Pontifícia Universidade Católica
do Rio de Janeiro

Bernardo de Campos Vidal Camilo

Uma Avaliação Experimental de *Hashing* Consistente com Cargas Limitadas na Distribuição de Vídeos *Online*

Dissertação de Mestrado

Dissertação apresentada como requisito parcial para obtenção do grau de Mestre pelo Programa de Pós-graduação em Informática da PUC-Rio.

Orientador: Prof.ª Noemi de La Rocque Rodriguez

Rio de Janeiro
Setembro de 2018

Pontifícia Universidade Católica
do Rio de Janeiro

Bernardo de Campos Vidal Camilo

Uma Avaliação Experimental de *Hashing* Consistente com Cargas Limitadas na Distribuição de Vídeos *Online*

Dissertação apresentada como requisito parcial para obtenção do grau de Mestre pelo Programa de Pós-graduação em Informática da PUC-Rio. Aprovada pela Comissão Examinadora abaixo assinada.

Prof.ª Noemi de La Rocque Rodriguez
Orientador
Departamento de Informática – PUC-Rio

Prof. Markus Endler
Departamento de Informática – PUC-Rio

Prof. Sérgio Colcher
Departamento de Informática – PUC-Rio

Prof. Márcio da Silveira Carvalho
Coordenador Setorial do Centro Técnico Científico – PUC-Rio

Rio de Janeiro, 06 de Setembro de 2018

Bernardo de Campos Vidal Camilo

Graduou-se em Engenharia de Computação e Informação pela Universidade Federal do Rio de Janeiro (UFRJ). Atualmente trabalha com desenvolvimento de *software* na Globo.com.

Ficha Catalográfica

Camilo, Bernardo de Campos Vidal

Uma Avaliação Experimental de *Hashing* Consistente com Cargas Limitadas na Distribuição de Vídeos *Online* / Bernardo de Campos Vidal Camilo; orientador: Noemi de La Rocque Rodriguez. – 2018.

54 f. : il. ; 30 cm

Dissertação (mestrado) – Pontifícia Universidade Católica do Rio de Janeiro, Departamento de Informática, 2018.

Inclui bibliografia

1. Informática – Teses. 2. Caching. 3. Hashing. 4. Balanceamento de carga. 5. Distribuição de vídeos online. I. Rodriguez, Noemi de La Rocque. II. Pontifícia Universidade Católica do Rio de Janeiro. Departamento de Informática. III. Título.

CDD: 004

À vida.

Agradecimentos

À Globo.com, pelo suporte financeiro.
Aos meus amigos, pelo companheirismo e amizade.
À minha namorada Rachel, pelo carinho e paciência.
À minha orientadora Noemi, pelos conselhos e ensinamentos.
Aos professores da PUC-Rio, pelo empenho em transmitir seus conhecimentos.
À minha família, em especial aos meus pais e irmãos, pelo apoio incondicional.

Resumo

Camilo, Bernardo de Campos Vidal; Rodriguez, Noemi de La Rocque. **Uma Avaliação Experimental de *Hashing* Consistente com Cargas Limitadas na Distribuição de Vídeos *Online***. Rio de Janeiro, 2018. 54p. Dissertação de Mestrado – Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

O consumo de vídeos representa grande parte do tráfego na Internet hoje e tende a aumentar ainda mais nos próximos anos. Neste trabalho, investigamos formas de aprimorar o *caching* em redes de distribuição de conteúdo (*Content Delivery Networks* – CDNs) de vídeo para reduzir o tempo de resposta das mesmas e aumentar a qualidade de experiência dos usuários. A partir da análise de diferentes técnicas, concluímos que o *hashing* consistente com cargas limitadas possui características interessantes para esse fim e se encaixa adequadamente ao cenário de distribuição de vídeos. Para verificar o seu desempenho, criamos uma plataforma de experimentação e, usando dados de uma CDN de vídeos real, o confrontamos com o *hashing* consistente e com o método de balanceamento *least connections*, todos implementados de maneira equivalente para permitir uma comparação justa. Por fim, discutimos os resultados dessa avaliação, destacando os benefícios e limitações dessa técnica no contexto considerado.

Palavras-chave

Caching; hashing; balanceamento de carga; distribuição de vídeos online.

Abstract

Camilo, Bernardo de Campos Vidal; Rodriguez, Noemi de La Rocque (Advisor). **An Experimental Evaluation of Consistent Hashing with Bounded Loads in Online Video Distribution**. Rio de Janeiro, 2018. 54p. Dissertação de Mestrado – Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Video consumption accounts for a large part of Internet traffic today and tends to increase further in the next years. In this work, we investigate ways to improve caching in video content delivery networks (CDNs) to reduce their response time and increase the users' quality of experience. From the analysis of different techniques, we concluded that consistent hashing with bounded loads has interesting characteristics for this purpose and fits adequately to the video delivery scenario. In order to verify its performance, we created an experimentation platform and, using data from a real video CDN, confronted it with the consistent hashing and the least connections balancing method, all implemented in an equivalent manner to permit a fair comparison. Lastly, we discussed the results of this evaluation, highlighting the benefits and limitations of this technique in the considered context.

Keywords

Caching; hashing; load balancing; online video distribution.

Sumário

Lista de figuras

Lista de tabelas

Lista de siglas

API – *Application Programming Interface*
CARP – *Cache Array Routing Protocol*
CDN – *Content Delivery Network*
DNS – *Domain Name Server*
EC2 – *Elastic Compute Cloud*
HAS – *HTTP Adaptive Streaming*
HLS – *HTTP Live Streaming*
HTTP – *Hypertext Transfer Protocol*
ICP – *Internet Cache Protocol*
IP – *Internet Protocol*
LRU – *Least Recently Used*
RPS – Requisições por segundo
SSL – *Secure Sockets Layer*
URI – *Uniform Resource Identifier*
URL – *Uniform Resource Locator*

1
Introdução

Vídeos representam uma grande parcela do tráfego na Internet hoje e essa parcela tende a aumentar nos próximos anos. Desconsiderando empresas e governos, é estimado que o tráfego de vídeos na Internet represente cerca de 82% do total em 2021 (CISCO, 2017). A Figura 1.1 mostra esse crescimento ao longo dos anos. Redes de distribuição de conteúdo (*Content Delivery Networks* – CDNs) desempenham um papel fundamental na entrega desse tipo de conteúdo. Em linhas gerais, essas redes mantêm múltiplos pontos de presença geograficamente espalhados com *clusters* de servidores de cache que armazenam cópias de um mesmo objeto. Dessa forma, a distribuição pode ser feita a partir da localidade mais adequada para cada cliente, trazendo, dentre outras vantagens, uma melhor qualidade de experiência (PALLIS; VAKALI, 2006). Esse panorama nos impulsiona a buscar melhorias na distribuição de vídeos através dessas redes.

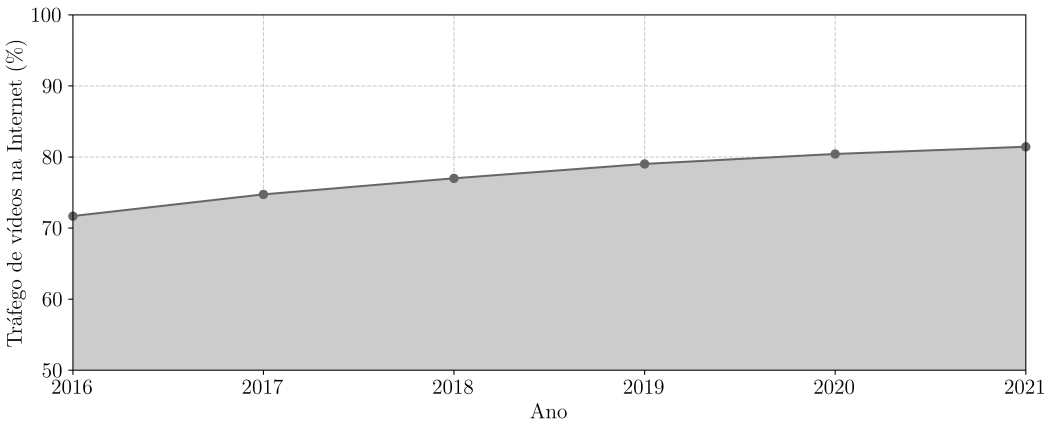


Figura 1.1: Tráfego de vídeos na Internet no período de 2016 a 2021 desconsiderando a parcela de empresas e governos. Os números foram obtidos em um relatório publicado pela Cisco (2017).

1.1
Objetivo e Proposta

O objetivo deste trabalho é avaliar uma forma de aprimorar o *caching* de redes de distribuição de conteúdo dedicadas a vídeos. Mais especificamente, focamos na coordenação dos servidores de cache que compõem os *clusters*

das CDNs. Fazendo com que esses servidores atendam as requisições de maneira coordenada, esperamos observar um aumento na eficiência de cache e, consequentemente, uma redução no tempo de resposta da CDN, melhorando a qualidade de experiência dos usuários.

A partir do estudo de diferentes esquemas de coordenação, concluímos que as abordagens que utilizam *hashing* para mapear requisições a servidores têm qualidades que se encaixam muito bem ao cenário em questão. No entanto, dependendo do algoritmo adotado, esse esquema pode apresentar sérias limitações. A principal delas está relacionada ao balanceamento de carga em situações nas quais o acesso é concentrado em poucos objetos. Usando um algoritmo de *hashing* convencional, podemos provocar um grande desbalanceamento no sistema, causando sobrecarga em servidores específicos e, consequentemente, degradando o desempenho do conjunto. Com indícios de que o acesso a vídeos na Internet possui essa característica (CHA et al., 2009; GILL et al., 2007; BRESLAU et al., 1999), tivemos que buscar um algoritmo capaz de lidar com essa situação. Assim, chegamos à nossa proposta, que se baseia na utilização do *hashing* consistente com cargas limitadas (MIRROKNI; THORUP; ZADIMOGHADDAM, 2018). Combinando o funcionamento do *hashing* consistente convencional (KARGER et al., 1997) com um parâmetro que nos permite um controle mais refinado sobre o balanceamento de carga do sistema, enxergamos nesse algoritmo um grande potencial para realizar a coordenação de servidores de cache no cenário apresentado.

1.2 Metodologia

Para verificar a efetividade da nossa proposta, adotamos uma abordagem experimental. Assim, projetamos uma plataforma de testes e escolhemos dois outros algoritmos para servir como bases de comparação para o *hashing* consistente com cargas limitadas, o *hashing* consistente convencional e o *least connections*. Todos esses algoritmos foram implementados de forma equivalente e são executados sobre a mesma estrutura para permitir uma comparação justa. Em seguida, trabalhamos na preparação do arquivo de entrada que é usado no processo de geração de carga da plataforma. A fim de reproduzir um cenário mais próximo da realidade, usamos *logs* de uma CDN de vídeos real como base para esse arquivo. Com isso, iniciamos uma série de experimentos considerando diferentes níveis de tráfego e coletamos os resultados, tomando como métrica central a latência[1] observada em cada caso. Os resultados indicam que o uso desse algoritmo é vantajoso, mas também revelam uma limitação.

[1]Tempo entre o início da requisição e a leitura da resposta medido no lado do cliente.

1.3 Organização do Trabalho

O restante deste texto é dividido da seguinte maneira. Na Seção 2, apresentamos os conceitos preliminares para o trabalho, incluindo o modelo de CDN assumido, as técnicas para coordenação de caches distribuídos, o funcionamento do *hashing* consistente e da sua variação com cargas limitadas, e os detalhes da entrega de vídeos na Internet. Na Seção 3, detalhamos a montagem da nossa plataforma de experimentação, passando por cada um dos seus componentes. A Seção 4 apresenta a preparação para os experimentos e discute os resultados obtidos. Por fim, concluímos o trabalho e apontamos direções futuras na Seção 5.

2
Conceitos Preliminares

Este capítulo apresenta os conceitos preliminares que sustentam e contextualizam o nosso estudo. Inicialmente, ilustramos o modelo de CDN assumido, destacando em que parte o nosso trabalho se encaixa. Em seguida, apresentamos as diferentes técnicas de coordenação encontradas na literatura, justificando por que escolhemos focar em estratégias baseadas em *hashing*. A partir de então, nos concentramos em problemas específicos da coordenação com *hashing* e apresentamos algumas abordagens para contorná-los, chegando até a explicação do funcionamento do *hashing* consistente com cargas limitadas. Por fim, fornecemos uma breve descrição da técnica de *streaming* adaptativo, que é comumente utilizada para entrega de vídeos na *web*.

2.1
Modelo de Rede de Distribuição de Conteúdo

Para suportar as discussões realizadas ao longo deste trabalho, definimos um modelo genérico de rede de distribuição de conteúdo. Apesar de simples, acreditamos que esse modelo é capaz de representar os principais componentes das CDNs. Com base nele, evidenciamos a parte da distribuição na qual estamos focados e delimitamos o escopo da nossa pesquisa.

Como ilustrado na Figura 2.1, nosso modelo é composto por diversos *clusters* de servidores de cache geograficamente distribuídos, que podem estar em *datacenters* privados, em pontos de troca de tráfego ou até na infraestrutura de provedores de serviço de Internet (MOKHTARIAN; JACOBSEN, 2017). Esses servidores armazenam cópias dos objetos servidos pelo provedor de conteúdo e evitam que as requisições tenham que percorrer longas distâncias até a origem. Assim, a latência percebida pelos clientes é reduzida e, consequentemente, há uma melhora na qualidade de experiência. Além disso, ao replicar o conteúdo em diversos servidores, as CDNs oferecem outros benefícios como o aumento da capacidade de entrega do provedor de conteúdo e a redução da carga nos servidores de origem.

O foco do nosso trabalho está no funcionamento interno dos *clusters*. Mais especificamente, buscamos uma forma efetiva de coordenar os diversos servidores de cache para operar em conjunto, pensando sempre no cenário de

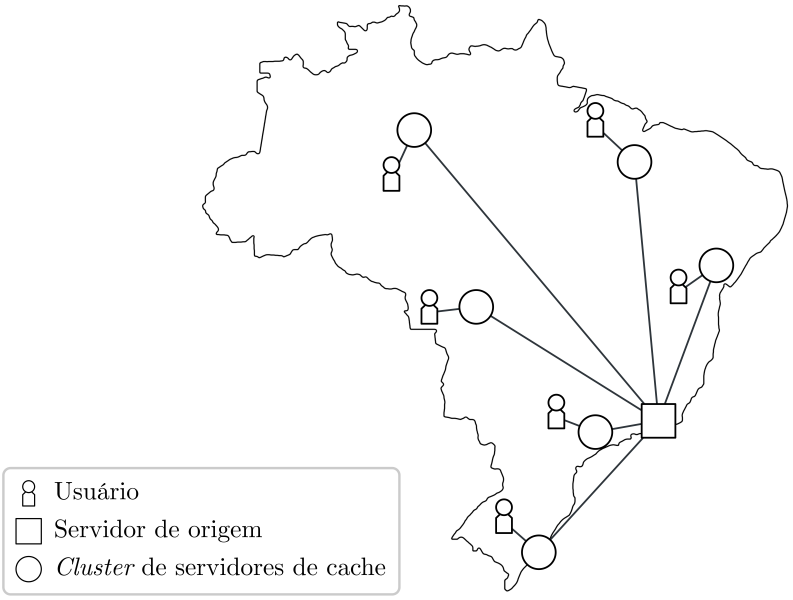


Figura 2.1: Arquitetura genérica de uma CDN com pontos de presença geograficamente espalhados.

distribuição de vídeos. Por isso, não nos preocupamos tanto com a estrutura da rede, nem com o direcionamento das requisições fora do *cluster*. Assumimos simplesmente que a nossa CDN tem uma estrutura plana e opera de forma não cooperativa baseada em *pull* (PALLIS; VAKALI, 2006). Portanto, as requisições são direcionadas ao *cluster* mais próximo e, caso o conteúdo requisitado não seja encontrado, os servidores buscam diretamente na origem.

Na literatura, encontramos várias técnicas que visam coordenar um conjunto de caches distribuídos para fazê-los operar como uma unidade coerente. No contexto deste trabalho, essas técnicas são usadas para aumentar a eficiência de cache dos *clusters* e reduzir o número de vezes que um servidor pede conteúdo à origem. Isso afeta diretamente o tempo de resposta da CDN e a utilização dos recursos físicos presentes em cada *cluster* e, por isso, consideramos importante conhecer essas técnicas.

2.2 Técnicas para Coordenação de Caches

Mesmo antes do termo CDN se tornar popular, *web caching* já era um assunto bastante estudado e diversas propostas para a criação de um sistema de cache distribuído para a *web* já haviam sido publicadas. Uma questão recorrente nessas propostas era como promover a cooperação entre diferentes servidores para aumentar a eficiência do conjunto. Nesse contexto, surgiram diversas abordagens, e o termo *caching* cooperativo acabou se tornando comum para descrevê-las. Muitos dos conceitos introduzidos por esses sistemas foram

importados para o mundo das CDNs e, neste trabalho, as técnicas para a coordenação dos caches de um *cluster* da CDN e os esquemas de *caching* cooperativo estão intimamente relacionados. Para apresentá-las, seguimos a mesma classificação utilizada no trabalho de Ni et al. (2003) e as separamos em quatro grupos: baseados em consulta, baseados em resumos, baseados em diretório e, finalmente, baseados em *hashing*.

No esquema baseado em consulta, quando o servidor que recebeu a requisição não encontra o conteúdo requisitado em seu cache (cache *miss*), ele dispara uma mensagem de consulta para todos os outros que compõem o conjunto a fim de tentar obtê-lo. Chankhunthod et al. (1996) apresentam um exemplo de sistema de cache baseado em consulta que foi desenvolvido no contexto do projeto Harvest[1]. Dentre outras contribuições, esse projeto serviu de berço para o protocolo ICP (*Internet Cache Protocol*) (WESSELS; CLAFFY, 1997) que, até hoje, é utilizado por diversos sistemas para realizar essa comunicação entre os caches. Os principais problemas desse esquema estão relacionados à inundação de mensagens de consulta e ao potencial atraso na resposta, pois um servidor tem que aguardar a resposta dos demais antes de concluir que o conjunto não tem o conteúdo requisitado (NI et al., 2003).

O esquema baseado em resumo mitiga os problemas da abordagem anterior. Como o próprio nome sugere, cada servidor mantém um resumo do conteúdo de todos os outros, que é consultado quando ocorre um cache *miss*. Apesar de evitar a inundação de mensagens de consulta e, potencialmente, reduzir o tempo de resposta, esse esquema depende de uma comunicação frequente entre os caches para manter os resumos atualizados. Fan et al. (2000) publicaram uma implementação desse esquema na qual utilizam *Bloom filters* para armazenar os resumos de forma eficiente. Além disso, argumentam que é possível atrasar a sincronização dos resumos para reduzir o tráfego entre caches sem trazer grandes impactos na eficiência do conjunto, mas isso impõe uma complexidade adicional no sistema.

No esquema baseado em diretório, mantém-se um mapa central que é consultado e atualizado por todos os caches do conjunto. Podemos considerá-lo uma versão centralizada do esquema anterior e, apesar de simplificar a sincronização dos resumos, introduz um potencial ponto único de falha (NI et al., 2003) e um gargalo no sistema. Uma implementação desse esquema foi proposta por Gadde, Rabinovich e Chase (1997), que argumentam que os problemas citados anteriormente são contornáveis via configuração. No entanto, a eficiência pode ser muito prejudicada e a complexidade de implementação do sistema tende a aumentar para tornar o esquema tolerante a falhas.

[1]<https://web.archive.org/web/19971221220012/http://harvest.transarc.com/>

Por fim, temos o esquema baseado em *hashing* no qual a escolha do servidor designado para atender a uma requisição é definida por uma função *hash*. No trabalho de Ni et al. (2003), esse esquema é descrito da seguinte forma: cada servidor do conjunto mantém a mesma função *hash* e, baseado na URL do conteúdo e no endereço dos servidores, a requisição é redirecionada para o servidor designado. O protocolo CARP (*Cache Array Routing Protocol*) (VALLOPPILLIL; ROSS, 1998) pode ser considerado um exemplo desse esquema. Além do modo de operação no qual o *hashing* é executado no lado do servidor, conforme descrito por Ni et al. (2003), o protocolo especifica uma implementação na qual o *hashing* é executado em navegadores *web*, no lado do cliente. Um outro exemplo é o sistema implementado por Karger et al. (1999). Eles adotam uma abordagem similar ao CARP no lado do cliente, porém o processo de *hashing* é essencialmente diferente, e parte dele é executada no servidor de DNS (*Domain Name Server*).

Considerando o cenário de distribuição de vídeos, o esquema baseado em *hashing* parece ser bastante promissor. Como o resultado de uma função *hash* é sempre igual para a mesma entrada, conseguimos evitar duplicações dentro do conjunto de caches com facilidade. Isso é útil para redução dos custos de armazenamento, principalmente quando tratamos de arquivos de vídeo que tendem a ser ordens de grandeza maiores que outros tipos de conteúdo, como imagens e texto. Além disso, a decisão de onde buscar o conteúdo é muito eficiente, causando pouco impacto no tempo de entrega. Essa característica é muito importante, visto que, de acordo com Krishnan e Sitaraman (2013), a taxa de abandono de um vídeo cresce significativamente quando o tempo para iniciar a sua reprodução é superior a dois segundos. Por esses motivos, concentramos nossos esforços na operação desse esquema, buscando melhorias para os seus problemas específicos.

2.3 Coordenação Baseada em *Hashing*

Apesar das vantagens mencionadas, a coordenação baseada em *hashing* tem questões que, se ignoradas, podem comprometer o seu desempenho. A primeira delas está relacionada à forma com que o processo de *hashing* é executado. O que aconteceria se utilizássemos uma função *hash* modular clássica, baseada no número de servidores do *cluster*, e um servidor ficasse indisponível? Com um número menor de servidores, muitos objetos já no cache de alguma máquina seriam remapeados para máquinas diferentes, inutilizando grande parte do que havia sido armazenado até então. O *hashing* consistente (KARGER et al., 1997) e o *rendezvous hashing* (THALER; RAVISHANKAR, 1998)

são dois algoritmos populares para lidar com esse ambiente dinâmico, no qual servidores podem ser removidos ou adicionados a qualquer momento. Considerando os exemplos da seção anterior, temos que o sistema implementado por Karger et al. (1999) faz uso do *hashing* consistente, enquanto o CARP faz uso do *rendezvous hashing*.

Uma outra questão fundamental é o balanceamento de carga desses esquemas. É razoável assumir que funções *hash* tendem a distribuir suas entradas de forma uniforme sobre as possíveis saídas (KARGER et al., 1999). Essa característica é muito importante, mas pode não ser conveniente quando o acesso é concentrado em poucos objetos. Em uma situação como essa, pode haver um grande desbalanceamento de carga e sobrecarregar servidores específicos, degradando o desempenho do sistema como um todo. No caso deste trabalho, o padrão de acesso é ditado pela popularidade dos vídeos presentes nos serviços de vídeo *online*.

2.3.1 Distribuição de Popularidade de Vídeos *Online*

Intuitivamente, é de se esperar que uma pequena minoria dos vídeos represente a grande maioria do consumo. Considerando vídeos de jornalismo, por exemplo, espera-se que as notícias mais recentes sejam acessadas em uma frequência muito superior às da última semana. Analogamente, o acesso ao último episódio de uma novela ou série tende a ser muito maior que o acesso a episódios de temporadas passadas. Por fim, embora de forma menos marcante que nos casos anteriores, filmes que são lançamentos também se tornam mais populares.

Encontramos na literatura artigos que buscam fazer uma caracterização do tráfego de vídeos de alguns serviços de vídeo *online* específicos. Gill et al. (2007) analisaram o consumo de vídeos do YouTube[2] dentro do campus de Universidade de Calgary, no Canadá, e concluíram que o acesso seguia uma distribuição similar à Zipf. Cha et al. (2009) fazem uma análise semelhante, coletando informações do YouTube e do Daum Videos[3]. Diferentemente de Gill et al. (2007), Cha et al. (2009) analisam separadamente o acesso de conteúdo popular e não popular, concluindo que a popularidade dos vídeos segue a distribuição de uma lei de potência com um *cutoff* exponencial. De qualquer maneira, os dois estudos indicam a presença de uma cauda longa na distribuição de popularidade dos vídeos, revelando que poucos objetos

[2] <https://www.youtube.com/>

[3] A referência encontrada no artigo não é mais válida. Cha et al. (2009) o descrevem como um serviço de vídeos gerados por usuários popular na Coreia.

representam grande parte do acesso. Diante disso, buscamos abordagens para evitar o desbalanceamento causado por esse padrão de acesso.

2.3.2 Técnicas para Lidar com Diferenças Acentuadas de Popularidade

Karger et al. (1999) dedicam uma breve seção em seu trabalho para discutir como lidar com a situação na qual há uma diferença acentuada na popularidade dos objetos acessados. Antes de apresentar a abordagem proposta, precisamos entender um pouco melhor o funcionamento do sistema proposto e implementado pelos autores. A ideia inicial era realizar o processo de *hashing* (consistente) nos navegadores dos clientes. No entanto, acabaram esbarrando em limitações e tiveram que usar o DNS para auxiliar nessa tarefa. Assim, a solução final acabou sendo dividida em duas partes. Primeiro, utilizam um *script* de autoconfiguração para mapear a URL de entrada para um intervalo de mil nomes virtuais, usando uma função de *hash* padrão. Em seguida, mapeiam os nomes virtuais nos endereços de IP dos caches usando um servidor de DNS, que é atualizado dinamicamente por um outro programa que executa o *hashing* consistente. Para lidar com a diferença de popularidade, os autores sugerem atribuir uma lista servidores de cache para os nomes virtuais mais "quentes" em vez de apenas um. Para isso, idealmente, o DNS teria que saber quais nomes virtuais são populares, mas obter essa informação não é uma tarefa trivial. O que o DNS possui é o mapeamento de nomes virtuais em endereços de IP e, possivelmente, um indicativo da carga em cada cache. Com isso, os autores propõem espalhar todos os nomes virtuais mapeados para um servidor com carga elevada para todos os caches do sistema e, em seguida, ir reduzindo o espalhamento removendo um endereço de IP por vez. Dessa forma, espera-se alcançar um equilíbrio com alguns nomes virtuais mapeados para listas de tamanho variável de caches.

Wu e Yu (2003) adotam uma outra abordagem, baseada em uma replicação controlada adaptável do conteúdo. O cenário adotado no trabalho desses autores é diferente do anterior, pois o processo de *hashing* é feito no lado do servidor. Cada cliente é configurado para se conectar em um cache que, quando necessário, computa uma função *hash* baseada na URL do objeto requisitado e dispara uma requisição HTTP para o servidor designado. Para lidar com acessos muito enviesados, os autores propõem que o espaço de cache de cada servidor seja dividido em duas pilhas LRU (*Least Recently Used*), denominadas local e regular, que operam conforme ilustrado na Figura 2.2. Quando uma requisição chega a um determinado servidor, ele verifica se possui o objeto em cache olhando a sua pilha local. Caso não tenha, computa a função *hash* base-

ada na URL e, dependendo do resultado, verifica a sua própria pilha regular ou requisita para o servidor designado. Com isso, espera-se que réplicas de conteúdo muito populares permaneçam no cache de outros servidores, evitando a sobrecarga nos servidores designados.

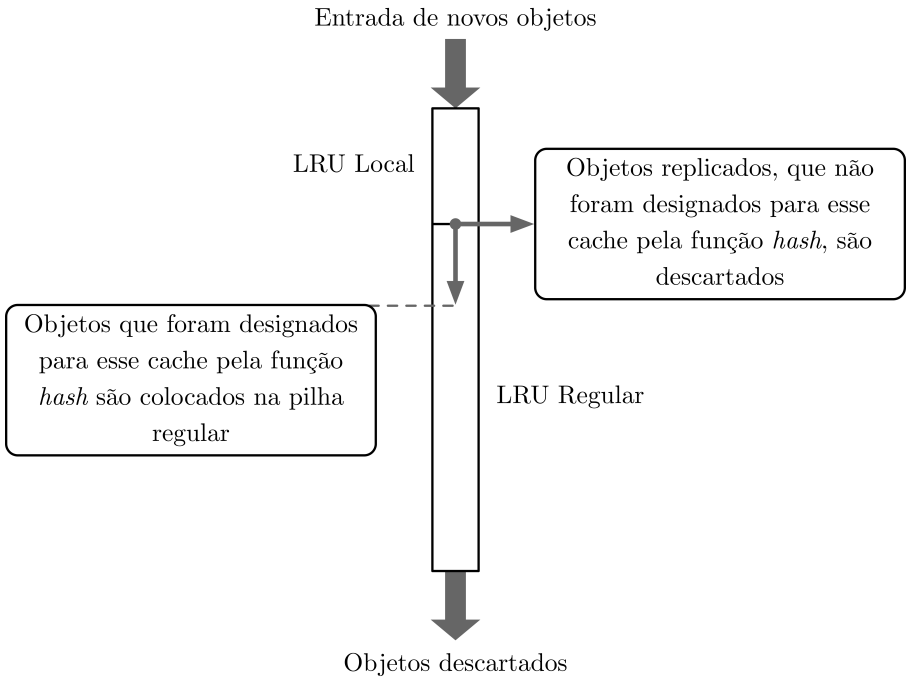


Figura 2.2: Pilhas LRU para replicação controlada adaptável. Adaptada do trabalho de Wu e Yu (2003).

A abordagem de Karger et al. (1999) é, de certa forma, acoplada ao funcionamento do sistema proposto pelos autores, que conta com um conjunto fixo de nomes virtuais. Além disso, a decisão de espalhar os nomes virtuais que são mapeados para o servidor sobrecarregado para todos os caches pode ser considerada agressiva demais e, considerando cenário muito dinâmico, pode ser difícil alcançar o equilíbrio esperado. A proposta de Wu e Yu (2003), por outro lado, faz uma replicação mais controlada. No entanto, foi pensada para o cenário em que o *hashing* é executado no lado do servidor, o que pode introduzir atrasos adicionais por conta da comunicação entre os caches. Por isso, exploramos outra forma de contornar esse problema, baseada no algoritmo *hashing* consistente com cargas limitadas (MIRROKNI; THORUP; ZADIMOGHADDAM, 2018). A seguir, apresentamos o funcionamento do *hashing* consistente convencional e da variação com cargas limitadas.

2.3.3 *Hashing* Consistente

O *hashing* consistente foi publicado em 1997 por Karger et al. (1997). Como mencionado anteriormente, esse algoritmo foi proposto como uma solução para evitar o problema de remapeamento causados por mudanças no intervalo de saída de uma função *hash*.

De forma sucinta, a ideia do *hashing* consistente é utilizar funções de *hash* comuns para mapear as entradas e saídas em um círculo com circunferência igual a um e, então, atribuir cada entrada ao primeiro ponto que corresponde a uma saída no sentido horário. A Figura 2.3 exemplifica o seu funcionamento. Nela, as letras representam servidores e os números representam os objetos. Se o servidor C se tornar indisponível, apenas os objetos um e dois são remapeados.

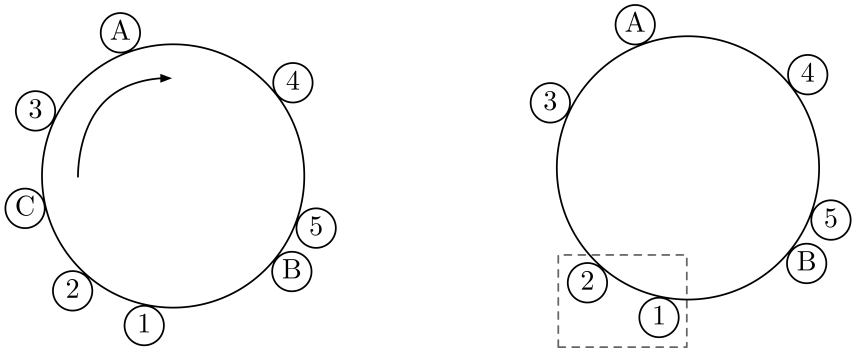


Figura 2.3: Exemplo de funcionamento do *hashing* consistente. Adaptada do trabalho de Karger et al. (1999).

Em 1999, Karger et al. (1999) implementaram um sistema de *web caching* usando esse algoritmo. De forma semelhante a esse sistema, o nosso trabalho assume que a entrada da função *hash* é baseada na URL do objeto requisitado e a saída é o endereço do servidor de cache designado.

2.3.4 *Hashing* Consistente com Cargas Limitadas

O *hashing* consistente com cargas limitadas teve a sua primeira publicação em 2016 (MIRROKNI; THORUP; ZADIMOGHADDAM, 2016) e permite um controle mais refinado sobre o balanceamento de carga do sistema. Em sua publicação mais recente, Mirrokni, Thorup e Zadimoghaddam (2018) descrevem o algoritmo com enfoque na análise e no embasamento teórico. De forma sucinta, o algoritmo combina o funcionamento do *hashing* consistente com um limite superior para a carga de cada servidor. Para definir esse limite, os autores usam um parâmetro de balanceamento $c = 1 + \varepsilon$, que garante

que a carga máxima em cada servidor é até $\lceil cm/n \rceil$, onde m é o número de objetos sendo servidos e n é o número de servidores. Em termos práticos, a escolha do valor desse parâmetro captura o *tradeoff* entre o balanceamento de carga e a estabilidade em resposta à mudanças no sistema (MIRROKNI; THORUP; ZADIMOGHADDAM, 2018). Quanto menor o valor do parâmetro, mais balanceado o sistema fica, porém menos estável em relação a mudanças.

Sua operação é bem similar à do *hashing* consistente, mas são necessários alguns passos a mais para possibilitar o balanceamento de carga mais controlado. Inicialmente, atribuímos capacidades aos servidores. Os $\lceil cm \rceil - n\lfloor cm/n \rfloor$ primeiros servidores recebem a capacidade $\lceil cm/n \rceil$, enquanto os demais recebem a capacidade $\lfloor cm/n \rfloor$. Com as capacidades estabelecidas, mapeamos os objetos no círculo e o percorremos no sentido horário até encontrar um servidor, da mesma forma que o *hashing* consistente. Porém, ao encontrar o servidor, fazemos uma verificação do seu estado. Caso o servidor esteja com capacidade disponível, o objeto pode ser designado ao mesmo. Caso contrário, continuamos percorrendo o círculo até encontrar um servidor com capacidade disponível.

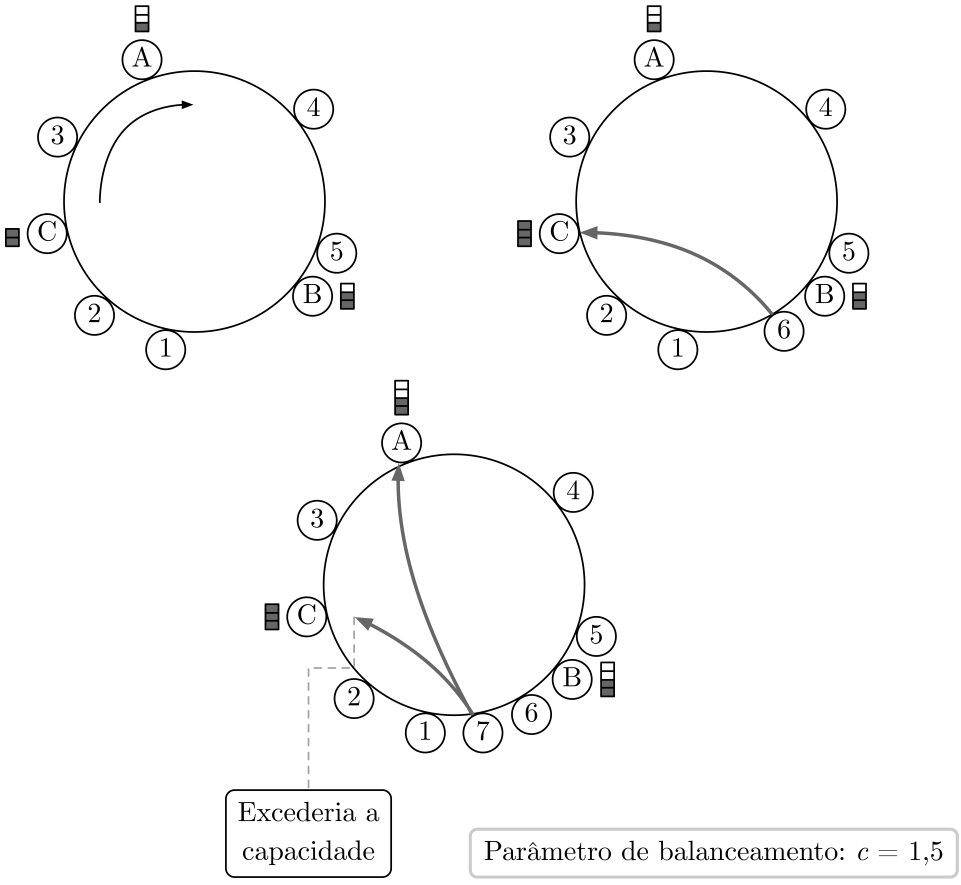


Figura 2.4: Exemplo de funcionamento do *hashing* consistente com cargas limitadas.

A Figura 2.4 exemplifica o funcionamento do algoritmo. Usando apenas

hashing consistente, o objeto sete seria mapeado para o servidor C, deixando-o muito mais sobrecarregado que os demais. No entanto, a modificação proposta por Mirrokni, Thorup e Zadimoghaddam (2018) evita essa situação e o objeto acaba sendo mapeado para o servidor A. No exemplo, assumimos que o parâmetro de balanceamento é 1,5, o que significa que nenhum servidor pode ficar com a carga superior a 150% da carga média arredondada para cima ($\lceil m/n \rceil$).

Apesar de recente, já existem implantações desse algoritmo em aplicações da indústria. Destacamos o exemplo descrito por Rodland (2016) por dois motivos: primeiro, porque também aborda o cenário de vídeos; segundo, porque foi a partir da leitura desse *post* que encontramos o *hashing* consistente com cargas limitadas. Resumidamente, o autor usa esse algoritmo para melhorar o processo de *caching* de metadados de vídeo para uma importante aplicação da plataforma de vídeos do Vimeo[4]. Antes de usar o *hashing* consistente com cargas limitadas, as requisições eram distribuídas entre os servidores da aplicação com o método de balanceamento *least connections*. Ao mudar para o *hashing* consistente com cargas limitadas, houve um aumento de mais de 20% na taxa de *hits* local da aplicação[5].

Enxergamos no *hashing* consistente com cargas limitadas um grande potencial para realizar uma coordenação efetiva de caches nos *clusters* de uma CDN. A partir da análise do seu funcionamento, acreditamos que ele é capaz de mitigar o problema de desbalanceamento de carga e, ao mesmo tempo, de se aproveitar dos benefícios dos esquemas baseados em *hashing*. Portanto, criamos uma plataforma de experimentação e conduzimos uma análise experimental desse algoritmo, apresentadas em detalhes nos próximos capítulos.

2.4 Entrega de Vídeos na *Web*

Para ilustrar como é realizada a entrega de vídeos na *web*, apresentamos a técnica de *streaming* adaptativo. Os conceitos introduzidos aqui são relevantes para melhor compreendermos os próximos capítulos.

2.4.1 *Streaming* Adaptativo

O *streaming* HTTP adaptativo (HAS – *HTTP Adaptive Streaming*) é a técnica utilizada por diversos serviços de vídeo na Internet, como YouTube,

[4] <https://vimeo.com/>

[5] Valor estimado com base no gráfico apresentado pelo autor com a taxa de *hits* antes e depois da aplicação do *hashing* consistente com cargas limitadas.

Vimeo e GloboPlay[6] para transmitir o seu conteúdo na *web* via HTTP. Seu funcionamento requer que os vídeos estejam disponíveis em múltiplas *bit rates*, representando diferentes níveis de qualidade, e segmentados em fragmentos de apenas alguns segundos para permitir que o cliente reaja rapidamente às variações nas condições de rede, solicitando a representação mais adequada em cada instante. Dessa forma, evitam-se travamentos na reprodução do vídeo e utiliza-se a banda disponível da melhor maneira possível (SEUFERT et al., 2015).

Apesar de existirem diversas soluções de HAS diferentes, o princípio de operação delas é bastante similar e pode ser descrito de forma genérica. A Figura 2.5 ilustra o funcionamento de uma solução HAS. Optamos por utilizar a nomenclatura da solução proprietária da Apple[7], HTTP Live Streaming (HLS) (PANTOS; MAY, 2017), pois a nossa avaliação experimental, apresentada em detalhes no Capítulo 4, é conduzida a partir de *logs* de entrega no qual essa solução foi empregada. Inicialmente, o cliente realiza uma requisição HTTP para obter arquivos de índice, que contêm os metadados das diferentes representações disponíveis. No HLS, obtém-se primeiro a *Master Playlist*, que contém diferentes *Media Playlists*, uma para cada representação. Por sua vez, as *Media Playlists* têm uma lista dos segmentos de uma dada representação, que devem ser reproduzidas sequencialmente pelo cliente. Com esses arquivos, o cliente consegue formular as requisições HTTP para a representação adequada, que é escolhida com base nas condições atuais da rede e do estado do *buffer* no cliente. Como as mudanças de representação só podem ocorrer entre diferentes segmentos, é recomendável adotar durações curtas, tipicamente de um a quinze segundos, para que a reação a uma variação na rede seja rápida o suficiente, e garantir que todos os segmentos estejam perfeitamente alinhados no tempo, para que a transição ocorra de maneira suave (SEUFERT et al., 2015).

[6] <https://globoplay.globo.com/>
[7] <https://www.apple.com/>

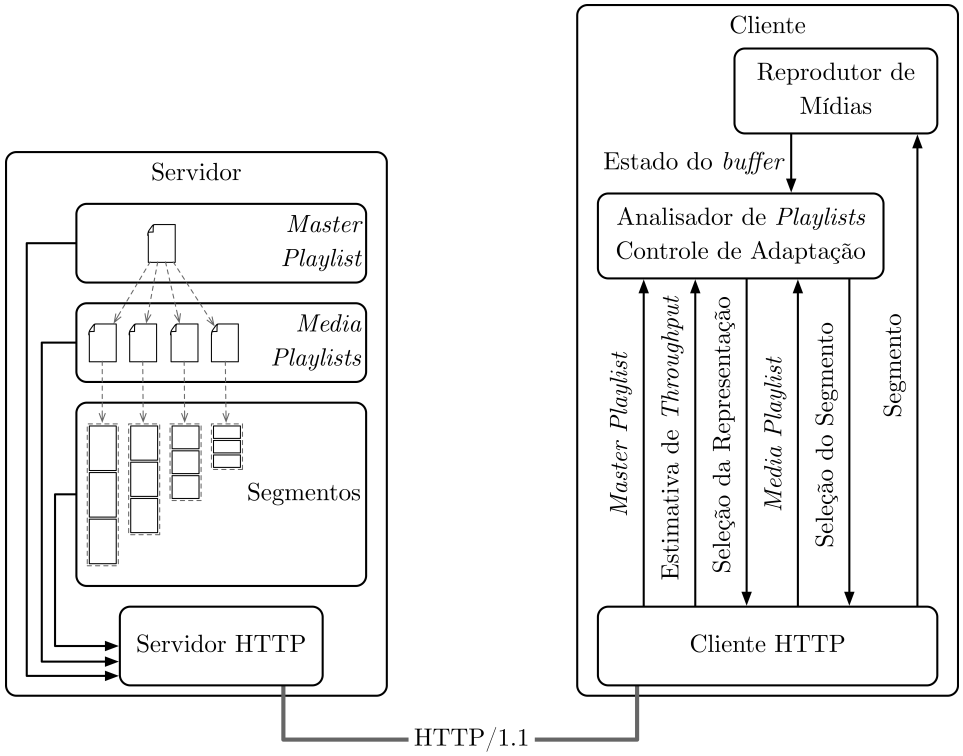


Figura 2.5: Funcionamento do *streaming* HTTP adaptativo. Adaptada do trabalho de Seufert et al. (2015).

3
Plataforma de Experimentação

Este capítulo detalha a plataforma criada para a avaliação experimental do *hashing* consistente com cargas limitadas e de outros algoritmos que nos servem de bases de comparação. Inicialmente, apresentamos o funcionamento geral da plataforma, passando por cada um dos seus componentes. Em seguida, descrevemos brevemente a ferramenta que desenvolvemos para a execução da plataforma em um ambiente local. Por fim, especificamos a infraestrutura final utilizada durante os nossos experimentos.

3.1
Funcionamento Geral

Buscamos criar uma plataforma de experimentação que nos permitisse avaliar o desempenho de diferentes estratégias para o funcionamento interno de um *cluster* de servidores de cache de maneira simples. Para isso, projetamos os seguintes componentes:

- Gerador de Carga: Simula o tráfego gerado pelos usuários.
- Coordenador: Responsável pela estratégia de coordenação empregada no *cluster*. Esse componente é dividido em dois:
  - Decisor: Define o servidor de cache designado para atender cada requisição baseado no estado atual do *cluster*.
  - Monitor: Realiza verificações periódicas, coletando informações dos Caches para compartilhar com o Decisor.
- Cache: Armazena cópias do conteúdo que foi buscado no servidor de origem do provedor de conteúdo.
- Origem: Gera e fornece qualquer arquivo requisitado.

Considerando o modelo de CDN apresentado no capítulo anterior, a nossa plataforma é a representação de um *cluster*, que tem conexão com a origem e recebe requisições dos usuários que estão nas proximidades. Esse *cluster* é composto por um Coordenador e pode ter um número qualquer de Caches. A Figura 3.1 ilustra como esses componentes são interligados e como

se comunicam de forma genérica. Todas as requisições partem do Gerador de Carga, e o fluxo de cada uma é o seguinte:

1. Gerador de Carga realiza a requisição para o Coordenador.
2. Baseado na estratégia configurada, o Coordenador decide qual Cache deve atender a essa requisição e retorna o endereço do servidor escolhido para Gerador de Carga.
3. O Gerador de Carga redireciona a requisição para o Cache escolhido.
4. O Cache verifica se possui uma cópia do objeto requisitado armazenado.
   - 4.1 Caso afirmativo, envia o objeto para o Gerador de Carga.
   - 4.2 Caso contrário, repassa a requisição para a Origem.
     - 4.2.1 A Origem gera o conteúdo e envia para o Cache.
     - 4.2.1 O Cache armazena uma cópia do objeto e envia para o Gerador de Carga.
5. A requisição é completada quando todos os dados chegam ao Gerador de Carga.

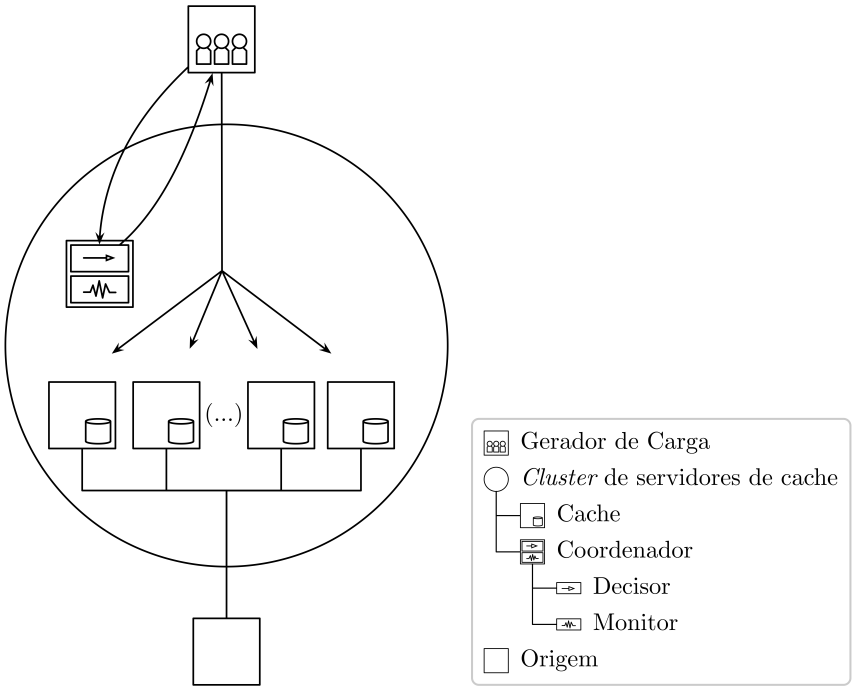


Figura 3.1: Componentes da plataforma de experimentação.

Idealmente, a consulta ao Coordenador deveria ser feita apenas uma vez por sessão de vídeo. Em um cenário real, o cliente (e.g., reprodutor de vídeos)

realizaria uma requisição para o Coordenador antes do início da reprodução do vídeo, que retornaria o endereço do servidor de cache escolhido dentro do *cluster*. Em seguida, o endereço recebido seria usado para requisitar os arquivos de índice e segmentos de vídeo, mantendo a conexão ativa com o Cache durante toda a sessão. No entanto, para simplificarmos a execução dos nossos experimentos, assumimos que cada requisição é precedida por uma consulta ao Coordenador. Como todos os algoritmos são avaliados sob as mesmas condições, acreditamos que isso não traga impactos para os resultados. A seguir, fazemos uma descrição detalhada de cada um dos componentes.

3.1.1 Gerador de Carga

Neste componente, utilizamos a ferramenta de teste de carga HTTP denominada Vegeta[1] (versão 7.0.2). Os principais fatores que nos levaram a escolher essa ferramenta foram:

- Suporte à realização de requisições para múltiplas URLs em sequência: Algumas ferramentas aceitam apenas uma URL como entrada, mas isso é insuficiente para avaliar os efeitos de diferentes estratégias de coordenação. Afinal, nesse caso, qualquer abordagem apenas prejudicaria o sistema, pois o ideal seria simplesmente armazenar o único objeto em todos os servidores e adotar alguma técnica de balanceamento de carga. Além disso, como a carga gerada na nossa avaliação experimental é baseada em *logs* de uma CDN de vídeos real, é essencial que o gerador de carga utilizado seja capaz de receber uma lista de URLs como entrada e de realizar as requisições de forma sequencial.
- Geração de requisições a uma taxa constante: Como veremos no próximo capítulo, usamos o número de requisições por segundo para definir os diferentes níveis de tráfego usados nos experimentos.
- Simplicidade: A Vegeta pode ser usada tanto como uma ferramenta de linha de comando quanto uma biblioteca Go[2]. Nós a utilizamos como uma ferramenta de linha de comando. Sua interface é muito clara e conseguimos iniciar testes de carga sem a necessidade de muita configuração.

Para esclarecer os pontos listados acima, apresentamos brevemente os comandos oferecidos pela ferramenta, mostrando um exemplo bem próximo à

[1] <https://github.com/tsenart/vegeta/>
[2] <https://golang.org/>

forma que utilizamos. Conforme especificado no manual de uso[3], iniciamos um teste de carga com o comando `attack`, conforme o exemplo a seguir:

```
$ vegeta attack -targets=targets.txt -rate=10 -lazy > \
> results.bin
```

A opção `targets` especifica o arquivo de entrada `targets.txt`, contendo as URLs e os métodos HTTP que devem ser utilizados para gerar as requisições. Apesar de não usarmos, também é possível especificar o corpo e os cabeçalhos de cada requisição nesse arquivo. A opção `rate` especifica quantas requisições por segundo devem ser disparadas pela ferramenta. Nesse caso, especificamos o valor de dez requisições por segundo. Por fim, a opção `lazy` torna a avaliação do arquivo de entrada preguiçosa em vez de ansiosa. Para nós, o efeito prático dessa opção é que cada entrada passa a ser requisitada apenas uma vez. Na avaliação ansiosa, as requisições são feitas em sequência por tempo indeterminado ou por um tempo especificado na opção `duration`. A saída desse comando é gravada no arquivo `results.bin`.

Com o comando `report`, a ferramenta interpreta a saída do teste e fornece métricas importantes sobre a sua execução, como a latência, o número de *bytes* trafegados e a taxa de sucesso.

```
$ vegeta report -inputs=results.bin
```

Por fim, a ferramenta ainda oferece o comando `dump` que recebe o arquivo binário de resultados e mostra as informações de cada requisição individualmente. Esse comando nos foi particularmente útil para depurar comportamentos inesperados. Cada comando também possui diversos outros parâmetros que não foram mencionados aqui, mas podem ser verificados no manual.

3.1.2 Coordenador

A base deste e de todos os demais componentes da plataforma é o servidor *web* OpenResty[4] (versão 1.13.6.1). Os principais motivos que nos levaram à escolha dessa ferramenta foram o seu desempenho e a sua flexibilidade. Integrando o núcleo de alto desempenho do servidor Nginx[5] com o LuaJIT[6], um compilador *just-in-time* para a linguagem Lua (IERUSALIMSCHY, 2016), o OpenResty nos permite executar código Lua em diferentes momentos. A Figura 3.2 mostra algumas diretivas introduzidas por um dos módulos do

[3] <https://github.com/tsenart/vegeta#usage-manual>
[4] <https://openresty.org/>
[5] <https://nginx.org/>
[6] <http://luajit.org/>

OpenResty que servem como pontos de entrada para a execução de código Lua. Na fase de inicialização, as diretivas nos permitem atuar no momento em que os arquivos de configuração do servidor estão sendo carregados e durante a inicialização dos processos *workers*. Na fase de reescrita/acesso, temos ganchos que possibilitam a execução de código imediatamente antes do *handshake* SSL, e logo após reescritas da requisição e verificações de acesso. Além disso, podemos atribuir variáveis programaticamente. Durante a fase de conteúdo, conseguimos manipular a geração da resposta e, caso o servidor esteja atuando como um *proxy* reverso, podemos programar a escolha do servidor designado. Por fim, na fase de *log* temos uma diretiva que permite operar durante o processo de *logging* da requisição.

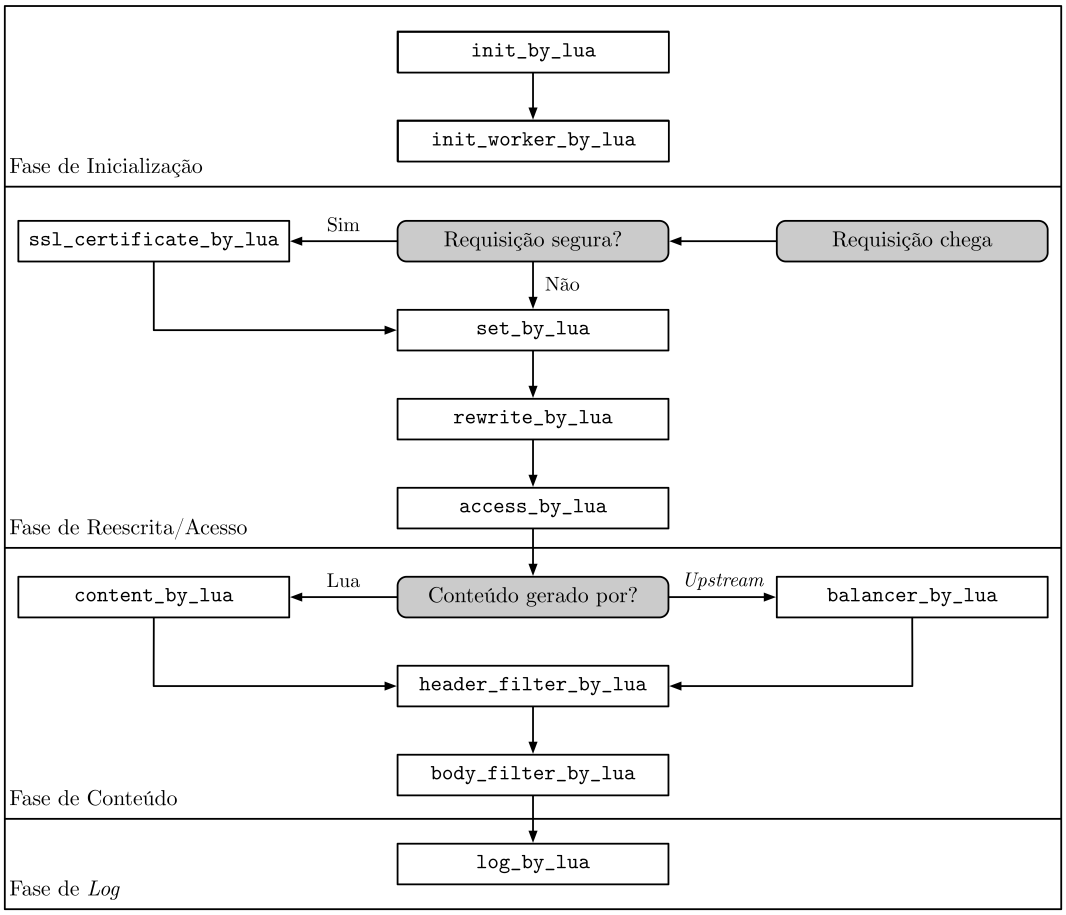


Figura 3.2: Diretivas do módulo `ngx_http_lua_module`. Adaptada da documentação do projeto[7].

Como vimos anteriormente, o Coordenador escolhe o servidor de cache designado para atender a uma requisição e informa ao Gerador de Carga. Seu funcionamento é dividido em duas partes, representadas pelo Decisor e pelo Monitor, que são apresentadas em detalhes nas Seções 3.1.2.1 e 3.1.2.2, respectivamente.

[7]<https://github.com/openresty/lua-nginx-module#readme>

3.1.2.1 Decisor

O ponto de partida para a criação desse elemento foi a biblioteca de código aberto `lua-resty-balancer`[8] (versão 0.02rc4), que provê uma implementação genérica de *hashing* consistente. Em cima desse código, implementamos o algoritmo *hashing* consistente com cargas limitadas (MIRROKNI; THORUP; ZADIMOGHADDAM, 2018), usando o número de conexões ativas em cada servidor como a métrica de carga. Também implementamos o método de balanceamento *least connections*, que seleciona o servidor com o menor número de conexões ativas.

Na fase de inicialização do servidor, carregamos o código com os algoritmos mencionados acima, passando, dentre outros parâmetros, qual deve ser utilizado. Depois, a cada requisição, executamos o *script* decisor para definir o servidor designado, e enviamos o seu IP no cabeçalho *Location* da resposta de redirecionamento do Coordenador. Todos os algoritmos dependem de informações sobre o estado de atividade dos servidores e, no caso do *hashing* consistente com cargas limitadas e do *least connections*, também sobre o número de conexões ativas em cada um. Como mencionado anteriormente, essas informações são coletadas pelo Monitor.

3.1.2.2 Monitor

Para criação do Monitor, também utilizamos uma biblioteca de código aberto como base. A `lua-resty-upstream-healthcheck`[9] (versão 0.05) realiza monitorações ativas periódicas em um conjunto de servidores para verificar quais estão responsivos e evitar que sejam feitas requisições para servidores inativos. Essa biblioteca nos oferece uma base confiável para as monitorações, mas além de não se adequar totalmente à nossa plataforma, não coleta dados sobre a carga em cada servidor. Por isso, fizemos algumas modificações em seu código.

Em primeiro lugar, tratamos de integrá-la à nossa plataforma. O Nginx e, consequentemente, o OpenResty podem ser usados como *proxy* reverso e serem configurados para realizar balanceamento de carga entre vários servidores de aplicação. A `lua-resty-upstream-healthcheck` assume esse tipo de configuração e depende de diretivas que são usadas apenas nesse cenário. Assim, removemos essa dependência e introduzimos um dicionário compartilhado[10]

[8] <https://github.com/openresty/lua-resty-balancer/>
[9] <https://github.com/openresty/lua-resty-upstream-healthcheck/>
[10] <https://github.com/openresty/lua-nginx-module#lua_shared_dict>

para manter o estado dos servidores, permitindo que a biblioteca seja usada mesmo quando o servidor não atua como um *proxy* reverso.

A outra modificação está relacionada à coleta do número de conexões ativas em cada servidor, que é utilizado no *least connections* e no *hashing* consistente com cargas limitadas. Todos os Caches da plataforma são configurados para escutar em duas portas, uma para o tráfego padrão e outra para o Monitor. Quando uma requisição chega com a URI esperada na porta dedicada a atender o Monitor, o Cache retorna o número de conexões ativas no corpo da resposta. Recebendo essa resposta, o Monitor grava o número de conexões ativas no dicionário compartilhado que mencionamos anteriormente. Por fim, esse dicionário também serve para estabelecer a comunicação entre o Monitor e o Decisor, que realiza uma consulta a cada requisição para tomar a decisão do servidor adequado.

3.1.3 Cache

Na Seção anterior, mencionamos que os Caches escutam em duas portas. A operação na porta reservada à monitoração já foi explicada e, por isso, focamos apenas na operação da porta que recebe o tráfego padrão. Neste componente, não usamos nenhum *script* Lua, apenas configuramos o servidor para atuar como um *proxy* reverso com suporte a *caching*. Além disso, como o número de conexões é utilizado pelo *hashing* consistente com cargas limitadas e pelo *least connections* e todas as requisições partem de um mesmo cliente, decidimos desabilitar a persistência de conexões do lado do servidor.

Operando entre o Gerador de Carga e a Origem, o Cache verifica se possui o conteúdo requisitado armazenado em cache. Caso afirmativo, entrega diretamente para o cliente. Caso contrário, busca o conteúdo na Origem e o armazena antes de entregar ao cliente. Quando o volume de dados armazenados ultrapassa o limite estabelecido para armazenamento em cache, ocorre o processo de substituição de conteúdo seguindo a política LRU.

3.1.4 Origem

Por fim, temos a Origem, que atua como o "oráculo" da plataforma, sendo capaz de servir qualquer conteúdo requisitado. Como o arquivo de entrada da nossa avaliação experimental é baseado em *logs* de uma CDN de vídeos real, que conta com milhões de mídias diferentes, seria inviável armazenar todo o conteúdo na Origem da plataforma experimental. Mesmo limitando esse

conteúdo a apenas arquivos requisitados no arquivo de entrada, o volume ainda seria muito grande. Portanto, tivemos que buscar uma solução alternativa.

Como não precisamos entregar vídeos de fato durante os nossos experimentos, criamos um arquivo contendo apenas *bytes* nulos com o tamanho do maior arquivo que é trafegado na plataforma (`generator`). Esse é o único objeto armazenado na Origem. Além disso, durante a preparação da entrada, adicionamos o tamanho do arquivo no nome. Assim, através de requisições internas e algumas manipulações feitas com código Lua na URI e nos cabeçalhos da requisição, conseguimos gerar arquivos de qualquer tamanho, atendendo a qualquer requisição com apenas um arquivo na origem. A Figura 3.3 descreve as operações realizadas para entregar os objetos requisitados.

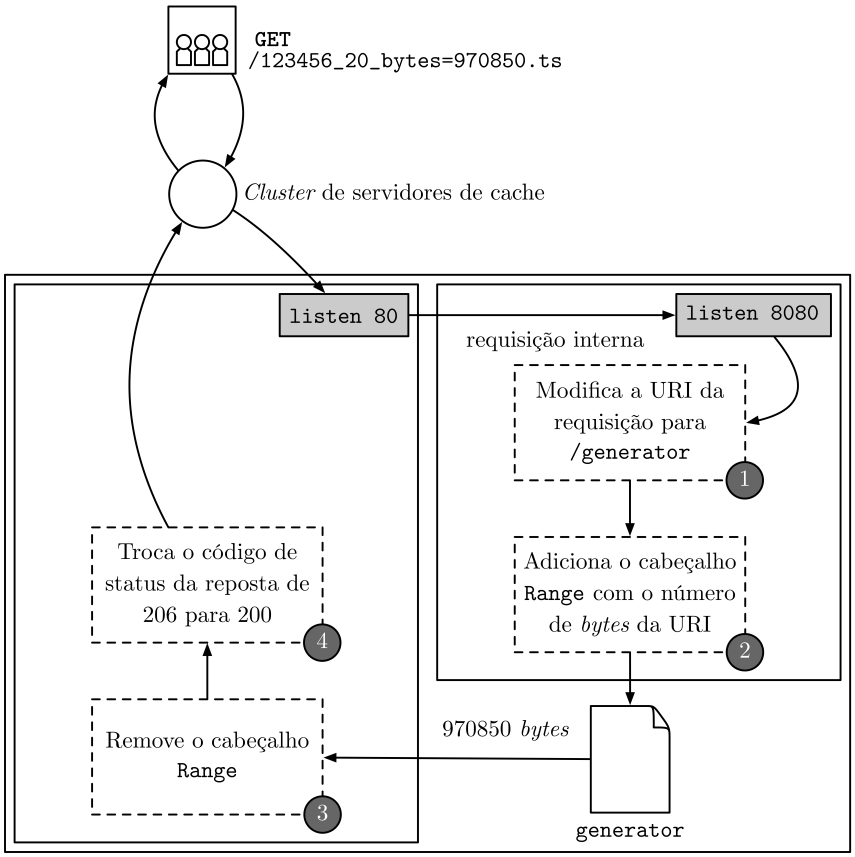


Figura 3.3: Operação da Origem.

3.2 Execução em Ambiente Local

No decorrer deste trabalho, desenvolvemos uma ferramenta que possibilita a montagem e configuração de uma plataforma de testes virtual com todos os elementos descritos acima em ambiente local. Dessa forma, podemos testar o funcionamento de cada componente e todas as suas interações de forma sim-

ples, evitando os esforços e os custos de criação de uma plataforma de testes usando máquinas físicas ou contratando serviços na nuvem.

Essa ferramenta foi desenvolvida em Python[11] e se baseia fortemente na virtualização em nível de sistema operacional usando Docker[12]. Em linhas gerais, cada componente é executado em um *container* isolado dos demais e se comunicam apenas através de uma rede privada. Usando a biblioteca `docker-py` (versão 2.6.1) para interagir com a API do Docker, conseguimos controlar toda a criação e configuração da plataforma, assim como a execução de cada um dos componentes.

3.3 Infraestrutura na Nuvem

Após realizarmos diversos testes localmente para confirmar o funcionamento de todos os componentes, configuramos a plataforma na infraestrutura final, na qual cada componente conta com capacidades de processamento, memória, armazenamento e rede garantidas até um certo nível. Isso é fundamental para a realização dos experimentos, visto que a disputa por recursos no ambiente local impede a medição adequada do desempenho de cada estratégia de coordenação.

Optamos por usar o serviço EC2 (*Elastic Compute Cloud*) da Amazon[13]. Nele, criamos um total de sete instâncias: uma para o Gerador de Carga, uma para o Coordenador, quatro para Caches e uma para a Origem. Os tipos de instância usadas para cada um desses componentes, assim como as principais características, são sumarizadas na Tabela 3.1. A única característica que não consta na descrição fornecida pela Amazon é a capacidade de banda. Os valores de banda presentes na tabela foram encontrados em um *benchmark* disponível *online*[14] e verificados experimentalmente através de alguns testes preliminares. Apesar de estarmos interessados apenas na taxa de transferência mínima garantida pela instância (*baseline*), também mostramos a taxa máxima (*burst*). Os critérios para a escolha de cada instância estão relacionados à metodologia usada para a condução dos experimentos e, por isso, são discutidos no próximo capítulo.

[11] <https://www.python.org/>
[12] <https://www.docker.com/>
[13] <https://aws.amazon.com/pt/ec2/>
[14] <https://cloudonaut.io/ec2-network-performance-cheat-sheet/>

Tabela 3.1: Instâncias EC2 utilizadas pelos componentes da plataforma.

Tipo	**Qtd.**	**Componente**	**Descrição**
c4.8xlarge	2	Coordenador e Origem	Otimizada para computação. vCPUs: 36 Memória: 60 GiB Armazenamento: — Banda (*baseline*): 9,85 Gbps Banda (*burst*): —
c5.large	1	Coordenador	Otimizada para computação. vCPUs: 2 Memória: 4 GiB Armazenamento: — Banda (*baseline*): 0,74 Gbps Banda (*burst*): 10,04 Gbps
i3.2xlarge	4	Cache	Otimizada para armazenamento. vCPUs: 8 Memória: 61 GiB Armazenamento: 1,9 TB (SSD NVMe) Banda (*baseline*): 2,48 Gbps Banda (*burst*): 10,09 Gbps

4
Avaliação Experimental

Este Capítulo apresenta como a nossa avaliação experimental foi conduzida em cima da plataforma detalhada anteriormente e discute os resultados obtidos. Inicialmente, discutimos a origem e a preparação dos dados que serviram de entrada para os experimentos. Em seguida, explicamos a metodologia adotada, assim como o cenário final e as métricas coletadas. Por fim, apresentamos e discutimos os resultados.

4.1
Preparação da Entrada

A base para a criação da entrada utilizada nos experimentos deste trabalho foi obtida a partir de arquivos de *log* de uma CDN de vídeos real, na qual trafegam diversos tipos de vídeos, como jornalismo, esportes, séries e filmes. O primeiro passo para a preparação da entrada foi selecionar um recorte temporal inicial. Coletamos, então, os *logs* de todos os servidores da CDN de um dia útil durante o período da noite, no qual costumam ocorrer os picos de acesso diário, e juntamos em um único arquivo. Em seguida, filtramos apenas requisições bem-sucedidas (código de *status* 200) para segmentos de vídeo HLS. As demais tecnologias de *streaming* HTTP adaptativo representam apenas cerca de 3,2% do volume total trafegado, enquanto os arquivos de índices, apenas cerca de 0,3% do volume trafegado com HLS. Desse *log* filtrado, separamos quarenta milhões de entradas contíguas em torno do pico observado naquele dia. Para viabilizar os experimentos na escala reduzida da nossa plataforma de testes, fizemos uma amostragem sistemática, resultando em um arquivo de quatrocentas mil entradas.

O gráfico na Figura 4.1 mostra a distribuição de popularidade dos arquivos antes e depois da amostragem. Os eixos da esquerda e de baixo indicam a escala do arquivo original, enquanto os eixos da direita e de cima indicam a escala da amostra. Horizontalmente, temos o *rank* dos arquivos, indo do mais popular ao menos popular. Verticalmente, temos o número de vezes com que um arquivo é requisitado. Como o volume de dados é muito grande, todos os eixos encontram-se na escala logarítmica. Observando o formato das duas curvas, notamos que são bem similares e, por isso, consideramos

razoável usar a amostra para a realização dos experimentos. Como mencionado anteriormente, a utilização do arquivo original seria inviável. Considerando o tamanho da nossa plataforma de testes, o tempo para reproduzir todas as linhas desse arquivo seria muito longo e isso resultaria em um custo financeiro extremamente elevado.

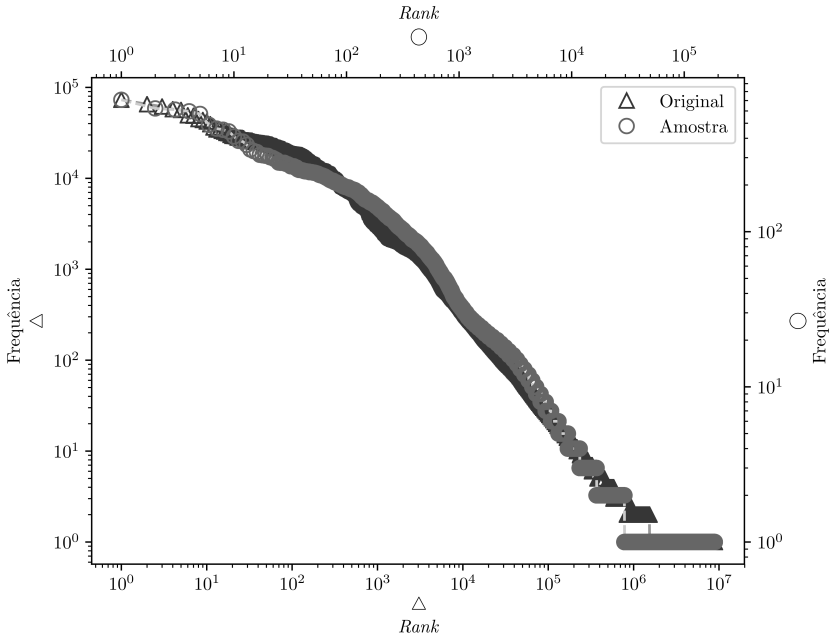
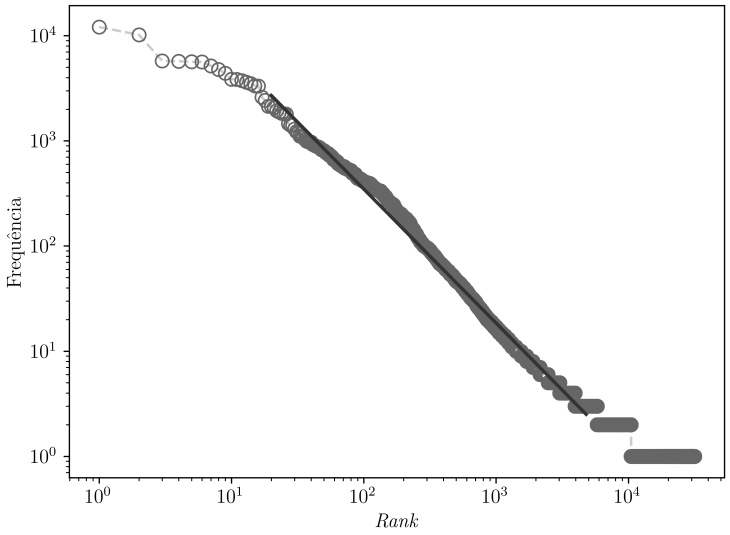


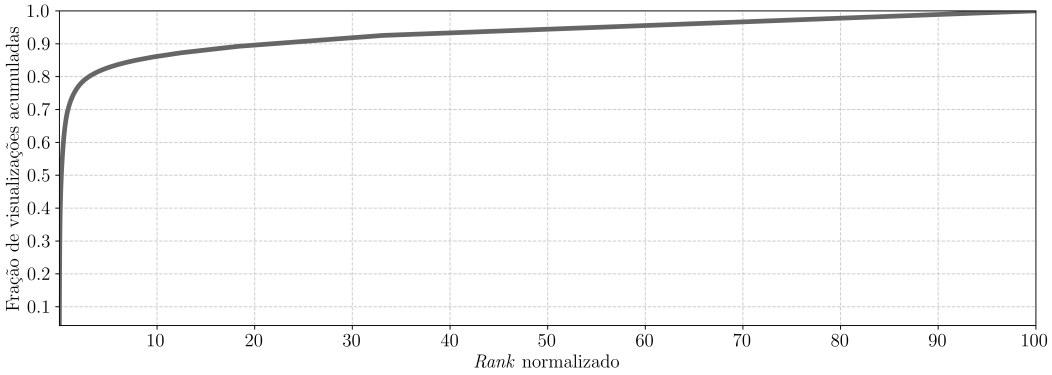
Figura 4.1: Distribuição de popularidade dos arquivos antes e depois da amostragem.

Para mostrar que a popularidade dos vídeos da nossa amostra apresenta o comportamento de cauda longa discutido na Seção 2.3.1, traçamos os gráficos da Figura 4.2. Os valores presentes nos gráficos foram obtidos acumulando as requisições feitas por diferentes endereços de IP para um determinado vídeo. Requisições para segmentos distintos de um vídeo originadas do mesmo endereço de IP contam como apenas uma visualização. Na Figura 4.2(a), mostramos a distribuição de popularidade dos vídeos de forma análoga à Figura 4.1. Nesse tipo de gráfico, uma linha reta representa a distribuição de uma lei de potência. Sabendo disso, notamos que o formato da nossa curva, principalmente na parte central, é próximo ao de uma lei de potência, que possui a cauda longa. Para evidenciar esse fato, fizemos uma regressão linear considerando apenas a parte central da distribuição e ajustamos a reta no gráfico. Uma outra forma de visualizar esse comportamento é ilustrada na Figura 4.2(b). O eixo horizontal apresenta os *ranks* normalizados entre zero e cem, e o eixo vertical mostra o valor da fração do número de visualizações acumulado pelo total de visualizações. Nesse gráfico, podemos ver que apenas 20% dos vídeos representam quase 90% das visualizações, enquanto os demais

80% representam apenas cerca de 10% das visualizações. Isso mostra que existem muitos vídeos com baixa popularidade formando a cauda longa da nossa distribuição.



(a) Distribuição de popularidade.



(b) Representação cumulativa de visualizações.

Figura 4.2: Análise da popularidade dos vídeos.

Por fim, após obter a amostra, preparamos o *log* para ser efetivamente usado na plataforma. Para isso, geramos um novo arquivo no formato esperado pela ferramenta Vegeta, usada no Gerador do Carga, adotando o seguinte padrão para as URLs: `http://<ip_do_coordenador>/<identificador_do_vídeo>_<número_do_segmento>_bytes=<tamanho_do_arquivo>.ts`. Dessa forma, cada arquivo é identificado de maneira única na plataforma e o tamanho em *bytes* de cada um fica exposto para que a Origem possa gerar os arquivos sob demanda, conforme visto no Capítulo 3.

4.2 Metodologia de Teste

Inicialmente, fizemos uma análise do tamanho médio dos segmentos. Com exceção dos últimos segmentos de cada vídeo, que podem ser mais curtos, a duração dos arquivos presentes na amostra é de dez segundos. Apesar disso, como esses segmentos são entregues em múltiplas representações, o tamanho de cada um é bastante variado e observamos arquivos variando de apenas alguns *kilobytes* a dezenas de *megabytes*. Em seguida, realizamos alguns experimentos preliminares, observando a utilização de rede das instâncias com o `iftop`[1], e dimensionamos a plataforma para testar três situações distintas:

1. Tráfego excessivo, na qual capacidade somada das interfaces de saída de rede não é suficiente para atender a demanda média recebida. Calculamos essa demanda multiplicando o tamanho médio dos segmentos pelo número de requisições realizadas por segundo. Nesse caso, realizamos 400 requisições por segundo (RPS).

2. Tráfego intenso, na qual a demanda média recebida corresponde a cerca de 75% da capacidade total das interfaces de rede do sistema. Essa situação é alcançada quando geramos 300 RPS.

3. Tráfego leve, na qual o sistema opera abaixo do seu limite de rede, recebendo uma demanda média de cerca de 50% da sua capacidade com uma taxa de 200 RPS.

A Figura 4.3 ilustra a plataforma de testes com destaque na taxa de transferência máxima entre as diferentes instâncias. Essa figura nos ajuda a entender o principal motivo por trás da escolha da maioria das instâncias da nossa plataforma, que é a capacidade de banda. Os valores de RPS mencionados acima foram calculados considerando que a capacidade da interface de rede de cada Cache é de 2 Gbps. Portanto, escolhemos uma instância otimizada para armazenamento[2] com capacidade mínima superior a esse valor. Para garantir que essa capacidade permanecesse constante durante todos os experimentos, usamos a ferramenta de controle de tráfego `tc`[3] para limitar a interface dos Caches a 2 Gbps. A escolha das instâncias do Gerador de Carga e da Origem foram apenas consequência da escolha da capacidade de banda dos Caches. Como queremos avaliar o desempenho do *cluster* com diferentes estratégias

[1] <http://www.ex-parrot.com/pdw/iftop/>

[2] Essa família de instâncias fornece armazenamento temporário com discos que estão fisicamente anexados à máquina física sobre a qual a máquina virtual está sendo executada.

[3] <https://wiki.linuxfoundation.org/networking/iproute2/>

de coordenação, dimensionamos esses dois componentes para que eles não fossem gargalos no sistema. Assim, escolhemos instâncias que oferecessem uma capacidade constante de, pelo menos, 8 Gbps. A única instância que não foi influenciada pela capacidade de rede do sistema foi a que desempenha o papel de Coordenador. Como as entradas e saídas desse componente são da ordem de *bytes*, a capacidade da sua interface de rede não é um problema para o seu funcionamento. Com isso, escolhemos a instância otimizada para computação mais básica, que foi capaz de processar todas as respostas mantendo a utilização de recursos baixa. Outras características relevantes também são exibidas nessa figura. Como podemos notar, o monitor realiza suas monitorações periódicas de um em um segundo e a capacidade configurada para armazenamento em cache de cada servidor é de 20 GB.

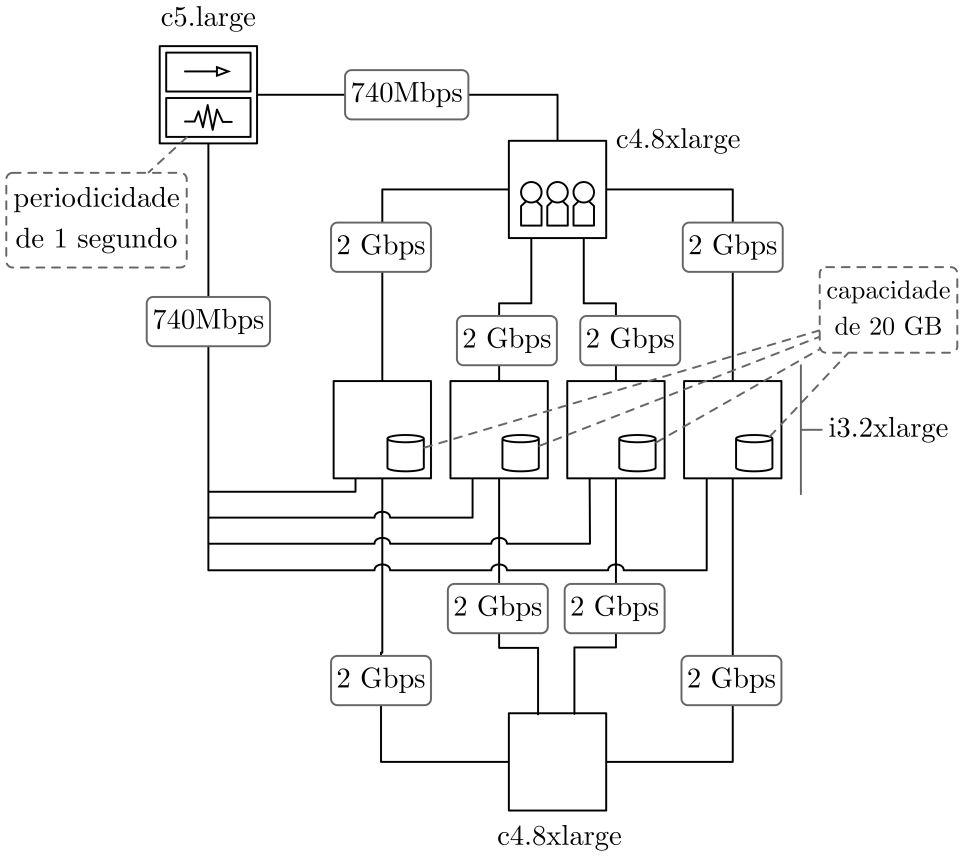


Figura 4.3: Plataforma de testes no EC2.

O objetivo da nossa avaliação experimental é explorar os benefícios e limitações do *hashing* consistente com cargas limitadas como forma de coordenar um grupo de caches para melhorar o desempenho de uma CDN de vídeos. Para mensurar o efeito do algoritmo sobre o sistema, consideramos que a métrica mais importante é a latência, pois afeta diretamente a qualidade de experiência dos usuários. Essa métrica é fortemente influenciada por dois

fatores, a eficiência dos caches e o balanceamento de carga do sistema. Quanto maior a eficiência do cache, mais requisições são atendidas sem acessar a origem e, consequentemente, menor a latência. O balanceamento de carga, por sua vez, evita que máquinas específicas fiquem sobrecarregadas e aumentem drasticamente o tempo de resposta do sistema.

Com isso em vista, escolhemos outros dois algoritmos para servir como bases de comparação para o *hashing* consistente com cargas limitadas, o *hashing* consistente convencional e o *least connections*. O primeiro prioriza a eficiência de cache, enquanto o segundo prioriza o balanceamento de carga. O *hashing* consistente com cargas limitadas busca um equilíbrio entre os dois, ditado pelo seu parâmetro de balanceamento. Quando colocamos o parâmetro de balanceamento próximo a um, o comportamento do algoritmo fica similar ao *least connections*, pois a tolerância de desbalanceamento entre os servidores é muito baixa. Por outro lado, quando o parâmetro de balanceamento é muito alto, seu comportamento se aproxima ao que observamos no *hashing* consistente convencional, pois o algoritmo permite um desbalanceamento tão alto que as decisões acabam sendo tomadas apenas com base no processo de *hashing*. No caso dos algoritmos baseados em *hashing*, usamos o identificador do vídeo como chave da função *hash*. Isso faz com que todos os segmentos de um vídeo sejam mapeados no mesmo ponto do círculo usado pelos algoritmos para atribuir requisições a servidores, conforme discutido no Capítulo 2. O mesmo não acontece com o *least connections* que sempre atribui a requisição ao servidor com o menor número de conexões ativas.

Finalmente, descrevemos o procedimento adotado para a realização dos experimentos.

- Inicialmente, separamos o arquivo de entrada em dois. As oitenta mil primeiras linhas (20% das entradas) foram destinadas para fazermos *precaching* do conteúdo, enquanto o restante (80% das entradas) foi usado nos testes efetivamente.
- Em seguida, fizemos o processo de *precache* utilizando o *hashing* consistente no Coordenador e gerando 200 RPS no Gerador de Carga. Decidimos realizar esse processo para criar um cenário mais próximo da realidade, no qual o sistema já se encontra em operação.
- Ao final desse processo, gravamos o estado de todos os Caches em um volume separado, no qual os dados são persistidos. Com isso, é possível copiar o conteúdo desse volume para o diretório de cache configurado antes de cada rodada de experimento e, assim, realizar todos os experimentos com o mesmo conteúdo inicial.

– Depois, usamos os 80% restante das entradas para testar cada situação de tráfego com os diferentes algoritmos. Executamos três repetições para cada experimento. Esse número de repetições foi definido levando em conta a baixa variação das medidas e os custos para cada iteração.

– No caso específico do *hashing* consistente com cargas limitadas, realizamos uma etapa adicional para a definição do parâmetro de balanceamento. Nela, variamos o parâmetro de balanceamento buscando pela menor latência. Realizamos somente uma repetição com parâmetros distintos, pois o objetivo dessa etapa não é encontrar o melhor valor possível, mas apenas definir um valor que possa ser considerado bom para mostrar o desempenho do algoritmo.

Seguindo esse procedimento, coletamos os resultados dos algoritmos nas situações planejadas e realizamos algumas análises que são apresentadas a seguir.

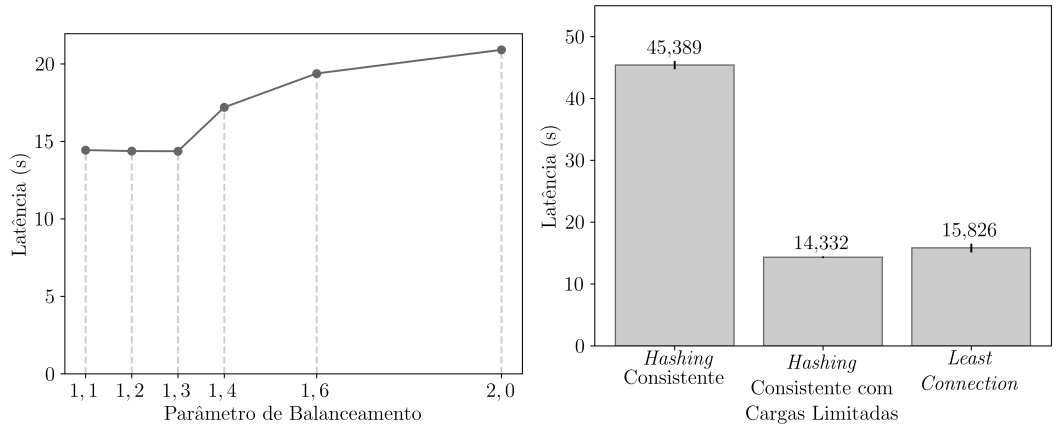
4.3 Resultados

Esta Seção discute os resultados obtidos na nossa avaliação experimental. Os resultados são agrupados de acordo com o nível de tráfego, e as suas interpretações são feitas com base na eficiência de cache e no desbalanceamento do sistema. Ao final da Seção, ainda fazemos algumas considerações gerais sobre os números mostrados.

4.3.1 Tráfego Excessivo

A Figura 4.4 mostra as medidas de latência obtidas no cenário de tráfego excessivo, no qual geramos 400 requisições por segundo. No gráfico da Figura 4.4(a), mostramos os resultados do *hashing* consistente com cargas limitadas usando diferentes parâmetros. O parâmetro que apresentou a menor latência e, consequentemente, foi usado nos experimentos comparativos é o 1,3. Na Figura 4.4(b), mostramos os resultados obtidos com cada algoritmo. Os números e traços acima das barras representam a média das três medições e o intervalo de confiança de 95%, respectivamente. O melhor caso foi obtido com o *hashing* consistente com cargas limitadas, enquanto o pior caso ocorreu usando o *hashing* consistente convencional.

Para entendermos melhor o que levou aos resultados apresentados, geramos os gráficos da Figura 4.5. O gráfico da Figura 4.5(a) representa a eficiência de cache, mostrando o número de requisições servidas diretamente do cache sobre o total de requisições recebidas (taxa de *hits*) em cada servidor de cache.



(a) Definição do parâmetro de balanceamento. (b) Comparativo dos algoritmos.

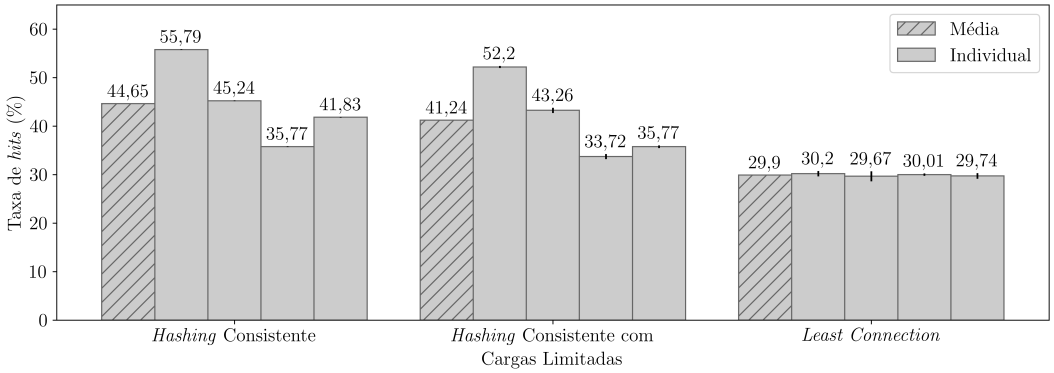
Figura 4.4: Latência com tráfego excessivo.

O gráfico da Figura 4.5(b), por sua vez, ilustra o balanceamento de carga, mostrando como ficou a distribuição das requisições geradas entre os servidores de cache.

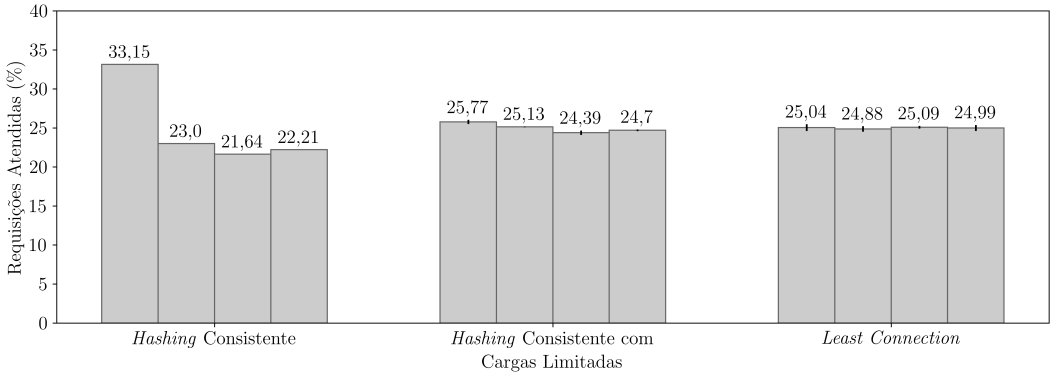
O primeiro ponto que nos chama atenção é o desbalanceamento provocado *hashing* consistente. Observando a Figura 4.5(b), notamos que o primeiro servidor recebe um número de requisições significativamente maior que os demais. Apesar de todos os servidores estarem operando acima do limite nesse cenário, esse desbalanceamento sobrecarrega ainda mais o primeiro servidor. A consequência disso é um grande aumento no tempo de resposta desse servidor, provocando a alta latência observada. O *hashing* consistente com cargas limitadas causa um desbalanceamento mais discreto e, quando analisamos a Figura 4.5(a), percebemos que sua taxa de *hits* é muito superior ao *least connections*. Por isso, observamos a menor latência com esse algoritmo.

4.3.2 Tráfego Intenso

De forma análoga à Seção anterior, a Figura 4.6 mostra os resultados de latência observados no cenário de tráfego intenso, no qual geramos 300 requisições por segundo. O ponto de menor latência na Figura 4.6(a) foi atingido com o parâmetro de balanceamento 2,7. É interessante notar a variação do parâmetro de balanceamento em relação ao cenário anterior. Em uma situação na qual o sistema está totalmente sobrecarregado, o valor do parâmetro de balanceamento "ideal" prioriza o balanceamento de carga e chega a um valor próximo a um. Quando a demanda no sistema se reduz, o parâmetro aumenta e permite que os servidores fiquem com uma carga de até 270% da média, tirando maior proveito da eficiência proporcionada por esquemas



(a) Eficiência de cache.



(b) Balanceamento de carga.

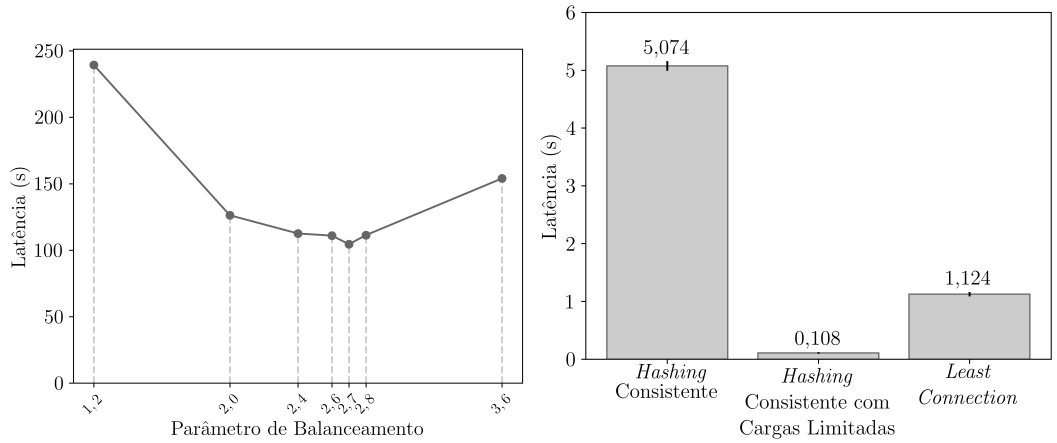
Figura 4.5: Análise da latência com tráfego excessivo.

baseados em *hashing*. Observando a Figura 4.6(b), percebemos um resultado equivalente ao anterior, em que o *hashing* consistente com cargas limitadas apresenta o melhor resultado, enquanto o *hashing* consistente convencional apresenta o pior resultado.

A Figura 4.7 traz detalhes da eficiência de cache e do balanceamento de carga. Analisando a Figura 4.7(b), percebemos que o motivo para o desempenho inferior do *hashing* consistente foi, novamente, o desbalanceamento. Na situação de tráfego intenso, o sistema não deveria ter tantos problemas com a demanda. No entanto, por conta do desbalanceamento, o primeiro servidor acaba ficando sobrecarregado e passa a responder lentamente. Considerando o gráfico da Figura 4.7(a), notamos que a diferença entre o *hashing* consistente com cargas limitadas e o *least connections* se deve outra vez à melhor eficiência de cache do primeiro.

4.3.3 Tráfego Leve

A Figura 4.8 mostra os resultados no cenário de tráfego leve, que recebe 200 requisições por segundo. De imediato, percebemos duas grandes diferenças



(a) Definição do parâmetro de balanceamento. (b) Comparativo dos algoritmos.

Figura 4.6: Latência com tráfego intenso.

em relação aos casos anteriores. A primeira é que não mostramos o resultado do *hashing* consistente com cargas limitadas. A segunda é que o *hashing* consistente foi o que apresentou o melhor desempenho, ao contrário do que ocorreu nos demais cenários.

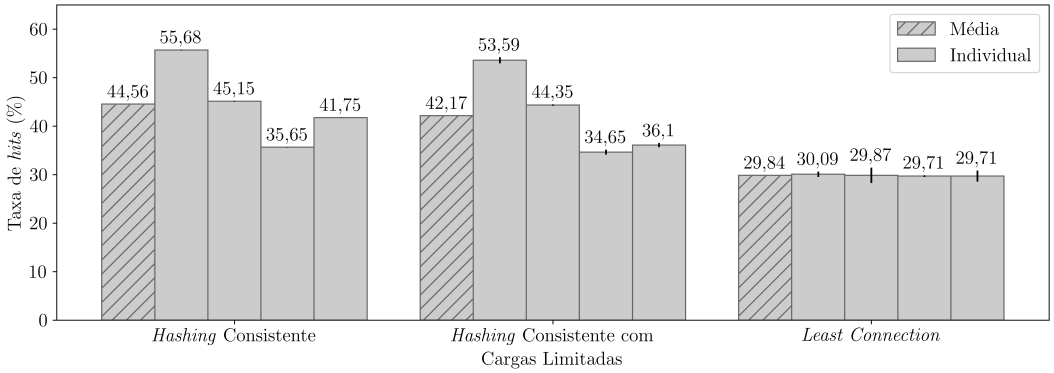
A Figura 4.9 esclarece a razão de não termos apresentado os resultados do *hashing* consistente com cargas limitadas. Como podemos ver, o valor da latência desse algoritmo tende a diminuir com o aumento do parâmetro de balanceamento. Com um valor suficientemente grande, o comportamento desse algoritmo se iguala ao *hashing* consistente e, na prática, as suas latências se tornam equivalentes.

Analisando a Figura 4.10, vemos que, assim como nos cenários anteriores, o *hashing* consistente tem a melhor eficiência de cache (gráfico da Figura 4.10(a)), mas causa o maior desbalanceamento (gráfico da Figura 4.10(b)). No entanto, dessa vez, esse algoritmo é o que tem a menor latência. A razão para isso é a baixa demanda recebida no cenário de tráfego leve. Mesmo causando desbalanceamento no sistema, a taxa de requisições que chega ao primeiro servidor não é o suficiente para sobrecarregá-lo. Dessa forma, apenas a eficiência de cache influencia o tempo de resposta.

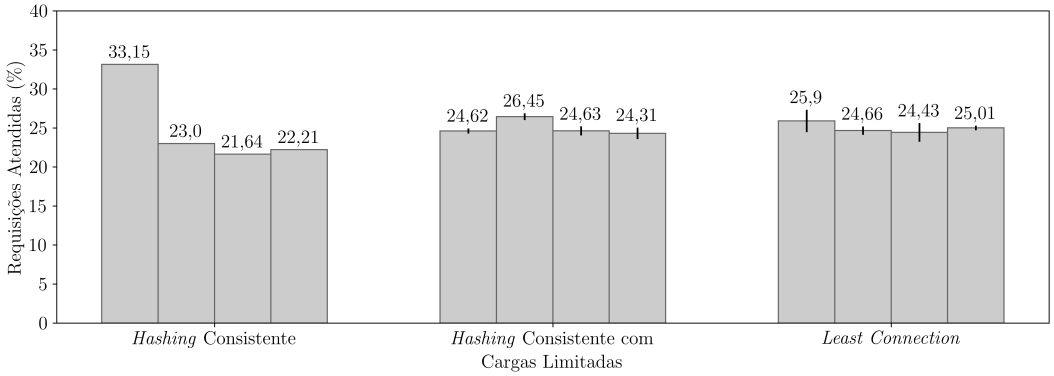
4.3.4 Considerações Gerais

Além das análises individuais de cada cenário, destacamos outros pontos mais gerais:

– Notamos que a taxa de *hits* observada nos nossos experimentos é baixa. Mesmo no melhor caso, observado durante os experimentos com o *hashing*



(a) Eficiência de cache.



(b) Balanceamento de carga.

Figura 4.7: Análise da latência com tráfego intenso.

consistente, a taxa de *hits* média ainda foi inferior a 50%. Acreditamos que isso esteja relacionado com a característica de cauda longa da distribuição de popularidade dos vídeos da nossa amostra. Para verificar, analisamos o arquivo de entrada utilizado nos experimentos e percebemos que cerca de 39,31% dos arquivos são requisitados apenas uma vez. Portanto, mesmo com um espaço de armazenamento ilimitado, a taxa máxima que poderíamos atingir seria algo em torno de 60,69%.

– Nossos experimentos foram realizados com parâmetros de balanceamento escolhidos previamente para cada situação. Em um cenário de distribuição de vídeos real, as variações de demanda são esperadas e ocorrem com frequência. Por isso, o uso de um parâmetro estático pode não ser ideal. Como podemos observar nos experimentos, não há um parâmetro específico que apresente o melhor desempenho em todas as situações.

– De certa forma, os impactos de cache *misses* são atenuados na nossa plataforma. Para isolar os efeitos da coordenação dos caches dentro de um *cluster*, assumimos que a banda entre os Caches e a Origem é abundante e que a Origem responde sempre rapidamente. No entanto, em um cenário real, a latência introduzida pelas idas à origem poderia ser

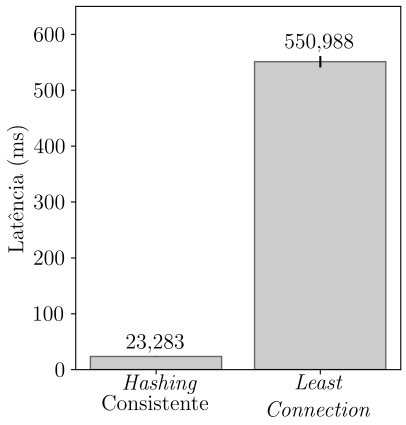


Figura 4.8: Latência com tráfego leve.

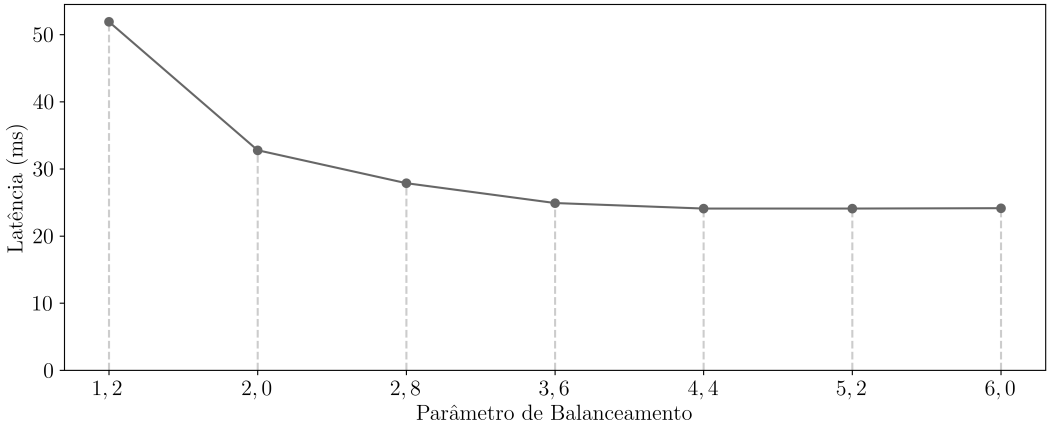
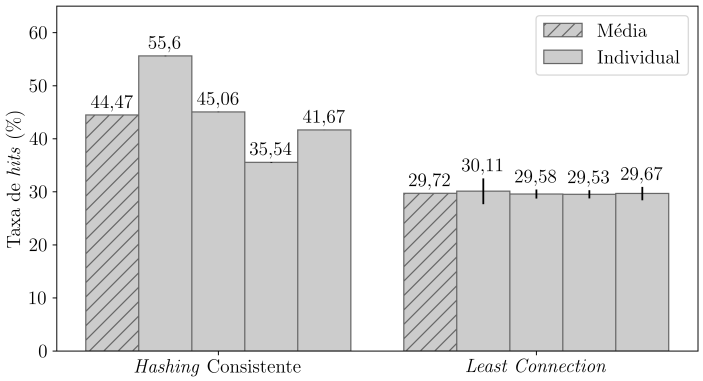
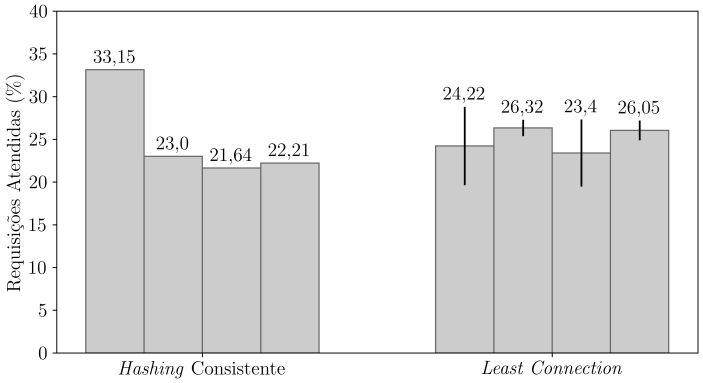


Figura 4.9: Latência do *hashing* consistente com cargas limitadas variando o parâmetro de balanceamento no cenário de tráfego leve.

maior por diversas razões, como a distância entre o *cluster* e a origem, a arquitetura da CDN e a infraestrutura utilizada para servir de origem do conteúdo. Mesmo assim, como a diferença entre a taxa de *hits* do *hashing* consistente com cargas limitadas e do *hashing* consistente convencional foi pequena nos experimentos, acreditamos que isso não deve impactar muito os resultados.



(a) Eficiência de cache.



(b) Balanceamento de carga.

Figura 4.10: Análise da latência com tráfego leve.

5
Conclusão

Neste trabalho, exploramos o uso do *hashing* consistente com cargas limitadas para aprimorar o tempo de resposta de redes de distribuição de conteúdo. Seguindo um modelo de CDN simples, projetamos uma plataforma de experimentação representando os *clusters* de servidores de cache presentes nessas redes, e discutimos como integrar o algoritmo para realizar a coordenação dos servidores em um cenário real. Com a plataforma devidamente montada e configurada, realizamos uma série de experimentos para avaliar o desempenho do *hashing* consistente com cargas limitadas. Esses experimentos consideraram diferentes níveis de tráfego e usaram como base arquivos de *log* de uma CDN de vídeos real.

Os resultados revelaram um grande potencial na utilização do *hashing* consistente com cargas limitadas. Nas situações de tráfego excessivo e intenso, o algoritmo apresenta a menor latência dentre os avaliados, pois é capaz tirar proveito do esquema de *hashing* sem provocar aumentos significativos na sobrecarga no sistema. Em especial, a situação de tráfego intenso mostrou um resultado surpreendente: a latência observada para o *hashing* consistente com cargas limitadas foi cerca de quarenta e sete vezes menor que a observada para o *hashing* consistente convencional, e cerca de dez vezes menor que a observada para o *least connections*. Na situação de tráfego leve, o desbalanceamento provocado pelo *hashing* consistente não provoca sobrecargas no sistema e, portanto, a variação com cargas limitadas não traz nenhum ganho.

Como trabalhos futuros, podemos listar alguns pontos. Como mencionado ao longo do texto, a nossa implementação atual do *hashing* consistente com cargas limitadas permite apenas a utilização de um parâmetro de balanceamento estático, definido no momento de inicialização do servidor. Queremos explorar meios de tornar esse parâmetro dinâmico, respondendo a variações na demanda recebida pelos servidores de cache. Outra possível melhoria da nossa implementação pode ser obtida com a investigação de outras métricas de carga. Utilizamos o número de conexões ativas em cada servidor, mas outras métricas, como o uso de banda, por exemplo, podem trazer resultados ainda mais vantajosos. Por fim, consideramos avaliar o desempenho desse algoritmo em uma CDN real. Apesar de buscarmos realizar os experimentos usando ca-

racterísticas realistas, sabemos que a nossa plataforma não captura diversas complexidades do mundo real. Por isso, acreditamos que a execução com tráfego real por um período mais longo traga intuições ainda mais valiosas.

Referências bibliográficas

BRESLAU, L. et al. Web caching and Zipf-like distributions: evidence and implications. In: ANNUAL JOINT CONFERENCE OF THE IEEE COMPUTER AND COMMUNICATIONS SOCIETIES, 18., 1999, New York, NY, USA. **Proceedings...** IEEE, 1999. v. 1, p. 126–134. ISBN 0-7803-5417-6. ISSN 0743-166X. Disponível em: <http://dx.doi.org/10.1109/INFCOM.1999.749260>.

CHA, M. et al. Analyzing the video popularity characteristics of large-scale user generated content systems. **IEEE/ACM Transactions on Networking**, IEEE, Piscataway, NJ, USA, v. 17, n. 5, p. 1357–1370, out. 2009. ISSN 1063-6692. Disponível em: <http://dx.doi.org/10.1109/TNET.2008.2011358>.

CHANKHUNTHOD, A. et al. A hierarchical Internet object cache. In: USENIX ANNUAL TECHNICAL CONFERENCE, 1996, San Diego, CA, USA. **Proceedings...** Berkeley, CA, USA: USENIX Association, 1996.

CISCO. **Cisco Visual Networking Index: forecast and methodology 2016–2021.** 2017. Disponível em: <https://www.cisco.com/c/dam/en/us-/solutions/collateral/service-provider/visual-networking-index-vni/complete-white-paper-c11-481360.pdf>. Acesso em: 22 jul. 2018.

FAN, L. et al. Summary cache: a scalable wide-area web cache sharing protocol. **IEEE/ACM Transactions on Networking**, IEEE, Piscataway, NJ, USA, v. 8, n. 3, p. 281–293, jun. 2000. ISSN 1063-6692. Disponível em: <http://dx.doi.org/10.1109/90.851975>.

GADDE, S.; RABINOVICH, M.; CHASE, J. Reduce, reuse, recycle: an approach to building large Internet caches. In: WORKSHOP ON HOT TOPICS IN OPERATING SYSTEMS, 6., 1997, Cape Cod, MA, USA. **Proceedings...** 1997. p. 93–98. ISBN 0-8186-7834-8. Disponível em: <http://dx.doi.org/10.1109/HOTOS.1997.595189>.

GILL, P. et al. YouTube traffic characterization: a view from the edge. In: ACM SIGCOMM INTERNET MEASUREMENT CONFERENCE, 7., 2007, San Diego, CA, USA. **Proceedings...** New York, NY, USA: ACM, 2007. p. 15–28. ISBN 978-1-59593-908-1. Disponível em: <http://dx.doi.org/10.1145/1298306.1298310>.

IERUSALIMSCHY, R. **Programming in Lua**. 4. ed. Rio de Janeiro: Lua.org, 2016. ISBN 978-85-903798-6-7.

KARGER, D. et al. Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the World Wide Web. In: ANNUAL ACM SYMPOSIUM ON THEORY OF COMPUTING, 29., 1997, El Paso, TX, USA. **Proceedings...** New York, NY, USA: ACM, 1997. p. 654–663. ISBN 0-89791-888-6. Disponível em: <http://dx.doi.org/10.1145/258533.258660>.

KARGER, D. et al. Web caching with consistent hashing. **Computer Networks**, Elsevier North-Holland, Inc., New York, NY, USA, v. 31, n. 11-16, p. 1203–1213, mai. 1999. ISSN 1389-1286. Disponível em: <http://dx.doi.org/10.1016/S1389-1286(99)00055-9>.

KRISHNAN, S. S.; SITARAMAN, R. K. Video stream quality impacts viewer behavior: inferring causality using quasi-experimental designs. **IEEE/ACM Transactions on Networking**, IEEE, Piscataway, NJ, USA, v. 21, n. 6, p. 2001–2014, dez. 2013. ISSN 1063-6692. Disponível em: <http://dx.doi.org/10.1109/TNET.2013.2281542>.

MIRROKNI, V.; THORUP, M.; ZADIMOGHADDAM, M. Consistent hashing with bounded loads. In: ANNUAL ACM-SIAM SYMPOSIUM ON DISCRETE ALGORITHMS, 29., 2018, New Orleans, LA, USA. **Proceedings...** Philadelphia, PA, USA: SIAM, 2018. p. 587–604. ISBN 978-1-61197-503-1. Disponível em: <http://dx.doi.org/10.1137/1.9781611975031.39>.

MIRROKNI, V. S.; THORUP, M.; ZADIMOGHADDAM, M. Consistent hashing with bounded loads. **Computing Research Repository**, abs/1608.01350, 2016. Disponível em: <http://arxiv.org/abs/1608.01350>.

MOKHTARIAN, K.; JACOBSEN, H.-A. Flexible caching algorithms for video content distribution networks. **IEEE/ACM Transactions on Networking**, IEEE, Piscataway, NJ, USA, v. 25, n. 2, p. 1062–1075, abr. 2017. ISSN 1063-6692. Disponível em: <http://dx.doi.org/10.1109/TNET.2016.2621067>.

NI, J. et al. Hierarchical content routing in large-scale multimedia content delivery network. In: IEEE INTERNATIONAL CONFERENCE ON COMMUNICATIONS, 2003, Anchorage, AK, USA. **Proceedings...** IEEE, 2003. v. 2, p. 854–859. ISBN 0-7803-7802-4. Disponível em: <http://dx.doi.org/10.1109/ICC.2003.1204454>.

PALLIS, G.; VAKALI, A. Insight and perspectives for content delivery networks. **Communications of the ACM**, ACM, New York, NY, USA, v. 49, n. 1, p. 101–106, jan. 2006. ISSN 0001-0782. Disponível em: <http://dx.doi.org/10.1145/1107458.1107462>.

PANTOS, E. R.; MAY, W. RFC, **HTTP Live Streaming**. RFC Editor, 2017. 1–60 p. Internet Requests for Comments. RFC 8216. Disponível em: <https://www.rfc-editor.org/rfc/rfc8216.txt>.

RODLAND, A. **Improving load balancing with a new consistent-hashing algorithm**. 2016. Vimeo Engineering Blog. Disponível em: <https://medium.com/vimeo-engineering-blog/improving-load-balancing-with-a-new-consistent-hashing-algorithm-9f1bd75709ed>. Acesso em: 29 abr. 2018.

SEUFERT, M. et al. A survey on quality of experience of HTTP adaptive streaming. **IEEE Communications Surveys & Tutorials**, IEEE, v. 17, n. 1, p. 469–492, 1. trim. 2015. ISSN 1553-877X. Disponível em: <http://dx.doi.org/10.1109/COMST.2014.2360940>.

THALER, D. G.; RAVISHANKAR, C. V. Using name-based mappings to increase hit rates. **IEEE/ACM Transactions on networking**, IEEE, Piscataway,

NJ, USA, v. 6, n. 1, p. 1–14, fev. 1998. ISSN 1063-6692. Disponível em: <http://dx.doi.org/10.1109/90.663936>.

VALLOPPILLIL, V.; ROSS, K. Internet-Draft, **Cache Array Routing Protocol v1.0**. 1998. Working Draft.

WESSELS, D.; CLAFFY, K. RFC, **Internet Cache Protocol (ICP), version 2**. RFC Editor, 1997. 1–9 p. Internet Requests for Comments. RFC 2186. Disponível em: <https://www.rfc-editor.org/rfc/rfc2186.txt>.

WU, K.-L.; YU, P. S. Replication for load balancing and hot-spot relief on proxy web caches with hash routing. **Distributed and Parallel Databases**, Kluwer Academic Publishers, Hingham, MA, USA, v. 13, n. 2, p. 203–220, mar. 2003. ISSN 0926-8782. Disponível em: <http://dx.doi.org/10.1023/A:1021519509203>.