são condizentes com as especificadas nas regras aplicadas na lógica fuzzy e as entradas observadas.

Referências

[1] VERGADOS, Dimitrios J.; MICHALAS, Angelos; SGORA, Aggeliki; et al. FDASH: A Fuzzy-Based MPEG/DASH Adaptation Algorithm. IEEE Systems Journal, v. 10, n. 2, p. 859–868, 2016. Disponível em: https://ieeexplore.ieee.org/document/7368091. Acesso em: 01 Out. 2021.

[2] RAHMAN, Waqas ur; HOSSAIN, Md Delowar ; HUH, Eui-Nam, Fuzzy-Based Quality Adaptation Algorithm for Improving QoE from MPEG-DASH Video, Applied Sciences, v. 11, n. 11, p. 5270, 2021.Disponível em: https://www.mdpi.com/2076-3417/11/11/5270. Acesso em: 11 Out. 2021.

[3] SCIKIT-FUZZY, GitHub - scikit-fuzzy/scikit-fuzzy: Fuzzy Logic SciKit (Toolkit for SciPy). Disponível em: https://github.com/scikit-fuzzy/scikit-fuzzy. Acesso em: 01 Out. 2021.

[4] Report: Where Does the Majority of Internet Traffic Come From?. Disponível em: https://www.ncta.com/whats-new/report-where-does-the-majority-of-internet-traffic-come. Acesso em: 21 Out. 2021.

[5] MFCAETANO, GitHub - mfcaetano/pydash: Dash Project, GitHub, disponível em: https://github.com/mfcaetano/pydash. Acesso em: 01 Out. 2021.

[6] mpeg-DASH. Disponível em: https://www.gta.ufrj.br/ensino/eel879/vf/mpeg-dash/. Acesso em: 01 Out. 2021.

[7] What is MPEG-DASH? — HLS vs. DASH. Disponível em: https://www.cloudflare.com/pt-br/learning/video/what-is-mpeg-dash/. Acesso em: 23 Out. 2021.