Universidade Federal do ABC
CMCC - Centro de Matemática, Computação e Cognição

Projeto 3 - Simulação de um servidor proxy

NA2MCTA002-17SA – Algoritmos e Estruturas de Dados II

Prof° Dr. Carlos da Silva dos Santos

Discentes - RA:

Fernando Gabriel Chacon F. T. do Prado - 11201811700
Felipe Oliveira Silva - 11201822479
Felipe de Souza Tiozo - 11201822483
Lucca Ianaguivara Kisanucki - 11201812090

Santo André
2021

1. INTRODUÇÃO

O objetivo dessa simulação é simular um servidor Proxy para um serviço de cache de requisições HTTPs e observar o uso de algoritmos para economizar dados trafegados desnecessariamente. Os servidores Proxy servem como intermediário entre um serviço final (servidor ou banco de dados) e o cliente solicitante. Na imagem abaixo, é possível visualizar um esquema comum de uso de servidor Proxy, onde um utilizador acessa sites da internet via protocolo HTTP ou HTTPs sendo intermediado por um servidor Proxy.

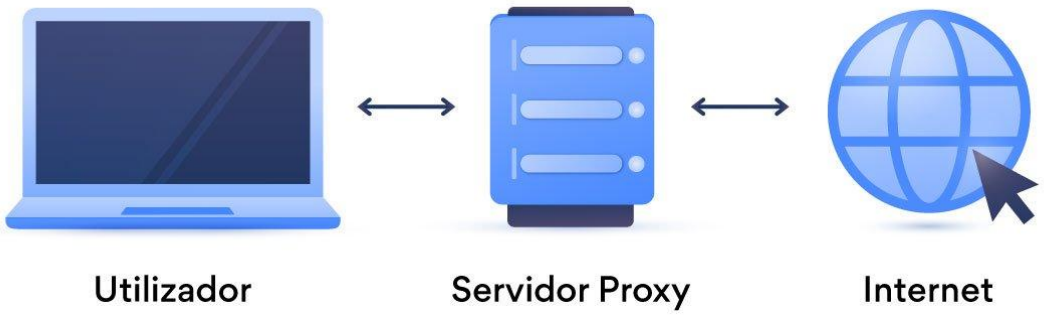

Figura 1 - Esquema de servidor Proxy

Existem três principais aplicações para servidores Proxy: **Controle de acesso/Filtro de conteúdo**, **Cache** e **Firewalls.**

Servidores proxy para controle de acesso são utilizados para bloquear determinados sites, tais como redes sociais, dentro de ambientes corporativos, logo, como toda requisição feita passa por um proxy, fica muito mais fácil para uma grande corporação bloquear acesso à conteúdos em que ele julga não necessários ou maliciosos para seus funcionários. A equipe de Tecnologia da Informação da companhia consegue configurar bloqueios através do IP do usuário ou das credenciais de login.

Firewall é um sistema de segurança utilizado para bloquear acessos inesperados a computadores ou servidores. Profissionais de segurança configuram firewalls para bloquear acesso indesejado às redes que estão tentando proteger, frequentemente como contramedidas anti malware ou anti-invasão. Um servidor proxy entre uma rede confiável e a internet é o local perfeito para implantar um firewall projetado para interceptar e aprovar ou bloquear o tráfego recebido antes que ele chegue à rede.

Por fim, temos servidores proxy para cache. A prática de cache está ligada com otimizações em processos repetitivos com o objetivo de facilitar um acesso à informação eficiente e otimizado. Serviços de cache para servidores HTTP são bem comuns e fundamentais nas aplicações da internet de hoje em dia. O consumo de internet duplicou nos últimos 10 anos no Brasil e atualmente cerca de 74% da população tem acesso à internet, aproximadamente 134 milhões de brasileiros.. Com isso, serviços de redes sociais, streaming e pagamentos foram popularizados com milhares de acessos simultâneos. Outro aparelho que virou fundamental no cotidiano das pessoas foi o smartphone que possui uma série de aplicativos e requer acesso constante à internet. Com isso, o tráfego de dados na internet cresce absurdamente a cada dia e conexões 3G/4G são cada vez mais utilizadas. Para não consumir dados desnecessários e manter uma experiência fluída, os aplicativos implementam camadas de cache tanto em servidores proxy quanto no armazenamento local do dispositivo.

Nesta prática, foi utilizado um arquivo de texto com entradas simulando requisições HTTP e contendo 3 informações: o **instante** (em segundos) em que a requisição foi feita a partir do tempo 0 que é quando o servidor foi iniciado. O segundo campo é o **recurso** que foi solicitado. Nesse caso, os recursos são referenciados ao site Wikipedia em português (pt.wikipedia.org). simulando uma requisição para essa página na web. Por fim, o terceiro campo representa o **tamanho do recurso** retornado do servidor. Nosso objetivo é identificar entradas que são para o mesmo recurso, armazenar isso em um sistema de cache e com isso determinar qual a quantidade de bytes economizados no tráfego utilizando esse sistema.

```
912 Transistor 12528
1004 Nina_Simone 12036
1247 Ciência 22176
```

Figura 2 - Exemplo de entrada de dados

2. MATERIAIS E MÉTODOS

Para simular o servidor de proxy e criar a estrutura de cache escolhemos a linguagem Java. Mais especificamente utilizamos a biblioteca hashMap que é uma estrutura do tipo dicionário que implementa a interface Map<K,V> que nada mais é do que um par key-value.

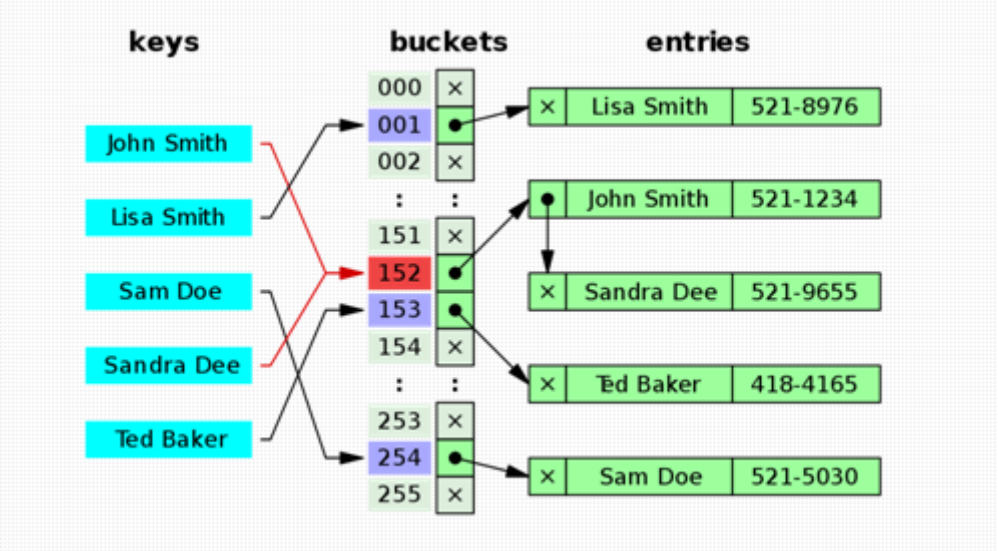

Figura 3 - Exemplo do mecanismo do algoritmo de hash

A própria implementação da interface Map utiliza Generics, desta forma nós podemos especificar tanto o tipo da nossa chave quanto o tipo do nosso valor. Desta forma, nós optamos por mapear uma String contendo o recurso em um valor do tipo Item que nada mais é do que um objeto criado por nós que contém em sua estrutura o instante de tempo em que a requisição é feita e o seu tamanho em bytes.

```java
package proxy;

public class Item {

    private long tempo;
    private long tamanho;

    public Item(long tempo,long bytes){
        this.tempo=tempo;
        this.tamanho=bytes;
    }

    public long getTempo(){
        return this.tempo;
    }
    public long getTamanho(){
        return this.tamanho;
    }

    void alterarTempo(long tempo){
        this.tempo=tempo;
    }

    @Override
    public String toString(){
        return String.format("Tempo: %d, Tamanho: %d", tempo,tamanho);

    }
}
```

Figura 4 - Estrutura mapeada pela String contendo o recurso

Um dos motivos que fizeram com que nós utilizássemos uma função de hash ao invés de uma estrutura de árvore foi a sua complexidade. O hashMap tem complexidade O(1) e caso a função de hash seja muito ruim, no pior dos casos sua complexidade será O(n). Como nosso algoritmo precisa verificar se o recurso já existe a cada requisição, nós consideramos essa a estrutura ideal, visto que também não precisamos realizar operações de balanceamento, fazer rotações ou algo do tipo para realizar a inserção.

No começo nós também tínhamos pensado em usar árvores digitais, porém percebemos que talvez não fosse uma boa ideia, visto que geralmente as URLs dos recursos tem uma quantidade pequena de caracteres e desta forma seu hash é gerado de forma extremamente rápida. Talvez as árvores digitais fossem uma opção mais interessante se a cadeia de caracteres fosse muito maior de forma que gerar seu hash passasse a ser bastante trabalhoso.

Basicamente nosso programa lê um arquivo linha a linha, onde cada linha representa uma requisição para o servidor de proxy. Para cada requisição é verificado se a chave contendo

o recurso já existe no HashMap, esta operação tem complexidade O(1) e O(n) no pior dos casos(já discutido anteriormente). Caso o recurso ainda não exista ele é adicionado ao cache e o tamanho do cache é incrementado de acordo com o tamanho do recurso, desta forma ficando disponível para ser enviado para os cliente em requisições futuras. Caso o recurso já exista, seu valor de tempo contendo o instante da última requisição é alterado para o tempo da nova requisição, e de forma análoga à operação anterior sua complexidade é O(1) e no pior dos casos O(n). A quantidade de bytes economizados também é incrementada com o tamanho do recurso após essa operação.

Ao final de todo o processo é gerado um arquivo CSV no caminho "SEUNOMEDEUSUARIO\Projeto_3\Projeto3.csv", onde há o registro da quantidade de recursos em cache, a quantidade total de requisições e a quantidade total de bytes economizados ao longo de cada requisição.

3. RESULTADOS

Utilizando o arquivo de exemplo disponível no moodle, obtivemos um tamanho final do cache de 7420112651 bytes e um total de 332818743628 bytes economizados, conforme podemos observar na figura abaixo.

```
Quantidade total de páginas em cache: 149792
Quantidade total de requisições feitas: 312646
Tamanho total do cache armazenado: 7420112651
Total de bytes economizados: 332818743628
Arquivo csv exportado com sucesso
Caminho:C:\Users\Casa\Projeto_3\Projeto3.csv
Finalizado em 0,643 segundos
CONSTRUÍDO COM SUCESSO (tempo total: 24 segundos)
```

Figura 5 - Resultado da execução do programa para o arquivo de testes

Conforme também podemos notar a nossa estrutura de cache contava com quase 150 mil páginas adicionadas ao final da execução.

Através dos arquivos csv gerados pudemos fazer algumas constatações interessantes. Por exemplo, traçando a curva Quantidade de recursos armazenados em cache X o total de bytes economizados, percebemos que há uma tendência exponencial, ou seja, quanto mais páginas armazenadas em cache maior é a economia de dados. Isso mostra exatamente o que imaginamos e evidencia a importância de se ter uma camada de cache ao desenvolver uma aplicação.

Figura 6 - Quantidade de recursos em cache X total de bytes economizados

Uma outra observação que notamos foi o fato da quantidade de requisições crescer de forma linear em relação aos bytes totais economizados, conforme observamos na figura abaixo.

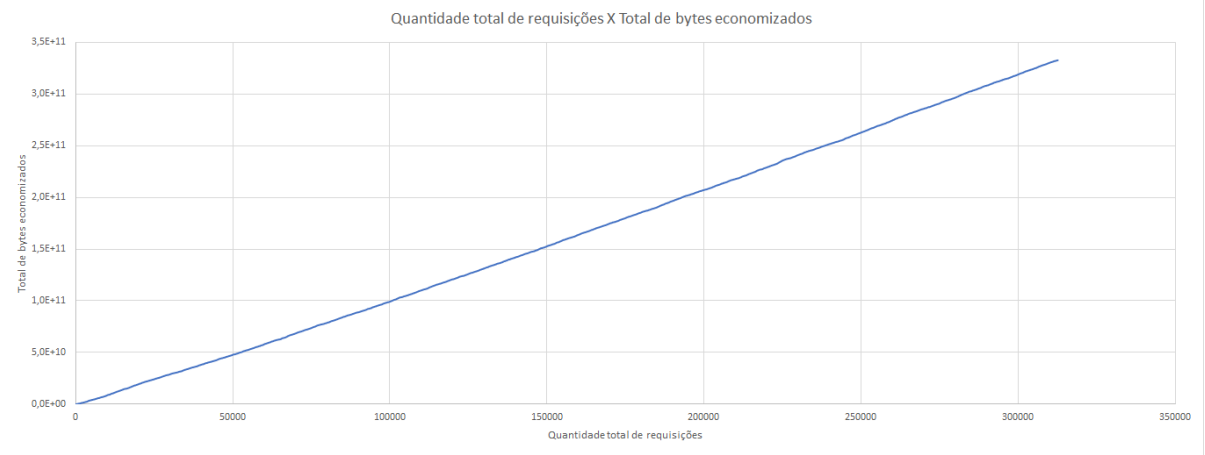

Figura 7 - Quantidade total de requisições X total de bytes economizados

Provavelmente essa linearidade foi causal e pode variar de acordo com os recursos solicitados. Possivelmente se tivéssemos mais arquivos de testes poderíamos verificar o comportamento da curva mais eficientemente.

De qualquer maneira, ficou comprovado através dos dados apresentados a importância de se ter um servidor de proxy para cache de internet, visando a economia de dados.