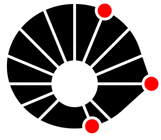

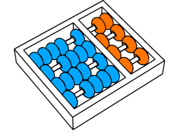

Simulações de Content Steering: explorando a Computação Contínua e o Aprendizado por Reforço para oferecer suporte ao Adaptive Video Streaming

C. D. Grazioti *V. K. Hiratsuka* *W. G. S. Lima*
L. F. Bittencourt *R. R. Filho*

Relatório Técnico - IC-PFG-24-29
Projeto Final de Graduação
2024 - Dezembro

UNIVERSIDADE ESTADUAL DE CAMPINAS
INSTITUTO DE COMPUTAÇÃO

The contents of this report are the sole responsibility of the authors.
O conteúdo deste relatório é de única responsabilidade dos autores.

Simulações de Content Steering: explorando a Computação Contínua e o Aprendizado por Reforço para oferecer suporte ao Adaptive Video Streaming

César Devens Grazioti* Vitor Kiyoshi Hiratsuka*
Wallace Gustavo Santos Lima* Luiz Fernando Bittencourt*
Roberto Rodrigues Filho†

13 de dezembro de 2024

Resumo

Com a crescente demanda por vídeo nos últimos anos e o crescimento da computação em nuvem, estamos vivenciando um maior investimento em streaming de vídeo adaptativo. Um protocolo popular que surgiu nesse contexto é o Dynamic Adaptive Streaming over HTTP (DASH). Ele propõe que um player de vídeo pode ajustar dinamicamente a taxa de bits com base na largura de banda disponível, maximizando o uso eficiente de recursos e permitindo uma experiência personalizada - conforme o dispositivo do usuário e as condições da rede. A arquitetura proposta nesse projeto encontra pauta no contexto do continuum edge-cloud, destacando como serviços de streaming de vídeo podem ser otimizados para lidar com usuários móveis. Para isso, foi explorado o problema do Multi-Armed Bandits e utilizado algoritmos de aprendizado por reforço, como Epsilon-Greedy e UCB1, combinados com métricas de latência para melhor redirecionar as requisições do cliente aos servidores cache. Os cenários explorados foram: cliente imóvel com estresse em um servidor e cliente móvel com cenários de estresse e sem estresse de servidor. Os resultados analisados mostram que a qualidade de experiência do usuário não depende somente da proximidade com o servidor cache que armazena o conteúdo de vídeo. Faz-se necessário capturar métricas adicionais. No nosso caso, capturamos o congestionamento do servidor e concluímos que o redirecionamento de requisições atua também como um balanceador de carga em sistemas de streaming de vídeo adaptativo. Além disso, vimos como a arquitetura de Content Steering pode permitir que o usuário final tenha uma menor latência durante sua experiência de streaming, explorando a Computação Contínua.

1 Introdução

A distribuição de vídeo evoluiu para constituir uma fração importante do tráfego atual da Internet na última década, graças aos avanços nas tecnologias de rede, recursos de dispositivos e esquemas de compressão de áudio e vídeo [2]. A popularização de plataformas como

*Instituto de Computação, Universidade Estadual de Campinas, 13081-970, Campinas, SP.
†Instituto de Informática, Universidade Federal de Goiás, 74690-900, Goiânia, GO.

YouTube, Netflix, e TikTok, além do aumento no uso de vídeos em redes sociais, exige que os sistemas sejam escaláveis para suportar bilhões de usuários simultaneamente. O aumento exponencial do tráfego de vídeo demanda investimentos contínuos em infraestrutura para evitar congestionamentos e garantir velocidades adequadas, especialmente para streaming em alta definição (HD) e resoluções superiores, como 4K e 8K. Dessa forma, implementar esquemas que ajustem automaticamente a qualidade do vídeo às condições da rede requer soluções sofisticadas para evitar interrupções ou quedas bruscas de qualidade.

Com o tempo, novas tecnologias surgiram para superar essas limitações e aprimorar o streaming de vídeo [2]. O streaming adaptável é uma técnica de entrega de vídeo dinâmica que ajusta a qualidade da reprodução de vídeo em tempo real com base nas condições de rede do usuário, com o objetivo principal de fornecer reprodução de visualização contínua, adaptando-se às mudanças na largura de banda disponível [10]. Hoje em dia, as soluções prevalentes em plataformas de streaming sob demanda e ao vivo, são o *Dynamic Adaptive Streaming over HTTP* (DASH) [7] e o *HTTP Live Streaming* (HLS) [6] da Apple. Nelas, os segmentos de mídia e arquivos de manifesto são hospedados em uma rede de entrega de conteúdo (CDN) e, de lá, entregues aos players de mídia [9].

O surgimento da computação em nuvem e da computação de borda possibilita a criação de uma orquestração no *Edge-Cloud Continuum*. A infraestrutura *edge-cloud* oferece para aplicações de streaming de vídeo uma ampla gama de recursos computacionais capazes de atender às crescentes demandas dos usuários finais [4]. Diante do elevado volume de dados, da qualidade variável das conexões de Internet e da mobilidade dos usuários, os sistemas precisam se adaptar constantemente a novas condições de operação [1]. Essa flexibilidade é viabilizada pela capacidade da borda de reduzir a latência e gerenciar tráfego localmente, enquanto a nuvem proporciona escalabilidade para processar grandes volumes de dados e personalizar serviços de acordo com as necessidades específicas. Assim, a integração eficiente entre borda e nuvem é fundamental para garantir uma experiência otimizada em streaming de vídeo, mesmo em cenários dinâmicos e desafiadores.

Com um protocolo de streaming adaptável baseado em HTTP, um reprodutor de vídeo pode ajustar dinamicamente (na granularidade de segundos) a taxa de bits do vídeo com base na largura de banda de rede disponível [5]. Em sua solução, o DASH implementa uma arquitetura de *Content Steering* para determinar em qual CDN o player de vídeo está buscando dados durante a reprodução do vídeo. Nas seções 2.1 e 2.2 apresentamos com mais detalhes como é a comunicação entre os componentes do *HTTP Adaptive Streaming* (HAS), bem como o funcionamento de um sistema HAS habilitado para *Content Steering*.

O corrente trabalho se concentra em explorar os recursos da organização *Edge-Cloud Continuum*, utilizando a arquitetura de *Content Steering* do DASH, para oferecer suporte ao *Adaptive Video Streaming*. Nesse contexto, foram utilizados algoritmos de aprendizado por reforço, como *Epsilon-Greedy* e *UCB1*, para resolver o problema de alocação do servidor cache destinado ao consumo de conteúdo de streaming, se apoiando nos conceitos de *exploring* e *exploiting*. Além disso, discutimos como o congestionamento dos servidores e o posicionamento geográfico entre o cliente (quem faz a requisição do conteúdo) o servidor (quem dispõe e entrega o conteúdo) influenciam no cálculo da latência. Para uma análise de desempenho desses algoritmos, foram criados diversos cenários, onde temos um cliente móvel que realiza requisições de conteúdo e três servidores de *cache*. A seção 6 exibe como

os algoritmos de escolha foram adaptados para métricas de latência, além de apresentar e discutir os resultados obtidos para os diferentes cenários.

2 Referencial Teórico

As subseções a seguir apresentam as fundamentações teóricas que embasam o desenvolvimento deste projeto. Nelas são explorados diversos conceitos que serviram como uma base técnica e norte para compreensão das simulações e seus resultados, incluindo uma explicação da arquitetura de *Content Steering*, funcionamento da hierarquia *edge-cloud* com intuito de auxiliar a melhoria da qualidade dos serviços, funcionamento de algoritmos de escolha (*Epsilon-Greedy* e *UCB1*), benefícios da conteinerização de processos no dia a dia e estimativa do cálculo da distância entre dois pontos na Terra a partir de suas latitudes e longitudes.

Tais temas permitiram a elaboração de uma visão geral consistente e integrada dos elementos necessários para orientar as decisões durante a construção do simulador e garantir a coerência entre as etapas de desenvolvimento e os resultados alcançados.

2.1 HTTP Adaptive Video Streaming

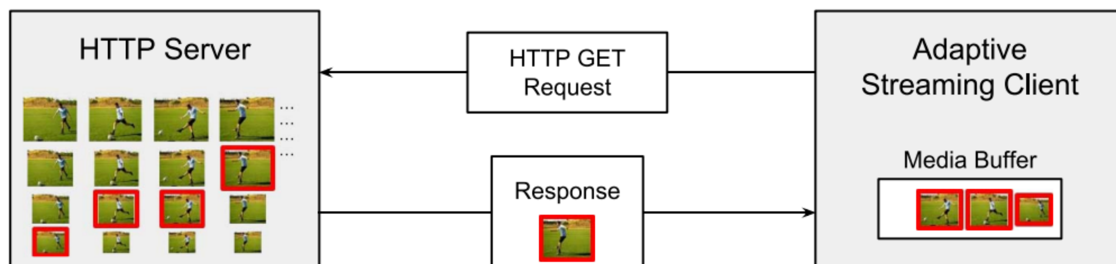

Figura 1: Comunicação no sistema HAS [2].

No contexto do HAS, os vídeos são divididos em segmentos curtos, geralmente com duração de um a dez segundos. O codificador então codifica cada segmento em diferentes versões com resoluções ou taxas de bits variadas. O arquivo manifesto *Media Presentation Description* (MPD) rastreia o local de armazenamento desses segmentos em um servidor. Quando um usuário assiste a um vídeo, seu player HAS utiliza essas informações, velocidade da Internet e recursos do dispositivo para selecionar a melhor versão para transmitir. Esse processo de segmentação e codificação é um aspecto fundamental do HAS, permitindo que ele se adapte às condições de rede do usuário e forneça uma experiência de visualização perfeita [10].

Abaixo temos uma interpretação bem detalhada dos autores [2] a respeito do HAS:

> O HAS usa HTTP como protocolo da camada de aplicação e TCP como protocolo de camada de transporte, conforme ilustrado na Figura 1, e os clientes

extraem os dados de um servidor HTTP padrão, que simplesmente hospeda o conteúdo de mídia. As soluções HAS empregam adaptação dinâmica com relação às condições variáveis da rede para fornecer uma experiência de streaming perfeita (ou pelo menos mais suave). Uma vez que um arquivo de mídia (ou fluxo) esteja pronto de uma fonte, ele é preparado para streaming antes de ser publicado em um servidor HTTP padrão e pronto para uso. O arquivo/fluxo original é particionado em segmentos (também chamados de chunks) de tempo de reprodução de comprimento igual. Várias versões (também chamadas de representações) de cada segmento são geradas, variando em taxa de bits/resolução/qualidade usando um codificador ou um transcodificador. Além disso, o servidor gera um arquivo de índice, que é um manifesto que lista as representações disponíveis, incluindo localizadores uniformes de recursos HTTP (URLs) para identificar os segmentos junto com seus tempos de disponibilidade. Durante uma sessão HAS típica, o cliente primeiro recebe o manifesto que contém os metadados para vídeo, áudio, legendas e outros recursos e, em seguida, mede constantemente certos parâmetros: largura de banda de rede disponível, status do buffer, níveis de bateria e CPU, etc. De acordo com esses parâmetros, o cliente HAS busca repetidamente o próximo segmento mais adequado entre as representações disponíveis do servidor [2].

2.2 Content Steering

Content Steering é uma capacidade determinística que permite aos distribuidores de conteúdo alterar a fonte de um conteúdo acessado por um player, seja no início da reprodução ou durante a transmissão. Essa funcionalidade é implementada por meio de um serviço remoto de direcionamento, que decide de forma eficiente qual URL alternativa de uma *Content Delivery Network* (CDN) o player deve acessar [8]. Essa solução promove resiliência, eficiência e flexibilidade em ambientes de distribuição complexos, uma vez que permite direcionar dinamicamente a fonte de conteúdo.

Os distribuidores de conteúdo geralmente usam várias redes de distribuição de conteúdo para distribuir seu conteúdo aos usuários finais. Eles podem carregar uma cópia de seu catálogo em cada CDN ou configurar as CDNs para buscar o conteúdo de uma origem comum. URLs alternativas são geradas, uma para cada CDN, que apontam para conteúdos idênticos. Isso permite que players compatíveis, como aqueles baseados em DASH (*Dynamic Adaptive Streaming over HTTP*), troquem para uma URL alternativa em caso de problemas de entrega [7, 9]. A Figura 2 ilustra como o DASH implementa o processo de *Content Steering* descrito anteriormente.

Na Figura 3 é mostrada a interação das partes de um sistema de vídeo adaptativo habilitado para Content Steering. Ele é composto pelo player dash.js no navegador, *Video Edge Service*, *Content Steering Service* e CDNs. O sistema de *Adaptive Video Streaming* foi projetado para otimizar a entrega de conteúdo de vídeo, posicionando estrategicamente o *Video Edge Service* próximo aos usuários. Este serviço captura vídeos de fontes CDN em casos de falha de cache, reduzindo a latência e otimizando o uso da rede central por meio do armazenamento eficiente na borda.

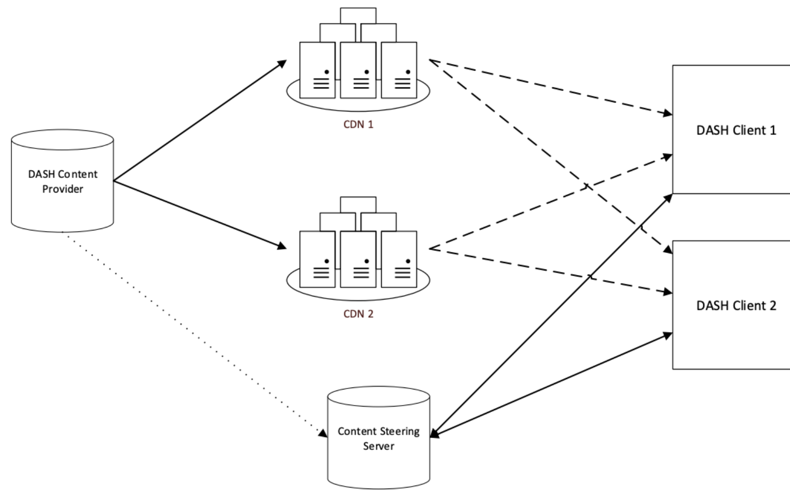

Figura 2: Arquitetura do processo de Steering pelo DASH [8].

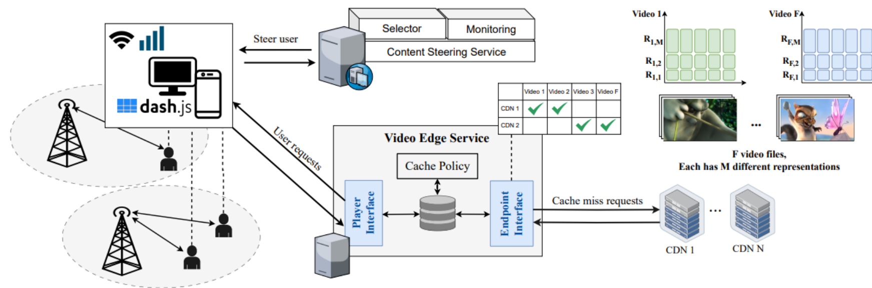

Figura 3: Workflow de um sistema de vídeo adaptativo habilitado para Content Steering [11].

O componente de camada de aplicação - *Content Steering Service* - fornece adaptabilidade ao sistema de vídeo ao considerar fatores como localização do usuário, recursos disponíveis na borda, características do conteúdo de vídeo e a qualidade da rede. Ele rastreia o fluxo do vídeo desde a fonte até o player do usuário, ajustando dinamicamente a entrega e os recursos da borda para eficiência máxima. Essa abordagem garante a versatilidade e robustez do sistema, permitindo operação eficiente em diferentes ambientes dentro do *continuum Edge-Cloud*.

Além disso, o *Content Steering Service* é composto por dois módulos principais, responsáveis por aprimorar a entrega de vídeos: o módulo de monitoramento e o módulo de seleção [11]. O módulo de monitoramento é o componente encarregado de coletar dados contextuais em tempo real, incluindo o identificador do conteúdo solicitado e a localização do usuário. Com essas informações, o sistema consegue diminuir a quantidade de saltos na

rede e aprimorar o armazenamento em cache direcionado, contribuindo para um uso mais eficiente dos recursos da rede e uma melhor experiência para o usuário. Já o módulo de seleção tem a função de identificar o melhor nó de borda para atender às solicitações dos usuários. Ele considera fatores como localização, condições da rede e o conteúdo requisitado, garantindo que a entrega seja realizada de maneira ágil e com baixa sobrecarga. Isso permite uma integração fluida entre borda e nuvem no contexto do continuum de computação.

Processo de entrega de vídeo

Origem e codificação O conteúdo de vídeo é obtido da fonte do provedor, codificado em múltiplas qualidades (taxas de bits) e segmentado em partes independentes, otimizando a adaptabilidade do streaming. Essas partes são armazenadas em CDNs estáticos na nuvem, e um manifesto contendo URLs e informações para acesso é gerado.

Interação com o usuário Quando um usuário solicita o manifesto, o dispositivo contata o *Content Steering Service*, que utiliza uma topologia de rede global e sessões persistentes para gerenciar conexões em tempo real no *continuum Edge-Cloud*. Isso permite ao *Video Edge Service* operar de forma autônoma, utilizando informações atualizadas para otimizar o gerenciamento de streaming.

Entrega e cache O *Video Edge Service*, hospedado em um nó de borda, possui duas interfaces principais:

1. ***Player Interface***: Conexão direta com os players do usuário.

2. ***Endpoint Interface***: Responsável por buscar segmentos de vídeo de servidores remotos, caso necessário.

Quando um segmento é solicitado, ele é entregue do cache se disponível; caso contrário, a *Endpoint Interface* recupera o segmento do CDN, o armazena no cache e o entrega ao usuário.

2.3 Edge-cloud Continuum

O *Edge-cloud Continuum* é uma infraestrutura hierárquica e altamente heterogênea com dispositivos espalhados por uma ampla localização geográfica. Ele varia de plataformas de computação em nuvem centralizadas localizadas no núcleo da rede, estendendo-se a dispositivos de usuário final localizados na borda da rede, resultando na computação contínua [11]. A Figura 4 ilustra a organização dessa infraestrutura, possibilitando uma visão geral das princiais características de seus níveis. O dispositivos finais estão localizados na parte inferior da hierarquia e possuem uma característica importante que caracteriza principalmente o *continuum* de computação. Os dispositivos de usuário final geralmente têm recursos limitados e pequena latência de rede [3]. À título de exemplificação, temos como dispositivos de borda: laptops, smartphones, dispositivos IoT, sensores, dentre outros.

Na segunda camada da infraestrutura, conhecida como plataforma de computação de borda (*edge computing platform*), encontramos clusters de computadores estrategicamente posicionados próximos aos usuários finais, abrigando o que chamamos de CDNs. Esses clusters possuem maior capacidade computacional, como processamento e armazenamento, em comparação com os dispositivos dos usuários. Contudo, isso vem acompanhado de um leve aumento na latência da rede. Apesar disso, a localização estratégica desses clusters garante que a latência permaneça reduzida, enquanto disponibilizam recursos computacionais mais robustos e eficientes. Essa camada intermediária da infraestrutura conecta regiões geográficas à nuvem e pode ser ampliada para formar múltiplas camadas adicionais, criando um *continuum* entre dispositivos locais e a computação em nuvem. As múltiplas camadas supracitadas estão representadas na Figura 4 como *"Layer 2...N"*.

A plataforma de computação em nuvem, localizada na camada superior da infraestrutura *edge-cloud*, destaca-se pela capacidade de oferecer recursos de computação escaláveis e aparentemente ilimitados, embora com maior latência de rede devido à distância dos usuários finais. Essa plataforma se organiza em diferentes modelos de implantação: nuvens públicas, acessíveis ao público com cobrança geralmente baseada no uso; nuvens privadas, restritas a um grupo específico, como empresas ou instituições; nuvens híbridas, que combinam recursos públicos e privados para atender a demandas dinâmicas; e nuvens comunitárias, que compartilham recursos entre organizações similares. Os serviços na nuvem operam sob Acordos de Nível de Serviço (SLAs), que definem como os usuários são cobrados, seja por tempo, volume de dados ou número de requisições [1]. Essa estrutura torna a nuvem ideal para tarefas intensivas em processamento e armazenamento, como grandes análises de dados ou execução de serviços corporativos complexos, consolidando-a como um elemento essencial na arquitetura moderna de computação distribuída.

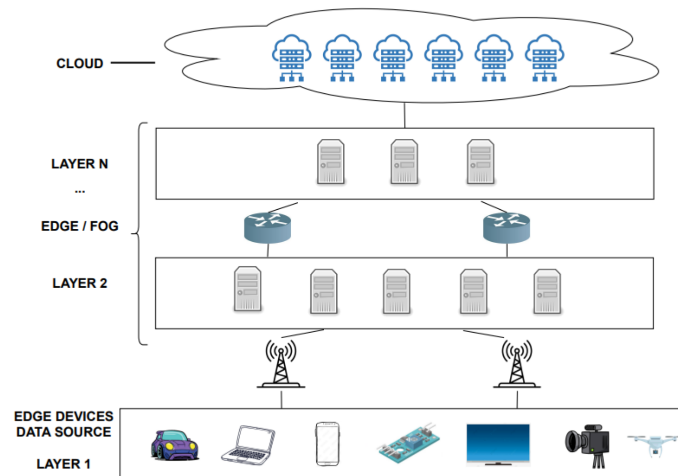

Figura 4: Visão geral da infraestrutura Edge-Cloud Continuum [10]

2.4 Multi-Armed Bandits

O **problema de *Multi-Armed Bandits* (MAB)** [15] é um dos problemas fundamentais na área de aprendizado por reforço e teoria de decisão. Ele modela cenários onde um agente deve fazer escolhas sequenciais para maximizar recompensas acumuladas enquanto aprende sobre o ambiente. O nome *"multi-armed bandit"* faz alusão às máquinas caça-níqueis (*bandits*), cada uma representando um braço que, ao ser puxado, fornece uma recompensa baseada em uma distribuição desconhecida.

Descrição do Problema

O problema pode ser descrito como segue:

- Um agente tem acesso a k braços (ou ações), enumerados de 1 a k.
- Cada braço i é associado a uma distribuição de recompensa desconhecida R_i, com valor esperado $\mu_i = \mathbb{E}[R_i]$.
- A cada rodada, o agente escolhe um braço a_t, observa uma recompensa R_{a_t}, e usa essa informação para decidir qual braço selecionar nas rodadas subsequentes.
- O objetivo é maximizar a recompensa total acumulada:

  $$G_T = \sum_{t=1}^{T} R_{a_t},$$

  ao longo de T rodadas.

Desafios: *Exploring* vs *Exploiting*

O problema central do MAB é o **trade-off entre *exploring* e *exploiting***:

- ***Exploring* (Exploração)**: Experimentar novos braços para descobrir se oferecem melhores recompensas do que os conhecidos.
- ***Exploiting* (Aproveitamento)**: Escolher os braços que, com base nas informações disponíveis, possuem a maior recompensa esperada.

Uma estratégia eficiente deve equilibrar esses dois objetivos para maximizar a recompensa acumulada.

Algoritmos para Resolver o Problema

Diversos algoritmos têm sido propostos para resolver o problema de MAB [16]. Entre os mais comuns, destacam-se:

- ***Epsilon-Greedy***: Alterna entre exploração (seleção aleatória) e aproveitamento (seleção do melhor braço conhecido) com probabilidade controlada por um parâmetro ϵ.
- ***UCB1 (Upper Confidence Bound)***: Seleciona o braço com o maior valor baseado em uma estimativa de recompensa média somada a um termo de exploração.

Aplicações

O problema de MAB encontra aplicações práticas em diversas áreas, como:

- **Teste A/B Adaptativo**: Para identificar rapidamente a melhor variante em experimentos de otimização.
- **Sistemas de Recomendação**: Para sugerir produtos ou conteúdos com base no feedback do usuário.
- **Alocação de Recursos**: Para distribuir investimentos ou esforços entre diferentes opções com retornos incertos.

2.4.1 Epsilon-Greedy

O algoritmo ***Epsilon-Greedy*** 1 é uma abordagem simples e amplamente utilizada para resolver o problema de Multi-Armed Bandits. Ele busca equilibrar a exploração de novas opções (*exploring*) e a exploração das melhores opções conhecidas (*exploting*), sendo esse equilíbrio controlado por um parâmetro ϵ.

Funcionamento do Algoritmo O algoritmo opera da seguinte maneira:

1. **Inicialização:**
   - Cada braço i é associado a um contador N_i, que indica o número de vezes que o braço foi selecionado, e uma estimativa de recompensa média Q_i, inicialmente definida como zero.
2. **Seleção de um Braço:**
   - Com probabilidade ϵ, o algoritmo escolhe um braço aleatoriamente para explorar (*exploring*).
   - Com probabilidade $1 - \epsilon$, o algoritmo escolhe o braço com a maior recompensa média observada até o momento Q_i (*exploiting*).
3. **Atualização:**
   - Após observar a recompensa R_t do braço selecionado, o contador N_i e a estimativa Q_i são atualizados:

   $$Q_i \leftarrow Q_i + \frac{1}{N_i}(R_t - Q_i)$$

   Esse é um método de média incremental que evita o armazenamento de todas as recompensas passadas.
4. **Iteração:**
   - O processo é repetido para o número de rodadas.

Controle de Exploração e Aproveitamento O parâmetro ϵ é fundamental para o comportamento do algoritmo:

- Quando ϵ é alto, o algoritmo explora mais, permitindo maior aprendizado sobre os braços menos conhecidos.
- Quando ϵ é baixo, o algoritmo explora menos, priorizando os braços com maior recompensa observada.
- Um ϵ decrescente ao longo do tempo (ϵ_t) é frequentemente utilizado para garantir maior exploração no início e mais exploração em estágios posteriores.

Algorithm 1 Epsilon-Greedy para Multi-Armed Bandit

Require: Número de braços k, parâmetro de exploração ϵ ($0 \leq \epsilon \leq 1$), número total de rodadas T
Ensure: Braço selecionado em cada rodada e recompensas acumuladas

```
                                                              ▷ Inicialização
 1: for i = 1 até k do
 2:     Q[i] ← 0                    ▷ Recompensa média estimada para o braço i
 3:     N[i] ← 0                    ▷ Número de vezes que o braço i foi escolhido
 4: end for
                                                              ▷ Iteração
 5: for t = 1 até T do
 6:     Gere um número aleatório r ∈ [0, 1)
 7:     if r < ε then
 8:         a ← Selecionar um braço aleatório em {1, 2, . . . , k}    ▷ Exploração
 9:     else
10:         a ← arg max_i Q[i]      ▷ Exploração: Selecionar o braço com maior Q[i]
11:     end if
12:     Obtenha a recompensa R ao puxar o braço a
                                    ▷ Atualizar estimativas do braço selecionado
13:     N[a] ← N[a] + 1
14:     Q[a] ← Q[a] + 1/N[a] (R − Q[a])
15: end for
```

2.4.2 UCB1

O algoritmo **UCB1 (*Upper Confidence Bound 1*)** 2 é uma abordagem determinística para resolver o problema de *Multi-Armed Bandits*. Ele utiliza princípios de otimização para equilibrar *exploring* e *exploting*, maximizando a recompensa acumulada. UCB1 baseia-se no conceito de intervalos de confiança, onde braços menos explorados recebem maior prioridade para exploração.

Intuição do Algoritmo A ideia principal do UCB1 é selecionar o braço que maximiza uma pontuação composta por duas partes:

1. **Recompensa média estimada** (Q_i): uma estimativa da recompensa média do braço baseada nas observações até o momento.
2. **Termo de exploração**: um valor adicional que diminui à medida que o braço é mais frequentemente selecionado, incentivando a exploração de braços menos conhecidos.

Esse balanceamento permite que o algoritmo explore novas opções no início e, com o tempo, se concentre nos braços que apresentam maior potencial de retorno.

Funcionamento do Algoritmo

1. **Inicialização**:
   - Cada braço é puxado pelo menos uma vez para inicializar suas estimativas.
2. **Cálculo do Valor UCB**:
   - Para cada braço i, calcula-se:

   $$UCB_i = Q_i + \sqrt{\frac{2 \ln t}{N_i}}$$

   Onde:
     - Q_i: recompensa média observada do braço i,
     - N_i: número de vezes que o braço i foi selecionado,
     - t: número total de rodadas até o momento.
3. **Seleção do Braço**:
   - Escolhe-se o braço i que maximiza UCB_i.
4. **Atualização**:
   - Após observar a recompensa R_t do braço selecionado, atualizam-se Q_i e N_i:

   $$Q_i \leftarrow Q_i + \frac{1}{N_i}(R_t - Q_i)$$
5. **Repetição**:
   - O processo é repetido até o número total de rodadas.

Algorithm 2 UCB1 (*Upper Confidence Bound 1*)

Require: k: Número de braços, T: Número total de rodadas
Ensure: Braço selecionado em cada rodada e recompensas acumuladas

▷ Inicialização: Selecionar cada braço uma vez

1: **for** $i = 1$ até k **do**
2: Observar recompensa $R[i]$
3: $Q[i] \leftarrow R[i]$ ▷ Recompensa média inicial do braço i
4: $N[i] \leftarrow 1$ ▷ Número de vezes que o braço i foi selecionado
5: **end for**

▷ Iteração para rodadas subsequentes

6: **for** $t = k + 1$ até T **do**
7: **for** $i = 1$ até k **do**
8: Calcular:

$$UCB[i] \leftarrow Q[i] + \sqrt{\frac{2\ln(t)}{N[i]}}$$

9: **end for**
10: Selecionar o braço:

$$a \leftarrow \arg\max_i \big(UCB[i]\big)$$

11: Observar recompensa R_a do braço selecionado
12: Atualizar:

$$N[a] \leftarrow N[a] + 1$$
$$Q[a] \leftarrow Q[a] + \frac{1}{N[a]}\big(R_a - Q[a]\big)$$

13: **end for**

2.5 Conteinerização

Conteinerização é um processo de implantação de software que agrupa o código de uma aplicação com todos os arquivos e bibliotecas de que ela precisa para ser executado em qualquer infraestrutura [12]. Com métodos tradicionais, os desenvolvedores escrevem código em um ambiente de computação específico que, quando transferido para um novo local, geralmente resulta em bugs e erros. Por exemplo, isso pode acontecer quando um desenvolvedor transfere o código de um computador desktop para uma VM (*Virtual Machine*) ou de um sistema operacional Linux para Windows. A conteinerização elimina esse problema ao agrupar o código da aplicação com os arquivos de configuração, bibliotecas e dependências relacionados necessários para sua execução [13].

A utilização de contêineres traz uma série de benefícios para o desenvolvimento e a implantação de aplicações, especialmente no contexto de ambientes modernos de TI que demandam agilidade, eficiência, resiliência e disponibilidade. Entre os principais pontos positivos está a leveza, pois os contêineres compartilham o kernel do sistema operacional host, reduzindo o consumo de recursos em comparação com as máquinas virtuais. Isso também resulta em inicialização mais ágil, acelerando processos de desenvolvimento, teste, escalonamento e atualização de aplicações. Além disso, a portabilidade proporcionada pelos contêineres permite que as aplicações sejam executadas de maneira consistente em diferentes ambientes, desde o desenvolvimento local até a produção em nuvem, reduzindo problemas de compatibilidade. Essa caracterização os torna ideais para arquiteturas baseadas em microsserviços, permitindo que desenvolvedores se concentrem em componentes específicos sem interferência mútua. Logo, um único contêiner com defeito não afeta os demais, contibuindo para o design de aplicações tolerantes a falhas.

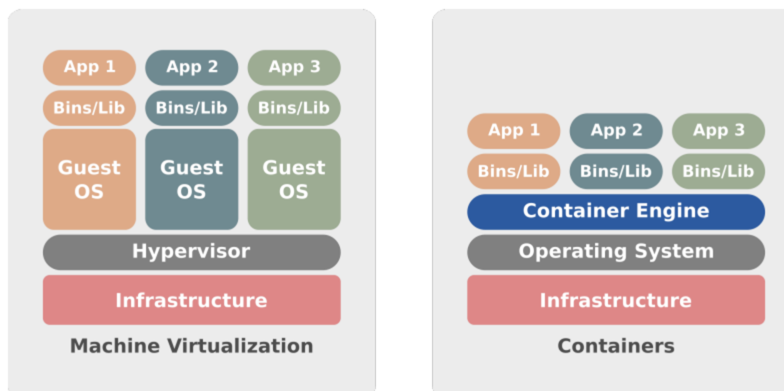

Figura 5: Diferença entre contêineres e máquinas virtuais [10].

As máquinas virtuais (VMs) são baseadas em um conceito de virtualização completa, onde um hipervisor emula o hardware físico de um sistema, permitindo que múltiplos sistemas operacionais completos sejam executados de maneira independente no mesmo servidor. Isso significa que cada VM carrega seu próprio kernel, sistema operacional e aplicações, resultando em um alto grau de isolamento e segurança entre as instâncias. No entanto, esse

nível de abstração também implica um consumo significativo de recursos, uma vez que cada VM requer uma quantidade substancial de CPU, memória e armazenamento para operar eficientemente.

Por outro lado, os contêineres utilizam uma abordagem mais leve, baseada na virtualização a nível de sistema operacional. Em vez de emular o hardware completo, os contêineres compartilham o mesmo kernel do sistema operacional host, isolando apenas os processos e as dependências necessárias para a execução de uma aplicação. Essa arquitetura reduz drasticamente o *overhead*, permitindo a criação de ambientes isolados que consomem menos recursos e se iniciam rapidamente. A Figura 5 mostra um panorama geral com as diferenças arquiteturais de contêineres e máquinas virtuais através de blocos hierárquicos.

O Docker Compose é uma ferramenta para definir e executar aplicativos multicontêineres. Ele permite o gerenciamento de serviços, redes e volumes em um único arquivo de configuração YAML [14]. Essa abordagem facilita tanto o controle quanto a replicação, uma vez que descomplica a tarefa de orquestrar e coordenar vários serviços. Ademais, o Compose permite a portabilidade entre ambientes, por meio de variáveis que servem para customizar a composição de diferentes ecossistemas. Isto posto, optou-se pelo uso do Docker Compose para a construção de 3 servidores cache - representados na forma de contêineres - e suas variáveis de ambiente, como latitude e longitude. Mais detalhes dessa implementação estão disponíveis na seção 5.5.

2.6 Cálculo de Distância entre Coordenadas na Terra

Uma das métricas utilizadas no cálculo de latência percebida pelo cliente ao fazer uma requisição para um servidor cache se baseia no posicionamento geográfico do cliente e do servidor, conforme será melhor apresentado na seção 5. Nesse contexto, é possível definir coordenadas para a localização dos servidores cache, bem como variar a posição do cliente a partir da simulação. A partir dessas coordenadas, pode-se calcular a distância entre dois pontos utilizando a Fórmula de Haversine.

A fórmula de haversine é uma importante equação usada em navegação, fornecendo distâncias entre dois pontos de uma esfera a partir de suas latitudes e longitudes. É um caso especial de uma fórmula mais geral de trigonometria esférica, a lei dos haversines, que relaciona os lados e ângulos de um triângulo contido em uma superfície esférica [19].

$$d = 2r \cdot \arcsin\left(\sqrt{\sin^2\left(\frac{\phi_2 - \phi_1}{2}\right) + \cos(\phi_1) \cdot \cos(\phi_2) \cdot \sin^2\left(\frac{\lambda_2 - \lambda_1}{2}\right)}\right) \tag{1}$$

Onde:

- d: Distância entre os dois pontos na superfície da esfera.
- r: Raio da esfera (para a Terra, geralmente $r = 6.371\,\text{km}$).
- ϕ_1, ϕ_2: Latitude do ponto 1 e latitude do ponto 2, respectivamente, em radianos.
- λ_1, λ_2: Longitude do ponto 1 e longitude do ponto 2, respectivamente, em radianos.

Pela teoria, devemos garantir que $h = hav(\theta) = \sin^2\left(\frac{\theta}{2}\right)$ não seja maior que 1, onde θ é o ângulo central entre dois pontos numa esfera. Isso ocorre quando tomamos pontos antipodais (pontos diametralmente opostos) — nesta região um número relativamente grande de erros tende a ocorrer na fórmula quando uma precisão finita é usada.

Vale ressaltar que essa distância trata-se de uma aproximação quando aplicada à Terra, que não é uma esfera perfeita, uma vez que o raio da Terra varia de 6356.752 km nos polos a 6378.137 km no equador. Além disso, para distâncias curtas, a distância obtida pela fórmula de Haversine tende a ser mínima. Para uma análise mais precisa - considerando a elipticidade da Terra - é possível utilizar as fórmulas de Vincenty [19]. Como um dos objetivos da simulação não é o cálculo da distância entre o cliente e o servidor com exatidão e sim estimar uma latência teórica, a Fórmula 1 é suficiente para o desenvolvimento do projeto.

3 Objetivos

Esse projeto objetiva o estudo e a criação de um simulador capaz fornecer suporte ao *Adaptative Video Streaming* utilizando a arquitetura de *Content Steering*, por meio da utilização de algoritmos de escolha. Com o simulador, busca-se reproduzir cenários que se baseiem em aspectos do mundo real, em que temos um cliente móvel consumindo conteúdo de vídeo de um serviço de *streaming*, e comparar o desempenho de diferentes algoritmos de escolha no processo de direcionamento de requisições aos servidores cache.

Desse modo, a criação do simulador e a exploração de diferentes cenários visa responder às questões abaixo:

- Quais métricas influenciam a latência percebida pelo cliente durante o streaming de vídeo e como podemos simulá-las?
- Quais são as informações que os algortimos utilizam para escolher o melhor servidor num determinado momento? Como saber que um algortimo convergiu?
- Por que escolher o servidor de cache mais próximo do cliente nem sempre é a melhor opção?
- Qual é o comportamento dos diferentes algoritmos de escolha?

4 Metodologia

4.1 Base do Projeto

O projeto do qual se trata este documento foi desenvolvido com base no trabalho [10], que implementa um simulador de um sistema de *content steering* para otimizar a entrega de conteúdos de vídeo utilizando servidores de cache e um orquestrador central. A seguir, detalhamos os principais componentes e ferramentas utilizadas no trabalho que serve como base para o projeto:

Orquestrador O **orquestrador** foi implementado utilizando **Flask**, um microframework web para Python que permite o desenvolvimento rápido de APIs e sistemas web. No contexto deste trabalho, o orquestrador é responsável por receber requisições de *content steering* do cliente e retornar uma lista de servidores de cache disponíveis.

Servidores de Cache O sistema utiliza três servidores Caddy para ler e enviar os fragmentos de vídeo.

- **Caddy** é um servidor web moderno e altamente configurável, projetado para oferecer uma experiência simples e eficiente. Ele suporta HTTPS por padrão, gerenciamento automático de certificados e uma configuração flexível, o que o torna ideal para implementar servidores de cache em sistemas de *content steering*.
- No contexto deste trabalho, os servidores Caddy foram configurados para:
  - Armazenar os fragmentos de vídeo e os arquivos de manifesto DASH.
  - Atender às requisições HTTP dos clientes para entrega de conteúdo.

Cliente O cliente é implementado como uma página web contendo um reprodutor **DASH**. Este reprodutor é responsável por:

- Fazer o download do **arquivo manifesto** e dos **fragmentos de vídeo** dos servidores de cache.
- Realizar requisições ao orquestrador para obter informações sobre os servidores de cache disponíveis e sua prioridade.
- Utilizar as informações recebidas do orquestrador para selecionar o servidor o para download de fragmentos de vídeo.

Fluxo de Operação

1. O cliente inicia o download de um vídeo, acessando os arquivos manifesto e fragmentos nos servidores de cache.
2. Periodicamente, o cliente faz uma requisição de *content steering* ao orquestrador.
3. O orquestrador mantém uma lista de servidores disponíveis e retorna à requisição do cliente com esta.
4. O cliente utiliza essa lista para direcionar as próximas requisições.

Contribuições do Trabalho Base O trabalho base proporciona uma estrutura base para a implementação de sistemas de content steering, destacando:

- A integração entre o orquestrador, os servidores de cache e o cliente DASH.
- O uso de **Caddy** como servidor de cache, devido à sua simplicidade de configuração e desempenho eficiente.
- O desenvolvimento de um cliente web que comunica-se dinamicamente com o orquestrador para otimizar a entrega de vídeo.

Essa base sólida permite explorar e estender o sistema com novos algoritmos de ordenação, diferentes métricas de avaliação e cenários de teste mais complexos, como foi realizado no projeto atual.

Incrementos Propostos O trabalho que serve como base para este projeto é projetado para ser resiliente à queda de servidores. Caso um dos servidores de cache se torne indisponível, o cliente é capaz de realizar um processo de *healing* (autocura), solicitando ao orquestrador uma nova lista de servidores disponíveis para continuar a entrega do conteúdo. Essa funcionalidade garante a continuidade do serviço mesmo em cenários de falha.

O objetivo do projeto atual é expandir essa arquitetura, adicionando um componente de *direcionamento inteligente* ao orquestrador. Com base nas latências experimentadas pelo cliente durante o consumo dos vídeos, o orquestrador passa a utilizar algoritmos para ordenar a lista de servidores disponíveis. Essa ordenação é feita de forma dinâmica, priorizando os servidores que oferecem a menor latência para o cliente, garantindo uma experiência mais fluida e otimizada. Essa abordagem busca não apenas melhorar o desempenho individual do cliente, mas também balancear a carga entre os servidores de cache, aumentando a eficiência do sistema como um todo.

4.2 Cenário Experimental

Com esse objetivo em foco, conduzimos experimentos seguindo as especificações detalhadas a seguir.

1. **Reprodutor DASH**:
   - Um cliente reprodutor que faz o download de fragmentos de vídeo para reprodução contínua.
   - Associado a coordenadas geográficas ($latitude, longitude$) que podem variar ao longo do tempo, simulando movimento.
   - A latência percebida pelo reprodutor é afetada pela distância física até os servidores cache e pelo tempo de processamento dos servidores.
2. **Servidores Cache**:

- Três servidores cache distribuídos geograficamente, cada um com coordenadas físicas específicas.
- Servem fragmentos de vídeo sob solicitação do reprodutor.
- A carga dos servidores pode variar ao longo do tempo, impactando seu tempo de resposta.

3. **Servidor Orquestrador de *Content Steering***:
   - Um componente central que recebe métricas do reprodutor e toma decisões sobre qual servidor cache o reprodutor deve utilizar.
   - Direciona o reprodutor para o servidor cache de acordo com algoritmo de escolha empregado e com base nas latências experimentadas pelo reprodutor.

4. **Medição e Monitoramento**:
   - O orquestrador recebe as latências experimentadas pelo reprodutor para cada servidor conforme os fragmentos de vídeo são baixados.
   - As decisões de redirecionamento são baseadas em métricas de punição e de recompensa conforme as latências observadas.

4.3 Dinâmica do Experimento

1. **Movimentação do Reprodutor**:
   - O reprodutor pode se deslocar periodicamente, alterando suas coordenadas geográficas.
   - A distância até cada servidor cache define um dos componentes da latência total observada.

2. **Solicitação de Fragmentos**:
   - O reprodutor solicita fragmentos de vídeo para reprodução contínua.
   - A decisão sobre qual servidor será utilizado é tomada pelo servidor orquestrador.

3. **Cálculo da Latência**:
   - A latência L experimentada pelo reprodutor é definida como:

   $$L = f(d_{client,server}) + f_{server_load} \tag{2}$$

   Onde:
     - $f(d_{client,server})$: Função que representa a latência baseada na distância física.
     - f_{server_load}: Função que representa o impacto do uso do servidor na latência.

4. **Redirecionamento (Content Steering)**:

- O orquestrador usa as latências reportadas como métricas para ordenar os servidores caches que deverão ser selecionados pelo cliente.
- As latências reportadas são convertidas em recompensa ou punição e utilizadas pelo algoritmo de escolha.

5 Implementações

5.1 Estimando a latência

A principal métrica utilizada para a análise foi o tempo de resposta (latência) dos servidores de cache durante o download dos fragmentos de vídeo. Para simular a latência, foi considerado um modelo que combina dois componentes principais: a distância geográfica entre o cliente e o servidor e o tempo de resposta do servidor, influenciado pela carga de trabalho. O componente de distância foi calculado utilizando a fórmula de Haversine, representando a latência teórica mínima com base na proximidade física. Já o tempo de resposta do servidor foi obtido dinamicamente, refletindo o impacto da carga em sua capacidade de atender requisições. Esse modelo integrado permitiu uma análise mais realista do desempenho dos servidores e da adequação dos algoritmos de *content steering* às condições variáveis do sistema.

5.1.1 Influência da distância

A lógica implementada para estimar um dos componentes da latência entre um cliente e um servidor baseia-se na distância geográfica entre eles, utilizando a fórmula de Haversine para calcular a distância em linha reta entre dois pontos na superfície terrestre. Essa distância é então utilizada para calcular uma latência mínima teórica, assumindo a transmissão de dados na velocidade da luz em fibras ópticas.

Inicialmente, a fórmula de Haversine 1 é empregada para determinar a distância entre as coordenadas de latitude e longitude do cliente e do servidor. O raio da Terra é considerado como 6371 km, e as coordenadas são convertidas de graus para radianos. A fórmula calcula a diferença angular em latitude e longitude e aplica funções trigonométricas para determinar a distância linear em quilômetros.

Com a distância calculada, a latência mínima teórica é estimada considerando a velocidade da luz em fibra óptica, aproximadamente 200.000 km/s. A fórmula utilizada divide a distância pela velocidade, convertendo o resultado para milissegundos. Esse cálculo fornece a latência mínima teórica possível, assumindo uma transmissão ideal sem interferências ou desvios no trajeto. Utilizamos essa estimativa como o componente de distância da equação (2)

É importante ressaltar que este modelo considera apenas a contribuição da distância geográfica para a latência, representando um limite teórico inferior. Outros fatores, como atrasos em roteadores, switches e congestionamento na rede, não são contemplados. Além disso, a trajetória real da fibra óptica pode ser significativamente mais longa do que a distância em linha reta calculada.

5.1.2 Influência da carga do servidor

Para representar a carga do servidor de cache, foi implementada uma estratégia em que, a cada download de um fragmento de vídeo, era enviada uma requisição *health* ao servidor selecionado. O tempo de resposta dessa requisição foi utilizado como um indicador do nível de carga do servidor, compondo assim a latência total experimentada pelo cliente.

Uma requisição *health* é uma solicitação simples, geralmente usada em sistemas distribuídos para verificar o estado e a capacidade de um servidor responder às requisições. Essas requisições geralmente retornam informações básicas sobre a saúde do sistema, como a disponibilidade do serviço. No contexto deste experimento, o foco estava no tempo de resposta, que reflete indiretamente a carga momentânea do servidor. Tempos de resposta mais elevados indicam maior sobrecarga, enquanto tempos mais baixos sugerem que o servidor está em um estado menos ocupado e mais capaz de atender a novas requisições de forma eficiente.

Essa abordagem permite integrar dinamicamente o impacto da carga do servidor na decisão sobre qual cache utilizar, contribuindo para a análise de algoritmos de *content steering* em cenários que simulam a realidade.

5.2 Teste de carga

Para realizar testes de carga nos servidores para compor os diferentes cenários, utilizamos a ferramenta **k6** [18], uma ferramenta de código aberto projetada para realizar testes de carga e desempenho em sistemas web e APIs. Ela permite simular múltiplos usuários simultâneos enviando requisições para um servidor, fornecendo métricas sobre tempos de resposta, erros e outras estatísticas de desempenho. O *k6* utiliza scripts escritos em JavaScript, proporcionando flexibilidade para configurar cenários complexos e customizar os testes conforme necessário.

5.2.1 Configuração do Teste de Carga

Neste experimento, a ferramenta *k6* foi utilizado para avaliar o desempenho dos servidores de cache em condições de carga simulada. O objetivo principal foi medir o impacto de múltiplos usuários simultâneos enviando requisições `health` ao servidor testado. Essas requisições foram configuradas para rodar em um loop, simulando múltiplas iterações de usuários ativos acessando o servidor.

5.2.2 Cenário de Teste

O teste foi configurado para simular um número específico de usuários simultâneos, cada um enviando requisições `health` ao servidor em intervalos regulares. O script desenvolvido para o *k6* seguia o seguinte fluxo:

1. **Inicialização**:
    - Configuração da URL de endpoint `health` do servidor de cache a ser testado.
    - Definição do número de usuários simultâneos e da duração do teste.

2. **Execução**:
    - Cada usuário virtual (VU, *Virtual User*) enviava uma requisição HTTP `GET` ao endpoint `health`.
3. **Loop de Requisições**:
    - Durante o período do teste, cada usuário continuava enviando requisições no loop, simulando um comportamento de uso contínuo.

5.3 Arquitetura

A Figura 6 representa a arquitetura do projeto desenvolvido. O sistema é composto pelos seguintes elementos principais:

- **Cliente Móvel:**
    - Um cliente cuja posição geográfica pode variar ao longo do tempo. Esse cliente é responsável por reproduzir o vídeo e interagir com o orquestrador para otimizar sua experiência.
- **Servidores de Cache:**
    - Três servidores de cache, cada um com uma posição geográfica fixa, definidos por suas coordenadas. Esses servidores são responsáveis por armazenar e enviar os fragmentos de vídeo requisitados pelo cliente.
- **Orquestrador:**
    - O orquestrador desempenha duas funções principais:
        * **Estimativa da Latência por Distância:**
            · Com base nas coordenadas geográficas enviadas pelo cliente e pela posição fixa dos servidores de cache, o orquestrador calcula o componente de latência relacionado à distância entre o cliente e o servidor escolhido.
        * **Processamento de Latência Total:**
            · O orquestrador recebe do cliente a latência total experimentada durante o consumo de vídeo, que inclui fatores como a distância e o tempo de resposta do servidor. Essa métrica é utilizada como base para os algoritmos de escolha implementados no orquestrador.

O objetivo do orquestrador é, ao realizar o *content steering*, retornar ao cliente uma lista ordenada de servidores de cache disponíveis, priorizando aqueles que oferecem a melhor experiência em termos de latência. O cliente, ao receber essa lista, realiza o download dos fragmentos de vídeo do servidor recomendado e relata ao orquestrador sua experiência de latência, permitindo um ciclo contínuo de aprendizado e otimização dinâmica.

As sessões a seguir detalharão sobre a implementação de cada componente.

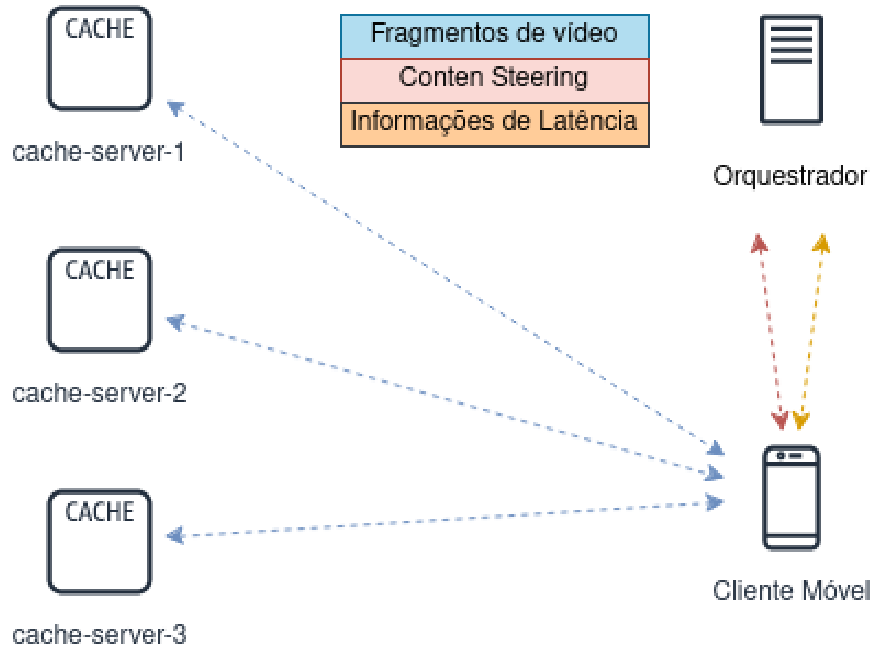

Figura 6: Arquitetura de um sistema do entrega de vídeos com Content Steering.

5.4 Cliente - Reprodutor

5.4.1 Player de vídeo

Na captação de latência dos servidores cache, o player dash foi utilizado para reproduzir o download dos trechos de vídeo. Este recebe um arquivo *manifest.pd* que primeiro identifica os servidores cache disponíveis, e recebe uma lista de prioridades dada pelo orquestrador via request do tipo *fetch*. Foi realizado um primeiro estudo do funcionamento deste streaming de vídeo verificando quando havia uma solicitação de carregamento ao servidor cache. Com isso, foram verificados os tempos em que o orquestrador realiza montagem de um arquivo em formato JSON, emite as informações de prioridade de servidores e *requests* que chegam ao orquestrador.

5.4.2 Coordenadas

Para a criação da interface do cliente, foram implementadas lógicas em JavaScript com tags HTML para controle, registrando como simulação um ponto inicial e final por meio de coordenadas longitudinais e latitudinais. Com base nesses pontos, foi gerada uma rota pré-estabelecida composta por 1.000 coordenadas intermediárias, formando um percurso que, a cada intervalo de tempo definido, enviava as coordenadas ao orquestrador. Tais coordenadas são geradas por uma função *thread* que recebe um valor arbitrário do intervalo

de tempo no qual as coordenadas serão mandadas na rota para o orquestrador. A escolha de um valor arbitrário ajudou na relação dos cálculos de latência, podendo também ser utilizados valores randômicos no cálculo para obtenção de uma rota com ruído.

```
function coords(flat, flong, initlat, initlong) {
    let time = 100;
    let steplat,steplong;
    steplat = (flat - initlat)/1000;
    steplong = (flong - initlong)/1000;
    latitude = parseFloat(initlat);
    longitude = parseFloat(initlong);


    function updatecoords(){
        latitude = latitude + steplat;
        longitude = longitude + steplong;
        document.getElementById("instlat").value = latitude;
        document.getElementById("instlong").value = longitude;
    }

    if (intervalID != null){
        clearInterval(intervalID)
    }

    intervalID = setInterval(updatecoords, 100);

}
```

Figura 7: Javascript usado para emissão das coordenadas.

Simulador de Movimento

latitude Init | latitude Final | Start Simulator | Stop Simulator
longitude Init | longitude Final

Instant Latitude:

Instant Longitude:

Figura 8: Simulador de Coordenadas.

5.4.3 Retorno ao cliente

Para o retorno, foi utilizado uma interface contendo a prioridade dos servidores. Sabendo que o orquestrador retorna uma lista ordenada com as prioridades de cache, foi feito uma sequência para ilustrar - durante a execução do vídeo - qual servidor está sendo selecionado. Essa abordagem não apenas facilita a identificação visual do servidor escolhido, mas também possibilitou a geração de gráficos relacionados à latência e ao desempenho dos ser-

vidores. Por exemplo, a Figura 9 relata a existência de três servidores cache, sendo que o `video-streaming-cache-1` é o que está sendo utilizado no momento.

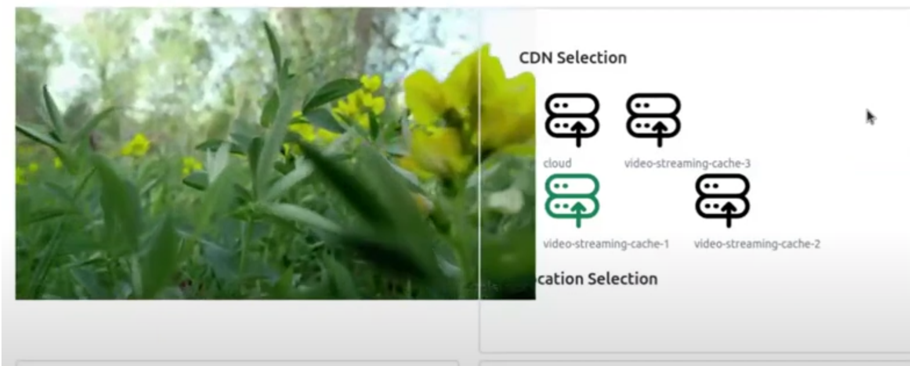

Figura 9: Exemplo de execução de escolha do servidor.

Quanto as métricas de latência, foi criado um timer na função dash, que começa ao iniciar a função de download e retorna ao finalizar. Essa escolha nos ajudou a verificar uma latência real ao congestionar um dos servidores, que ao final envia ao servidor as coordenadas em tempo de execução.

```
129         function _onFragmentLoadingStarted(e) {
130             try {
131                 if (e && e.mediaType && (e.mediaType === 'video' || e.mediaType === 'audio') && e.request) {
132                     if (e.request.serviceLocation) {
133                         const element = document.getElementById(`${e.mediaType}-service-location`);
134                         element.innerText = e.request.serviceLocation;
135 
136                         if (currentSelectedServiceLocation.video != undefined){
137                             let start = performance.now();
138                             let host = "https://"+currentSelectedServiceLocation.video+"/Eldorado/health-check.json";
139                             fetch(host)
140                                 .then(data => {
141                                     const end = performance.now();
142                                     const duration = end - start; // duração em milissegundos
143                                     console.log(`Request duration: ${duration.toFixed(2)} ms`);
144                                 })
145                                 fetch("https://steering-service:30500/coords", {
146                                 method: "POST",
147                                 body: JSON.stringify({
148                                     "lat" : latitude,
149                                     "long" : longitude,
150                                     "rt" : duration
151                                 }),
152                                 headers: {
153                                     "Content-type": "application/json; charset=UTF-8"
154                                 }
155                                 }).then(data => {console.log(data)});
156                                 // lat, long, duration
157                         }
```

Figura 10: Javascript para envio de parâmetros de latência ao orquestrador.

5.5 Servidores Cache

Os servidores de cache utilizados neste projeto foram desenvolvidos utilizando Caddy, um servidor web moderno e configurável. A configuração de cada servidor é realizada por meio de um arquivo chamado Caddyfile, que é um arquivo de texto simples no qual são definidos

os parâmetros de operação do servidor, como domínios, rotas, cabeçalhos, e comportamento de proxy ou cache. O uso do Caddyfile simplifica a configuração e a personalização dos servidores, tornando o Caddy uma escolha prática e eficiente.

```
(cors) {
    @cors_preflight method OPTIONS

    header {
        Access-Control-Allow-Origin "{header.origin}"
        Vary Origin
        Access-Control-Expose-Headers "Authorization"
        Access-Control-Allow-Credentials "true"
    }

    handle @cors_preflight {
        header {
            Access-Control-Allow-Methods "GET, POST, PUT, PATCH, DELETE"
            Access-Control-Max-Age "3600"
        }
        respond "" 204
    }
}


:80 {
    root * /srv
    file_server

    header Access-Control-Allow-Origin *
}

:443 {
    tls /root/certs/video-streaming-cache-1.pem /root/certs/video-streaming-cache-1-key.pem

    root * /srv
    file_server

    import cors {header.origin}
    header Access-Control-Allow-Origin *
}
```

Figura 11: Exemplo de arquivo Caddyfile.

Cada servidor Caddy foi configurado para ter acesso ao dataset de vídeo por meio de volumes, permitindo que os arquivos necessários para a entrega dos fragmentos de vídeo sejam acessíveis diretamente no ambiente de execução. Os servidores são executados em contêineres, garantindo portabilidade, isolamento e facilidade de gerenciamento.

Além disso, cada contêiner é associado a uma localização geográfica específica, definida por valores de latitude e longitude. Essas informações são utilizadas pelo orquestrador para calcular o componente de latência baseado na distância entre o cliente e o servidor cache escolhido.

5.6 Orquestrador

Em uma arquitetura de *content steering*, o orquestrador é um componente central responsável por gerenciar e otimizar a alocação de recursos, como servidores de cache, para atender às solicitações dos clientes. O objetivo principal do orquestrador é direcionar o

```yaml
services:
  cache-1:
    container_name: video-streaming-cache-1
    image: caddy
    mem_limit: 512m
    mem_reservation: 128M
    memswap_limit: 1g
    cpus: "0.3"
    cpuset: "1"
    volumes:
      - ./Caddyfile-1:/etc/caddy/Caddyfile
      - ./certs:/root/certs
      - ../dataset1:/srv
    environment:
      LATITUDE: "-23"
      LONGITUDE: "-47"

  cache-2:
    container_name: video-streaming-cache-2
    image: caddy
    mem_limit: 512m
    mem_reservation: 128M
    memswap_limit: 1g
    cpus: "0.3"
    cpuset: "1"
    volumes:
      - ./Caddyfile-2:/etc/caddy/Caddyfile
      - ./certs:/root/certs
      - ../dataset2:/srv
    environment:
      LATITUDE: "-33"
      LONGITUDE: "-71"

  cache-3:
    container_name: video-streaming-cache-3
    image: caddy
    mem_limit: 512m
    mem_reservation: 128M
    memswap_limit: 1g
    cpus: "0.3"
    cpuset: "1"
    volumes:
      - ./Caddyfile-3:/etc/caddy/Caddyfile
      - ./certs:/root/certs
      - ../dataset3:/srv
    environment:
      LATITUDE: "5"
      LONGITUDE: "-74"
```

Figura 12: Exemplo de arquivo docker-compose utilizado.

cliente ao servidor mais adequado com base em métricas como latência garantindo uma experiência de alta qualidade para o usuário e o balanceamento eficiente do sistema.

Adaptação do Orquestrador no Contexto do Projeto No contexto deste projeto, o orquestrador foi adaptado para receber informações dinâmicas do cliente, processar esses dados e tomar decisões inteligentes sobre qual servidor de cache deve atender às suas requisições.

Recebimento de Informações do Cliente

- O orquestrador recebe do cliente dados sobre a **latência experimentada** ao se comunicar com os servidores de cache. Essas informações permitem avaliar o desempenho atual de cada servidor sob diferentes condições de carga.
- Ele também recebe as coordenadas de **latitude e longitude** do cliente, permitindo calcular uma latência mínima teórica baseada na distância geográfica até cada servidor de cache.

Ordenação de Servidores de Cache O orquestrador implementa diferentes algoritmos para ordenar os servidores de cache com base nas métricas recebidas e calculadas:

- ***Random***: Escolhe servidores aleatoriamente, sem considerar métricas de desempenho.
- ***Epsilon-Greedy***: Balanceia exploração (testar servidores menos utilizados) e aproveitamento (priorizar servidores com menor latência média conhecida).
- ***UCB1 (Upper Confidence Bound)***: Explora inicialmente todos os servidores e, em seguida, prioriza aqueles que oferecem a melhor combinação de proximidade geográfica e desempenho, considerando a carga, equilibrando exploração e aproveitamento.

Redirecionamento Inteligente Após processar as informações e ordenar os servidores, o orquestrador devolve ao cliente uma lista priorizada de servidores de cache. O cliente utiliza essa lista para realizar seus downloads de fragmentos de vídeo, começando pelo servidor mais recomendado.

5.7 Algoritmos de escolha

Com base em [17], foram implementados três algoritmos de escolha: *Random*, *Epsilon-Greedy* e *UCB1*. Esses algoritmos utilizam as latências experimentadas pelo cliente durante a reprodução do vídeo para ordenar dinamicamente a lista de servidores disponíveis, com o objetivo de minimizar a latência ao longo do tempo e proporcionar uma experiência otimizada ao usuário.

5.7.1 Epsilon-Greedy

O algoritmo ***Epsilon-Greedy*** foi adaptado para cenários onde a escolha de servidores cache é baseada em uma métrica de punição, que pode refletir fatores como latência ou sobrecarga. Nesta adaptação, os braços correspondem aos servidores, e o algoritmo alterna entre **exploração** (testar servidores aleatoriamente) e **exploração** (selecionar o servidor com o menor valor médio de punição).

Estrutura do Algoritmo

1. **Inicialização**:
   - Cada servidor (ou braço) é associado a dois atributos:
     - `counts`: número de vezes que o servidor foi selecionado.
     - `values`: valor médio de punição associado ao servidor.
2. **Seleção de um Servidor**:
   - O algoritmo foi instanciado com $\epsilon = 0.3$;
   - Com probabilidade $1 - \epsilon$, o algoritmo realiza aproveitamento selecionando o servidor com o menor valor médio de punição conhecido.
   - Com probabilidade ϵ, o algoritmo realiza exploração escolhendo um servidor aleatoriamente, permitindo avaliar opções menos exploradas.
3. **Atualização das Métricas**:
   - Após selecionar um servidor, o algoritmo atualiza os valores de `counts` e `values` com base na punição observada para o servidor escolhido.

Detalhes da Implementação

1. **Exploração vs Aproveitamento**:
   - O parâmetro ϵ controla o equilíbrio entre exploração e exploração:
     - **Aproveitamento** $(1 - \epsilon)$: Seleciona o servidor com menor valor médio de punição conhecido.
     - **Exploração** (ϵ): Realiza escolhas aleatórias para explorar servidores menos utilizados.
2. **Atualização das Punições**:
   - A punição observada para o servidor selecionado é incorporada no cálculo de sua média, permitindo que o algoritmo se adapte dinamicamente às condições variáveis do sistema.
3. **Cálculo Incremental**:
   - Os valores médios são atualizados de forma incremental, evitando a necessidade de armazenar todas as observações anteriores.

Algorithm 3 EpsilonGreedy Adaptaptado para Métricas de Latência

```
1: Atributos:
2: ϵ: Probabilidade de realizar exploração
3: counts: Número de seleções para cada servidor
4: values: Punição média para cada servidor

5: procedure INICIALIZAR(servidores)
6:     for all servidor ∈ servidores do
7:         counts[servidor] ← 0
8:         values[servidor] ← 0.0
9:     end for
10: end procedure

11: procedure SELECIONAR_SERVIDOR(servidores)
12:     if random() > ϵ then
13:         return servidor com menor valor médio de punição
14:     else
15:         return servidor aleatório da lista
16:     end if
17: end procedure

18: procedure ATUALIZAR(servidor_escolhido, punição)
19:     counts[servidor_escolhido] ← counts[servidor_escolhido] + 1
20:     n ← counts[servidor_escolhido]
21:     value ← values[servidor_escolhido]
22:     values[servidor_escolhido] ← ((n−1)/n) · value + (1/n) · punição
23: end procedure
```

5.7.2 Random

A implementação do algoritmo *Random* foi realizada por meio da instanciação do algoritmo *Epsilon-Greedy* com o parâmetro $\epsilon = 1.0$. Com essa configuração, o algoritmo opera exclusivamente em modo de exploração, realizando escolhas de forma totalmente aleatória em todos os momentos.

5.7.3 UCB1

O algoritmo ***UCB1 (Upper Confidence Bound 1)*** foi adaptado para lidar com cenários onde a métrica de desempenho de servidores cache é baseada na latência observada pelo cliente. Nesta adaptação, os braços correspondem aos servidores disponíveis, e o algoritmo utiliza a latência medida para calcular recompensas inversamente proporcionais ao tempo de resposta, permitindo a seleção eficiente do servidor que proporciona a melhor experiência.

Estrutura do Algoritmo

1. **Inicialização**:
   - Cada servidor (ou braço) possui dois atributos:
     - `counts`: número de vezes que o servidor foi selecionado.
     - `values`: recompensa média associada ao servidor, calculada com base na latência.
2. **Seleção de um Servidor**:
   - Se um servidor ainda não foi selecionado, ele é priorizado.
   - Caso contrário, utiliza-se a fórmula do UCB para calcular o valor esperado ajustado por um termo de exploração:

     $$UCB_i = Q_i + \sqrt{\frac{2\ln(N)}{N_i}}$$

     Onde:
     - Q_i: recompensa média do servidor i,
     - N: número total de seleções realizadas,
     - N_i: número de seleções do servidor i.
3. **Atualização das Métricas**:
   - Após selecionar um servidor e observar sua latência, o algoritmo converte a latência em uma recompensa usando a fórmula:

     $$\text{reward} = \frac{1000}{\text{latency}}$$

   - Os valores de `counts` e `values` são atualizados para refletir o impacto da nova observação.

Algorithm 4 UCB1 Adaptado para Métricas de Latência

Atributos:
1: `counts`: Número de seleções para cada servidor.
2: `values`: Recompensa média para cada servidor.
3: **procedure** INICIALIZAR(servidores)
4: **for all** servidor em servidores **do**
5: `counts[servidor]` $\leftarrow 0$
6: `values[servidor]` $\leftarrow 0.0$
7: **end for**
8: **end procedure**
9: **procedure** SELECIONARSERVIDOR(servidores)
10: **for all** servidor em servidores **do**
11: **if** `counts[servidor]` $== 0$ **then**
12: **return** servidor ▷ Prioriza servidores ainda não selecionados.
13: **end if**
14: **end for**
15: `totalCounts` $\leftarrow \sum$ `counts.values()`
16: **for all** servidor em servidores **do**
17: `bonus` $\leftarrow \sqrt{\frac{2\ln(\texttt{totalCounts})}{\texttt{counts[servidor]}}}$
18: `UCBValues[servidor]` $\leftarrow$ `values[servidor]` $+$ `bonus`
19: **end for**
20: **return** servidor com maior `UCBValues`
21: **end procedure**
22: **procedure** ATUALIZAR(servidorEscolhido, latência)
23: `reward` $\leftarrow$ 1000/`latência`
24: `counts[servidorEscolhido]` $\leftarrow$ `counts[servidorEscolhido]` $+ 1$
25: $n \leftarrow$ `counts[servidorEscolhido]`
26: $value \leftarrow$ `values[servidorEscolhido]`
27: `values[servidorEscolhido]` $\leftarrow \frac{(n-1)}{n} \cdot \texttt{value} + \frac{1}{n} \cdot \texttt{reward}$
28: **end procedure**

Detalhes da Implementação

1. **Conversão de Latência em Recompensa**:
   - A recompensa é calculada como 1000/latência, garantindo que menores latências resultem em maiores recompensas.
2. **Seleção de Servidor**:
   - Servidores com menos seleções são priorizados inicialmente para garantir exploração.
   - Após explorar todos os servidores, o algoritmo seleciona aquele com o maior valor UCB_i.
3. **Atualização Dinâmica**:
   - O valor médio da recompensa de cada servidor é atualizado de forma incremental, permitindo ajustes em tempo real com base na performance observada.

6 Resultados

Para avaliar o desempenho do sistema de *content steering*, foram definidos e executados dois cenários distintos. Esses cenários simulam diferentes condições de carga nos servidores de cache e movimentação do cliente, permitindo analisar o impacto dessas variáveis no comportamento dos algoritmos implementados.

6.1 Estrutura do Sistema

Os testes consideram os seguintes componentes:

1. **Cliente**:
   - Um reprodutor DASH que se comunica com os servidores de cache para obter fragmentos de vídeo.
   - Pode permanecer imóvel ou se deslocar ao longo do experimento.
2. **Orquestrador**:
   - Responsável por receber as latências experimentadas pelo cliente em relação a cada servidor de cache e orquestrar a comunicação.
3. **Servidores de Cache**:
   - Três servidores de cache distribuídos geograficamente:
     - `cache-server-1`
     - `cache-server-2`
     - `cache-server-3`
   - Podem ser submetidos a diferentes níveis de carga (usuários simultâneos).

6.2 Cenário 1: Cliente Imóvel com Estresse em um Servidor

Neste cenário, o cliente permanece imóvel em um ponto equidistante dos três servidores de cache. Simultaneamente, o servidor `cache-server-1` é submetido a uma carga de **300 usuários simultâneos**, simulando uma situação de estresse operacional. O objetivo é avaliar como o sistema distribui as requisições do cliente em um ambiente onde um dos servidores apresenta incremento de latência devido à sobrecarga.

Aspectos avaliados:

- Capacidade do sistema de redirecionar o cliente para os servidores menos congestionados.

- Impacto do estresse em `cache-server-1` nas métricas de latência e na latência experimentada pelo cliente.

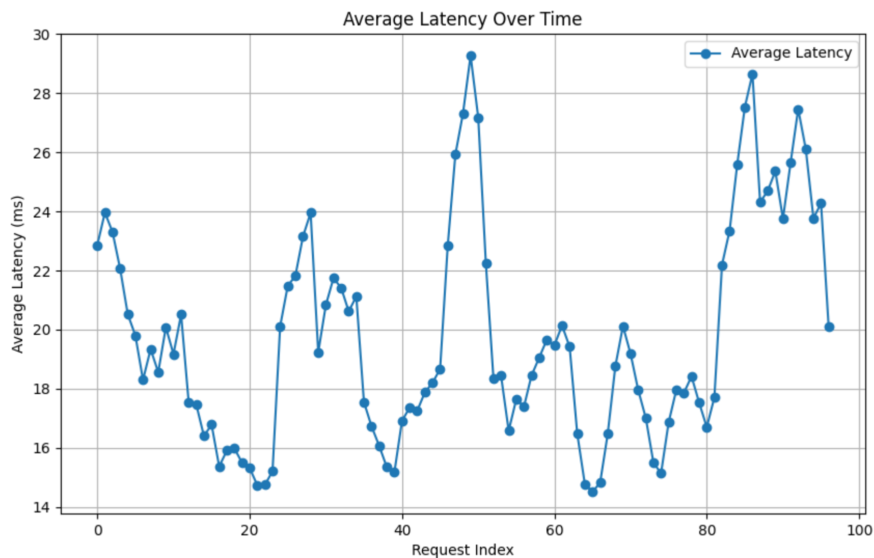

Figura 13: Cenário 1 com algoritmo *Random*

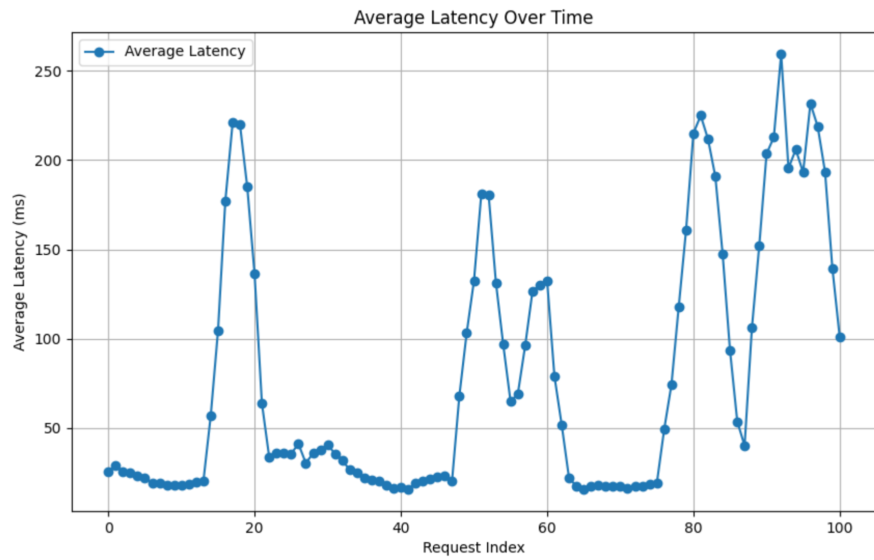

Figura 14: Cenário 1 com algoritmo *Epsilon Greedy*

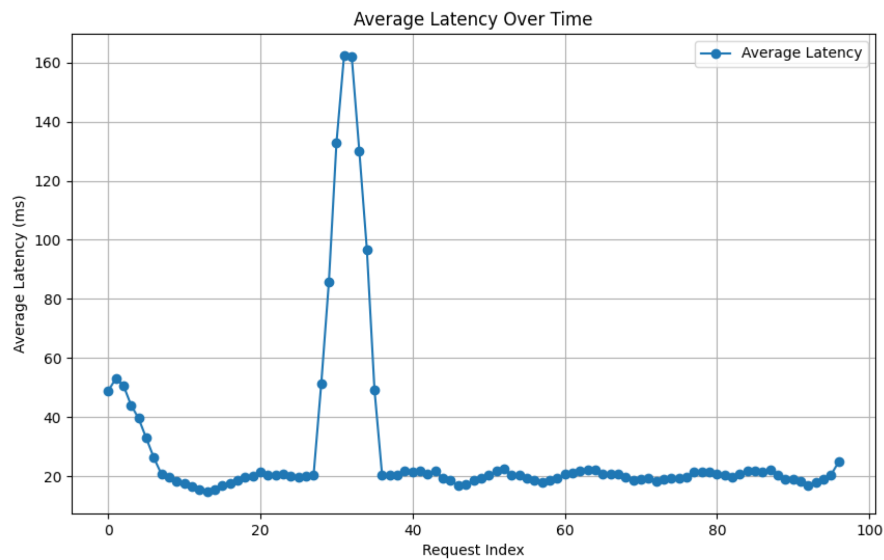

Figura 15: Cenário 1 com algoritmo *UCB1*

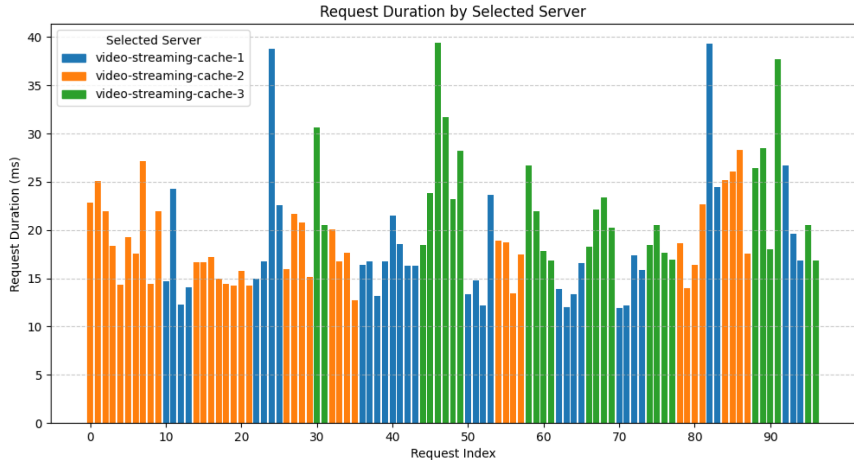

Figura 16: Cenário 1 com algoritmo *Random*

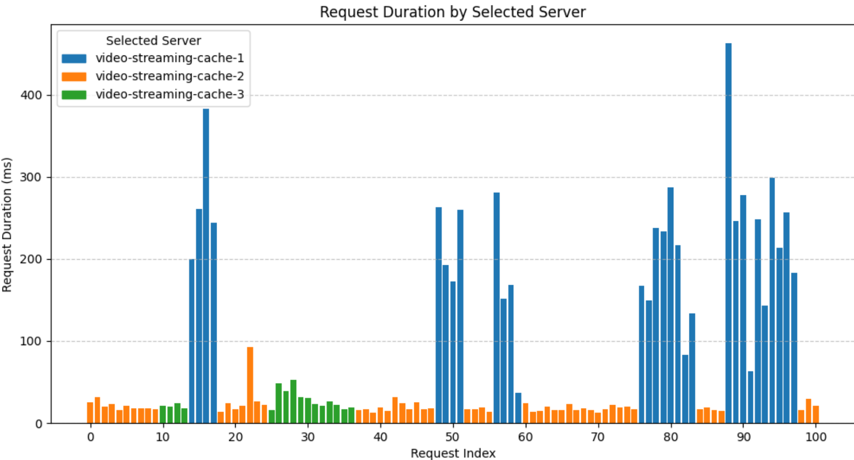

Figura 17: Cenário 1 com algoritmo *Epsilon Greedy*

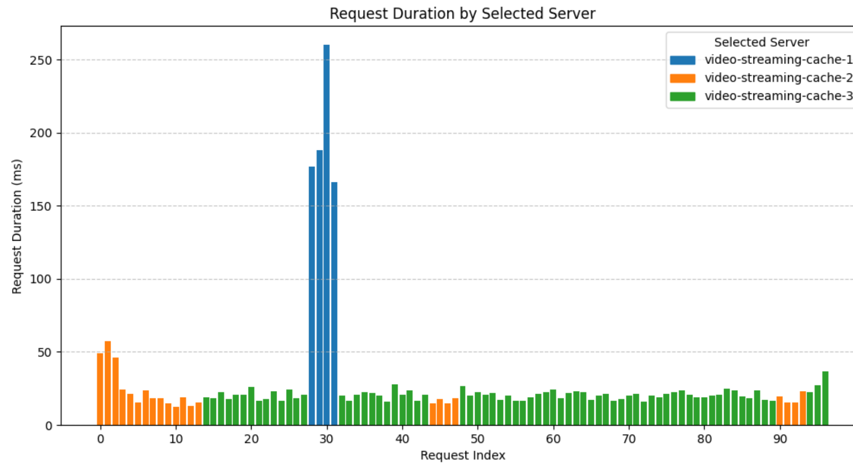

Figura 18: Cenário 1 com algoritmo *UCB1*

Resultados A análise da latência média ao longo dos downloads de fragmentos revela diferenças no desempenho dos algoritmos testados. O algoritmo *Random* (Figura 13) apresenta a pior experiência para o usuário, pois, em diversos momentos, realiza escolhas aleatórias que acabam direcionando o cliente para o servidor mais congestionado, resultando em altas latências. Por outro lado, o algoritmo *Epsilon Greedy* (Figura 14) demonstra uma experiência superior em comparação ao *Random*, priorizando servidores com menor latência média conhecida. No entanto, ocasionalmente, ele também seleciona servidores congestionados, ainda que com menor frequência.

Já o algoritmo *UCB1* (Figura 15) se destaca pelo equilíbrio entre exploração e aproveitamento. Ele apresenta, inicialmente, um período de descoberta, durante o qual explora todos os servidores para avaliar suas latências. Após esse período inicial, o *UCB1* converge para o servidor que oferece a melhor experiência de latência, permanecendo consistentemente nele e garantindo uma qualidade superior de serviço ao longo do tempo.

Também é possível observar diferenças no tempo de resposta de cada requisição em função do algoritmo de escolha que direciona o cliente para um servidor de cache. O algoritmo *UCB1* se destaca nesse aspecto, apresentando um tempo de resposta melhor ao longo do tempo, como pode ser observado nas Figuras 16 17 18

6.3 Cenário 2: Cliente Móvel com Cenários de Estresse e Sem Estresse

Neste cenário, o cliente inicia em um ponto equidistante dos três servidores de cache e se move gradualmente em direção ao `cache-server-1`. Dois subcenários são avaliados:

1. **Sem Estresse**:

    - Nenhum dos servidores está submetido a carga adicional.

  - O objetivo é observar como o sistema ajusta o *content steering* com base apenas na proximidade geográfica, priorizando o servidor mais próximo do cliente.

2. **Com Estresse em `cache-server-1`**:

  - O servidor `cache-server-1` está submetido a uma carga de **300 usuários simultâneos**.
  - O objetivo é avaliar se o sistema considera tanto a proximidade quanto o impacto da sobrecarga no redirecionamento do cliente, balanceando distância e carga.

Aspectos avaliados:

- Capacidade do sistema de adaptação a mudanças na posição do cliente.
- Balanceamento entre a proximidade geográfica e o estresse nos servidores.
- Diferenças de desempenho nos cenários com e sem estresse.

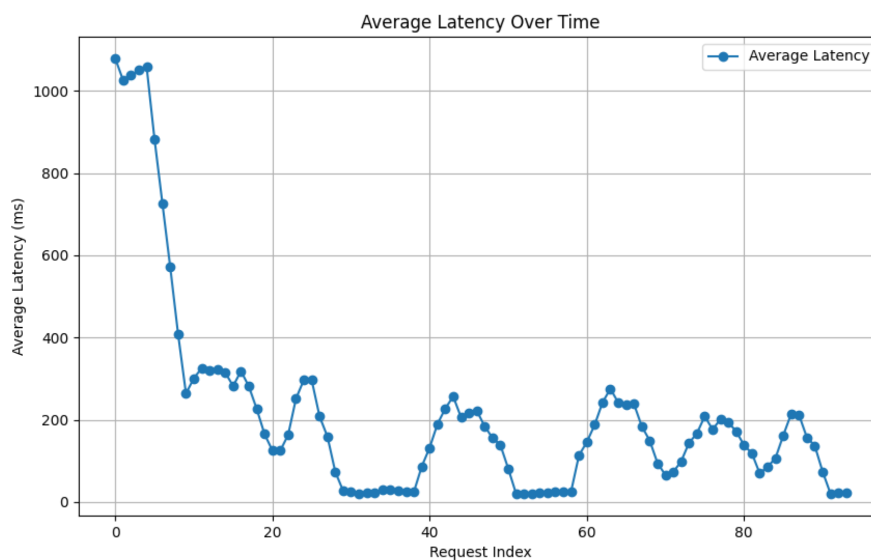

Figura 19: Cenário 2 com algoritmo *Random* e estresse no cache-server-1

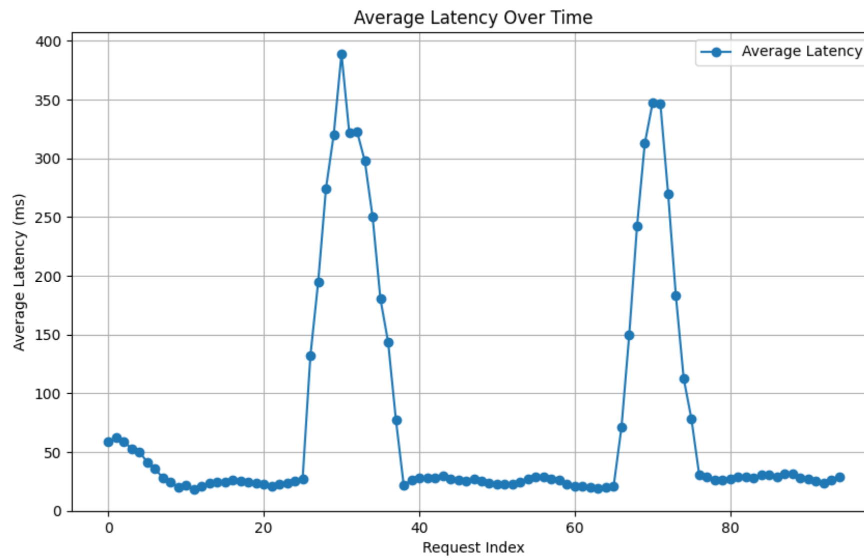

Figura 20: Cenário 2 com algoritmo *Epsilon Greedy* e estresse no cache-server-1

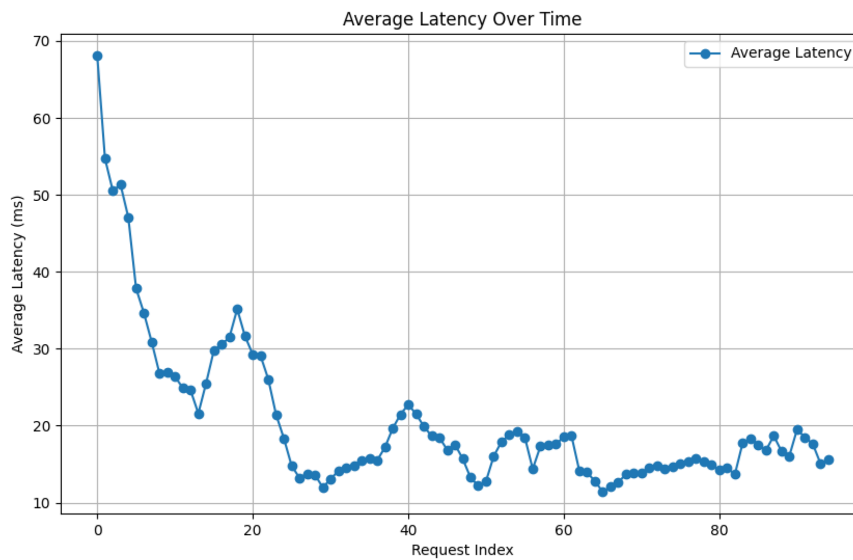

Figura 21: Cenário 2 com algoritmo *UCB1* sem estresse nos servidores de cache

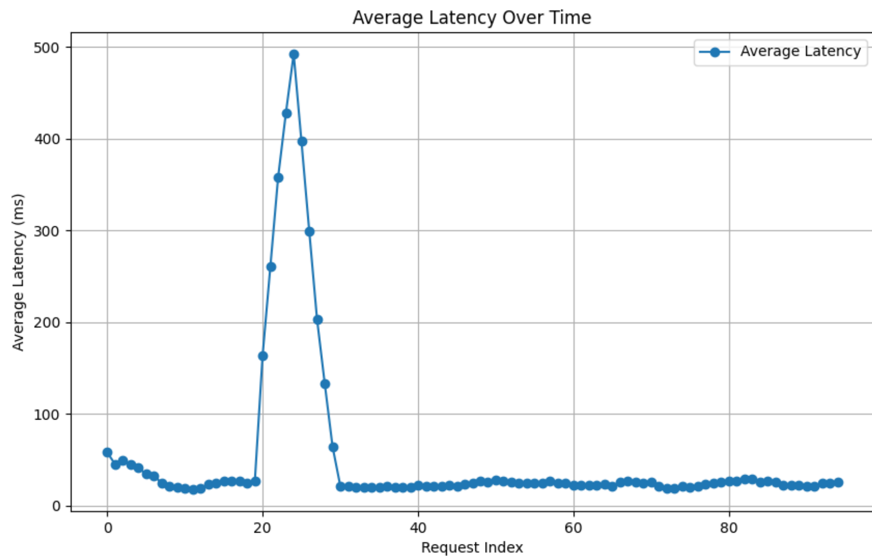

Figura 22: Cenário 2 com algoritmo *UCB1* e estresse no cache-server-1

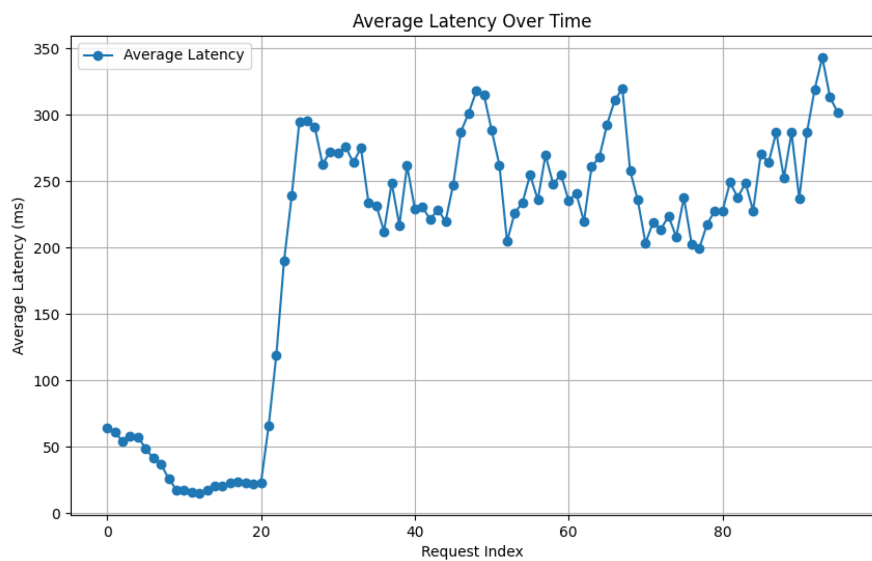

Figura 23: Cenário 2 com algoritmo *UCB1* considerando apenas a componente de distância para latência e com estresse no cache-server-1

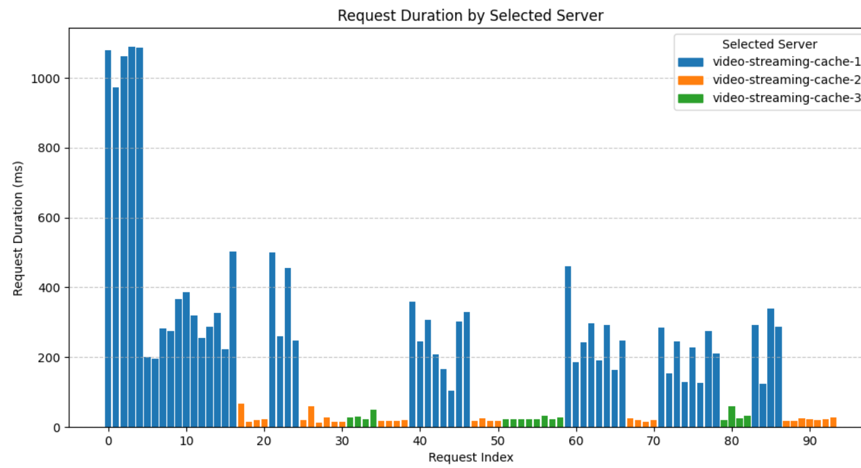

Figura 24: Cenário 2 com algoritmo *Random* e estresse no cache-server-1

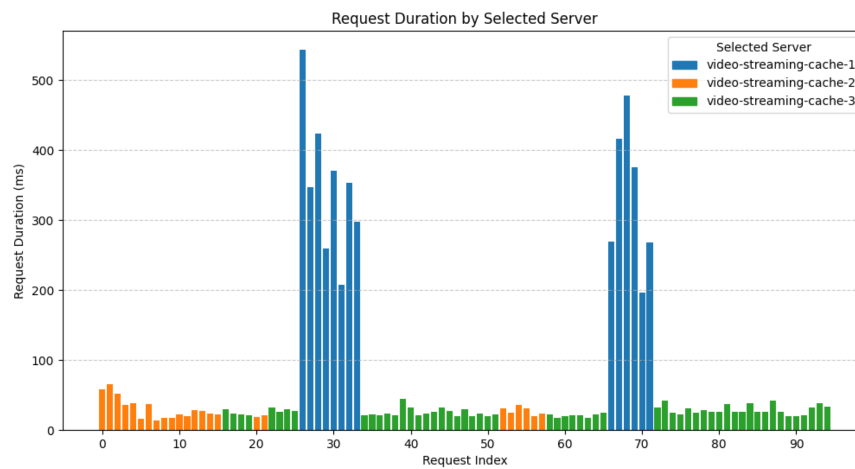

Figura 25: Cenário 2 com algoritmo *Epsilon Greedy* e estresse no cache-server-1

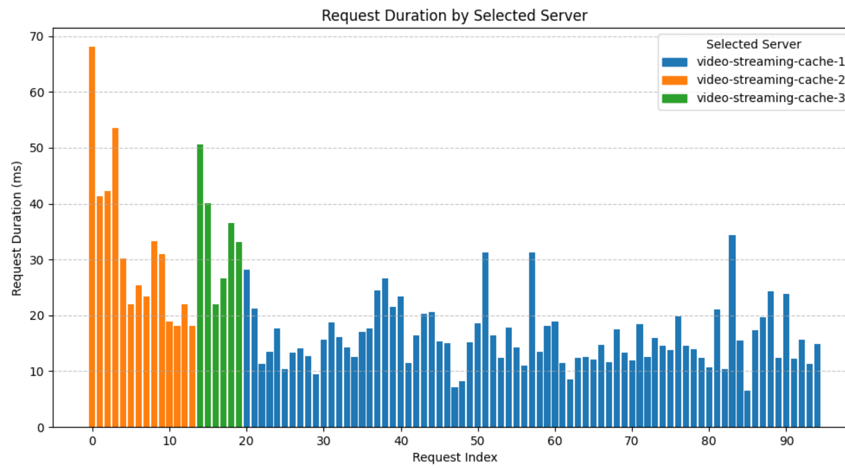

Figura 26: Cenário 2 com algoritmo *UCB1* sem estresse nos servidores de cache

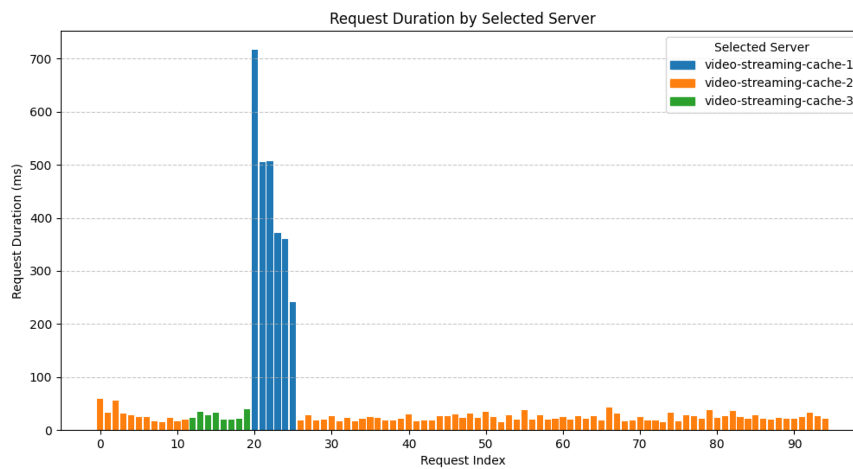

Figura 27: Cenário 2 com algoritmo *UCB1* e estresse no cache-server-1

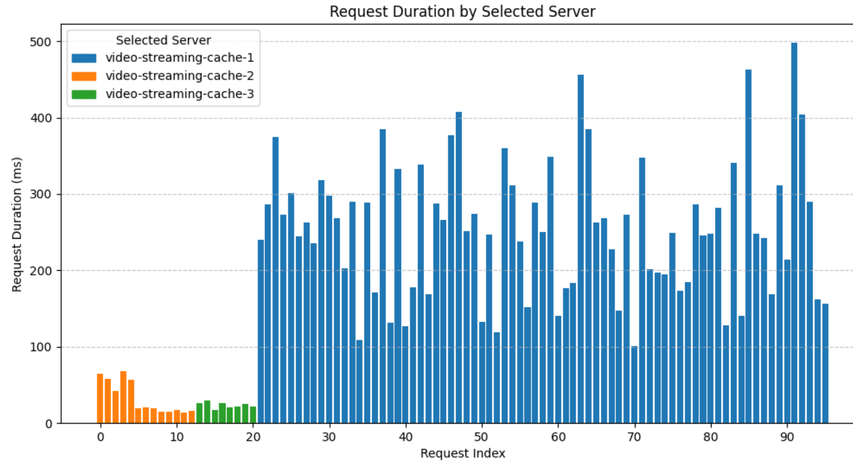

Figura 28: Cenário 2 com algoritmo *UCB1* considerando apenas a componente de distância para latência e com estresse no cache-server-1

Resultados Neste cenário, o objetivo era avaliar como os algoritmos se comportam ao tentar equilibrar os componentes de latência relacionados à distância geográfica e à carga dos servidores de cache.

O algoritmo *Random* apresentou desempenho semelhante ao observado no cenário anterior (Figuras 19 e 24). Ao realizar escolhas aleatórias, ele dedicou grande parte do tempo a servidores com maior latência experimentada, devido à ausência de um mecanismo que considere métricas como carga ou distância.

O algoritmo *Epsilon-Greedy* (Figuras 20 e 25)demonstrou um melhor equilíbrio entre a distância e a carga dos servidores. Embora ocasionalmente direcionasse o cliente para o servidor mais congestionado, esse comportamento está alinhado com sua natureza de explorar novas opções em parte do tempo. Ainda assim, ele conseguiu reduzir significativamente as latências médias em comparação ao algoritmo *Random*.

Por outro lado, o algoritmo *UCB1* destacou-se pela capacidade de tomar decisões mais informadas. Quando os servidores apresentavam cargas semelhantes (Figuras 21 e 26), ele priorizou a proximidade geográfica, otimizando o componente de latência relacionado à distância. No entanto, ao detectar diferenças significativas de carga, o algoritmo foi capaz de redirecionar as requisições para servidores mais distantes, mas que ofereciam tempos de resposta melhores devido à menor sobrecarga, como pode ser observado nas Figuras 22 e 27. Quando a entrada do algoritmo considerava exclusivamente a distância, o UCB1 priorizou consistentemente o servidor mais próximo, o que, em casos de sobrecarga, aumentou a latência experimentada pelo cliente, como mostrado nas Figuras 23 e 28. Esses resultados destacam a eficácia do UCB1 em adaptar-se dinamicamente às condições de carga e distância, maximizando o desempenho em cenários complexos.

7 Trabalhos Futuros

O projeto desenvolvido possui diversas possibilidades de aprimoramento e exploração de novos cenários:

1. **Exploração de Cenários Diferentes de Movimentação e Carga nos Servidores:**

    - Investigar como o sistema se comporta em cenários variados, onde o cliente apresenta padrões de movimentação mais complexos, como trajetórias não lineares ou mudanças bruscas de direção. Além disso, simular cargas dinâmicas e heterogêneas nos servidores de cache pode ajudar a entender melhor como os algoritmos de escolha se adaptam a situações mais próximas de ambientes reais. Esses estudos podem revelar limitações ou destacar oportunidades para otimização adicional.

2. **Adição de Novos Algoritmos de Escolha:**

    - Integrar novos algoritmos para ordenar os servidores de cache, como abordagens baseadas em aprendizado por reforço ou inteligência artificial. Explorar algoritmos híbridos, que combinem heurísticas existentes com técnicas avançadas, pode contribuir para melhorar a eficiência e a robustez do sistema em cenários dinâmicos e com múltiplos fatores influenciando a latência.

3. **Aumento do Número de Servidores de Cache:**

    - Ampliar a quantidade de servidores de cache utilizados nas simulações para avaliar a escalabilidade do sistema. Essa expansão permitirá investigar se o desempenho dos algoritmos de escolha se mantém consistente ou se surgem novos desafios, como maior tempo de convergência ou dificuldade em balancear a carga entre os servidores.

4. **Implementação de Novas Metodologias para Cálculo de Latência e Qualidade de Experiência (QoE):**

    - Desenvolver e incorporar novas metodologias para mensurar a latência e avaliar a qualidade da experiência do usuário (QoE). Isso pode incluir métricas mais sofisticadas, como análise de rebuffering, estabilidade de *bitrate* e percepção de qualidade. Além disso, modelos mais precisos para estimar a latência com base em características da rede e condições de tráfego podem enriquecer os algoritmos de escolha, tornando-os mais alinhados com situações do mundo real.

Esses caminhos permitirão aprofundar a análise do sistema, aprimorar sua eficiência e explorar sua aplicação em cenários mais complexos e realistas. A evolução do projeto nessas direções contribuirá para avanços na área de *content steering* e otimização de sistemas distribuídos.

8 Conclusão

O projeto explorou a implementação de uma arquitetura de *content steering* com foco na entrega otimizada de vídeos em sistemas distribuídos. Ao longo do desenvolvimento, foram integrados um orquestrador central, servidores de cache e um cliente móvel simulado, criando uma infraestrutura dinâmica capaz de adaptar-se às condições variáveis de latência e carga.

O orquestrador, como peça central do sistema, demonstrou ser satisfatório em gerenciar as comunicações entre cliente e servidores, utilizando algoritmos inteligentes para ordenar os servidores de cache disponíveis com base nas latências experimentadas pelo cliente. A funcionalidade de estimativa de latência mínima, baseada na distância geográfica, e o uso de latência total como métrica para os algoritmos de escolha, destacaram a capacidade do sistema de equilibrar fatores como proximidade, carga e experiência do usuário.

Foram implementados e avaliados três algoritmos de escolha: *Random*, *Epsilon-Greedy* e *UCB1*. O algoritmo *Random*, configurado como um caso especial do *Epsilon-Greedy* ($\epsilon = 1.0$), realizou escolhas aleatórias, servindo como baseline para comparações. O *Epsilon-Greedy* apresentou um balanceamento interessante entre exploração e exploração, enquanto o *UCB1* destacou-se pela sua capacidade de convergir rapidamente para servidores de cache que proporcionavam a melhor experiência em termos de latência e carga. Os experimentos demonstraram que a seleção inteligente de servidores pode reduzir significativamente a latência ao longo do tempo, otimizando a entrega de conteúdo.

Os experimentos realizados evidenciaram o comportamento do sistema em condições variadas, incluindo movimentação do cliente e diferentes níveis de carga nos servidores.

Como próximos passos, o projeto propõe a exploração de novos cenários e algoritmos, a inclusão de mais servidores de cache para testar a escalabilidade do sistema e a implementação de metodologias adicionais para avaliar a latência e a qualidade da experiência (QoE). A adição de algoritmos baseados em aprendizado por reforço e inteligência artificial também é uma direção promissora para tornar o sistema ainda mais adaptativo e eficiente.

Referências

[1] L. Bittencourt, R. Immich, R. Sakellariou, N. Fonseca, E. Madeira, M. Curado, L. Villas, L. DaSilva, C. Lee, and O. Rana. *The internet of things, fog and cloud continuum: Integration and challenges. Internet of Things*, 3-4:134–155, 2018, https://doi.org/10.48550/arXiv.1809.09972.

[2] A. Bentaleb, B. Taani, A. C. Begen, C. Timmerer, and R. Zimmermann. *A survey on bitrate adaptation schemes for streaming media over http.* IEEE Communications Surveys & Tutorials, 21(1):562–585, 2019. https://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=8424813

[3] F. Pisani, F. de Oliveira, E. S. Gama, R. Immich, L. F. Bittencourt, and E. Borin. *Fog computing on constrained devices: Paving the way for the future iot.* Advances in Edge Computing: Massive Parallel Processing and Applications, 35:22, 2020, https://arxiv.org/pdf/2002.05300.

[4] E. S. Gama, R. Immich, and L. F. Bittencourt. *Towards a multi-tier fog/cloud architecture for video streaming.* In 2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion), pages 13–14, 2018, https://www.ic.unicamp.br/~roger/pdf/2018UCCposter.pdf.

[5] J. Jiang, V. Sekar, and H. Zhang, "*Improving fairness, efficiency, and stability in http-based adaptive video streaming with festive*", TON, vol. 22, no. 1, pp. 326–340, 2014. https://conferences.sigcomm.org/co-next/2012/eproceedings/conext/p97.pdf

[6] "Http live streaming," https://developer.apple.com/streaming/, 2019.

[7] DASH, "Dash," 2019. [Online]. Available: https://dashif.org/

[8] Content Steering for Dash, 2022. Acessado em 05 de dezembro de 2024.

[9] D. Silhavy, W. Law, S. Pham, A. C. Begen, A. Giladi, and A. Balk. *Dynamic cdn switching-dash-if content steering in dash.js.* In Proceedings of the 2nd Mile-High Video Conference, pages 130–131, 2023, https://dl.acm.org/doi/pdf/10.1145/3588444.3591027.

[10] R. Rodrigues Filho, E. S. Gama, M. Assis, E. R. M. Madeira, R. Immich and L. F. Bittencourt, *"Content Steering: Leveraging the Computing Continuum to Support Adaptive Video Streaming"*, Minicursos do XLII Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos, SBRC, 2024, https://github.com/robertovrf/content-steering-tutorial.

[11] E. S. Gama, R. Rodrigues-Filho, E. R. M. Madeira, R. Immich and L. F. Bittencourt, *"Enabling Adaptive Video Streaming via Content Steering on the Edge-Cloud Continuum"*, 2024 IEEE 8th International Conference on Fog and Edge Computing (ICFEC),

Philadelphia, PA, USA, 2024, pp. 35-42, https://doi.org/10.1109/ICFEC61590.2024.00018.

[12] *O que é conteinerização?*, AWS, 2024, https://aws.amazon.com/what-is/containerization/

[13] S. Susnjara e I. Smalley, *O que é conteinerização?*, IBM, 2024, https://www.ibm.com/br-pt/topics/containerization.

[14] Docker Compose, 2024, https://docs.docker.com/compose/.

[15] Gittins, John, Kevin Glazebrook, and Richard Weber. Multi-armed bandit allocation indices. John Wiley And Sons, 2011.

[16] White, J. M. "Bandit Algorithms for Website Optimization."O'Relly (2013).

[17] Foo, K. "Bandit Simulations", GitHub. Disponível em: https://github.com/kfoofw/bandit_simulations

[18] Godinho, A., Rosado, J., Sá, F., And Cardoso, F. (2023, October). Method for Evaluating the Performance of Web-Based APIs. In International Conference on Smart Objects and Technologies for Social Good (pp. 30-48). Cham: Springer Nature Switzerland.

[19] Fórmula de Haversine, 2024, https://en.wikipedia.org/wiki/Haversine_formula.