UNIVERSIDADE FEDERAL DE PERNAMBUCO
CENTRO DE INFORMÁTICA
GRADUAÇÃO EM ENGENHARIA DA COMPUTAÇÃO

Gabriel Silva de Oliveira

Cache Adaptativa para o gRPC usando Teoria de Controle

Recife

2026

Gabriel Silva de Oliveira

Cache Adaptativa para o gRPC usando Teoria de Controle

Monografia apresentada na Graduação em Engenharia da Computação do Centro de Informática da Universidade Federal de Pernambuco, como requisito parcial para a obtenção do título de Bacharel em Engenharia da Computação.

Orientador (a): Nelson Souto Rosa

Recife

2026

Ficha de identificação da obra elaborada pelo autor,
através do programa de geração automática do SIB/UFPE

Oliveira, Gabriel Silva de.
Cache Adaptativa para o gRPC usando Teoria de Controle / Gabriel Silva de Oliveira. - Recife, 2026.
38 : il., tab.

Orientador(a): Nelson Souto Rosa
Trabalho de Conclusão de Curso (Graduação) - Universidade Federal de Pernambuco, Centro de Informática, Engenharia da Computação - Bacharelado, 2026.
10.
Inclui referências.

1. Memória Cache. 2. Teoria de Controle. 3. Sistemas Adapatativos. 4. gRPC. I. Rosa, Nelson Souto. (Orientação). II. Título.

000 CDD (22.ed.)

GABRIEL SILVA DE OLIVEIRA

Cache Adaptativa para o gRPC usando Teoria de Controle

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Engenharia da Computação da Universidade Federal de Pernambuco, como requisito parcial para obtenção do título de bacharel em Engenharia da Computação.

Aprovado em: 26/01/2021

BANCA EXAMINADORA

_____

Prof. Dr. Nelson Souto Rosa (Orientadora)

Universidade Federal de Pernambuco

_____

Prof. Dr. Carlos Ferraz (Examinador Interno)

Universidade Federal de Pernambuco

_____

AGRADECIMENTOS

Gostaria de agradecer primeiramente aos meus pais, Simone Rosa da Silva e Vanildo Souza de Oliveira, por serem meus maiores incentivadores na minha vida, que sempre acreditaram em mim e me ensinaram o valor da educação desde cedo, e como professores sempre foram exemplo dentro de casa. Agradeço também ao meu irmão Vinicius Silva de Oliveira que ao longo dos anos também foi uma grande referência e incentivador no campo do estudo.

Gostaria de agradecer também aos meus colegas de turma que me acompanharam durante a faculdade e partilham dessa conquista, em especial a Pedro Vitor e Rafinha por termos passado por tantas matérias juntos.

Agradeço Também aos meus professores do centro de informática por todo conhecimento adquirido e oportunidades abertas, e por termos um sistema de ensino superior público gratuito e de qualidade que transforma a vida de tantas pessoas.

RESUMO

Memória cache é essencial e amplamente utilizada com o intuito de melhorar o desempenho de aplicações. Contudo, gerenciar tamanho da cache em ambientes que sofrem variações de carga ainda é um desafio. Apesar de existirem diversos algoritmos para melhorar a eficiência da cache, a maioria destes algoritmos são baseados em configurações estáticas ou heurísticas que não conseguem se adaptar rápido o bastante a picos de tráfego. Nesse contexto, soluções clássicas podem levar a desperdícidio de recursos ou instabilidade do sistema por não conseguirem se ajustar a tempo. Este trabalho propõe uma cache adaptativa, para sistemas distríbuidos que utilizam o gRPC, baseada em Teoria de Controle. A solução proposta foi implementada como um sistema de malha fechada que monitora o erro entre a taxa de acerto da cache e a meta (*setpoint*) que se deseja atingir. O ajuste no tamanho da cache é baseado na amplitude deste erro. Essa abordagem tem como objetivo assegurar a estabilidade do sistema, permitindo que ele se mantenha eficiente mesmo em condições de carga variável, minimizando chamadas redundantes e otimizando recursos. Neste trabalho foram encontrados resultaods promissores ao realizadar simulações para observar a relação da taxa de acerto e tamanho da memória cache utilizando a abordagem da teoria de controle.

Palavras-chaves: Memória Cache, Teoria de Controle, Sistemas Adapatativos, gRPC

ABSTRACT

Cache memory is essential and widely used to improve applications performance. However, managing cache size in environments that experience load variations is still a challenge. Although several algorithms exist to improve cache efficiency, most of these algorithms are based on static or heuristic configurations that cannot adapt quickly enough to traffic spikes. In this context, classic solutions can lead to wasted resources or system instability because they cannot adjust in time. This work proposes an adaptive cache for distributed systems using gRPC, based on Control Theory. The proposed solution was implemented as a closed-loop system that monitors the error between the cache hit rate and the target (setpoint) to be achieved. The adjustment in cache size is based on the amplitude of this error. This approach aims to ensure system stability, allowing it to remain efficient even under variable load conditions, minimizing redundant calls and optimizing resources.This work yielded promising results by conducting simulations to observe the relationship between hit rate and cache memory size using a control theory approach.

Keywords: Cache Memory, Control Theory, Adaptive Systems, gRPC

LISTA DE FIGURAS

LISTA DE CÓDIGOS

LISTA DE TABELAS

SUMÁRIO

1 INTRODUÇÃO

Este capítulo apresenta inicialmente o contexto e a motivação para este trabalho. Em seguida, ele apresenta o problema e soluções existentes para resolvê-lo. Finalmente, é apresentada uma visão geral da solução proposta e a estrutura do trabalho.

1.1 CONTEXTUALIZACAO E MOTIVAÇÃO

Com um mundo altamente conectado e digital, as aplicações atuais enfrentam padrões de tráfego voláteis e imprevisíveis, o que torna o dimensionamento de recursos essencial para enfrentar esses cenários a fim de não saturar a aplicação e nem desperdiçar recursos. Como exemplo, temos a memória cache. Caso ela seja subdimensionada pode acarretar em uma baixa taxa de acerto, o que prejudicaria o desempenho do sistema por aumentar sua latência. Por outro lado, o superdimensionamento da cache resultará em um desperdício de memória RAM.

A evolução dos sistemas distribuídos e a adoção crescente de microserviços e várias tecnologias como middlewares (TANENBAUM; STEEN, 2023) tem levado à comunicação entre serviços ser cada vez mais eficiente.O gRPC (Google Remote Procedure Call) é um middleware muito utilizado para intermediar conexões entre serviços. Contudo, ele possui algumas limitações como não possuir um suporte nativo a cache, que é de extrema importância para serviços de alto desempenho executarem mais rapidamente (PASCHOS et al., 2018). Além disso, existem diversas estratégias de cache muito populares (MAYER; RICHARDS, 2025) que utilizam parâmentros estáticos como a política de substituição e tamanho da cache. Contudo, abordagens dinâmicas ainda são escassas.

A Teoria de Controle (NISE, 2012) é amplamente usada em sistema físicos, nas mais diversas áreas da engenharia tal como em aeronaves, robôs industriais, circuitos elétricos, entre outras muitas aplicações. Contudo, na engenharia de software, a teoria de controle ainda não é uma abordagem amplamente utilizada, baseando-se no fato de que por sistemas de softwares são discretos e lógicos e não se comportam como plantas físicas contínuas (JANERT, 2013). Porém, com o aumento da complexidade de sistemas distribuídos, a abordagem manual vai se mostrando cada vez mais insustetável e surge uma busca pela automação. Diante desse contexto, a motivação desse trabalho é implementar um sistema de cache adaptativa para chamadas gRPC que adapte o tamanho da cache de acordo com a variação de tráfego da

aplicação.

1.2 PROBLEMA

O problema central do trabalho é conciliar o uso eficiente da memória cache sem comprometer o desempenho de uma aplicação em sistemas distribuídos submetidos a tráfegos variáveis.

Em várias aplicações de software a latência é um requisito fundamental, que usa da cache como uma solução para reduzir o tempo de chamadas. Porém, a memória RAM é um recurso finito e mais caro que memória em disco, e com isso temos um *trade-off* em relação ao seu dimensionamento:

Subdimensionamento: Uma cache configurada com tamanho insuficiente para a demanda irá acarretar em uma baixa de taxa de acerto, fazendo com que e aplicação tenha que realizar novas chamadas em rede ou consulta a banco de dados, assim degradando o tempo de resposta.

Superdimensionamento: Por outro lado, uma cache maior do que a necessária (superdimensionada) pode ocasionar desperdício de recursos computacionais e, consequentemente, recursos financeiros.

1.3 ESTRATÉGIAS EXISTENTES

As abordagens usadas para dimensionamento de recursos normalmente seguem uma estratégia estática, provisionando os recursos manualmente ou de forma ad-hoc (MENASCÉ; ALMEIDA, 2001). Contudo, com o uso crescente de sistemas em nuvem, existem algumas abordagens para lidar com o ambiente dinâmico das aplicações, como exemplo, o mecanismo de autoscaling de memória cache da Amazon (Amazon Web Services, 2025). Arman et al. (2012) utilizam o sistema de feedback para escalar horizontalmente armazenamentos chave- valor e reforçam como a teoria do controle pode ser usada em sistemas computacionais.

Entre as abordagens existentes, a que mais se aproxima do trabalho aqui apresentado, é o trabalho de (FAROKHI et al., 2015) que propõem uma uma adaptação vertical da memória baseada na perfomance da aplicação utilizando teoria do controle. Nesse trabalho, os autores reduziram em entre 47% e 57% o consumo de memória quando comparado com a abordagem

sem o redimensionamento de recursos, ficando em evidência a importância da adaptatividade da memória cache. Os autores usaram o tempo de resposta como setpoint (meta) do sistema de controle.

1.4 SOLUÇÃO PROPOSTA

Tendo em vista o fato de que estratégias de cache estáticas não reagem a mudanças de demanda do servidor e com o uso cada vez maior de sistemas distribuídos em nuvem, esse trabalho propõe a implementação, de uma sistema de cache adaptativa. Tendo como base o trabalho de (FAROKHI et al., 2015), que teve uma enfoque maior no desempenho da aplicação, este trablho foca na utilização de memória. Desse modo foi utilizada a taxa de acerto da cache como critério de tomada de decisão, e as análises foram focadas em o quanto a memória alocada está sendo usada em relação ao tamanho teórico ideal.

1.5 ESTRUTURA

O restante deste trabalho está organizado em 5 capítulos. O Capítulo 2 apresenta as bases teóricas necessárias para a compreensão dos experimentos. No Capítulo 3 será apresentada a solução proposta, descrevendo tecnologias utilizadas, configurações de experimentos e como os resultados foram obtidos. Já no Capítulo 4 são apresentados os resultados dos experimentos e suas análises. Por fim o Capítulo 5 apresenta as conclusões e propostas de trabalhos futuros.

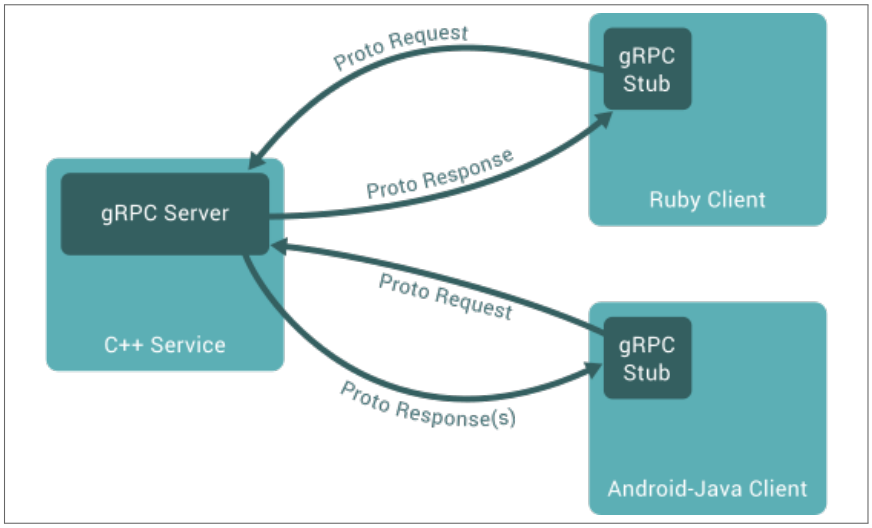
2 FUNDAMENTAÇÃO TEÓRICA

Este capítulo introduz a base teórica necessária para a compreensão do trabalho. Primeiramente, é feita uma apresentação do gRPC; logo após, são apresentados os conceitos de interceptador, cache e teoria do controle, que são os objetos centrais deste estudo.

2.1 GRPC

O gRPC (GOOGLE, 2025) é um *middleware open-source* desenvolvido pela Google que atua na comunicação entre serviços. Ele é utilizado para realizar chamadas de procedimento remoto (RPC), isto é, permite que um cliente realize chamadas para um servidor que está em uma máquina diferente como se estivesse fazendo uma chamada local. Servidores e clientes gRPC podem se comunicar em qualquer linguagem de programação que seja suportada pelo gRPC; como podemos ver no exemplo da Figura 1, um servidor em C++ pode ter um cliente em Ruby e outro em Java. Isso é possível pois o gRPC utiliza *protocol buffers* (GOOGLE, 2024) como IDL (*Interface Definition Language*), uma linguagem para definir as estruturas de dados e serviços. Uma vez que definimos essas estruturas, usamos o compilador de *protocol buffer* para gerar classes de acesso na linguagem a ser utilizada, chamadas de *stub* no cliente e *gRPC server* no servidor, que fazem a codificação e decodificação de requisições.

Figura 1 – Funcionamento do gRPC

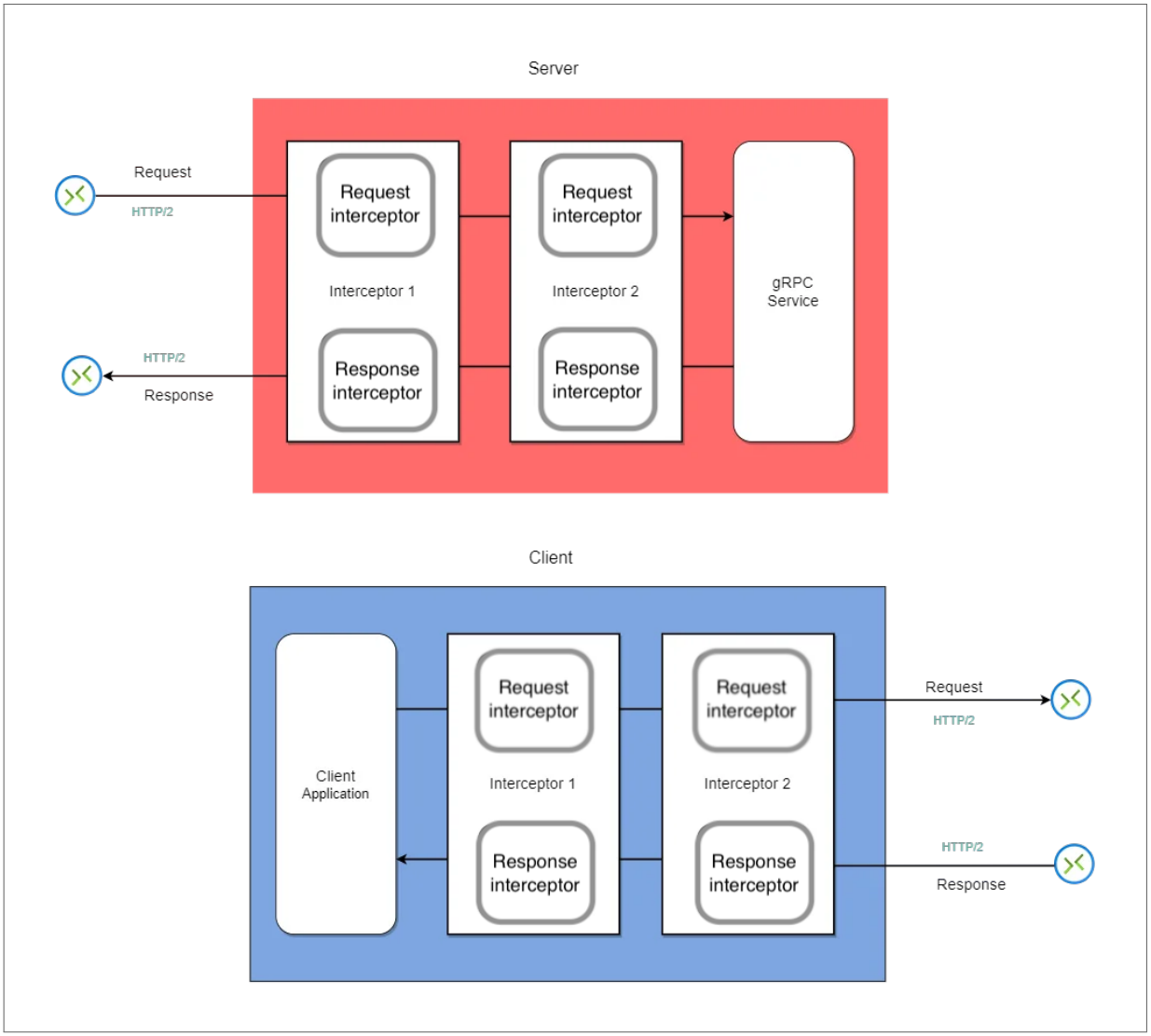


Fonte: (GOOGLE, 2025)

2.1.1 Interceptadores

Existem lógicas que precisam ser implementadas em todos os métodos da RPC; para esses casos, o gRPC possui uma abstração chamada interceptador. Ela é utilizada para interceptar requisições e respostas do servidor ou do cliente, como podemos ver na Figura 2. É possível utilizar um interceptador como *logger*, cache, autenticador, entre muitas outras funcionalidades. Neste trabalho, utilizaremos o interceptador para realizar a implementação de cache em requisições que chegam ao servidor.

Figura 2 – Funcionamento de um interceptador



Fonte: (KUMAR, 2022)

2.2 MEMÓRIA CACHE

A cache é uma memória volátil de acesso rápido para armazenar dados, de forma que uma aplicação não precise acessar fontes de dados mais lentas, como um banco de dados, ou aplicar algum processamento custoso. Dentre suas aplicações, a cache ajuda a diminuir a latência de requisições recorrentes. É importante ressaltar que existem diversas estratégias de cache (MAYER; RICHARDS, 2025) e vários parâmetros que podem ser calibrados em um algoritmo de cache.

2.2.1 Cache Estática

A cache estática, como o próprio nome dugere, possui apenas parâmetros fixos que não se alteram de acordo com a demanda. Como exemplo de parâmetros, temos o tamanho total da cache e as políticas de substituição, que são definidos previamente à sua execução. Como lado positivo dessa abordagem, temos a facilidade de implementação e uma previsibilidade maior, facilitando os testes, entre outros pontos. Contudo, como lado negativo, pode ocorrer a ociosidade de recursos e queda de desempenho no sistema.

2.2.2 Cache Adaptativa

Muitas vezes, os sistemas não possuem uma demanda constante, como pode ser observado em (GMACH et al., 2007). Na realidade, encontramos vários sistemas que possuem horários de pico e recursos que são populares para vários usuários. Para lidar com essa dinâmica e evitar a ociosidade, a cache adaptativa se mostra uma abordagem eficaz, na qual os parâmetros se ajustam continuamente de acordo com a demanda e métricas observadas ao longo do tempo, como taxa de acerto e volume de requisições. A partir dessas métricas, podemos alterar parâmetros como o tamanho da cache para melhor acomodar a demanda. Para fazer essa adaptação, podemos utilizar diversas estratégias, como heurísticas, inteligência artificial, análises estatísticas e teoria do controle. Neste trabalho, focaremos na abordagem utilizando teoria do controle como algoritmo de adaptação.

2.3 TEORIA DE CONTROLE

Na engenharia, a teoria de controle é amplamente utilizada para regular sistemas a partir de observações de sinais de entrada e saída, e comparação com um valor de referência. Um sistema de controle é geralmente composto por quatro elementos principais:

Referência (*setpoint*): Valor desejado para a saída do sistema (por exemplo, manter a taxa de acerto em 90%).

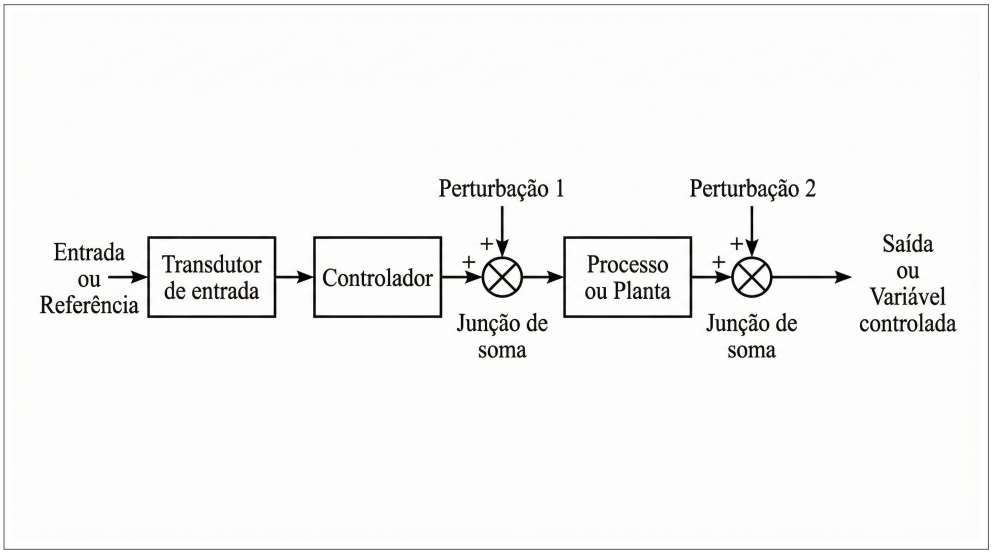
Sensor/Monitor: Componente responsável por medir a saída real do sistema, como o monitoramento da taxa de acerto.

Controlador: Calcula o erro entre o valor desejado e o valor real, gerando uma ação corretiva.

Planta: Parte do sistema que é influenciada pela ação do controlador, como a cache.

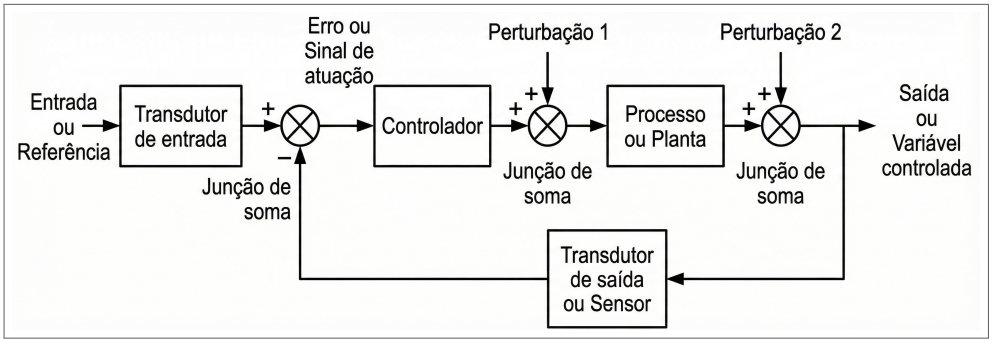
Existem duas principais abordagens para a arquitetura do sistema de controle: malha aberta (Figura 3) e malha fechada (Figura 4). A principal diferença entre os sistemas de malha aberta e fechada é que o controlador de malha fechada utiliza a saída como parâmetro da ação de controle — também conhecido como controle de *feedback* —, sendo este o tipo de sistema utilizado neste trabalho.

Figura 3 – Sistema de malha aberta



Fonte: (NISE, 2012)

Figura 4 – Sistema de malha fechada



Fonte: (NISE, 2012)

2.3.1 Controlador PID

Como mencionado previamente, o controlador é responsável por calcular o erro entre o valor desejado e o valor real de saída e, a partir disso, gerar uma ação corretiva. No caso da memória cache, o controlador poderia aumentar o tamanho da cache para elevar o *hit-rate*. Os controladores servem para calcular essa ação corretiva com base em cálculos matemáticos. Embora existam diversos tipos, neste projeto foi estudado o controlador PID (Proporcional Integral Derivativo), amplamente utilizado na engenharia. Esse controlador, apresentado matematicamente pela Equação 2.1, utiliza o erro corrente, o histórico acumulado de erros e a taxa de variação do erro.

$$u(t) = K_p e(t) + K_i \int_0^t e(\tau)\,d\tau + K_d \frac{de(t)}{dt} \tag{2.1}$$

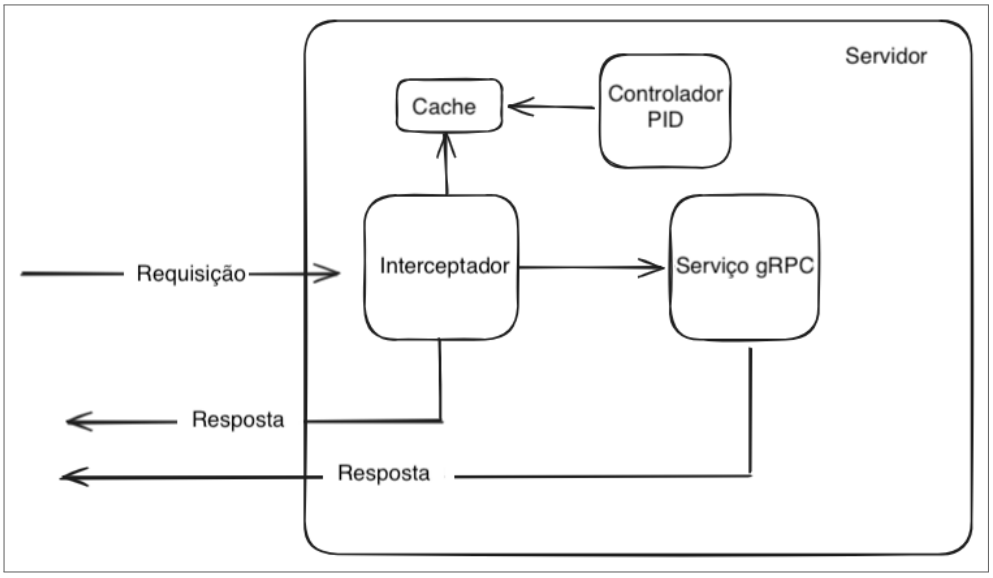
3 CACHE ADAPTATIVA

Neste capítulo, apresentaremos a solução proposta. Começaremos pela arquitetura, fornecendo uma visão macro e serão apresentados detalhes de cada componente.

3.1 ARQUITETURA

A arquitetura do projeto é apresetada na Figura 5. Primeiramente, as requisições chegam ao servidor; então, o interceptador da requisição é responsável por verificar se a resposta está na memória cache. Em caso positivo, retornamos a resposta; caso contrário, devemos fazer uma chamada para o serviço, que naturalmente é mais custosa do que retornar algo armazenado na cache.

Figura 5 – Arquitetura do servidor



Fonte: Elaborado pelo autor

O servidor foi desenvolvidos na linguagem de programação Go (GOLANG, 2024), por ser uma linguagem amplamente utilizada em sistemas distribuídos. Foi utilizado o gRPC como *middleware* e *protocol buffers* para definir as estruturas de dados e serviços; para o banco de dados, utilizou-se o MySQL (ORACLE CORPORATION, 2023).

3.2 INTERCEPTADOR

O interceptador é responsável por processar todas as requisições que chegam ao servidor. Como podemos ver no Código Fonte 1, a Linha 16 acessa a cache; a Linha 18 verifica se há um valor para a respectiva chave e, em caso positivo, o valor já é retornado na resposta. Caso contrário, o fluxo prossegue para o serviço gRPC na Linha 27.

Código Fonte 1 – Interceptador

```
1  func CacheInterceptor(lruCache *LRUCache) grpc.UnaryServerInterceptor {
2      return func(
3          ctx context.Context,
4          req interface{},
5          info *grpc.UnaryServerInfo,
6          handler grpc.UnaryHandler,
7      ) (interface{}, error) {
8          totalRequests = totalRequests + 1
9          // Type assert the request
10         r, ok := req.(*pb.Request)
11         if !ok {
12             return nil, status.Errorf(codes.Internal, "invalid request type")
13         }
14
15         var clientId = int(r.GetClientId())
16         var cachedValue = cache.Get(clientId)
17
18         if cachedValue != "not found" {
19             cacheHits = cacheHits + 1
20             address := cachedValue
21             response := &pb.Response{
22                 ClientAddress: address,
23             }
24             return response, nil
25         }
26
27         resp, err := handler(ctx, req)
28         if err != nil {
29             return nil, err
30         }
31
32         return resp, nil
33     }
34 }
```

3.3 CACHE ADAPTATIVA

Para a cache, como podemos ver no Código Fonte 2, criamos uma estrutura que consiste em um *hashmap* para armazenar e acessar os valores rapidamente, uma lista com a ordem de acesso das chaves e, por fim, o tamanho da cache.

Código Fonte 2 – Definição do cache

```
1 type LRUCache struct {
2     capacity int
3     cache    map[int]*list.Element
4     order    *list.List
5 }
```

No Código Fonte 3, podemos observar como funciona a inserção de novos valores. Primeiramente, verificamos na Linha 2 se o elemento já está na cache; caso positivo, apenas alteramos sua posição para o início da lista, para refletir o uso recente da chave, e a função é encerrada. Já na Linha 8, verificamos se o tamanho máximo foi atingido; caso positivo, a cache utiliza o algoritmo LRU (*Least Recently Used*), que consiste em retirar o último elemento da lista e apagá-lo do *hashmap*. Então, o novo elemento é adicionado, como mostrado nas Linhas 14 a 16.

Código Fonte 3 – Função de inserção de novo elemento na cache

```
1  func (lru *LRUCache) Put(key int, value string) {
2      if element, found := lru.cache[key]; found {
3          element.Value.(*CacheItem).value = value
4          lru.order.MoveToFront(element)
5          return
6      }
7
8      if lru.order.Len() == lru.capacity {
9          backElement := lru.order.Back()
10         lru.order.Remove(backElement)
11         delete(lru.cache, backElement.Value.(*CacheItem).key)
12     }
13
14     item := &CacheItem{key: key, value: value}
15     element := lru.order.PushFront(item)
16     lru.cache[key] = element
17 }
```

3.4 SISTEMA DE CONTROLE

3.4.1 Definição da Planta

Na abordagem da teoria de controle, a planta trata-se do elemento a ser controlado. Em nosso contexto, o foco é o uso eficiente da cache; portanto, nosso elemento controlado será a própria cache.

3.4.2 Setpoint

Uma das métricas mais importantes para avaliar um algoritmo de cache é a sua taxa de acerto. Dessa forma, utilizaremos o valor de 90% como nosso *setpoint*, isto é, o valor a ser atingido. É importante observar que não é viável definir um *setpoint* de 100%, visto que isso implicaria em uma taxa de variação crescente e, portanto, o sistema nunca convergiria.

3.4.3 Controlador PID

O controlador teve suas constantes (Kp, Ki e Kd), como mostrado na Equação 1, calculadas através de um *tuning* empírico, que consiste em ajustar os parâmetros de acordo com a execução de experimentos e selecioná-los conforme os melhores resultados. Para isto, foram observados o *overshoot*, a oscilação e o tempo de estabilização. O Código Fonte mostra a implementação do controlador, que basicamente implementa a equação 1. Note que também foi adicionado uma lógica de *clamping*, que se trata de definir limites para o erro integal, para evitar um possível fenômeno do termo integral crescer indefinidamente conhecido como *windup* (ÅSTRÖM; HÄGGLUND, 2006).

Código Fonte 4 – Definição do controlador

```
1  func (pid *PIDController) calculate(currentError float64) float64 {
2      now := time.Now()
3      dt := now.Sub(pid.lastControllerTime).Seconds()
4      pid.lastControllerTime = now
5
6      if dt <= 0 {
7          pid.previousError = currentError
8          pid.integralError = 0
9          return pid.kp * currentError
10     }
```

```
11
12      proportionalTerm := pid.kp * currentError
13
14      pid.integralError += currentError * dt
15
16      pid.integralError += currentError * dt
17      if pid.integralError > 1.0 {
18          pid.integralError = 1.0
19      } else if pid.integralError < -1.0 {
20          pid.integralError = -1.0
21      }
22
23      integralTerm := pid.ki * pid.integralError
24
25      derivativeTerm := pid.kd * (currentError - pid.previousError) / dt
26      pid.previousError = currentError
27
28      output := proportionalTerm + integralTerm + derivativeTerm
29
30      return output
31  }
```

O código fonte 5 mostra como é feita a atualização do tamanho da cache. A entrada do controlador é erro, isto é, a diferença entre a nossa taxa de acerto de referência (*setpoint*) e a taxa de acerto atual. A taxa de acerto corrente é calculada a cada período de amostragem, que corresponde ao intervalo de tempo entre duas invocações consecutivas do controlador. Após isso o erro é passado como entrada para o controlador e a partir disso recebemos o fator de correção, note que como taxa de acerto é adimensional e normalizada temos de multiplicar o fator de correção pelo tamanho da cache atual para obtermos o ajuste na escala correta e por fim calcular o novo tamanho da cache. Também foi adicionado um limite inferior como uma trava de segurança para não obtermos valores negativos de cache, o que não faria sentido fisicamente.

Código Fonte 5 – Atualização do tamanho da cache

```
1
2       errorHitRate := setpoint - currentHitRate
3
4       controlOutput := pid.calculate(errorHitRate)
5
6       adjustment := int(controlOutput * float64(lruCache.capacity))
7
8       var newCacheSize = lruCache.capacity + adjustment
9
```

```
10      if newCacheSize < 10 {
11          newCacheSize = 10
12      }
13
14      lruCache.SetCacheSize(newCacheSize)
```

4 AVALIAÇÃO DE DESEMPENHO

Este capítulo apresenta uma avaliação de desempenho da cache adaptativa proposta e, para isso, segue a metodologia de Jain (JAIN, 1991). Inicialmente, aborda-se a definição do sistema, seguida pela escolha das métricas que guiarão a comparação de desempenho, a definição da carga à qual o sistema será submetido e, por fim, a análise dos resultados.

4.1 DEFINIÇÃO DO SISTEMA

O sistema analisado é a cache adaptativa implementada e utilizada pelo serviço gRPC. A cache recebe como entrada o padrão de requisições e a taxa de acertos, processa tais informações por meio de um controlador (PID) e produz, como saída, o ajuste dinâmico do tamanho da cache.

4.2 MÉTRICAS AVALIADAS

A primeira métrica a ser avaliada é a taxa de acerto da cache, fundamental para determinar o grau de utilização da memória. Ela é calculada pela razão entre o número de requisições atendidas pela memória cache e o número total de requisições do sistema no intervalo de ajuste do controlador. Note que, para uma avaliação mais precisa, essa métrica deve ser analisada em conjunto com o tamanho da cache.

A importância de monitorar o tamanho da cache reside na premissa de que, hipoteticamente, uma memória suficientemente grande resultaria em uma taxa de acertos altíssima. No entanto, tal configuração não seria necessariamente eficiente, pois consumiria recursos excessivos e manteria memória ociosa.

Nesta avaliação, considera-se a métrica de GR (*Goal Range*), com base na pesquisa de (ROSA; CAVALCANTI, 2024). O GR representa o intervalo aceitável da métrica avaliada, refletindo a estabilidade do controlador. Neste trabalho, utiliza-se um intervalo de 90% a 110% do *setpoint*.

4.3 PARÂMETROS

Os parâmetros referem-se às características do ambiente que afetam as métricas, sendo fundamentais para determinar as condições sob as quais a análise será realizada. Eles podem ser divididos em duas categorias: parâmetros do sistema e parâmetros de carga. Os parâmetros do sistema referem-se a atributos internos, como o *hardware* no qual o serviço está sendo executado; por outro lado, os parâmetros de carga correspondem à forma como o sistema é estimulado, como, por exemplo, o número de requisições por segundo.

Os parâmetros foram especificados na Tabela 1. Os experimentos foram executados localmente em uma máquina com as configurações detalhadas, a política de cache foi fixada como LRU e a implementação do controlador também foi mantida constante. Para os parâmetros de carga, fixou-se o valor de 100 requisições por segundo para simular um ambiente de alta demanda. Todos os experimentos tiveram duração de 20 minutos, e o grupo de requisições mais frequentes foi alterado a cada 5 minutos para observar a capacidade de adaptação do sistema.

Para fins de simplificação, denominamos o grupo de requisições mais frequentes como *top-k*. Note que o *setpoint* foi definido como 90% para ter uma margem de 5 pontos percentuais abaixo da cobertura percentual de *top-k* para respeitar os limites estatísticos do sistema, visto que o desempenho máximo teórico da cache corresponderia à própria porcentagem da cobertura do *top-k*.

Tabela 1 – Parâmetros

Categoria	**Parâmetro**	**Descrição**
sistema	Máquina	Processador: i5-1235u SO: Ubuntu 20.04 Ram: 16gb SSD: 512gb
sistema	Política de cache	LRU
sistema	Implementação do controlador	PID
sistema	Kp	0.25
sistema	Ki	0.1
sistema	Kd	0.05
carga	Requisições por segundo	100
carga	duração dos experimentos	20 minutos
carga	Intervalo de mudança *top-k*	5 minutos
carga	Cobertura percentual *top-k*	95%
carga	Setpoint	90%

4.4 FATORES

Os fatores são os parâmetros alterados nos experimentos com o objetivo de mensurar o impacto destes parâmetros sobre as métricas durante os experimentos. Foram escolhidos como fatores o intervalo de tempo de ajuste do controlador e o tamanho absoluto do grupo *top-k*, visando analisar o comportamento do sistema diante de pequenas e grandes mudanças no número de usuários mais frequentes.

Fatores	**Valores**
Intervalo de ajuste do controlador	10 e 30 segundos
Variação do tamanho top-k	50 e 1000

Tabela 2 – Fatores

4.5 CARGA DE TRABALHO

Para gerar a carga, foi utilizada a ferramenta *Locust*, que permite um alto controle sobre as requisições do sistema e a personalização das simulações, tais como a definição da porcentagem de requisições mais frequentes e o intervalo de tempo entre elas. Neste trabalho, ajustamos

as requisições do *top-k* de acordo com a tabela de fatores, e foram realizadas chamadas para um único método do serviço gRPC.

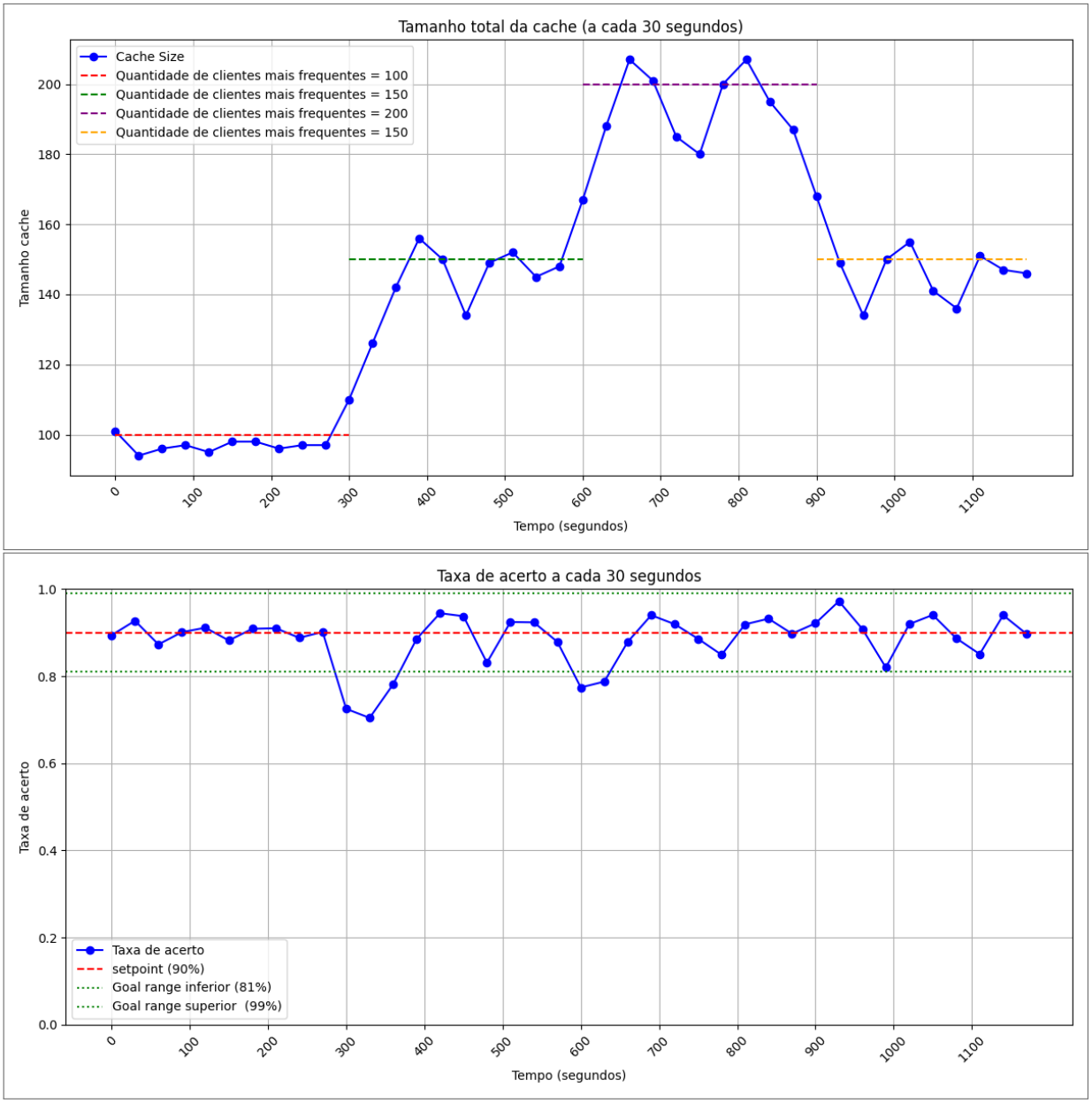
4.6 RESULTADOS

Os resultados serão apresentados dentro do universo de cada cobertura percentual de *top-k*. A apresentação será feita graficamente e, em seguida, os resultados serão discutidos com base nos dados e na variação dos fatores.

4.6.1 Cobertura percentual *top-k* de 95%, variação de 50 no tamanho do *top-k* e intervalo de 30 segundos

A Figura 6 mostra que o tamanho da cache ficou aproximadamente igual ao tamanho do *top-k* (indicado pela linha pontilhada no gráfico). Isso mostra que o sistema otimizou o tamanho da cache sem que houvesse grande ociosidade de memória. É importante notar que, além de manter o tamanho da cache condizente com o *top-k*, a taxa de acerto permaneceu, na maior parte do tempo, dentro do *goal range*. É importante ressaltar que os momentos em que ocorre um maior desvio do *goal range* são as transições de tamanho do *top-k*, o que é esperado, visto que esses novos valores ainda não haviam sido incorporados à memória cache.

Figura 6 – Variação de *top-k*= 50 e intervalo de 30s



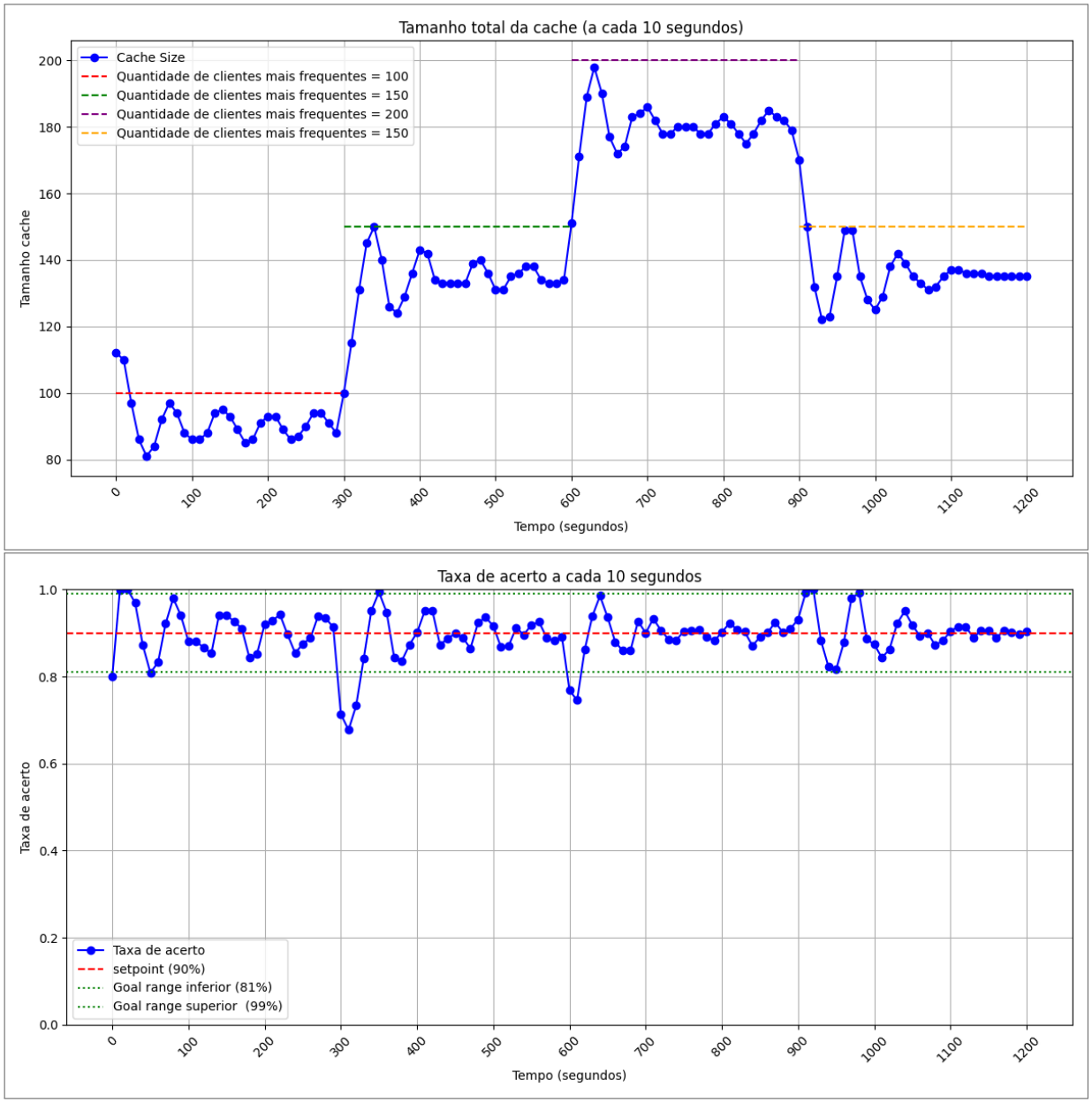
Fonte: Elaborado pelo autor

4.6.2 Cobertura percentual *top-k* de 95%, variação de 50 no tamanho do *top-k* e intervalo de 10 segundos

Neste experimento, reduzimos o intervalo de 30 para 10 segundos. Podemos perceber, pela Figura 7, que o tamanho da cache se manteve abaixo do tamanho do *top-k*. Isso ocorre porque, com um intervalo de ajuste menor e um tamanho de *top-k* reduzido, a distribuição das requisições consegue ser reproduzida dentro desses 10 segundos. Como o *setpoint* é definido 5 pontos percentuais abaixo da porcentagem real do *top-k*, conseguimos manter a taxa de

acerto dentro do *goal range*, apesar de o sistema utilizar um tamanho menor de memória cache. Também notamos que os momentos em que a taxa de acerto se desvia do *goal range* são as transições de tamanho do *top-k*, conforme o esperado.

Figura 7 – Variação de *top-k* = 50 e intervalo de 10s

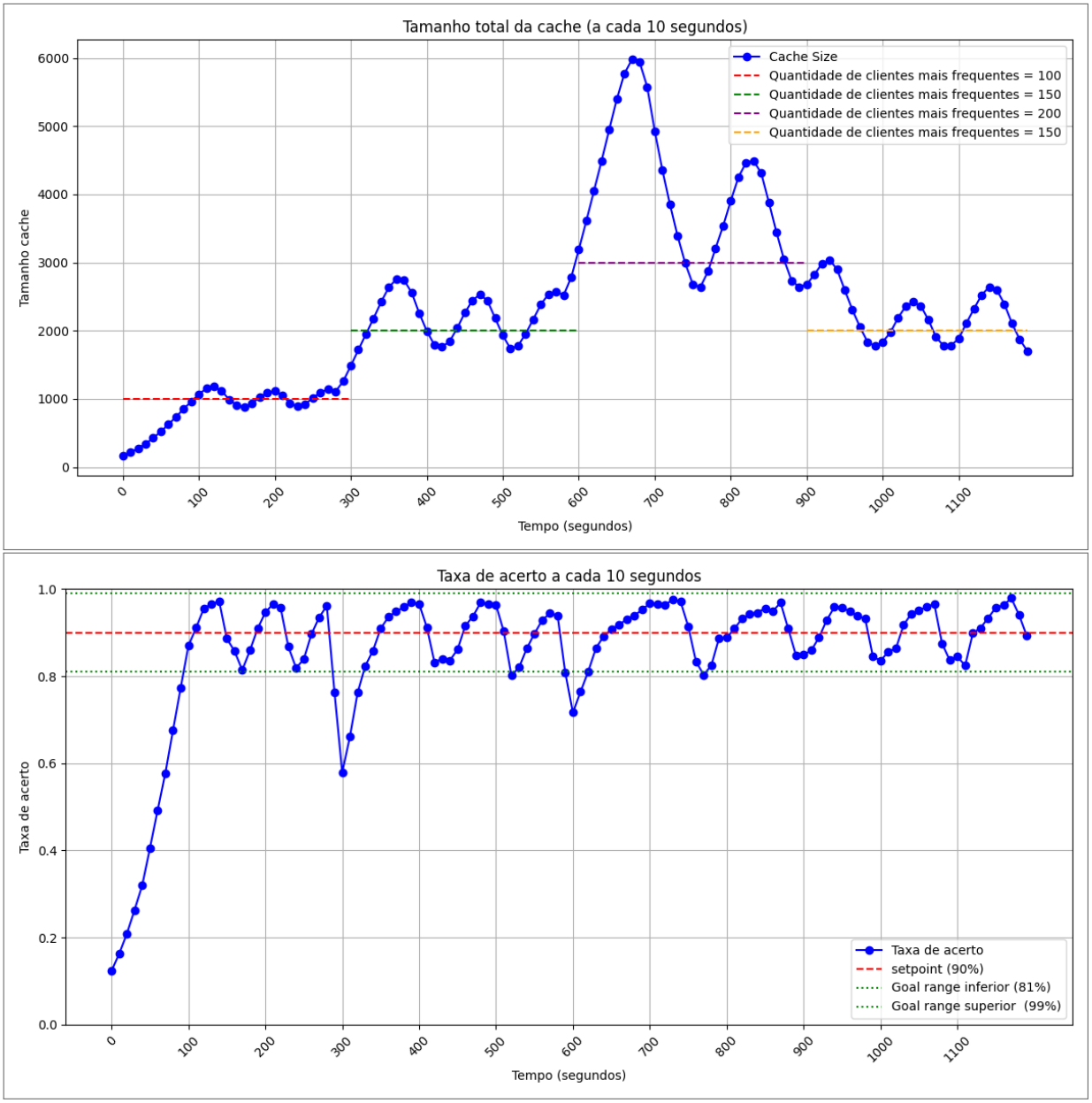


Fonte: Elaborado pelo autor

4.6.3 Cobertura percentual *top-k* de 95%, variação de 1000 no tamanho do *top-k* e intervalo de 10 segundos

Neste experimento, alteramos a variação do *top-k* de 50 para 1000 em relação ao cenário anterior. Podemos perceber, pela Figura 8, que à medida que o tamanho do grupo mais frequente aumentava, o tamanho da cache desviava mais do tamanho real do *top-k*. Isso ocorre porque, sendo o número do *top-k* elevado — atingindo o pico de 3000 itens no intervalo de 600 a 900 segundos — e considerando que os parâmetros definem 100 requisições por segundo, em 10 segundos teríamos apenas 1000 requisições. Este volume não é suficiente para refletir a distribuição real; dessa forma, um intervalo de amostragem maior beneficiaria esse tamanho de *top-k*. Nota-se também na Figura 8 que, apesar dessa diferença no tamanho da cache, a taxa de acerto manteve-se na maior parte do tempo dentro do *goal range*, com desvios significativos apenas nos momentos de transição.

Figura 8 – Variação de *top-k* = 1000 e intervalo de 10

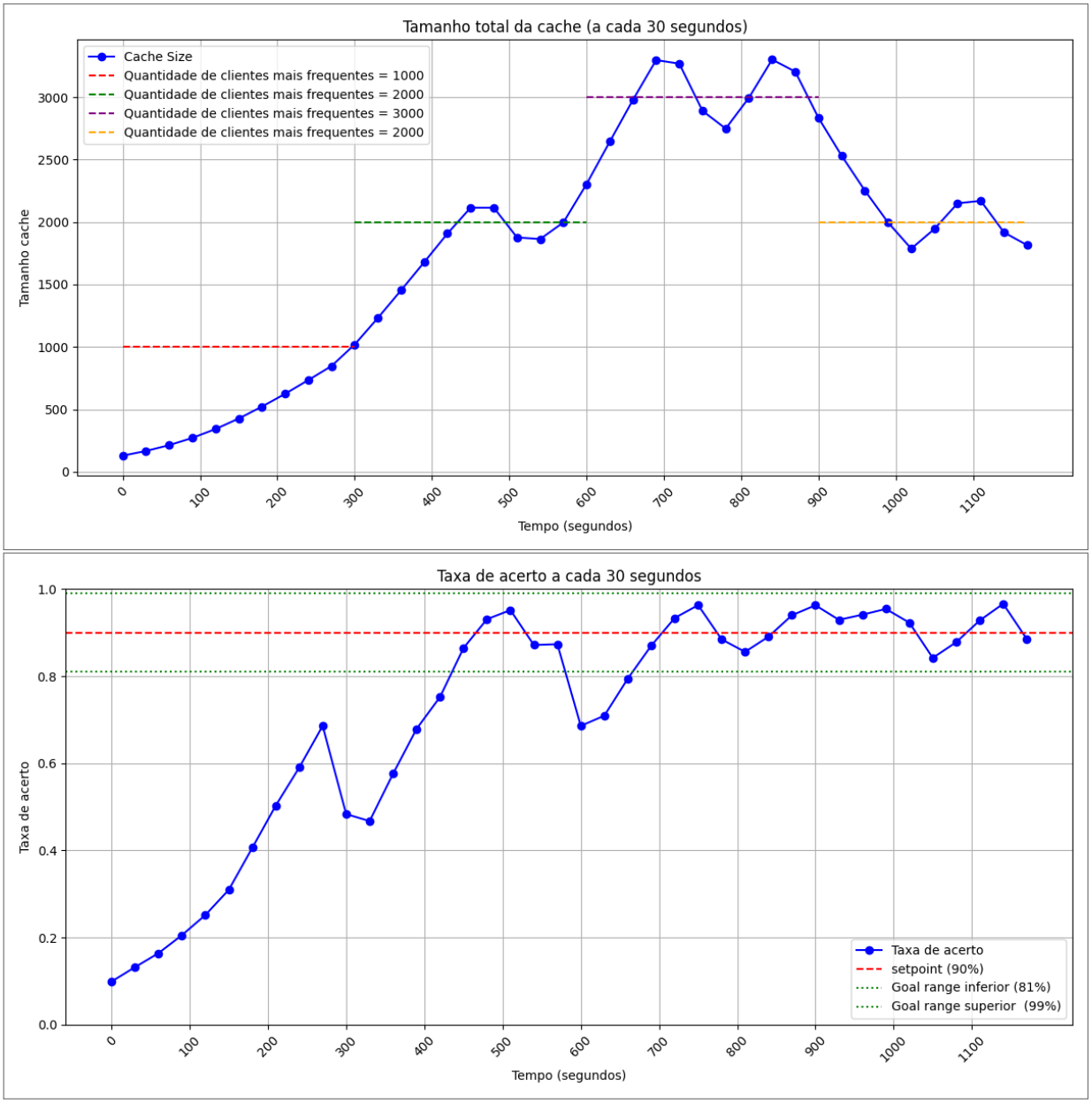


Fonte: Elaborado pelo autor

4.6.4 Cobertura percentual *top-k* de 95%, variação de 1000 no tamanho do *top-k* e intervalo de 30 segundos

Em relação ao experimento anterior, aumentamos o intervalo de ajuste para 30 segundos. Podemos observar, pela Figura 9, que embora o tamanho da cache não se desvie significativamente do tamanho do *top-k*, a taxa de acerto demora mais para atingir o *goal range*. Isso mostra que, para um intervalo de amostragem maior, há a necessidade de um tempo de aquecimento (*warm-up*) mais longo para a estabilização do sistema nesses cenários.

Figura 9 – Variação de *top-k* = 1000 e intervalo de 30s



Fonte: Elaborado pelo autor

5 CONCLUSÃO E TRABALHOS FUTUROS

Este capítulo apresenta inicialmente as contribuições deste trabalho. Em seguida, ele apresenta as limitações e sugestões de trabalhos futuros..

5.1 CONTRIBUIÇÕES

Este trabalho apresentou uma aplicação de Teoria de Controle ao desenvolvimento de uma cache adaptiva para o gRPC. A cache proposta foi implementada no lado do Consumidor e tem o seu tamanho definido dinamicamente por um controlador PID. A utilização de um controlador permite que propriedades (e.g., overshoot e estabilidade) sobre o desempenho da cache sejam garantidas em tempo de execução.

5.2 LIMITAÇÕES

É importante ressaltar que os experimentos foram realizados em um ambiente controlado, utilizando uma topologia simplificada de um cliente e um servidor. Embora este modelo funcional tenha atingido o objetivo de validar a proposta, ele representa uma simplificação que pode ser estendida em cenários mais complexos. Vale também ressaltar que diversos parâmetros foram fixados para a viabilização dos testes, os quais podem sofrer variações dinâmicas em aplicações reais. Adicionalmente, foram selecionadas métricas específicas para manter o escopo do trabalho focado, embora outras variáveis de desempenho pudessem ter sido integradas à análise.

5.3 TRABALHOS FUTUROS

Com base nas limitações expostas, há uma série de possibilidades para trabalhos futuros.

- **Explorar outros controladores**: Como foi pontuado na seção 4.3, se um setpoint for maior que a porcentagem do *top-k*, o tamanho da cache não irá convergir. Dessa forma, seria um grande avanço determinar o setpoint dinamicamente, caso contrário a abordagem apresentada só deve ser utilizada em sistemas em que se tenha uma estimativa da porcentagem do grupo *top-k*.

- **Avaliação de outros fatores**: Em trabalhos futuros podem ser avaliadas as variações de outras varáveis, como tempo entre requisições, políticas de cache, outros tipos de controladores, etc.
- **Utilização de cache distribuída**:Podem ser avaliadas a operação de vários nós paralelos que utilizem uma mesma memória cache global.

REFERÊNCIAS

Amazon Web Services. *Amazon ElastiCache User Guide: On-demand scaling for Memcached clusters*. [S.l.], 2025. Acessado em: 29 dez. 2025. Disponível em: <https://docs.aws.amazon.com/AmazonElastiCache/latest/dg/Scaling-self-designed.mem-heading.html>.

ÅSTRÖM, K. J. et al. *Advanced PID Control*. Research Triangle Park, NC: ISA - The Instrumentation, Systems, and Automation Society, 2006. ISBN 978-1-55617-942-6.

FAROKHI, S. et al. Performance-based vertical memory elasticity. In: IEEE. *2015 IEEE International Conference on Autonomic Computing*. [S.l.], 2015.

GMACH, D. et al. Workload analysis and demand prediction of enterprise data center applications. *Performance Evaluation*, v. 64, n. 9-12, p. 1052–1073, 2007. Disponível em: <https://doi.org/10.1016/j.peva.2007.04.018>.

GOLANG. *The Go Programming Language Documentation*. 2024. `https://go.dev/doc/`. Disponível em: <https://go.dev/doc/>. Acesso em: 9 jul. 2025.

GOOGLE. *Protocol Buffers Documentation*. 2024. `https://protobuf.dev/`. Disponível em: <https://protobuf.dev/>. Acesso em: 9 jul. 2025.

GOOGLE. *gRPC - Documentação oficial*. 2025. Disponível em: <https://grpc.io/docs/>. Acesso em: 11 jun. 2025.

JAIN, R. *The art of computer systems performance analysis*. [S.l.]: John Wiley Sons, 1991.

JANERT, P. K. *Feedback Control for Computer Systems: Introducing Control Theory to Enterprise Programming*. [S.l.]: O'Reilly Media, 2013.

KUMAR, P. *gRPC Interceptor: unary interceptor with code example*. 2022. Disponível em: <https://techdozo.dev/grpc-interceptor-unary-interceptor-with-code-example/>. Acesso em: 12 jun. 2025.

MAYER, H. et al. Comparative analysis of distributed caching algorithms: Performance metrics and implementation considerations. *arXiv preprint*, arXiv:2504.02220, 2025. Preprint. Disponível em: <https://arxiv.org/abs/2504.02220>.

MENASCÉ, D. A. et al. *Capacity Planning for Web Services: metrics, models, and methods*. Upper Saddle River, NJ, USA: Prentice Hall, 2001. ISBN 0-13-065901-1.

NISE, N. S. *Engenharia de sistemas de controle*. 6. ed. [S.l.]: LTC, 2012.

ORACLE CORPORATION. *MySQL 8.0 Reference Manual*. [S.l.], 2023. Acesso em: 14 jul. 2025. Disponível em: <https://dev.mysql.com/doc/refman/8.0/en/>.

PASCHOS, G. S. et al. The role of caching in future communication systems and networks. *IEEE Journal on Selected Areas in Communications*, v. 36, n. 6, p. 1111–1125, 2018.

ROSA, N. S. et al. Exploiting controllers to adapt message-oriented middleware. In: *2024 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)*. [S.l.]: IEEE, 2024. p. 91–100.

TANENBAUM, A. S. et al. *Distributed Systems: Principles and Paradigms*. [S.l.]: Maarten Van Steen, 2023. v. 4.