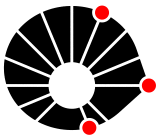

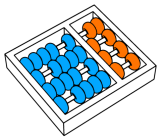

Estudo de Políticas de Enfileiramento para Transmissão de Vídeo 360º em HTTP3

G. F. Costa *C. Melo* *D. M. Casas-Velasco*
N. L. S. da Fonseca

Relatório Técnico - IC-PFG-21-01
Projeto Final de Graduação
2021 - Dezembro

UNIVERSIDADE ESTADUAL DE CAMPINAS
INSTITUTO DE COMPUTAÇÃO

Estudo de Políticas de Enfileiramento para Transmissão de Vídeo 360º em HTTP3

Gustavo Fernandez da Costa* César Melo† Daniela M. Casas-Velasco†
Nelson L. S. da Fonseca†

Resumo

O vídeo em 360º trás uma experiência totalmente diferente de um vídeo normal para o usuário, possibilitando uma imersão total no conteúdo. Contudo, essa liberdade e proximidade à imagem torna alterações de qualidade muito mais perceptíveis ao usuário. O objetivo deste trabalho é avaliar a performance de políticas de enfileiramento *First In First Out*, *Strict Priority* e *Weighted Fair Queuing* em um servidor de transmissão de vídeo 360º em HTTP3/QUIC.

Foram realizados 72 cenários de experimentos com variação de Largura de Banda, Tráfego em Plano de Fundo e Atraso para analisar as políticas e não houve distinção na qualidade da experiência do usuário entre elas.

1 Introdução

Serviços de streaming de vídeos estão cada vez mais populares e tomando o mercado de criação de conteúdo, permitindo que seus usuários acessem seu acervo através de qualquer dispositivo conectado. Contudo, os vídeos em 360º ainda permanecem fora dos principais serviços, aparecendo apenas em alguns como Youtube[1] e Facebook[2], mas sem chamar muita atenção. Grande parte da causa da tímida aparição desse formato, deve-se à dificuldade em entregar para o usuário final uma boa experiência.

Se considerarmos um vídeo como uma sequência de imagens, cada imagem estática destas é o que chamamos de *frame*. Cada *frame* de um vídeo 360º possui uma quantidade de *pixels* muito maior do que de um vídeo normal. Ou seja, são muito mais dados a serem transferidos para cada instante de vídeo exigindo uma maior largura de banda do usuário, o que muitas vezes não é possível. Outro fator agravante da necessidade de uma boa qualidade de vídeo é que estes vídeos por vezes são visualizados através de óculos de realidade virtual. Essa total imersão do usuário no vídeo aumenta o valor desse tipo de conteúdo, contudo torna variações na qualidade muito mais perceptíveis também.

Um mecanismo importante para começar a pensar melhorias na transmissão destes vídeos é o *tiling*. Isto consiste em dividir a imagem de um vídeo em quadrantes que podem

*Faculdade de Engenharia Elétrica e de Computação, Universidade Estadual de Campinas, 13083-852 Campinas, SP.

†Instituto de Computação, Universidade Estadual de Campinas, 13081-970 Campinas, SP

ter seus conteúdos acessados individualmente. Esta divisão permite que ao realizar a transmissão seja possível definir prioridades diferentes a *tiles* diferentes, de acordo com algum critério da aplicação. Para o nosso estudo iremos atribuir uma prioridade maior aos *tiles* no campo de visão do usuário e uma prioridade menor aos demais. Diminuindo a chance de que um *tile* requerido não ter sido transmitido ainda seja visto pelo usuário.

Além de definir um critério de priorização dos *tiles*, é necessário definir uma política de enfileiramento destas prioridades no momento de requisição no servidor. Dá-se então o objetivo de nosso estudo, analisar e comparar políticas de enfileiramento para *tiles* de vídeo 360º transmitidos em HTTP3[3]. As políticas definidas para o nosso estudo foram: *First In First Out* (FIFO), *Strict Priority* (SP) e *Weighted Fair Queuing* (WFQ).

2 Vídeo 360º

O vídeo em 360º permite ao usuário total controle sob a câmera enquanto o vídeo está rodando. Isso é possível graças a uma câmera especial que grava todas as direções ao mesmo tempo sem que haja um campo de visão definido. O resultado inicial é uma imagem disforme com tudo que foi gravado em 2D. É então necessário que o *player* interprete e transforme a imagem para um contexto 3D e que permita o movimento do usuário de alguma maneira. Esse controle pode ser feito tanto por uma interface gráfica, como o Youtube [1] usa, por exemplo, ou até mesmo por um óculos de realidade virtual.

Figura 1: Exemplo de câmera utilizada para captura de vídeos 360º [14]

2.1 *Video Tiling* e Segmentação

Um mecanismo importante para começar a pensar melhorias na transmissão destes vídeos é o *tiling*. Isto consiste em dividir a imagem de um vídeo em quadrantes que podem ter seus conteúdos acessados individualmente.

Veja o exemplo da Figura 4 que ilustra o *tiling* da Figura 3 em uma grade 10x20.

Figura 2: Aparência inicial de um vídeo gravado em 360º, obtido do YouTube [9]

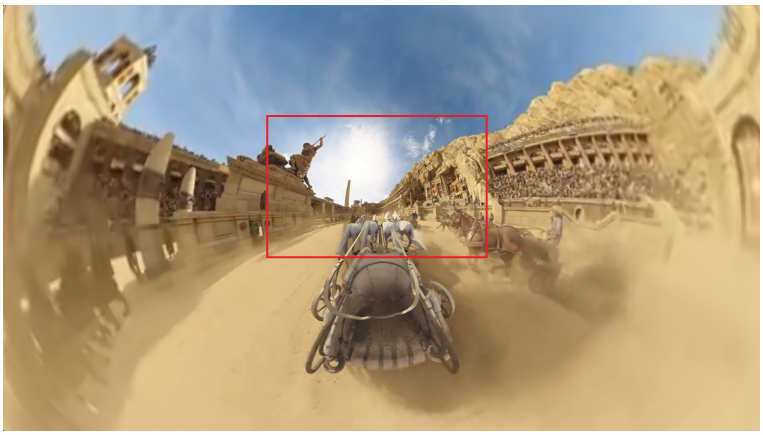

Figura 3: Campo de visão em um vídeo 360º [9]

Além disso, realiza-se também a divisão temporal do vídeo em segmentos. O tamanho do segmento é determinado pela aplicação, sendo que segmentos mais longos resultam em um tempo maior de transferência e mais curtos resultam em mais requisições ao servidor.

No nosso experimento iremos segmentar os vídeos em trechos de 1 segundo. Então para um vídeo de 30 segundos teríamos 30 segmentos. Além disso, iremos dividir cada segmento em 200 arquivos, um para cada *tile*. Isso significa que a cada segundo, 200 arquivos de *tiles* são enviados do servidor para o cliente.

Dividir o vídeo em segmentos temporais, apenas permite que o streaming ocorra. Ou seja, o conteúdo seja consumido a medida que é recebido pelo cliente. Contudo, dividir o vídeo em *tiles*, confere ao servidor uma liberdade em enviá-los em qualquer ordem. Esta ordem pode influenciar bastante na qualidade da experiência do usuário e deve ser considerada.

Como dito anteriormente, o vídeo 360º não possui um campo de visão pré-definido. A câmera grava todo o ambiente em volta e o usuário pode decidir a direção de interesse.

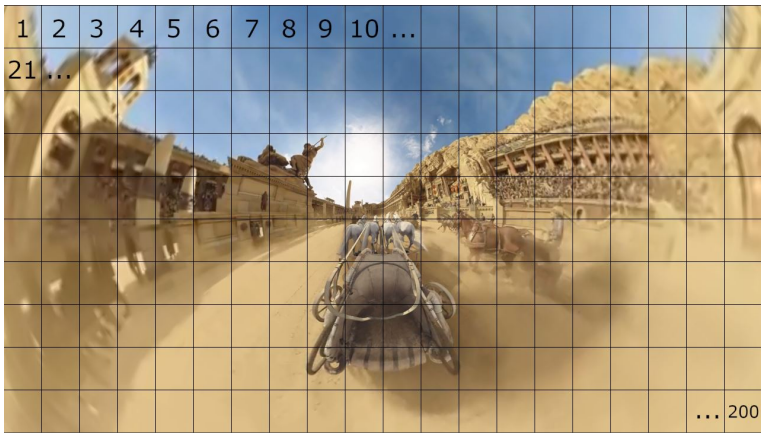

Figura 4: *Tiling* 10x20 aplicado à tela de um vídeo 360º [9]

Ou seja, se o usuário estiver olhando para o *tile* 0 e os demais *tiles* em volta, aqueles que estão diametralmente opostos não são observados. Pode ser que após alguns segundos o usuário vire para o outro lado e veja o *tile* que não era observado, porém a cada instante há um campo de visão definido e o que estiver fora dele não é importante para a experiência. Levando isso em consideração, podemos definir uma prioridade maior para *tiles* dentro do campo de visão e uma menor para aqueles que estão fora. O modo como o servidor irá gerenciar tais prioridades será discutido mais adiantes.

3 HTTP/3 e Aioquic

O HTTP/3 é a nova versão do protocolo implementada sobre o QUIC[4] (Quick UDP Internet Connections), este que foi criado por Jim Roskind na Google com a intenção de diminuir o tempo de latência em toda a internet.

O QUIC pode ser caracterizado em duas principais características: uso de UDP e criação de múltiplas *streams* de conexão entre os hosts.

Até agora, tanto a primeira quanto a segunda versão do HTTP utilizava TCP para suas conexões. Com nova versão baseada em UDP, os *round trips* presentes no TCP são eliminados e esse tráfego redundante é reduzido.[4]

Além disso, o QUIC faz uso de *multiplexing*, o que permite o envio de diferentes *streams* de dados entre servidor e clientes sem a necessidade de portas adicionais de comunicação. Isso resulta em um aumento do tráfego de dados, mas em troca de uma redução de latência. O envio paralelo de dados já estava presente no HTTP/2, contudo, por este ser baseado em TCP, o *head-of-line blocking*[5] poderia bloquear todas transações presentes. No QUIC esse problema já se torna inexistente pelo uso de UDP.

Percebe-se que tais características do novo protocolo, podem apresentar bons resultados para os problemas anteriormente tratados do streaming de vídeo 360º. Reduzir latência e aumentar o fluxo de dados para a nossa aplicação é fundamental para a entrega de uma boa experiência.

Para a nossa simulação iremos utilizar uma implementação em Python de uma API do QUIC chamada Aioquic[6]. Tal biblioteca foi escolhida por facilitar a prototipagem do servidor e cliente, e permitir que a aplicação gerencie a entrada e saída por conta própria.

4 Implementação

Para alcançar nosso objetivo do estudo da transmissão de vídeo 360º sobre QUIC, foi necessário implementar os hosts por completo[7].

O servidor foi arquitetado com a funcionalidade de receber os requests de múltiplos clientes, poder utilizar diversas políticas de enfileiramento para ordenação das prioridades e retornar os arquivos requisitados.

O cliente então deve considerar uma entrada simulada de um usuário, atribuir prioridades a esta entrada seguindo uma política pré-definida, enviar os requests ao servidor, receber os arquivos de streaming do servidor e simular um consumo destes em tempo real.

Além disso, é necessário um tratamento prévio sobre os arquivos de vídeo para que estejam prontos para transmissão. Veja abaixo mais detalhes sobre estas implementações e a etapa de preparação.

4.1 Arquivos de vídeo e entrada de usuário

Utilizamos o dataset disponibilizado pelo trabalho "*360° Video Viewing Dataset in Head-Mounted Virtual Reality*"[8]. O estudo confere dados de movimento e entrada de usuário sobre vídeos 360º encontrados no Youtube. Iremos utilizar o vídeo "*Chariot Race*"[9] e o arquivo do dataset que retrata os *tiles* observados por um usuário a cada frame. Este arquivo será nossa simulação de uma interação com um usuário.

O vídeo utilizado foi obtido como um arquivo MP4 de resolução 3840x2160. Precisamos prepará-lo para a transmissão por segmentos divididos pelos 200 *tiles*.

Primeiramente é necessário convertê-lo para o formato yuv, o que pode ser feito utilizando a ferramenta *ffmpeg*[10]. A conversão é realizada para o formato tornar-se compatível com o codificador Kvazaar[11], o responsável pela divisão do vídeo em *tiles*.

Temos então:

```
$ ffmpeg -i original_file.mp4 converted.yuv
$ kvazaar -i converted.yuv --input-res 3840x2160
-o output_file.hvc
--tiles 10x20 --slices tiles
--mv-constraint frametilemargin
--bitrate 5000000 --period 30 --input-fps 30
```

Os parâmetros utilizados:

- -i: Arquivo de entrada no formato yuv.
- - -input-res: Resolução do arquivo de entrada.
- -o: Arquivo de saída no formato hvc.

- - -tiles: Dividir a tela em *tiles* uniformes de largura x altura.
- - -slices: Determina o tipo de divisão.
- - -mv-constraint: Modo de restrição dos vetores de movimento.
- - -bitrate: Bitrate em bps do arquivo de saída.
- - -period: A cada quantas imagens uma é "intra".
- - -input-fps: FPS do arquivo de entrada.

Esse processo resulta em um arquivo codificado do tipo HVC dividido nos *tiles* desejados. Contudo ainda é necessário separar os *tiles* em diferentes arquivos e também segmentá-los para envio. Para isso, a segunda etapa é utilizar o empacotador MP4Box[12]. Utilizando-o primeiramente para converter o arquivo hvc para um mp4 com divisão por *tiles* e em seguida segmentá-lo em trechos de 1 segundo cada. Veja os comandos a seguir:

```
$ MP4Box -add output_file.hvc:split_tiles -new video_tiled.mp4
$ MP4Box -dash 1000 -rap -frag-rap -profile live
-out dash_tiled.mpd video_tiled.mp4
```

Parâmetros da segmentação:

- -dash: Duração dos segmentos em ms.
- -rap: Garante que os segmentos comecem com Random Access Points.
- -frag-rap: Garante que todos fragmentos comecem com Random Access Points.
- -profile: Especificação do perfil da segmentação. Live é utilizado para streaming.
- -out: Arquivo de saída.

Ao fim desta última etapa teremos nossos arquivos prontos para envio. Para um vídeo de 60 segundos temos sua divisão temporal em 60 segmentos de 1 segundo. Cada segmento é dividido em 200 *tiles*, resultando em 12000 arquivos de mídia.

4.2 Servidor

Tanto o servidor com o cliente foram implementados em Python utilizando o aioquic para estabelecer a conexão.

O servidor escuta a porta e endereço determinado no momento de execução para pedidos de conexão e permite a requisição por múltiplos clientes concorrentes.

Os requests recebidos possuem o ID do cliente, o *tile* e segmento requisitados e a prioridade atribuída. Com estas informações, então, o arquivo localizado no servidor é enviado. Um dos primeiros pontos de implementação foi a fila de entrada destas requisições, que define a ordem de atendimento dos requests do cliente. Como atribuímos diferentes prioridades para as requisições, podemos implementar diversas políticas de enfileiramento e comparar seus resultados.

4.2.1 Políticas de Enfileiramento

As políticas de enfileiramento utilizadas estão definidas a seguir.

A *First In First Out* (FIFO) é uma fila ordenada por ordem de chegada. Os *tiles* cujos request são recebidos primeiro pelo servidor, serão enviados primeiro. Para ser utilizado em nosso experimento, foi utilizada a estrutura *Queue* da biblioteca Asyncio[15] que é uma estrutura do tipo FIFO.

A *Strict Priority* (SP) é uma fila que utiliza de prioridades para a ordenação, de modo que as prioridades maiores devem ser completamente atendidas antes da baixa. No exemplo do nosso experimento, temos duas prioridades: alta para um *tile* no campo de visão e baixa para os demais. Então no SP, todos os *tiles* de alta prioridade serão enviados antes daqueles de baixa prioridade. Implementamos a SP através de uma estrutura de *Heap* para retirar os elementos de maior prioridade com um custo computacional menor.

Por fim, a *Weighted Fair Queuing* (WFQ) trata-se de uma política em que o algoritmo divide o canal de transmissão de acordo com os pesos atribuídos a cada fluxo. No nosso caso, cada "peso"seria a prioridade dos *tiles* requisitados e o "fluxo"seria a própria transmissão dos arquivos. Como a transmissão não é feita paralelamente, o algoritmo simula esta divisão ao determinar a ordem de entrada e saída da fila de requisições. Por exemplo, se atribuirmos aos pacotes de alta prioridade um peso de 0,75 e aos de baixa prioridade, 0,25. A cada 4 pacotes enviados, 3 serão de prioridade maior e 1 menor. Para a implementação do WFQ utilizamos o algoritmo desenvolvido no estudo "Differentiated Service Queuing Disciplines in NS-3"[13] com pequenas alterações para o nosso cenário.

4.2.2 *Multiplexing*

Uma das principais funcionalidades do QUIC foi o uso de *multiplexing* utilizando UDP, assim evitando o *head-of-line blocking*. Utilizando o aioquic, podemos utilizar da função apenas instanciando diferentes *streams* de transmissão dentro de uma mesma conexão.

Desta maneira aumentamos a taxa de transmissão afim de adquirir uma melhor qualidade de serviço para o streaming e evitar que *tiles* de maior prioridades fiquem aguardando o reenvio de *tiles* menos importantes em caso de erro.

Para avaliarmos de forma mais simples o impacto das *streams*, iremos utilizar apenas duas para cada cliente, sendo uma para cada prioridade.

4.2.3 *Server Push*

Uma outra funcionalidade presente desde o HTTP/2 incluída em nosso servidor é o *push*. Resumidamente, o *server push* é uma resposta do servidor para um request que o cliente ainda não fez. Para isso, o servidor deve "prever"qual será o request e antecipar a resposta, o que é mais simples do que parece. Há muitos processos de requisições que seguem uma ordem já conhecida e repetida diversas vezes. Por exemplo, ao se conectar com uma página WEB o cliente enviará o pedido de conexão ao servidor e em seguida requisitar os arquivos necessários para a navegação (*e.g.* o html da página, o css, os arquivos de script, imagens presentes e etc.). Ao invés de esperar cada request individualmente, ao receber o pedido pelo html, o servidor pode enviar em seguida os demais arquivos sem ser necessário novas chamadas.

Contudo, isso deve ser uma política já definida e acordada entre os hosts da aplicação. Afinal isso gera um tráfego a mais para o cliente que, no caso de uma "previsão"incorreta, pode ser até desnecessário.

Em nosso experimento o *push* pode ser bem aplicado, afinal em um contexto de visualização de um vídeo, se o usuário requisitou o segmento 1 dos *tiles* em exibição, em seguida

o trecho a ser assistido será o 2. Desta forma o *push* foi implementado para enviar os segmentos seguintes após o envio do atual. Para estes próximos, a mesma prioridade atribuída aos *tiles* deste será mantida. Tal política foi estabelecida levando em consideração que em situações onde não haja movimentos bruscos do campo de visão, muitos *tiles* anteriormente vistos permanecerão nele.

No cenário do *push*, então, temos duas constatações sendo pressupostas pelo servidor: de que o cliente continuará assistindo o vídeo e que seu campo de visão permancerá aproximadamente o mesmo.

A primeira pode ser verdadeira na maioria dos casos, contudo para evitar um excesso de tráfego desnecessário, o cliente deve reconhecer o *push* enviado pelo servidor para permitir o envio dos próximos. Já a segunda constatação pode ser mais arriscada, afinal o diferencial do vídeo 360º é permitir o movimento livre do campo de visão. Assumir que este não irá alterar-se muito entre cada segmento pode ser muitas vezes uma previsão falsa. Contudo, por tal lógica ser aplicada apenas para o *push*, esperamos que pelo fato de estarmos antecipando o próximos request do cliente, todos os *tiles* já terão sido enviados no momento de sua visualização. Então a prioridade que é atribuída para garantir que os *tiles* mais importantes estejam presentes até lá, não será tão relevante.

4.3 Cliente

Como dito anteriormente, o cliente também foi desenvolvido em Python utilizando a biblioteca do aioquic.

O cliente é instanciado utilizando o endereço do servidor para conexão e utiliza de um arquivo CSV como simulação de entrada de um usuário. Este arquivo possui os *tiles* no campo de visão a cada frame do vídeo observado.

O algoritmo do cliente segue a seguinte lógica em repetição:

1. Conexão com o servidor
2. Requisição dos *tiles* do segmento atual considerando suas prioridades
3. Aguardo do tempo de visualização do usuário para cálculo do *Missing Ratio*

O recebimento dos arquivos de mídia é feito em uma *thread* separada da principal.

Após a conexão com o servidor, o primeiro passo do cliente, então, é a requisição dos *tiles* do primeiro segmento. A requisição é feita em ordem numérica, começando pelo *tile* de número 1 até o de número 200. Para cada um destes, é checado se encontra-se no campo de visão do frame atual através do arquivo CSV e atribuído sua prioridade para cada caso (*i.e.* 1 para alta e 2 para baixa) e então é enviado o request para o servidor na *stream* adequada para a prioridade designada.

É importante ressaltar que o request é feito por segmento de vídeo e não frame. Sendo cada segmento um trecho de 1 segundo de vídeo a 30 fps, então apenas a cada 30 frame que os requests de novos segmentos de *tiles* são feitos.

Após o envio dos requests do segmento atual do vídeo, a *thread* principal passa a verificar se a cada frame o *tile* requisitado está disponível para visualização. Para isso foi necessário simular o tempo de requisição do usuário levando em conta o FPS do vídeo. Ou seja, para

30 fps, a cada 1s/30 = 33,33ms o frame requisitado pelo arquivo de entrada deve estar presente para o usuário. Caso não esteja, isso pode ser caracterizado como um *miss* para a métrica de *Missing Ratio*.

O recebimento dos arquivos enviados pelo servidor dos segmentos dos *tiles* ocorre em uma outra *thread* paralela a todo este processo. De modo que enquanto a *thread* de envio de requests está verificando os arquivos recebidos e simulando o tempo de visualização, esta pode estar recebendo a todo momento tais arquivos e inclusive, recebendo o *server push* de segmentos futuros.

Enquanto a *thread* de recebimento dos arquivos aguarda o tempo todo os pacotes do servidor, a *thread* de requests repete as etapas descritas acima até que o vídeo chegue ao fim e então encerre a conexão com o servidor.

5 Experimentos

Com o servidor e cliente implementados e os arquivos preparados, torna-se possível começar os experimentos. Para facilitar e criar um ambiente de testes que possibilita a alteração de parâmetros de rede, iremos utilizar o Mininet[16].

Ao todo, serão 24 cenários de rede criados no Mininet, como explicitado na tabela 1.

Cada cenário será feito alternando as políticas de enfileiramento utilizadas pelo servidor (FIFO, SP e WFQ), sendo então um total de 72 cenários realizados. Cada cenário será executado 5 vezes para obtenção dos dados, com um total de 360 execuções. Para cada uma, será analisado o cálculo das seguintes métricas:

- *Missing Ratio* Total, calculado pela quantia de *misses* pelo total de *tiles* em todos frames. Um *miss* ocorre quando um *tile* de um frame do cliente ainda não foi transmitido pelo servidor no momento de visualização.

- *Missing Ratio* no Campo de Visão, calculado pela quantia de *misses* pelo total de *tiles* presentes no campo de visão de todos frames. Um *miss* ocorre quando um *tile* presente no campo de visão de um frame do cliente ainda não foi transmitido pelo servidor no momento de visualização.

- Contagem de *Rebuffering*, calculado pela quantidade de vezes que o vídeo foi pausado para o carregamento do *buffer* do cliente.

- Duração de *Rebuffering*, calculado pelo tempo total gasto em segundos carregando o *buffer* do cliente ao longo do vídeo.

- Uso do canal, calculado pela porcentagem ocupada de banda pela transmissão do vídeo.

#	Bandwidth (Mbps)	Delay (ms)	Load (%)
1	10	5	10
2	10	50	10
3	10	75	10
4	10	100	10
5	10	125	10
6	10	150	10
7	8	5	10
8	8	50	10
9	8	75	10
10	8	100	10
11	8	125	10
12	8	150	10
13	10	5	30
14	10	50	30
15	10	75	30
16	10	100	30
17	10	125	30
18	10	150	30
19	8	5	30
20	8	50	30
21	8	75	30
22	8	100	30
23	8	125	30
24	8	150	30

Tabela 1: Parâmetros de cenários

6 Resultados

6.1 *Missing Ratio* Total

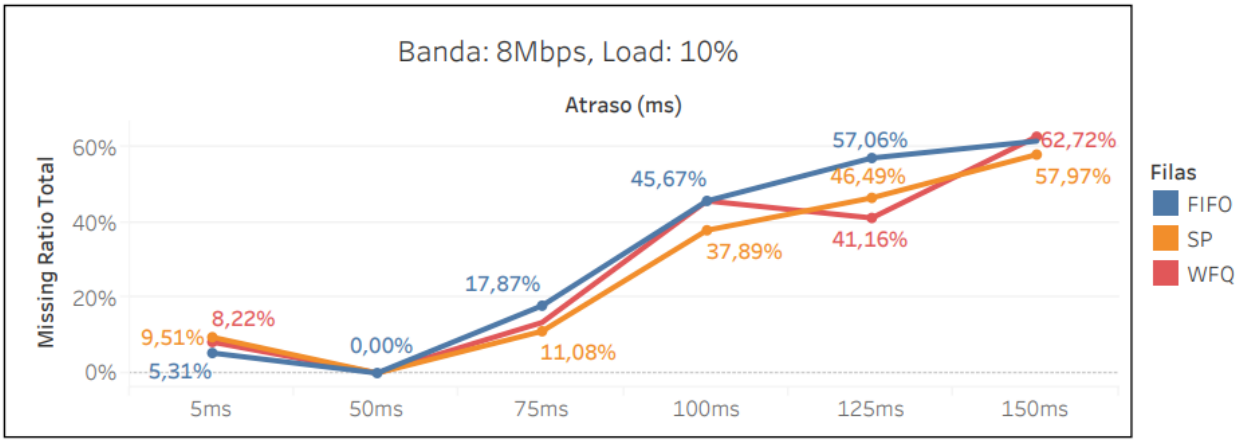

Figura 5: Missing Ratio Total: 8Mbps e 10% de Banda

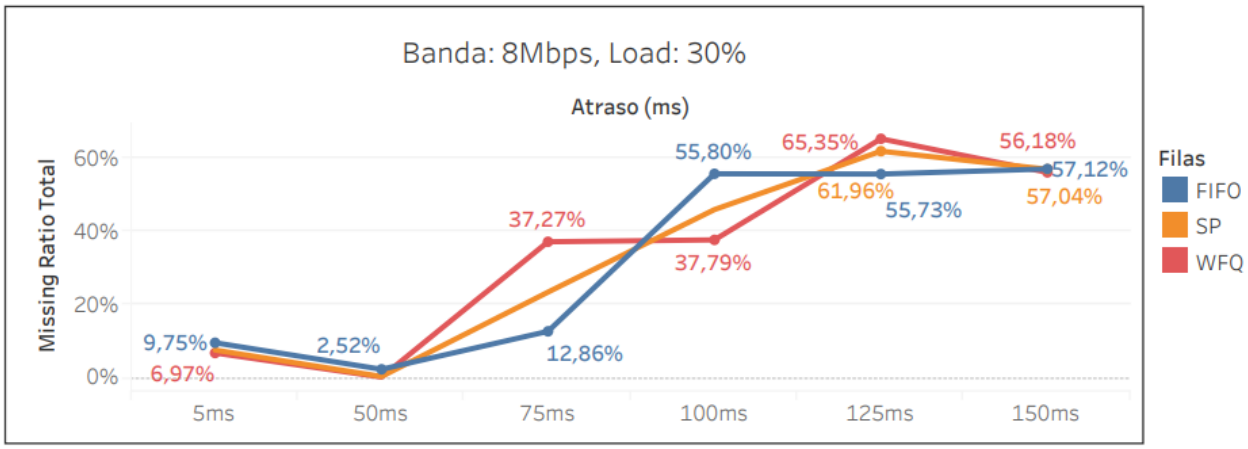

Figura 6: Missing Ratio Total: 8Mbps e 30% de Banda

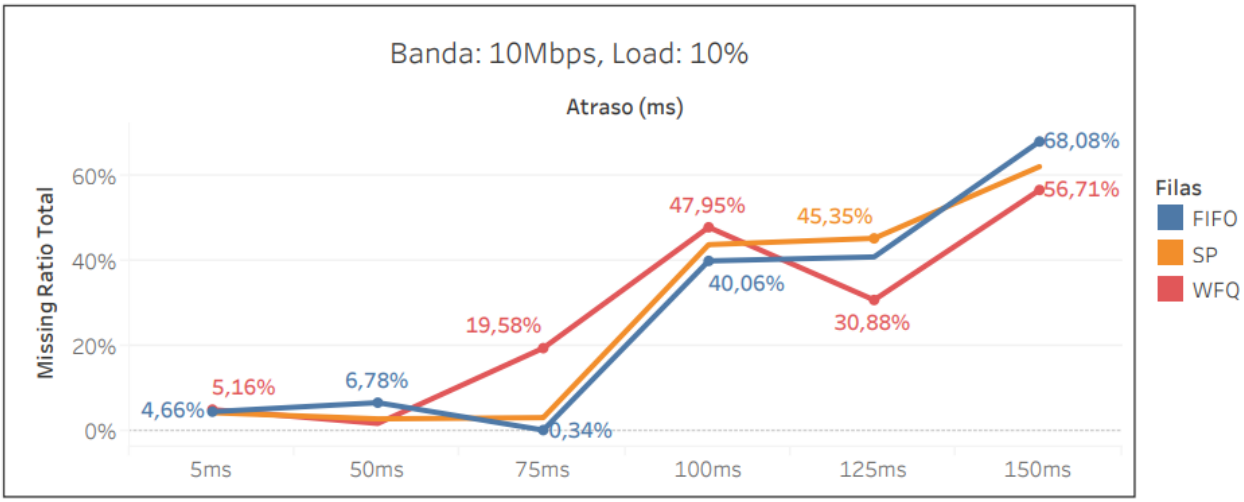

Figura 7: Missing Ratio Total: 10Mbps e 10% de Banda

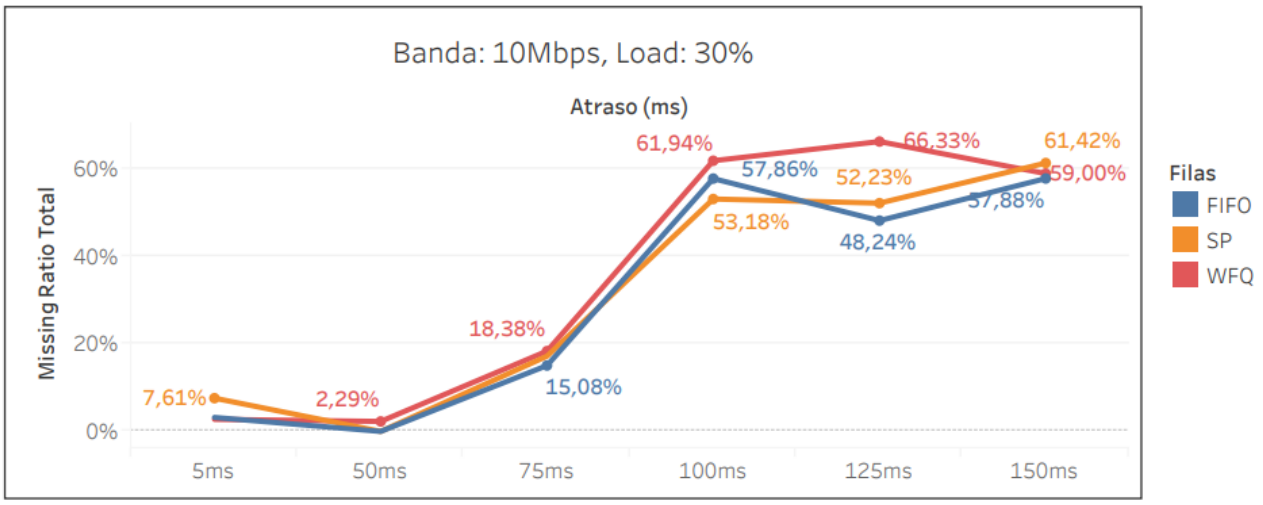

Figura 8: Missing Ratio Total: 10Mbps e 30% de Banda

6.2 *Missing Ratio* no Campo de Visão

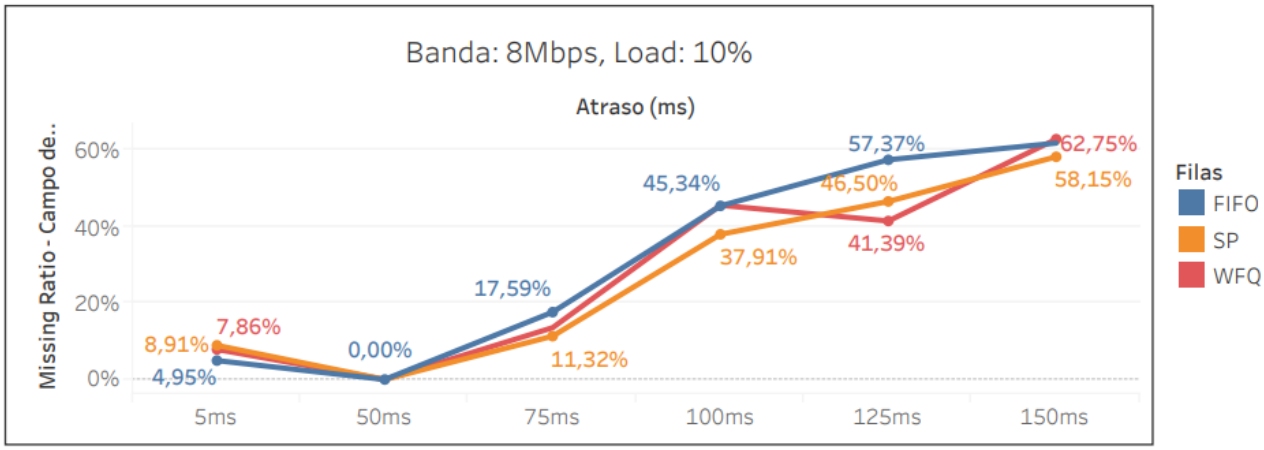

Figura 9: Missing Ratio no Campo de Visão: 8Mbps e 10% de Banda

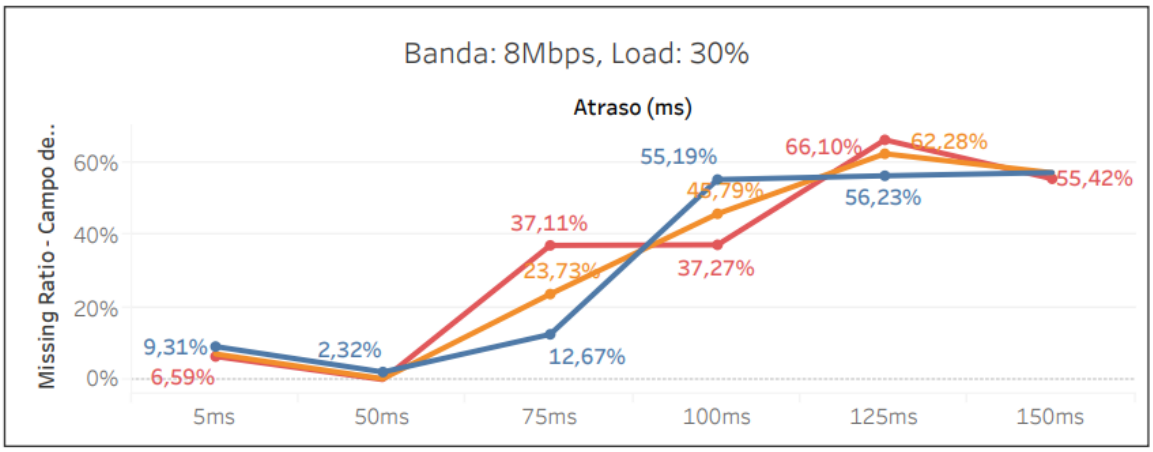

Figura 10: Missing Ratio no Campo de Visão: 8Mbps e 30% de Banda

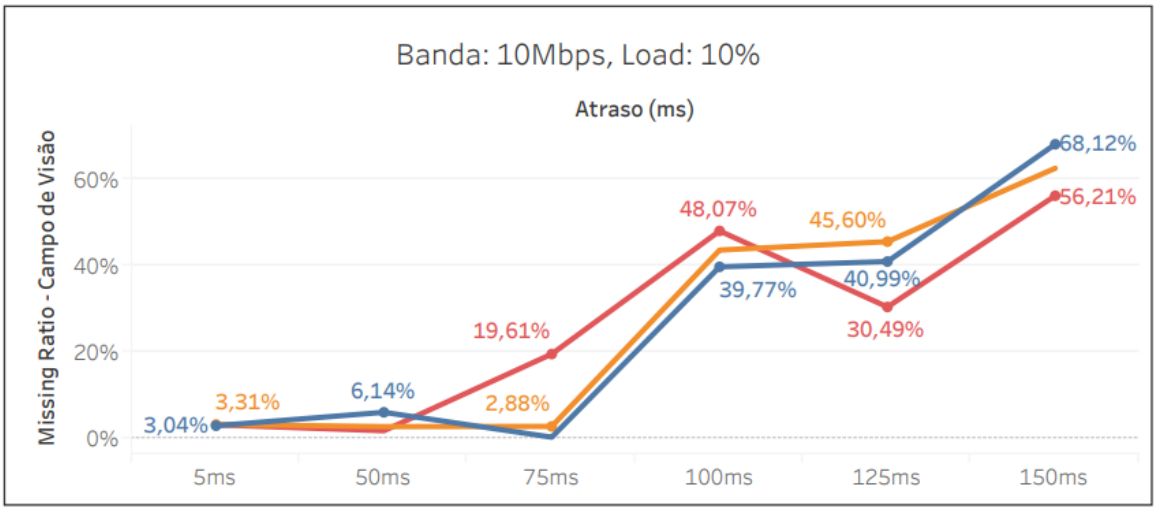

Figura 11: Missing Ratio no Campo de Visão: 10Mbps e 10% de Banda

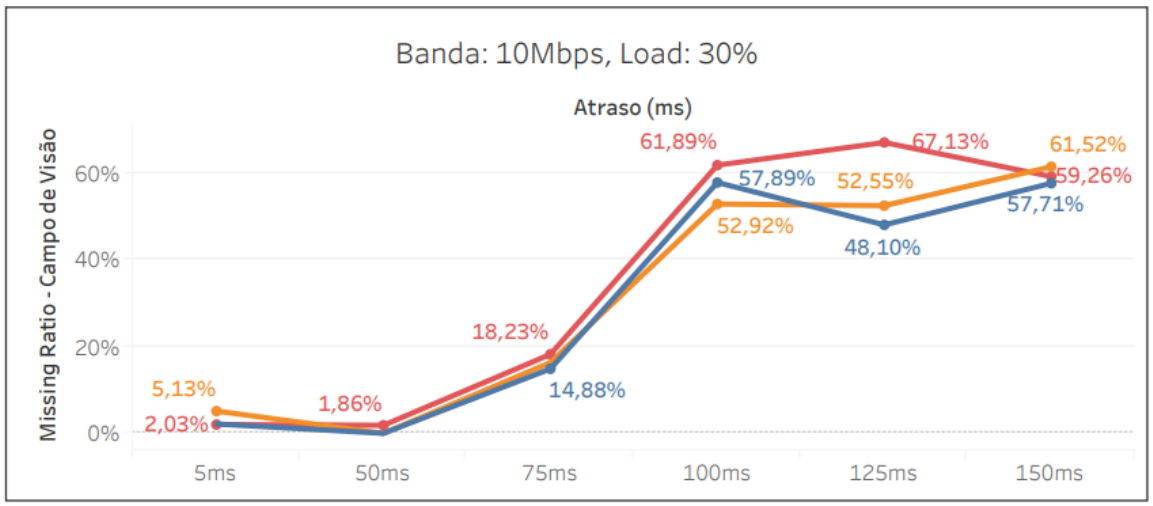

Figura 12: Missing Ratio no Campo de Visão: 10Mbps e 30% de Banda

6.3 Contagem de *Rebuffering*

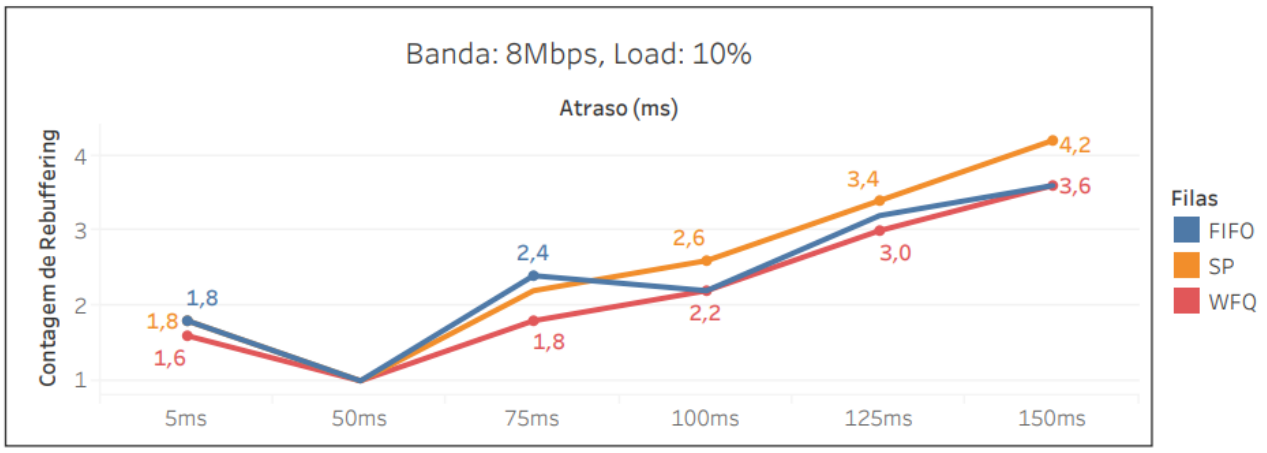

Figura 13: Contagem de *Rebuffering*: 8Mbps e 10% de Banda

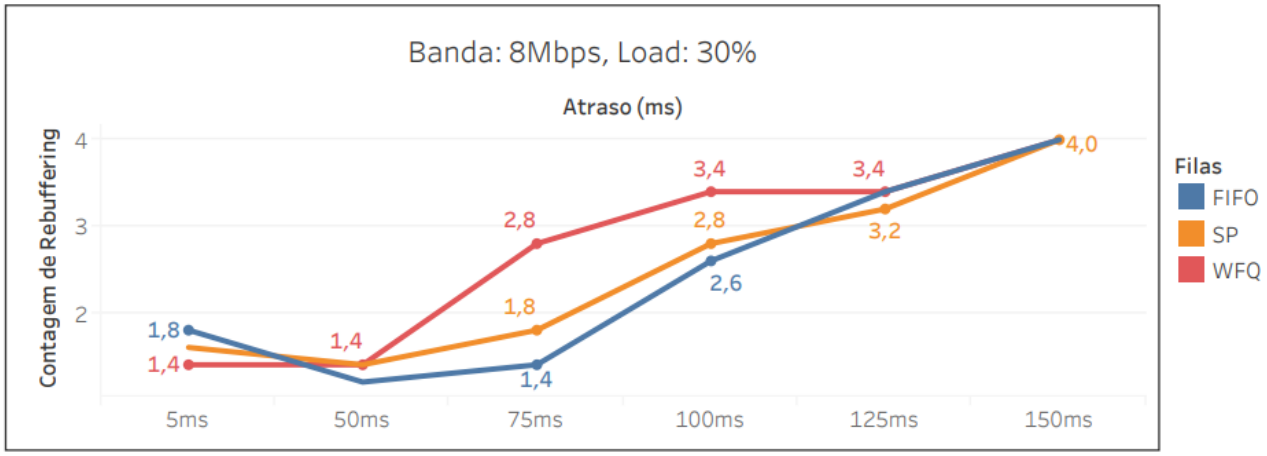

Figura 14: Contagem de *Rebuffering*: 8Mbps e 30% de Banda

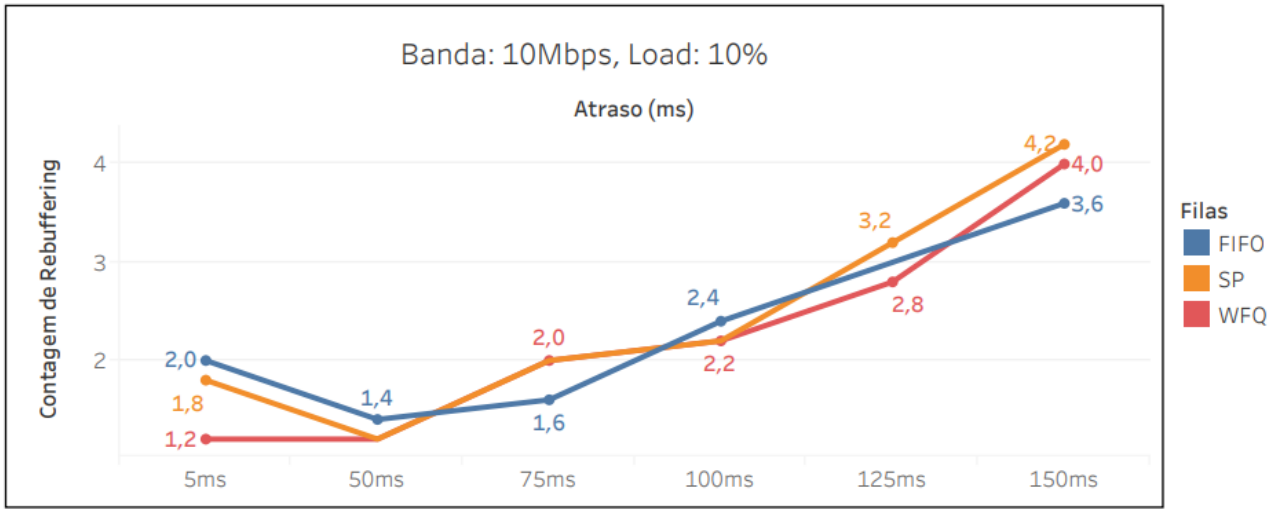

Figura 15: Contagem de *Rebuffering*: 10Mbps e 10% de Banda

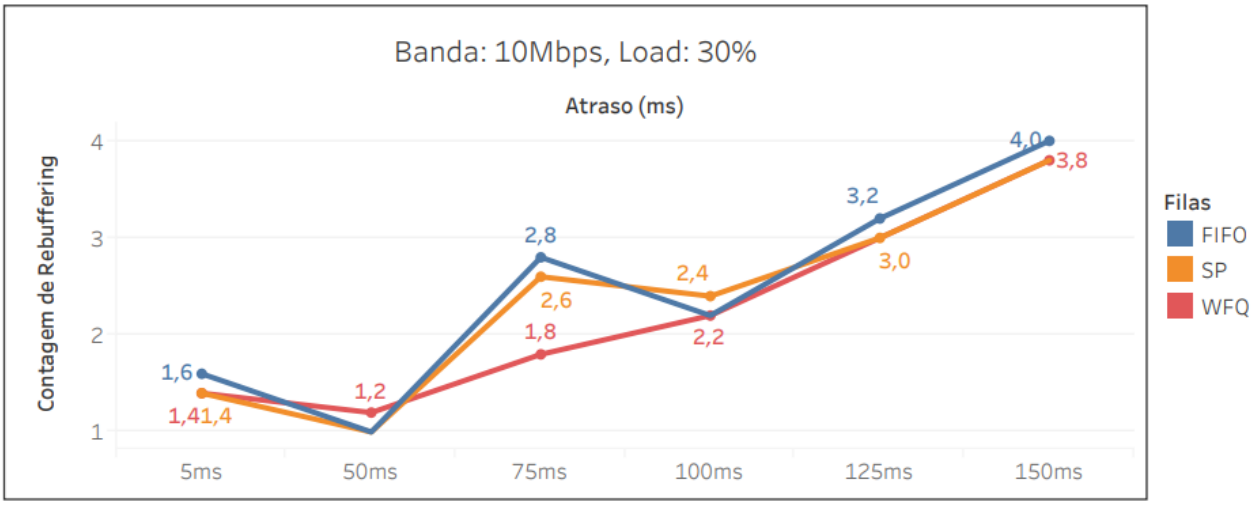

Figura 16: Contagem de *Rebuffering*: 10Mbps e 30% de Banda

6.4 Tempo de *Rebuffering*

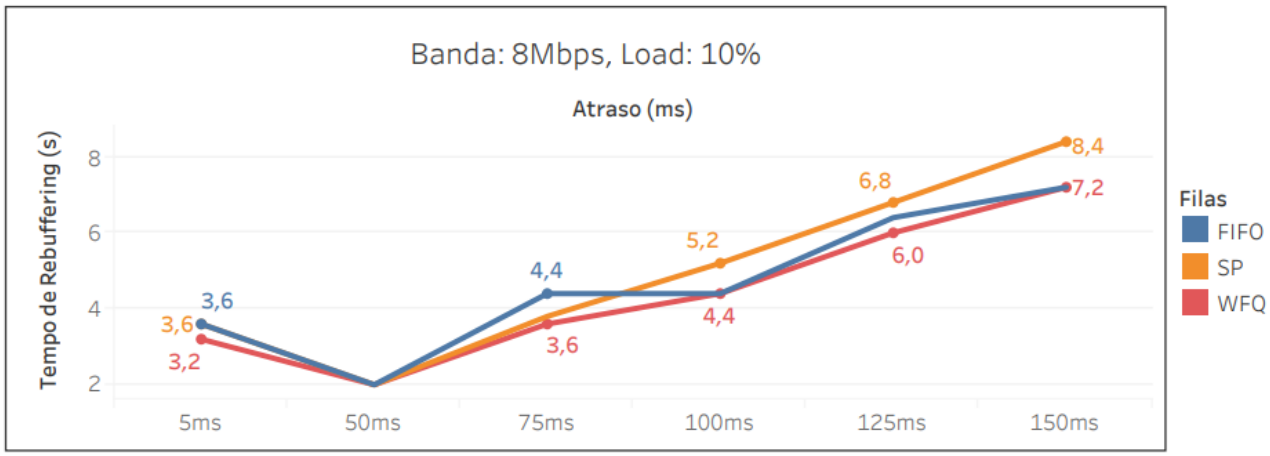

Figura 17: Tempo de *Rebuffering*: 8Mbps e 10% de Banda

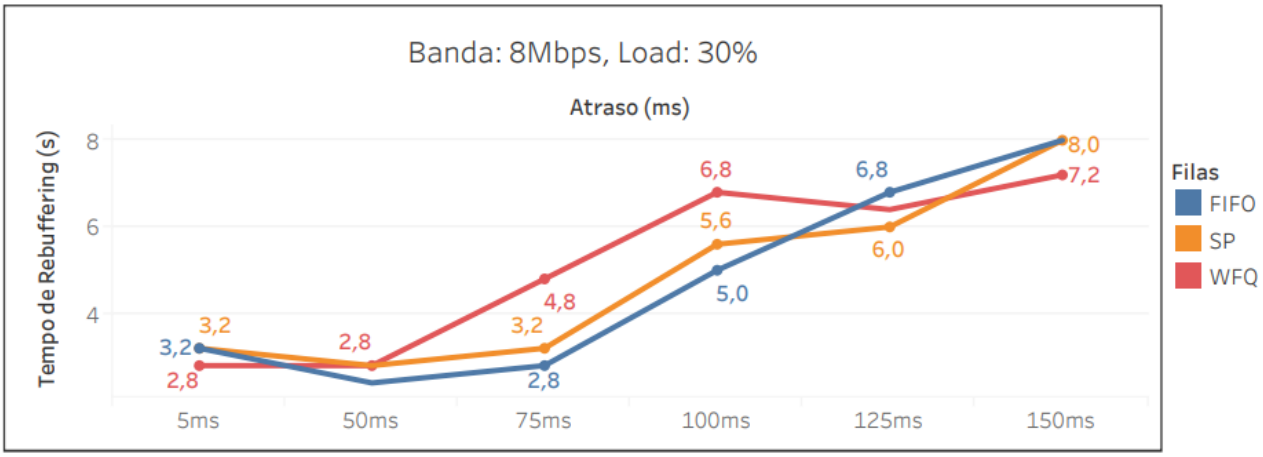

Figura 18: Tempo de *Rebuffering*: 8Mbps e 30% de Banda

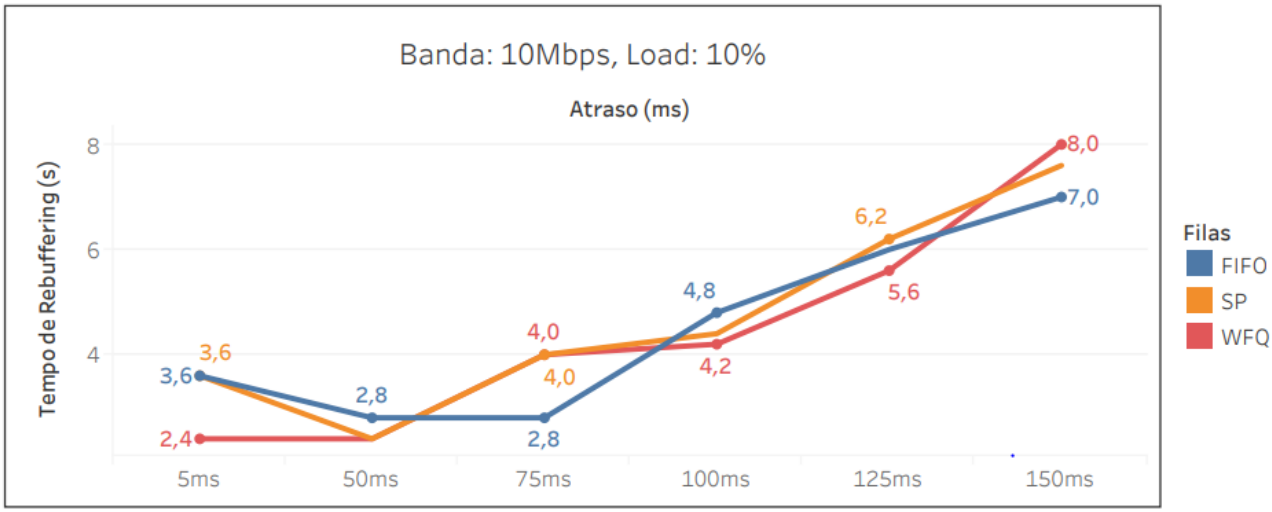

Figura 19: Tempo de *Rebuffering*: 10Mbps e 10% de Banda

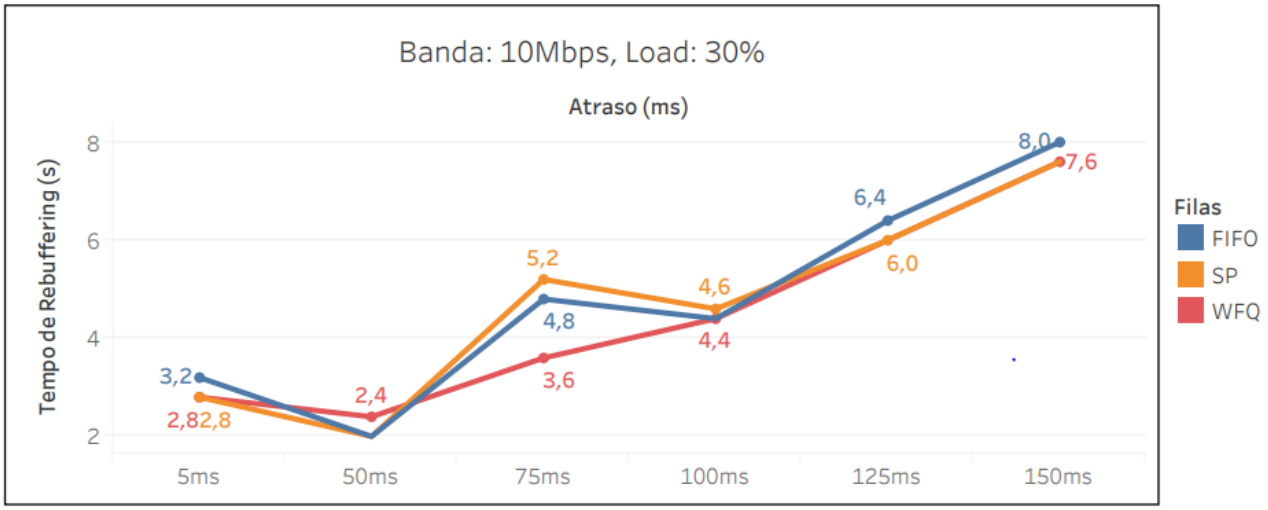

Figura 20: Tempo de *Rebuffering*: 10Mbps e 30% de Banda

6.5 Uso do Canal

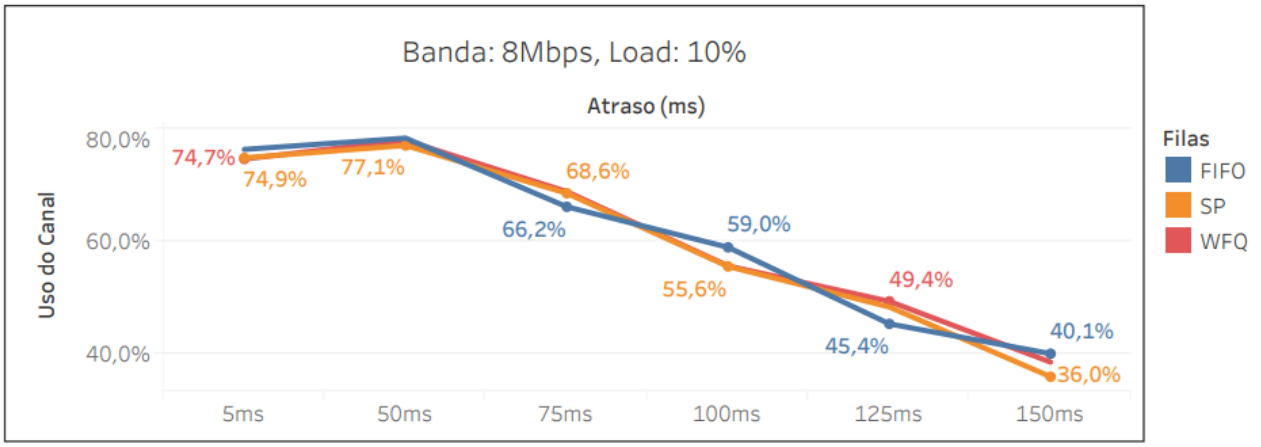

Figura 21: Uso do Canal: 8Mbps e 10% de Banda

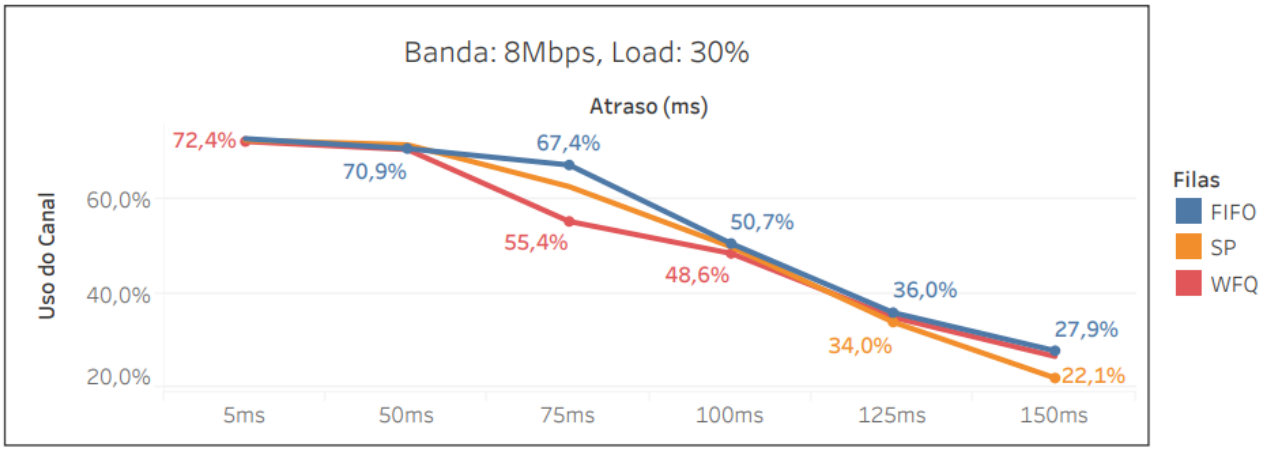

Figura 22: Uso do Canal: 8Mbps e 30% de Banda

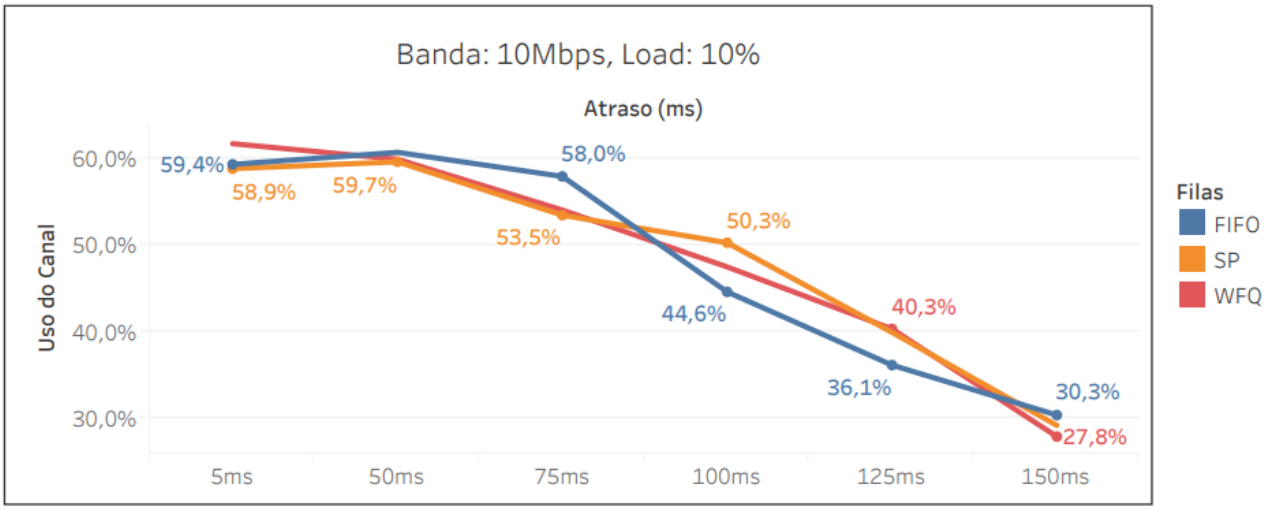

Figura 23: Uso do Canal: 10Mbps e 10% de Banda

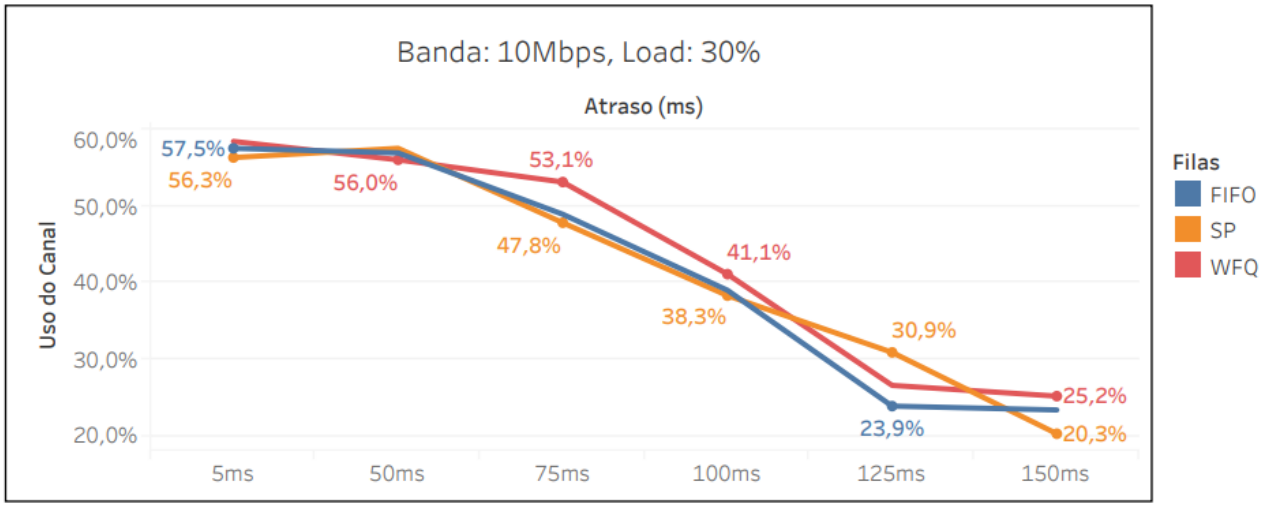

Figura 24: Uso do Canal: 10Mbps e 30% de Banda

7 Análise

Pela observação dos gráficos da Figura 5 até a Figura 12 é possível perceber um aumento do *Missing Ratio* em conjunto ao aumento do atraso. Este passa a ser significante a partir da conexão com 100ms de atraso com valores já próximos a 50% *Missing Ratio* e possui resultados melhores no atraso de 50ms, melhor mesmo até que o anterior de 5ms. Em todos cenários não é possível notar alguma variação entre as políticas, todas apresentam resultados próximos e não indicam alguma vantagem de uso entre elas.

Dos gráficos da Figura 13 até a Figura 20 é possível analisar o mesmo padrão da progressão do *Missing Ratio* se repetindo para o *Rebuffering*. As políticas pouco se diferenciam entre si nos cenários tratados à medida que o atraso aumenta. E como observado anteriormente, o atraso com menor tempo e contagem de *Rebuffering* ocorre com atraso de 50ms.

Por fim, os gráficos da Figura 21 até a Figura 24 indicam que o uso do canal também não é influenciado pela política de enfileiramento apresentada e que foi definido uma largura de banda suficiente para os experimentos em questão. Contudo, a banda utilizada para cenários com atraso de 5ms se manteve a mesma para os de 50ms. Podemos interpretar que o atraso menor necessitava de mais banda para processar o vídeo em baixa latência e por isso apresentou piores resultados que o atraso maior nas métricas descritas anteriormente.

8 Trabalhos Relacionados

- *360° Video Viewing Dataset in Head-Mounted Virtual Reality* [8]

  Estudo com *dataset* de dados da análise de conteúdo e movimento de diferentes vídeos disponíveis no Youtube. Além do acesso aos vídeos analisados, também utilizamos a análise de movimento de um usuário como uma simulação de entrada para os nossos experimentos.

9 Conclusão

Ao fim das 360 execuções feitas nos 72 cenários descritos, com as configurações apresentadas neste relatório, pode-se concluir que não há clara diferença de qualidade de serviço na transmissão de vídeo 360º para as políticas de enfileiramento: FIFO, SP e WFQ.

Por serem políticas tão distintas em suas abordagens, era esperado que alguma diferença fosse encontrada nos resultados. Deste modo, suspeita-se de que alguma configuração de enfileiramento interna do Aioquic tenha interferido nos resultados.

Para uma melhor análise destas políticas, recomenda-se a investigação mais profunda da biblioteca e seu impacto, ou a utilização de outra implantação de QUIC.

Referências

[1] YouTube. https://www.youtube.com/. Acesso: 2022-07.

[2] Facebook. https://www.facebook.com/. Acesso: 2022-07.

[3] Cloudflare. *What is HTTP3?*. https://www.cloudflare.com/learning/performance/what-is-http3/. Acesso: 2022-07-05.

[4] J. Roskind. *QUIC - Quick UDP Internet Connections*. (Março 2012).

[5] HAXX. *TCP head of line blocking*. https://http3-explained.haxx.se/en/why-quic/why-tcphol. Acesso: 2022-07-05.

[6] Aioquic - Repositório. *aiortc/aioquic*. https://github.com/aiortc/aioquic. Acesso: 2022-07-05.

[7] Implementação do Experimento - Repositório. *aiortc/aioquic*. https://github.com/gufernandez/aioquic-360-video-streaming. Acesso: 2022-07-05.

[8] W. Lo, C. Fan, J. Lee, C. Huang, K. Chen, C. Hsu. *360° Video Viewing Dataset in Head-Mounted Virtual Reality*. (2017)

[9] Paramout Pictures at Youtube. *BEN-HUR (2016) - Chariot Race 360° Video - Paramount Pictures*. https://www.youtube.com/watch?v=jMyDqZe0z7M. Acesso: 2022-07.

[10] FFmpeg. *FFmpeg*. https://ffmpeg.org/. Acesso: 2022-07-05.

[11] Kvazaar - Repositório. *ultravideo/kvazaar*. https://github.com/ultravideo/kvazaar. Acesso: 2022-07-05.

[12] MP4Box - Repositório. *gpac/gpac*. https://github.com/gpac/gpac/wiki/MP4Box. Acesso: 2022-07-05.

[13] R. Chang, M. Rahimi, V. Pournaghshband. *Differentiated Service Queuing Disciplines in NS-3*. (2015)

[14] GCF Global. *What is 360 video?*. https://edu.gcfglobal.org/en/thenow/what-is-360-video/1/. Acesso: 2022-07.

[15] Asyncio - Biblioteca. *asyncio — Asynchronous I/O*. https://docs.python.org/3/library/asyncio.html. Acesso: 2022-07-05.

[16] Mininet - Repositório. *mininet/mininet*. https://github.com/mininet/mininet. Acesso: 2022-07-05.