UNIVERSIDADE ESTADUAL DO OESTE DO PARANÁ - UNIOESTE

CENTRO DE CIÊNCIAS EXATAS E TECNOLÓGICAS

Colegiado de Ciência da Computação

Curso de Bacharelado em Ciência da Computação

Um estudo sobre bugs de concorrência em aplicações Go open-source

Trabalho de Conclusão de Curso

Alonso Lucca Fritz

Cascavel–PR

2024

UNIVERSIDADE ESTADUAL DO OESTE DO PARANÁ - UNIOESTE

CENTRO DE CIÊNCIAS EXATAS E TECNOLÓGICAS

Colegiado de Ciência da Computação

Curso de Bacharelado em Ciência da Computação

Alonso Lucca Fritz

Um estudo sobre bugs de concorrência em aplicações Go open-source

Monografia apresentada como requisito parcial para obtenção do grau de Bacharel em Ciência da Computação, do Centro de Ciências Exatas e Tecnológicas da Universidade Estadual do Oeste do Paraná - Campus de Cascavel.

Orientador(a): Marcio Seiji Oyamada

Cascavel–PR

2024

ALONSO LUCCA FRITZ

UM ESTUDO SOBRE BUGS DE CONCORRÊNCIA EM APLICAÇÕES GO OPEN-SOURCE

Monografia apresentada como requisito parcial para obtenção do Título de Bacharel em Ciência da Computação, pela Universidade Estadual do Oeste do Paraná, Campus de Cascavel, aprovada pela Comissão formada pelos professores:

Prof. Marcio Seiji Oyamada (Orientador)
Colegiado de Ciência da Computação, UNIOESTE

Prof. Guilherme Galante
Colegiado de Ciência da Computação, UNIOESTE

Prof. Ivonei de Freitas Silva
Colegiado de Ciência da Computação, UNIOESTE

Cascavel, DD de MMMM de AAAA

Este trabalho é dedicado às crianças adultas que,
quando pequenas, sonharam em se tornar cientistas.

Agradecimentos

Digite os agradecimentos que quiser, e se quiser, pois não é obrigatório, mas pode jogar confete a vontade, desde que não fique muito extenso (portanto nada de agradecimentos ao cachorro, ao gato, ao papagaio, etc.)

Este trabalho, além de cultural, filosófico e pedagógico
É também medicinal, preventivo e curativo
Servindo entre outras coisas para pano branco e pano preto
Curuba e ferida braba
Piolho, chulé e caspa
Cravo, espinha e berruga
Panarismo e água na pleura
Só não cura o velho chifre
Por que não mata a raiz
Pois fica ela encravada
No fundo do coração
(Falcão)

Resumo

Segundo a ABNT (2003, 3.1-3.2), o resumo deve ressaltar o objetivo, o método, os resultados e as conclusões do documento. A ordem e a extensão destes itens dependem do tipo de resumo (informativo ou indicativo) e do tratamento que cada item recebe no documento original. O resumo deve ser precedido da referência do documento, com exceção do resumo inserido no próprio documento. (. . .) As palavras-chave devem figurar logo abaixo do resumo, antecedidas da expressão Palavras-chave:, separadas entre si por ponto e finalizadas também por ponto.

Palavras-chave: latex. abntex. editoração de texto.

Lista de figuras

Lista de quadros

Lista de tabelas

Lista de códigos

Lista de abreviaturas e siglas

CETIC	Centro Regional de Estudos para o Desenvolvimento da Sociedade da Informação
SMTP	*Simple Mail Transfer Protocol* - Protocolo de Transferência de Correio Simples
POSIX	*Portable Operating System Interface* - Interface de Sistema Operacional Portátil
CSP	*Communicating Sequential Processes* - Processos Sequenciais Comunicantes
ABNT	Associação Brasileira de Normas Técnicas
abnTeX	ABsurdas Normas para TeX

Lista de símbolos

Γ	Letra grega Gama
Λ	Lambda
ζ	Letra grega minúscula zeta
$\in$	Pertence

Sumário

1 Introdução

Com a evolução atual dos dispositivos de hardware, o hardware multi-core tornou-se uma realidade não somente para grandes corporações e servidores de ponta, mas também para computadores desktop, notebooks, tablets, smartphones e outros dispositivos que possuem algum tipo de processamento de mais de um núcleo de processamento. Realizada anualmente desde 2005, a pesquisa realizada pela Cetic (2022) tem o objetivo de mapear o acesso às tecnologias da informação e comunicação nos domicílios urbanos e rurais do país e as suas formas de uso por indivíduos de 10 anos de idade ou mais, segundo dados de 2022, existem 149 milhões de usuários de Internet no território nacional, 142 milhões se conectam todos, ou quase todos os dias, sendo que 92 milhões de brasileiros acessam a internet apenas pelo telefone celular. A presença de Internet nos domicílios estabilizou entre 2021 e 2022, alcançando 60 milhões de domicílios correspondendo a 80% do total no país (CETIC, 2022).

Consequentemente a quantidade de desenvolvimento de sistemas de software concorrentes aumentou, trazendo consigo a necessidade das empresas por desenvolver sistemas que utilizem ao máximo os recursos disponíveis dos dispositivos, dessa forma a computação concorrente tem se tornado crucial para se aproveitar da evolução do hardware. Porém escrever programas concorrentes não é trivial e essa complexidade tem atingido a comunidade de desenvolvimento de software, desde que escrever programas concorrentes de boa qualidade e corretos tornou-se importante (STOICA et al., 2014).

Escrever programas concorrentes corretamente não somente é difícil, como a maioria dos programadores pensam de forma sequencial, portanto, cometem erros facilmente ao escrever programas concorrentes, além disso a natureza não determinística torna extremamente difícil para os desenvolvedores detectarem, diagnosticarem e corrigirem bugs relacionados. Estes chamados bugs de concorrência, são causados por problemas de sincronização em software multithread e podem causar bilhões de dólares em perdas econômicas, tornando-se uma ameaça à confiabilidade do uso de softwares multithread (LU et al., 2008).

Estudos software de código aberto realizados por Lu et al. (2008) mostraram que geralmente leva vários meses para os desenvolvedores corrigirem corretamente os bugs de concorrência. Além disso, os bugs de concorrência são os mais difíceis de corrigir corretamente entre os tipos de bugs comuns (YIN et al., 2011), com muitos patches incorretos lançados. Partindo deste ponto podemos dizer que desenvolver programas concorrentes de forma correta é um desafio que ainda necessita de conhecimento e avanços em diversos cenários, incluindo entre eles a detecção de bugs de concorrência, teste e verificação de modelo de programas concorrentes e o design de uma linguagem de programação concorrente.

Algumas linguagens entretanto passaram por diversas adaptações para então poder disponibilizar o conjunto de ferramentas necessárias para o desenvolvimento concorrente, desde que inicialmente essas linguagens não nasceram com esse tema específico em mente. A linguagem Java criou um modelo de green threads, um tipo de mecanismo de concorrência que facilita o uso de fluxos concorrentes, sem utilizar múltiplos processadores, e só depois disponibilizou threads nativas do Java (ORACLE, 2000). C++ disponibilizou uma biblioteca própria de threads apenas na versão de 2011, antes tendo de utilizar threads POSIX, ou a biblioteca pthreads do C (ISO, 2011). A linguagem Python, por exemplo, passou a oferecer os recursos de *async/await*, que permite a escrita de códigos assíncronos, com o lançamento da versão 3.5, em setembro de 2015 (PRANSKEVICHUS; SELIVANOV, 2015), estes são alguns exemplos de adaptações realizadas pelas linguagens para atenderem as necessidades atuais de hardware.

A linguagem Go, originalmente desenvolvida pelo Google em 2009 (GO, 2023), tem conquistado rapidamente seu espaço na indústria de software, sendo adotada em uma ampla gama de aplicações, desde bibliotecas como gRPC (GOOGLE, 2023) e softwares de alto nível como clientes SMTP (HECTANE, 2023) até sistemas de infraestrutura em nuvem, como sistemas de contêineres, que serão o principal foco nesta pesquisa (DOCKER, 2023). Uma das propostas fundamentais do Go é simplificar e tornar mais segura a programação concorrente, buscando soluções que minimizem os erros nesse contexto, sendo esse o foco de nossa pesquisa.

Um dos principais objetivos do design da linguagem Go é aprimorar a programação concorrente em comparação com as linguagens tradicionais multithreaded. Para alcançar esse objetivo, o Go concentra seu design de concorrência em dois princípios essenciais: primeiramente tornar as threads, chamadas goroutines, leves e de fácil criação; e em segundo adotar o uso de mensagens explícitas, conhecidas como canais, para facilitar a comunicação entre essas threads. Esses princípios fundamentais não apenas introduzem novas primitivas e bibliotecas de concorrência, mas também promovem uma reinterpretação das semânticas já existentes (TU et al., 2019).

Compreender o impacto dessas novas primitivas e mecanismos de concorrência do Go em relação aos bugs de concorrência é crucial. Esses tipos de bugs são notoriamente complexos de diagnosticar e têm sido amplamente estudados em linguagens de programação tradicionais multithread. Tu et al. (2019), realizou um estudo sobre bugs de concorrência na linguagem Go,

nas seguintes aplicações open-source: Docker, Kubernetes, gRPC, etcd, CockroachDB e BoltDB. Conforme o autor avaliou, existe uma incerteza se os mecanismos de concorrência realmente tornam a programação em Go mais acessível e menos propensa a erros de concorrência quando comparada às abordagens tradicionais.

Neste contexto, este trabalho tem como objetivo estender o trabalho de Tu et al. (2019), realizando uma busca nos repositórios open-source de projetos populares em Go verificando se ainda existe a ocorrência de bugs de concorrência.

Ao longo desta pesquisa, será identificado não apenas a causa raiz desses bugs, mas também suas correções e padrões comuns de uso inadequado das primitivas de concorrência do Go. Essa pesquisa contribuirá significativamente para uma melhor compreensão dos modelos de concorrência do Go, além de fornecer orientações valiosas para desenvolvedores e pesquisadores que desejam escrever software Go mais confiável e desenvolver ferramentas de depuração e diagnóstico mais eficazes para aplicações concorrentes.

1.1 Objetivos Gerais

Este trabalho tem como objetivo estender o trabalho de Tu et al. (2019), realizando um estudo sobre bugs de concorrência na linguagem de programação Go, e em como suas primitivas e mecanismos de concorrência afetam os bugs de concorrência em aplicações open source como Docker (DOCKER, 2023), Terraform (TERRAFORM, 2023) e CockroachDB (LABS, 2023a). Go é uma linguagem que possui estruturas tanto para memória compartilhada quanto passagem de mensagens, no entanto ele incentiva o uso de canais em vez de memória compartilhada, considerando assim a passagem explícita de mensagens menos propensa a erros, assim nos concentramos em uma questão antiga e fundamental da programação concorrente qual dos mecanismos de comunicação entre threads é menos propenso a erros, passagem de mensagens ou memória compartilhada.

O foco principal é investigar como os mecanismos de concorrência do Go impactam a ocorrência e a natureza dos bugs de concorrência, bem como identificar e classificar seu comportamento e causa. A pesquisa visa aprofundar a compreensão bugs em aplicações concorrentes em Go, em entender se aplicações em Go que utilizam passagem explícita de mensagens menos propensa a erros e em contribuir para o desenvolvimento de software mais confiável e ferramentas de diagnóstico eficazes nessa linguagem.

1.2 Objetivos Específicos

- Definir aplicações Open Source a serem utilizadas como fonte de pesquisa para Bugs;
- Identificar e Selecionar os Bugs de concorrência em repositórios de aplicações open-source,

segundo palavras chave definidas;

- Entender a taxonomia de bugs proposta por Tu et al. (2019), e aplica-la aos bugs selecionados
- Compreender as causas raiz dos bugs de concorrência identificados, incluindo o uso inadequado das primitivas de concorrência do Go;
- Investigar as issues, tempos de resposta e solução para problemas de concorrência encontrados;
- Relatar pontos positivos e negativos em relação ao uso do escalonador personalizado.
- Analisar os padrões comuns de erro relacionados à concorrência observados nas aplicações open-source estudadas;
- Fornecer orientações e boas práticas para desenvolvedores e pesquisadores que desejam escrever código Go mais confiável e criar ferramentas de depuração e diagnóstico de bugs de concorrência mais eficazes para essa linguagem;
- Contribuir para o avanço do conhecimento sobre programação concorrente em Go e aprimorar a compreensão dos desafios e das soluções associados à concorrência nessa linguagem.

2

Revisão Bibliográfica

2.1 Concorrência

O conceito de concorrência na computação está associado à ideia de independência da ordem de execução entre programas, onde fluxos distintos não possuem uma dependência causal entre si, permitindo que sejam considerados como fluxos concorrentes (LAMPORT, 2019). A palavra "concorrência"pode ter diferentes interpretações para aqueles na área de computação, sendo algumas vezes utilizada de forma intercambiável com termos como "assíncrono", "paralelo"ou "*threaded*". Alguns consideram essas variações como tendo os mesmos significados, enquanto outros as consideram inteiramente distintas entre si (COX-BUDAY, 2017). A programação concorrente tem sido cada vez de mais importância na ciência da computação, destacando sua complexidade, desafios e a necessidade de um estudo cuidadoso. A discussão ressalta que, apesar dos desafios, a linguagem de programação Go oferece primitivas de concorrência que prometem tornar os programas mais claros e eficientes (COX-BUDAY, 2017).

Entretanto, o código concorrente é conhecido por ser notoriamente difícil de ser desenvolvido corretamente, muitas vezes exigindo iterações para alcançar o funcionamento esperado. Bugs em códigos concorrentes podem permanecer não detectados por longos períodos, sendo revelados apenas quando ocorrem mudanças nas condições temporais, como aumento na utilização de disco ou maior número de usuários no sistema. A comunidade científica, ciente desses desafios, identificou questões comuns relacionadas ao código concorrente, possibilitando discussões sobre como esses problemas surgem, por que ocorrem e como podem ser solucionados (COX-BUDAY, 2017).

2.1.1 Processos

Em um sistema operacional, a representação de um programa em execução é denominada processo. Essa abstração possibilita a execução simultânea de vários processos no mesmo

sistema, aparentando que cada um tem uso exclusivo do hardware. O conceito de simultaneidade refere-se à intercalação de instruções entre processos, sendo que, na maioria dos sistemas, há mais processos para execução do que CPUs disponíveis. Nos sistemas tradicionais, apenas um programa era executado de cada vez, ao passo que processadores multi core mais modernos têm a capacidade de executar diversos programas simultaneamente. Mesmo em uma única CPU, a ilusão de execução simultânea entre processos é obtida por meio da troca de contexto, um mecanismo operado pelo sistema operacional (BRYANT; O'HALLARON, 2011).

Em sistemas operacionais contemporâneos, cada processo é alocado em um espaço de memória exclusivo, prevenindo assim problemas de integridade dos dados do programa, uma vez que cada processo possui seu próprio espaço de endereçamento. O sistema operacional desempenha o papel de fornecer e proteger esse espaço de memória contra acessos indevidos por outros processos. A comunicação com dispositivos ocorre frequentemente por meio de chamadas de sistema, operações realizadas por um processo para requisitar ao sistema operacional a execução de uma tarefa específica, como desenhar um ponto na tela (TANENBAUM; FILHO, 2016).

Além disso, os processos em sistemas operacionais modernos não são a menor abstração de um fluxo de execução, uma vez que podem conter uma ou mais threads, fluxos de execução independentes. Threads compartilham o espaço de memória e código de um processo, mas cada uma possui seus próprios registradores e ponteiros de instrução. Embora compartilhem dados, as threads não interferem umas nas outras, garantindo a independência de execução. A criação e o gerenciamento de threads são responsabilidades do sistema operacional, que provê recursos como espaço de memória e registradores, além de controlar a execução das threads (TANENBAUM; FILHO, 2016).

2.1.2 *Threads*

Em sistemas operacionais modernos, o conceito de processo evoluiu para além de um único fluxo de controle, permitindo que um processo consista em múltiplas unidades de execução conhecidas como threads. As threads operam no contexto do processo, compartilhando o mesmo código e dados globais. Essa abordagem destaca a importância crescente das threads como um modelo de programação vital, especialmente diante da necessidade de concorrência em servidores de rede, por exemplo (BRYANT; O'HALLARON, 2011).

A relevância das threads é evidenciada pela facilidade de compartilhamento de dados entre elas, comparado com a complexidade de compartilhar dados entre processos distintos. Além disso, as threads, em geral, são mais eficientes que processos isolados. O uso de multi-threading não apenas atende à demanda por concorrência, mas também representa uma estratégia para acelerar a execução de programas em sistemas com múltiplos processadores disponíveis (TANENBAUM; FILHO, 2016).

Nos sistemas operacionais tradicionais, um processo era associado a um único espaço de endereçamento e a um único thread de controle. Contudo, em muitas situações, a necessidade de operações quase paralelas levou à preferência por múltiplos threads de controle compartilhando o mesmo espaço de endereçamento. Essa abordagem permite que as threads ajam quase como processos separados, embora compartilhem o espaço de endereçamento (TANENBAUM; FILHO, 2016). Essa flexibilidade no uso de threads dentro de um processo destaca a adaptação dos sistemas operacionais modernos para atender às demandas de eficiência e concorrência na execução de programas.

Atualmente podemos destacar dois tipos principais de threads, sendo elas as kernel threads e as chamadas user-space threads, ou threads de espaço de usuário. A diferença essencial entre elas está no detentor da responsabilidade pela coordenação de execução de fluxos. Sendo para as kernel threads, o sistema operacional é o responsável por essa gerência, é comum em sistemas UNIX e Windows essas threads serem do padrão POSIX. Enquanto para as de espaço de usuário, um programa criado por um usuário que é o responsável por essa gerência, não tendo uma visão global que o sistema operacional possui, como exemplo conhecido podemos citar a máquina virtual do Java com o recurso de green threads, outro exemplo, importante para esse trabalho é o escalonador de Go, que gerencia as goroutines (TANENBAUM; FILHO, 2016), mostradas em subseção 2.3.1.

2.1.3 Condição de Corrida

Em um contexto de abstração para threads, uma condição de ocorrida ocorre sempre que um programa depende de uma thread atingir um objetivo x em seu fluxo de controle antes que outra thread atinja o objetivo y (BRYANT; O'HALLARON, 2011). E como iremos trabalhar com Go, dentro desse contexto pode-se considerar que uma condição de corrida ocorre quando o programa não fornece o resultado correto para algumas intercalações das operações de múltiplas goroutines (DONOVAN; KERNIGHAN, 2015).

No contexto geral então uma condição de corrida ocorre quando duas ou mais operações devem ser executadas em uma ordem correta, mas o programa não foi escrito de forma a garantir que essa ordem seja mantida, se manifestando no que é chamado de corrida de dados, onde uma operação concorrente tenta ler uma variável enquanto, em um determinado momento, outra operação concorrente tenta escrever na mesma variável (COX-BUDAY, 2017).

Um exemplo é a maneira mais clara de entender a natureza desse problema. Considere o Código 1. Em Go voce pode usar a o termo *go* para executar uma função concorrentemente, fazendo isso criamos o que chamamos de *goroutines*, discutido em maior detalhes em subseção 2.3.1. Dessa maneira são criadas duas *goroutines* que incrementam uma variável compartilhada chamada *counter* em um *loop*. A função *main* utiliza *sync.WaitGroup* para garantir que as duas goroutines concluam antes de imprimir o valor final de *counter*, isso é discutido em detalhes na subseção 2.3.2.1.

Código 1 – Exemplo de Condição de Corrida

```go
package main

import (
  "fmt"
  "runtime"
  "sync"
)

var counter int
var wg sync.WaitGroup

func main() {
  runtime.GOMAXPROCS(2)
  wg.Add(2)

  go incrementCounter()
  go incrementCounter()

  wg.Wait()

  fmt.Println("Final Counter:", counter)
}

func incrementCounter() {
  defer wg.Done()
  for i := 0; i < 1000000; i++ {
    // Critic! Race condition here!
    counter++
  }
}
```

A condição de corrida ocorre porque duas goroutines estão concorrendo para acessar e modificar a variável counter sem uma sincronização adequada. Sendo esta uma situação em que o comportamento do programa depende da ordem de execução das threads ou goroutines.

Ambas as goroutines estão incrementando counter simultaneamente sem qualquer mecanismo de sincronização, o que pode resultar em leituras e gravações concorrentes na variável compartilhada.

Na maioria das vezes, as corridas de dados são introduzidas porque os desenvolvedores estão pensando no problema de forma sequencial assumindo que, porque uma linha de código está antes de outra, ela será executada primeiro (COX-BUDAY, 2017). De forma mais específica ocorre porque os desenvolvedores assumem que as threads seguirão alguma trajetória específica pelo espaço de estados de execução, esquecendo a regra fundamental de que programas com threads devem funcionar corretamente para qualquer trajetória viável (BRYANT; O'HALLARON, 2011).

Condições de corrida são perigosas porque podem permanecer latentes em um programa e aparecer raramente, talvez apenas sob carga intensa ou ao usar determinados compiladores,

plataformas ou arquiteturas. Isso torna difícil reproduzi-las e diagnosticá-las (DONOVAN; KERNIGHAN, 2015).

2.1.4 *Deadlocks*

Um cenário de deadlock ocorre quando todos os processos concorrentes aguardam uns aos outros. Nesse estado, o programa torna-se irrecuperável sem intervenção externa (COX-BUDAY, 2017). O tempo de execução (*runtime*) do Go faz tentativas de detectar alguns *deadlocks*, nos quais todas as *goroutines* devem estar bloqueadas ou "adormecidas" (Golang Team, 2023). No entanto, essa abordagem não oferece uma solução abrangente para prevenir *deadlocks*.

Para uma compreensão mais clara do que constitui um deadlock, examinemos inicialmente um exemplo no Código 2. O programa apresenta um potencial deadlock devido à possibilidade de duas goroutines ficarem bloqueadas uma esperando pela outra.

Código 2 – Exemplo de *Deadlock*

```
1  type value struct {
2    mu    sync.Mutex
3    value int
4  }
5  var wg sync.WaitGroup
6  printSum := func(v1, v2 *value) {
7    defer wg.Done()
8    v1.mu.Lock()
9    defer v1.mu.Unlock()
10   time.Sleep(2 * time.Second)
11   v2.mu.Lock()
12   defer v2.mu.Unlock()
13   fmt.Printf("sum=%v\n", v1.value+v2.value)
14 }
15 var a, b value
16 wg.Add(2)
17 go printSum(&a, &b)
18 go printSum(&b, &a)
19 wg.Wait()
```

Analisando as sessões críticas segundo Cox-Buday (2017):

- **Linha 8**: Aqui, a goroutine adquire a trava (lock) do mutex associado à estrutura v1. Isso significa que ela está bloqueando o acesso a qualquer outra goroutine que tente adquirir o mesmo mutex. No entanto, observe que a goroutine também chama v2.mu.Lock() mais tarde no código.
- **Linha 9**: O uso de defer garante que o mutex seja desbloqueado mesmo se ocorrer uma exceção ou se a função retornar. Neste caso, a goroutine desbloqueia o mutex associado a v1 quando a função printSum é concluída.

- **Linha 10**: Este é um atraso de 2 segundos adicionado intencionalmente para aumentar a chance de ocorrer um deadlock. Durante esse tempo, a primeira goroutine ainda mantém o lock de v1, mas está prestes a solicitar o lock para v2.
- **Linha 11**: A segunda goroutine tenta adquirir o lock para o mutex associado a v2. No entanto, este mutex está atualmente bloqueado pela primeira goroutine.
- **Linha 12**: Assim como na seção 2, esta instrução defer garante que, mesmo se ocorrer uma exceção ou se a função retornar, o mutex associado a v2 será desbloqueado.

O deadlock potencial ocorre porque a primeira goroutine bloqueia v1 e espera acessar v2, enquanto a segunda goroutine bloqueia v2 e espera acessar v1. Ambas as goroutines estão esperando uma pela outra para desbloquear o mutex que precisam para prosseguir, resultando em um impasse (COX-BUDAY, 2017).

2.1.5 Sincronização por Memória Compartilhada

A memória compartilhada, uma técnica de sincronização concorrente em Go, permite que múltiplas *goroutines* acessem simultaneamente a mesma variável ou recurso, evitando a sobrecarga de comunicação entre elas. Para implementar a memória compartilhada em Go, utilizam-se estruturas como *channels* e *mutexes*. *Channels* possibilitam comunicação assíncrona entre *goroutines*, enquanto *mutexes* protegem o acesso exclusivo a recursos compartilhados. Contudo, essa eficiente técnica é mais propensa a condições de corrida. Por outro lado, a passagem de mensagens, uma alternativa que envolve o envio de mensagens entre *goroutines*, é menos propensa a race conditions, mas pode apresentar menor eficiência em termos de desempenho. A escolha entre essas técnicas é crucial, e a análise de Tu et al. (2019) sobre 171 bugs de concorrência em Golang destaca a importância de usar a memória compartilhada com cautela, visto que a maioria dos bugs identificados estava relacionada a race conditions (TU et al., 2019).

O Go oferece suporte a acessos tradicionais à memória compartilhada, proporcionando primitivos de sincronização como Mutex, RWMutex, Cond e atomic. O RWMutex no Go difere da implementação pthread_rwlock_t em C, conferindo maior privilégio a solicitações de trava de escrita. Introduzindo o primitivo "Once", o Go assegura que uma função seja executada apenas uma vez, sendo valioso para a inicialização única de variáveis compartilhadas por várias goroutines. Além disso, o "WaitGroup"é empregado para permitir que múltiplas goroutines concluam seus acessos a variáveis compartilhadas antes que uma goroutine de espera prossiga. Contudo, o uso inadequado do WaitGroup pode resultar em bugs de bloqueio e não bloqueadores (TU et al., 2019). Essas ferramentas oferecidas pelo Go refletem a preocupação em equilibrar eficiência e segurança em ambientes concorrentes.

2.1.6 Sincronização por Passagem de Mensagem

A sincronização por passagem de mensagem é uma técnica de sincronização de threads que utiliza a troca de mensagens entre threads para garantir a consistência de dados (TU et al., 2019). Essa técnica é baseada no princípio de que threads só podem acessar dados compartilhados se receberem uma mensagem autorizando-os a fazê-lo.

Uma das vantagens da sincronização por passagem de mensagem é que ela é mais segura do que a sincronização por memória compartilhada, pois reduz a possibilidade de condições de corrida (CORMEN et al., 2009). Isso ocorre porque, nesta sincronização, as threads só podem acessar dados compartilhados se receberem uma mensagem autorizando-os a fazê-lo. Isso elimina a possibilidade de um thread acessar dados compartilhados que estão sendo modificados por outro thread.

Outra vantagem da sincronização por passagem de mensagem é que ela é mais flexível do que a sincronização por memória compartilhada (CORMEN et al., 2009). Isso ocorre porque, na sincronização por passagem de mensagem, os threads podem trocar mensagens de qualquer tipo, incluindo dados estruturados complexos. Isso permite que os threads compartilhem dados de forma mais eficiente e flexível.

No entanto, a sincronização por passagem de mensagem também apresenta algumas desvantagens. Uma das desvantagens é que ela pode ser menos eficiente do que a sincronização por memória compartilhada. Isso ocorre porque, na sincronização por passagem de mensagem, os threads precisam trocar mensagens para acessar dados compartilhados. Essa troca de mensagens pode adicionar overhead ao sistema (TANENBAUM; FILHO, 2016).

Outra desvantagem da sincronização por passagem de mensagem é que ela pode ser mais complexa do que a sincronização por memória compartilhada. Isso ocorre porque, na sincronização por passagem de mensagem, os desenvolvedores precisam implementar mecanismos para gerenciar a troca de mensagens entre threads (TANENBAUM; FILHO, 2016).

2.2 A linguagem de programação Go

Bugs de concorrência representam desafios significativos para o desenvolvimento e a confiabilidade de sistemas de software. No âmbito de linguagens de programação projetadas para enfrentar tais desafios, o Go, também conhecido como Golang, surgiu como uma ferramenta poderosa e eficiente para programação concorrente. Esta introdução fornece uma breve visão geral da linguagem de programação Go, destacando seus princípios de design, recursos e contexto histórico, estabelecendo as bases para a exploração subsequente de bugs de concorrência em aplicativos Go de código aberto.

Go é uma linguagem de propósito geral explicitamente criada com programação de sistemas em mente (PIKE, 2023g). Desenvolvido como um projeto de código aberto, o Go

tem como objetivo aprimorar a produtividade do programador, oferecendo uma linguagem expressiva, concisa e eficiente, com forte suporte para programação concorrente (PIKE, 2023a). A linguagem é fortemente tipada, possui coleta de lixo e utiliza uma sintaxe direta que permite análise fácil, tornando-a propícia para análises automáticas por ferramentas como ambientes de desenvolvimento integrado (PIKE, 2023b).

Programas em Go são construídos a partir de pacotes, facilitando a gestão eficiente de dependências. Os princípios de design do Go buscam inspiração na família C em termos de sintaxe básica, com influências adicionais da família Pascal/Modula/Oberon para declarações e pacotes, e ideias de linguagens inspiradas por Processos Sequenciais Comunicantes (CSP), como Newsqueak e Limbo, para concorrência (PIKE, 2023e).

A linguagem de programação Go, apresentada ao público como um projeto de código aberto em 2009, angariou uma comunidade ampla de desenvolvedores chamados "gophers"(PIKE, 2023d). Seu sucesso superou as expectativas iniciais, tornando-se a linguagem de escolha para inúmeros programadores em todo o mundo (PIKE, 2023d).

Notavelmente, o Go possui uma extensa biblioteca de tempo de execução, integrada a cada programa Go. Essa biblioteca implementa recursos críticos, como coleta de lixo, concorrência e gerenciamento de pilhas (PIKE, 2023h). É essencial esclarecer que o tempo de execução do Go difere de ambientes tradicionais de máquinas virtuais, já que os programas Go são compilados antecipadamente para código nativo da máquina (**??**).

O Go enfrenta os desafios da concorrência construindo sobre as ideias de CSP (PIKE, 2023e). Este modelo enfatiza interfaces de nível mais alto para simplificar o código, desviando da complexidade associada a detalhes de baixo nível, como mutexes e barreiras de memória. As primitivas de concorrência do Go, inspiradas pelo CSP, incluem o conceito inovador de canais como objetos de primeira classe (PIKE, 2023e).

A linguagem de programação Go, com suas raízes na família C e influências do CSP, oferece uma abordagem única para a programação concorrente. Seus princípios de design, sistema de tempo de execução eficiente e ênfase na simplicidade contribuem para sua popularidade entre os desenvolvedores. As seções subsequentes desta irão explorar as características, e primitivas que dão suporte ao diferencial da programação concorrente (PIKE, 2023c).

2.3 Concorrência em Go

A importância da concorrência no cenário computacional contemporâneo, especialmente no contexto de máquinas multicore e servidores web lidando com múltiplos clientes. Go, como linguagem, se destaca ao incorporar uma variante de CSP com canais de primeira classe. A adoção do CSP é motivada não apenas pela familiaridade, mas também pela sua integração perfeita a um modelo de programação procedural sem exigir mudanças profundas na estrutura da

linguagem. Essa composição de funções executando de forma independente dentro de código procedural regular permite que o Go una suavemente a concorrência à computação, oferecendo uma solução prática para a construção de servidores web, um caso de uso comum no Google (PIKE, 2012).

Embora o modelo de concorrência do Go prove ser vantajoso para cenários típicos, ele vem com uma ressalva significativa: o Go não é puramente seguro em termos de memória na presença de concorrência. A linguagem permite compartilhamento, incluindo a passagem de ponteiros por canais, o que é tanto idiomático quanto eficiente. Apesar de alguma decepção por parte de especialistas em concorrência e programação funcional que esperavam uma abordagem de escrita única para semântica de valor, semelhante a linguagens como Erlang, as escolhas de design do Go priorizam a familiaridade e adequação ao domínio do problema. A linguagem promove programação concorrente simples e segura, confiando em convenções e treinamento para incentivar programadores a adotar um paradigma de passagem de mensagens como uma forma de controle de propriedade. O lema "Não se comunique compartilhando memória, compartilhe memória se comunicando"encapsula a abordagem do Go, promovendo simplicidade e robustez em software de rede concorrente (PIKE, 2012).

2.3.1 *Goroutines*

Goroutines, fundamentais no Go, simplificam o uso de concorrência ao serem funções que operam de maneira independente ou coroutines, multiplexadas em um conjunto de threads. O sistema de tempo de execução, conforme destacado por Pike (2023f), gerencia automaticamente situações de bloqueio, movendo coroutines entre threads, garantindo eficiência na utilização dos recursos do sistema.

No sentido de atingir eficiência, o tempo de execução do Go implementa pilhas redimensionáveis e limitadas para goroutines. A dinâmica de ajuste do tamanho da pilha possibilita a criação de inúmeras goroutines com uma quantidade moderada de memória. Essa escolha de design diferencia as goroutines das threads tradicionais, contribuindo para a reputação do Go como uma linguagem que facilita a programação escalável e concorrente, conforme destacado por Pike (2023f).

Goroutines, sendo unidades fundamentais de organização em programas Go, merecem compreensão aprofundada. Todo programa Go, ao iniciar, possui pelo menos uma goroutine, denominada goroutine principal, automaticamente criada e iniciada durante o processo (COX-BUDAY, 2017).

A definição de goroutines em Go, como atividades executadas concorrentemente, difere de um programa sequencial, onde chamadas para diferentes funções ocorrem de maneira linear. Donovan e Kernighan (2015) introduzem a ideia de que, em um programa concorrente com duas ou mais goroutines, chamadas para diferentes funções podem estar ativas simultaneamente,

proporcionando novas possibilidades de execução.

Embora a sintaxe para lançar goroutines seja semelhante à chamada de threads em outras linguagens, Donovan e Kernighan (2015) esclarecem que as diferenças entre threads e goroutines são predominantemente quantitativas. O lançamento de goroutines, realizado pela palavra-chave go, introduz a execução de contextos leves (green threads), proporcionando uma eficiente multiplexação em threads nativas (BINET, 2018).

Ao iniciar um programa, sua única goroutine é aquela que chama a função principal, sendo denominada goroutine principal. A criação de novas goroutines é realizada pela instrução go, que, sintaticamente, precede uma chamada de função ou método comum, levando à execução da função em uma goroutine recém-criada (DONOVAN; KERNIGHAN, 2015).

Código 3 – Exemplo de *Goroutine*

```
package main

import (
  "fmt"
  "sync"
)

func main() {
  var wg sync.WaitGroup

  // Adiciona 2 ao WaitGroup para indicar que estamos esperando duas
    goroutines
  wg.Add(2)

  // Goroutine 1
  go func() {
    defer wg.Done() // Decrementa o contador quando a goroutine termina
    printMessage("Hello from Goroutine 1")
  }()

  // Goroutine 2
  go func() {
    defer wg.Done() // Decrementa o contador quando a goroutine termina
    printMessage("Hello from Goroutine 2")
  }()

  // Espera at  que todas as goroutines tenham terminado
  wg.Wait()

  fmt.Println("Main function exiting")
}

func printMessage(message string) {
  fmt.Println(message)
}
```

O Código 3, representa uma simples de utilização de goroutines em Go. Este exemplo cria duas goroutines anônimas usando a palavra-chave go antes de uma função anônima. A

função printMessage é chamada em cada goroutine para imprimir uma mensagem específica. A sync.WaitGroup é usada para aguardar a conclusão de ambas as goroutines antes de sair da função main. Isso garante que a função main aguarde a conclusão de todas as goroutines antes de terminar.

As duas goroutines são criadas de forma concorrente para executar as funções anônimas que chamam printMessage. A utilização de defer wg.Done() garante que o contador do WaitGroup seja decrementado quando as goroutines terminam.

Ao executar esse código, você verá mensagens "Hello from Goroutine 1"e "Hello from Goroutine 2"sendo intercaladas na saída, demonstrando a execução concorrente das goroutines. O uso do WaitGroup garante que a função main aguarde até que ambas as goroutines tenham terminado antes de encerrar.

Goroutines, uma característica distintiva do Go, diferenciam-se por não serem threads do sistema operacional e também não serem green threads, que são threads gerenciadas pelo tempo de execução de uma linguagem. Elas representam um nível mais elevado de abstração conhecido como coroutines. As coroutines, por sua vez, são sub-rotinas concorrentes, como funções, closures ou métodos em Go, caracterizadas pela não preempção, ou seja, não podem ser interrompidas. Coroutines proporcionam vários pontos ao longo dos quais é possível a suspensão ou reentrada (COX-BUDAY, 2017).

A singularidade das goroutines no Go está relacionada à sua profunda integração com o tempo de execução da linguagem. Ao contrário de outras implementações, as goroutines não especificam seus próprios pontos de suspensão ou reentrada. O tempo de execução do Go monitora dinamicamente o comportamento em tempo de execução das goroutines, suspendendo-as automaticamente quando bloqueadas e retomando-as quando desbloqueadas. Isso as torna, de certa forma, preemptivas, mas apenas nos pontos específicos em que a goroutine fica bloqueada. Essa colaboração eficaz entre o tempo de execução e a lógica de uma goroutine as configura como uma classe especial de coroutine (COX-BUDAY, 2017).

2.3.2 Pacote *Sync*

Diretamente da documentação da versão go1.121.5, o pacote sync fornece primitivas básicas de sincronização, como fechaduras de exclusão mútua. Com exceção dos tipos Once e WaitGroup, a maioria destes é destinada ao uso por rotinas de bibliotecas de baixo nível. Sincronização em níveis mais altos é mais bem realizada através de canais e comunicação (PIKE, 2023h). No contexto do Go, a diferença marcante em relação a essas linguagens é que o Go desenvolveu um conjunto inovador de primitivas de concorrência, construído sobre as primitivas de sincronização de acesso à memória, proporcionando assim uma gama ampliada de elementos para trabalhar (COX-BUDAY, 2017).

Quando a modelagem por troca de mensagem se mostra difícil e complexa, o Go oferece

recursos clássicos, tais como mutexes e semáforos, por meio da biblioteca sync (COX-BUDAY, 2017). Além disso, a biblioteca sync disponibiliza outras estruturas que podem ser usadas de maneira complementar aos canais (channels) e goroutines.

2.3.2.1 *WaitGroup*

O mecanismo de WaitGroup no Go oferece uma forma eficaz de coordenar a execução de goroutines, permitindo que a goroutine principal aguarde a conclusão de um conjunto específico delas. Conforme descrito por Pike (2023h), a principal funcionalidade da WaitGroup envolve a chamada do método Add para definir o número de goroutines a serem aguardadas, seguido por cada goroutine chamando o método Done quando conclui sua execução. A utilização do método Wait possibilita que a goroutine principal bloqueie sua execução até que todas as goroutines tenham sido finalizadas. No contexto do modelo de memória do Go, é importante notar que uma chamada ao método Done "sincroniza antes"do retorno de qualquer chamada Wait que ela desbloqueia.

Conforme sugerido por Cox-Buday (2017), o WaitGroup se destaca como uma ferramenta valiosa para a espera de operações concorrentes quando o resultado dessas operações não é relevante ou quando existem outros meios de coletar esses resultados. No entanto, se essas condições não forem atendidas, a autora recomenda o uso de canais e uma instrução select como uma alternativa. A versatilidade e utilidade do WaitGroup são evidentes, justificando sua apresentação como um recurso fundamental no desenvolvimento de aplicações concorrentes em Go.

O Código 4 demonstra um exemplo básico de como utilizar o WaitGroup to aguardar a finalização de goroutines. Inicialmente importa-se os pacotes fmt para formatação de saída e sync para utilização do sync.WaitGroup, que nos permite esperar pela conclusão de múltiplas goroutines.

Na função principal (main), declaramos uma variável do tipo sync.WaitGroup chamada wg. Essa variável é crucial para sincronizar a execução entre a função principal e as goroutines, permitindo que a função principal aguarde a conclusão de todas as goroutines antes de encerrar.

Adicionamos 2 ao contador do WaitGroup, indicando que estamos esperando duas goroutines. Em seguida, criamos duas goroutines anônimas utilizando a palavra-chave go. Cada goroutine executa uma função anônima que imprime uma mensagem específica, como "Hello from Goroutine 1"ou "Hello from Goroutine 2".

Ao utilizar defer wg.Done(), garantimos que o contador do WaitGroup seja decrementado quando cada goroutine termina, permitindo que a função principal saiba quando todas as goroutines concluíram a execução. Chamamos wg.Wait() para bloquear a execução da função principal até que o contador do WaitGroup retorne a zero, indicando que todas as goroutines terminaram.

Código 4 – Exemplo de *WaitGroup*

```go
package main

import (
  "fmt"
  "sync"
)

func main() {
  var wg sync.WaitGroup

  // Adiciona 2 ao WaitGroup para indicar que estamos esperando duas
    goroutines
  wg.Add(2)

  // Goroutine 1
  go func() {
    defer wg.Done() // Decrementa o contador quando a goroutine termina
    printMessage("Hello from Goroutine 1")
  }()

  // Goroutine 2
  go func() {
    defer wg.Done() // Decrementa o contador quando a goroutine termina
    printMessage("Hello from Goroutine 2")
  }()

  // Espera at  que todas as goroutines tenham terminado
  wg.Wait()

  fmt.Println("Main function exiting")
}

func printMessage(message string) {
  fmt.Println(message)
}
```

Por fim, imprimimos uma mensagem indicando que a função main está saindo. A execução das goroutines demonstra a natureza concorrente, e a utilização do WaitGroup garante uma coordenação eficiente entre as diferentes partes do programa, proporcionando um controle ordenado da execução concorrente.

2.3.2.2 *Mutex* e *RWMutex*

Mutex, abreviação de "mutual exclusion", é uma poderosa ferramenta para garantir exclusão mútua em regiões críticas de um programa concorrente. Conforme descrito por Pike (2023h), o Mutex funciona como uma trava que concede acesso exclusivo a recursos compartilhados, prevenindo condições de corrida e garantindo a consistência dos dados em ambientes concorrentes. Ao seguir a convenção estabelecida pelo Go, os desenvolvedores utilizam o Mutex para coordenar o acesso à memória compartilhada, protegendo-a de acessos simultâneos

não seguros. Esta abordagem é crucial para evitar problemas como race conditions e garantir a integridade dos dados em situações de concorrência, proporcionando uma base sólida para o desenvolvimento de sistemas concorrentes confiáveis.

Código 5 – Exemplo de *Mutex*

```go
package main

import (
  "fmt"
  "sync"
)

var count int
var mutex sync.Mutex

func increment() {
  mutex.Lock()
  defer mutex.Unlock()
  count++
  fmt.Printf("Incrementing: %d\n", count)
}

func decrement() {
  mutex.Lock()
  defer mutex.Unlock()
  count--
  fmt.Printf("Decrementing: %d\n", count)
}

func main() {
  var arithmetic sync.WaitGroup

  // Increment
  for i := 0; i < 5; i++ {
    arithmetic.Add(1)
    go func() {
      defer arithmetic.Done()
      increment()
    }()
  }

  // Decrement
  for i := 0; i < 5; i++ {
    arithmetic.Add(1)
    go func() {
      defer arithmetic.Done()
      decrement()
    }()
  }

  arithmetic.Wait()
  fmt.Println("Final Count:", count)
}
```

Na terminologia do modelo de memória do Go, a n-ésima chamada para Unlock

"sincroniza antes"da m-ésima chamada para Lock para qualquer n < m. Uma chamada bem-sucedida para TryLock é equivalente a uma chamada para Lock. Uma chamada mal sucedida para TryLock não estabelece nenhuma relação "sincroniza antes"de forma alguma (PIKE, 2023h).

Para demonstrar o uso de um mutex em Go, confira o Código 5 onde é feito um contador que pode ser acessado por várias goroutines de forma segura utilizando um mutex.

Neste exemplo, temos duas funções, increment e decrement, que modificam a variável count de forma segura usando um mutex. Cada função adquire o mutex com Lock() antes de modificar a variável compartilhada e o libera com Unlock() após a modificação.

Na função main, utilizamos um sync.WaitGroup chamado arithmetic para garantir que a função principal espere a conclusão de todas as goroutines antes de imprimir o valor final de count. As goroutines são criadas para incrementar e decrementar o contador em paralelo. O uso do mutex garante que as operações de leitura e gravação na variável count sejam atomicamente seguras.

As seções críticas, assim denominadas por refletirem um ponto de estrangulamento em um programa, representam áreas onde a concorrência pode gerar conflitos e impactar adversamente o desempenho. Como mencionado por Cox-Buday (2017), o custo associado à entrada e saída dessas seções é significativo, motivando esforços para minimizar o tempo gasto dentro delas. Estratégias eficazes buscam reduzir a seção transversal da seção crítica, e para alcançar esse objetivo, o uso de mutexes se torna essencial.

Uma abordagem para otimizar o acesso concorrente à memória compartilhada é a utilização do sync.RWMutex, conforme proposto por Cox-Buday (2017). O sync.RWMutex compartilha o mesmo propósito fundamental que um Mutex, protegendo o acesso à memória compartilhada, mas oferece uma funcionalidade adicional de controle mais refinado. Ao solicitar um bloqueio para leitura, o RWMutex concede acesso a múltiplos leitores simultâneos, desde que nenhum bloqueio de escrita esteja sendo mantido. Essa característica proporciona uma flexibilidade valiosa, permitindo um equilíbrio entre concorrência e exclusividade no acesso a dados compartilhados. No Código 6 demonstra a utilização do sync.RWMutex para controle de acesso concorrente.

Este código cria três funções: producer, observer, e test. A função producer representa uma goroutine que realiza operações de Lock e Unlock em um Locker fornecido, simulando uma seção crítica. A função observer é uma goroutine que apenas adquire um Lock e o libera imediatamente. A função test simula um teste onde uma quantidade especificada de leitores (observers) e um produtor (producer) operam simultaneamente em uma seção crítica protegida por um sync.Locker (que pode ser um sync.Mutex ou um sync.RWMutex).

2.3.2.3 *Cond*

A implementação de uma variável de condição, como descrita por (**????**) é fundamental para coordenar a execução de goroutines em Go, proporcionando um ponto de encontro para a comunicação entre elas em relação à ocorrência de eventos. Cada variável de condição, representada pela estrutura Cond, está associada a um Locker (frequentemente um Mutex ou RWMutex), que deve ser mantido durante alterações na condição e chamadas ao método Wait. É crucial notar que, conforme as diretrizes apresentadas por Pike (2023h), uma variável de condição não deve ser copiada após o primeiro uso. No contexto do modelo de memória do Go, a Cond estabelece uma ordem de sincronização, indicando que chamadas para Broadcast ou Signal ocorrem "sincroniza antes"de chamadas para Wait que desbloqueiam a execução.

Seguindo as recomendações de Cox-Buday (2017), é destacado que um "evento"refere-se a qualquer sinal arbitrário entre goroutines, indicando apenas a ocorrência de um evento sem carregar informações adicionais. Em muitos casos, aguardar por esses sinais torna-se essencial antes de prosseguir com a execução de uma goroutine. A definição de eventos, conforme Cox-Buday (2017), destaca a simplicidade e eficácia do uso de canais para muitos casos de uso, onde o Broadcast corresponde ao fechamento de um canal, e o Signal equivale ao envio em um canal. Essas abstrações facilitam a comunicação entre goroutines, contribuindo para o desenvolvimento de sistemas concorrentes robustos em Go.

A estrutura sync.Cond em Go é utilizada para implementar variáveis de condição, proporcionando um ponto de encontro para goroutines que desejam esperar ou anunciar a ocorrência de eventos. O Código 7 demonstra o uso prático de sync.Cond. Neste exemplo, a estrutura Data contém um valor inteiro e uma variável de condição (sync.Cond). Duas goroutines são criadas: uma para atualizar o valor de Data a cada 2 segundos e sinalizar a condição, e outra para esperar pelo sinal de atualização e imprimir o valor.

A função sync.NewCond(&sync.Mutex) cria uma nova variável de condição associada a um mutex. O método Signal() é usado para notificar a goroutine adormecida que a condição foi satisfeita. A função Wait() é usada para aguardar até que a condição seja satisfeita, liberando temporariamente o mutex associado.

O uso do sync.Cond facilita a coordenação eficiente entre goroutines, permitindo que elas cooperem na execução de tarefas específicas quando as condições desejadas são atendidas. Este tipo é particularmente útil em casos onde canais podem ser inadequados ou quando se deseja coordenar o comportamento de múltiplas goroutines.

Como ocorre com a maioria das outras funcionalidades do pacote sync, o Cond é mais eficaz quando utilizado em um escopo específico e restrito, ou quando exposto a um escopo mais amplo por meio de um tipo que o encapsula (COX-BUDAY, 2017).

2.3.2.4 *Once*

Como uma nova primitiva introduzida pelo Go, Once é projetado para garantir que uma função seja executada apenas uma vez (TU et al., 2019). Um Once é um objeto que realizará exatamente uma ação e não deve ser copiado após o primeiro uso (PIKE, 2023h). Podemos acompanhar no Código 8 um programa representando a utilização de sync.Once.

Neste exemplo, a função initializer será executada apenas uma vez, independentemente de quantas goroutines chamem once.Do(initializer). Isso é alcançado pelo mecanismo interno do sync.Once, que garante que a função seja executada apenas na primeira chamada e, subsequentemente, seja ignorada.

O uso do sync.Once é especialmente útil quando há a necessidade de realizar uma inicialização única em um programa concorrente, evitando a repetição desnecessária da mesma operação por várias goroutines. Isso é particularmente útil em situações em que a inicialização é custosa ou envolve acesso a recursos compartilhados.

2.3.3 *Channels*

Os canais, primitivas de sincronização em Go inspiradas no CSP de Hoare, são empregados não apenas para sincronizar acesso à memória, mas principalmente para facilitar a comunicação entre goroutines, conforme destacado por Cox-Buday (2017). Funcionando como condutores de fluxo de informações, os canais permitem a transmissão de valores entre partes distintas do programa, proporcionando uma forma eficiente de compor funcionalidades em programas de qualquer tamanho. A comunicação por meio de canais simplifica a interação entre goroutines, permitindo que valores sejam transmitidos e lidos a jusante, sem a necessidade de conhecimento detalhado entre as partes envolvidas. A criação de canais pode ser realizada de maneira concisa com o operador :=, evidenciando a simplicidade e a eficácia dessa primitiva em Go (COX-BUDAY, 2017). Em paralelo, se as goroutines são as atividades concorrentes em um programa Go, os canais representam as conexões entre elas, funcionando como mecanismos de comunicação para a transmissão de valores específicos entre goroutines. Cada canal é associado a um tipo específico de elemento, sendo expresso, por exemplo, como chan int para um canal que transmite valores inteiros. A criação de canais é realizada utilizando a função make, como exemplificado por 'ch := make(chan int)', proporcionando uma abordagem simples e poderosa para facilitar a comunicação efetiva entre as atividades concorrentes.

No Código 9, temos um canal chamado messageChannel que é usado para comunicação entre duas goroutines: uma produtora e outra consumidora. A goroutine produtora envia mensagens para o canal, e a goroutine consumidora as recebe.

A relação dos canais com a sincronização por passagem de mensagem está no fato de que os canais proporcionam um meio seguro e eficiente para que as goroutines se comuniquem trocando mensagens. A sincronização por passagem de mensagem refere-se à troca de dados

entre threads ou processos para coordenar suas atividades. Em Go, os canais desempenham um papel essencial nesse modelo de comunicação, garantindo que a troca de mensagens seja sincronizada e segura para ambientes concorrentes.

Ao utilizar canais, as goroutines podem se comunicar sem a necessidade de mutexes ou outras estruturas de sincronização, simplificando o desenvolvimento de programas concorrentes e garantindo a prevenção de condições de corrida. Isso torna os canais uma ferramenta poderosa para implementar a sincronização por passagem de mensagem em Go.

2.3.4 *Select*

A instrução select em Go desempenha um papel crucial na composição de canais, conectando efetivamente goroutines e permitindo a formação de abstrações mais amplas em um programa concorrente, conforme enfatizado por Cox-Buday (2017). Ao ser a "cola que une os canais,"o select pode ser localmente empregado em uma única função ou tipo, bem como globalmente, na interseção de diversos componentes em um sistema. Além de unir componentes, as instruções select podem ser estrategicamente utilizadas em pontos críticos do programa para integrar canais de forma segura, incorporando conceitos como cancelamentos, limites de tempo, espera e valores padrão.

A funcionalidade do select assemelha-se a um switch, apresentando vários casos e um padrão opcional, conforme destacado por Donovan e Kernighan (2015). Cada caso no select especifica uma comunicação, representando operações de envio ou recebimento em algum canal, junto a um bloco associado de declarações. O select aguarda até que uma dessas comunicações esteja pronta para prosseguir, realizando a comunicação associada ao caso correspondente e executando as declarações relacionadas. Caso nenhum caso esteja pronto, um select sem casos (select) aguarda indefinidamente.

O select funciona como um mecanismo de controle de fluxos concorrentes, proporcionando decisões condicionais em operações de escrita ou leitura em canais, conforme mencionado por Cox-Buday (2017) e Donovan e Kernighan (2015). Assim como no switch, é possível escolher um comportamento padrão através do caso especial default, acionado se todas as operações resultarem em bloqueio. O select possibilita a escrita de fluxos de controle comuns em códigos sequenciais clássicos, enriquecendo códigos concorrentes com a capacidade de tomar decisões em múltiplos fluxos. Essa abordagem flexível e poderosa do select torna-o uma ferramenta essencial na construção de programas concorrentes robustos em Go.

No Código 10, temos duas goroutines produtoras que enviam mensagens para dois canais diferentes (ch1 e ch2). A goroutine consumidora utiliza select para receber mensagens de ambos os canais. O select permite que a goroutine consumidora aguarde a comunicação em qualquer um dos canais prontos para enviar, garantindo que a execução não seja bloqueada.

A relação com canais é fundamental no contexto do select. O select é frequentemente

utilizado em conjunto com canais para coordenar a comunicação entre goroutines. Cada caso dentro do select representa uma possível operação de leitura ou escrita em um canal, e o select aguarda até que uma dessas operações esteja pronta para prosseguir. Isso proporciona uma forma eficiente de lidar com múltiplos canais de forma concorrente, permitindo a tomada de decisões dinâmicas com base na disponibilidade de comunicação em diferentes canais.

2.4 Aplicações Go

Atualmente Go é a décima linguagem mais popular no Github, ocupando também a décima oitava posição quando considerarmos o percentual de crescimento por contribuidores (GITHUB, 2023). As linguagens de programação não estão mais restritas ao âmbito do desenvolvimento de software convencional. Observamos uma equivalência significativa entre as linguagens mais amplamente utilizadas em projetos desenvolvidos em 2023 em comparação com aquelas predominantes no GitHub. Alguns casos atípicos notáveis incluem Kotlin, Rust, Go e Lua, que demonstraram um aumento mais expressivo em popularidade em projetos mais recentes hospedados no GitHub (GITHUB, 2023). No momento dessa pesquisa foram encontrados 566k repositórios escritos em Go (GITHUB, 2024).

Este estudo selecionou três aplicações de mundo real escritas em Go, incluindo um sistema de container (Docker), um banco de dados (CockroachDB) e um sistema de automação e provisionamento de infraestrutura como código (Terraform). Essas aplicações são projetos de código aberto que ganharam ampla utilização em ambientes de data center.

O Docker é um sistema de containerização que permite a criação, o empacotamento e o envio de aplicações como unidades executáveis portáteis. É um dos sistemas de containerização mais populares do mundo, com mais de 48,9 mil estrelas no GitHub. O Docker é mantido ativamente por uma comunidade de desenvolvedores e é usado por empresas de todos os tamanhos (DOCKER, 2023).

O CockroachDB é um banco de dados relacional distribuído que oferece alta disponibilidade, escalabilidade e desempenho. É uma alternativa popular ao PostgreSQL e ao MySQL, e é usado por empresas como Netflix, Spotify e Uber. O CockroachDB tem mais de 36,5 mil estrelas no GitHub e é mantido ativamente pela equipe da Cockroach Labs (LABS, 2023b).

O Terraform é uma ferramenta de automação e provisionamento de infraestrutura como código (IaC) que permite aos desenvolvedores gerenciar recursos de infraestrutura, como servidores, redes e armazenamento, usando um único arquivo de configuração. O Terraform é uma ferramenta popular para DevOps e arquitetos de infraestrutura, e é usado por empresas de todos os tamanhos. O Terraform tem mais de 30 mil estrelas no GitHub e é mantido ativamente pela equipe da HashiCorp (HASHICORP, 2023).

(PRECISA ALTERAR OS VALORES INFORMADOS)

2.5 Trabalhos Relacionados

Código 6 – Exemplo de *RWMutex*

```go
package main

import (
  "fmt"
  "math"
  "os"
  "sync"
  "text/tabwriter"
  "time"
)

func producer(wg *sync.WaitGroup, l sync.Locker) {
  defer wg.Done()
  for i := 5; i > 0; i-- {
    l.Lock()
    l.Unlock()
    time.Sleep(1 * time.Millisecond)
  }
}

func observer(wg *sync.WaitGroup, l sync.Locker) {
  defer wg.Done()
  l.Lock()
  defer l.Unlock()
}

func test(count int, mutex, rwMutex sync.Locker) time.Duration {
  var wg sync.WaitGroup
  wg.Add(count + 1)
  beginTestTime := time.Now()
  go producer(&wg, mutex)
  for i := count; i > 0; i-- {
    go observer(&wg, rwMutex)
  }
  wg.Wait()
  return time.Since(beginTestTime)
}

func main() {
  tw := tabwriter.NewWriter(os.Stdout, 0, 1, 2, ' ', 0)
  defer tw.Flush()
  var m sync.RWMutex

  fmt.Fprintf(tw, "Readers\tRWMutex\tMutex\n")
  for i := 0; i < 5; i++ {
    count := int(math.Pow(2, float64(i)))
    fmt.Fprintf(
      tw,
      "%d\t%v\t%v\n",
      count,
      test(count, &m, m.RLocker()),
      test(count, &m, &m),
    )
  }
}
```

Código 7 – Exemplo de *Cond*

```go
package main

import (
  "fmt"
  "sync"
  "time"
)

type Data struct {
  value int
  cond  *sync.Cond
}

func main() {
  data := &Data{
    value: 0,
    cond:  sync.NewCond(&sync.Mutex{}),
  }

  // Goroutine para atualizar o valor ap s um certo tempo
  go func() {
    for {
      time.Sleep(2 * time.Second)
      data.cond.L.Lock()
      data.value++
      fmt.Println("Valor atualizado:", data.value)
      data.cond.Signal() // Sinaliza que o valor foi atualizado
      data.cond.L.Unlock()
    }
  }()

  // Goroutine para imprimir o valor quando atualizado
  go func() {
    for {
      data.cond.L.Lock()
      data.cond.Wait() // Aguarda o sinal de atualiza  o
      fmt.Println("Valor atual:", data.value)
      data.cond.L.Unlock()
    }
  }()

  // Aguarda indefinidamente
  select {}
}
```

Código 8 – Exemplo de *Once*

```go
package main

import (
  "fmt"
  "sync"
)

func main() {
  var once sync.Once

  // Fun  o que ser  executada apenas uma vez
  initializer := func() {
    fmt.Println("Executando inicializa  o.")
  }

  // Goroutine 1
  go func() {
    once.Do(initializer)
    fmt.Println("Goroutine 1 conclu da.")
  }()

  // Goroutine 2
  go func() {
    once.Do(initializer)
    fmt.Println("Goroutine 2 conclu da.")
  }()

  // Goroutine 3
  go func() {
    once.Do(initializer)
    fmt.Println("Goroutine 3 conclu da.")
  }()

  // Aguarda a conclus o das goroutines
  select {}
}
```

Código 9 – Exemplo de *Channels*

```go
package main

import (
  "fmt"
  "sync"
)

func main() {
  // Criando um canal de comunica  o entre goroutines
  messageChannel := make(chan string)

  // WaitGroup para garantir que todas as goroutines sejam conclu das
    antes do t rmino do programa
  var wg sync.WaitGroup

  // Goroutine produtora
  wg.Add(1)
  go func() {
    defer wg.Done()
    for i := 1; i <= 5; i++ {
      message := fmt.Sprintf("Mensagem %d", i)
      messageChannel <- message // Enviando mensagem para o canal
    }
    close(messageChannel) // Fechando o canal ap s a conclus o do envio
    de mensagens
  }()

  // Goroutine consumidora
  wg.Add(1)
  go func() {
    defer wg.Done()
    for message := range messageChannel {
      fmt.Println("Recebido:", message) // Recebendo mensagem do canal
    }
  }()

  // Aguardando a conclus o de ambas as goroutines
  wg.Wait()
}
```

Código 10 – Exemplo de *Select*

```go
package main

import (
  "fmt"
  "time"
)

func main() {
  ch1 := make(chan string)
  ch2 := make(chan string)

  // Goroutine produtora 1
  go func() {
    for i := 0; i < 5; i++ {
      time.Sleep(time.Second)
      ch1 <- fmt.Sprintf("Mensagem do Canal 1: %d", i)
    }
    close(ch1)
  }()

  // Goroutine produtora 2
  go func() {
    for i := 0; i < 5; i++ {
      time.Sleep(2 * time.Second)
      ch2 <- fmt.Sprintf("Mensagem do Canal 2: %d", i)
    }
    close(ch2)
  }()

  // Goroutine consumidora com select
  for {
    select {
    case msg, ok := <-ch1:
      if ok {
        fmt.Println(msg)
      } else {
        fmt.Println("Canal 1 fechado.")
      }
    case msg, ok := <-ch2:
      if ok {
        fmt.Println(msg)
      } else {
        fmt.Println("Canal 2 fechado.")
      }
    }
  }
}
```

3

Metodologia

4

Próximos Passos

5
Conclusão

texto texto texto

Referências

ASSOCIAÇÃO BRASILEIRA DE NORMAS TÉCNICAS. *NBR 6028*: Resumo - apresentação. Rio de Janeiro, 2003. 2 p. Citado na página 6.

BINET, S. Go-hep: writing concurrent software with ease and go. In: IOP PUBLISHING. *Journal of Physics: Conference Series*. [S.l.], 2018. v. 1085, n. 5, p. 052012. Citado na página 28.

BRYANT, R. E.; O'HALLARON, D. R. *Computer systems: a programmer's perspective*. [S.l.]: Prentice Hall, 2011. Citado 3 vezes nas páginas 20, 21 e 22.

CETIC. *92 milhões de brasileiros acessam a internet apenas pelo telefone celular, aponta TIC Domicílios 2022*. 2022. Disponível em: <https://nic.br/noticia/releases/92-milhoes-de-brasileiros-acessam-a-internet-apenas-pelo-telefone-celular-aponta-tic-domicilios-2022/>. Acesso em: 21 dez 2023. Citado na página 15.

CORMEN, T. et al. *Book: introduction to algorithms*. [S.l.]: MIT Press, 2009. Citado na página 25.

COX-BUDAY, K. *Concurrency in Go: Tools and Techniques for Developers*. [S.l.]: "O'Reilly Media, Inc.", 2017. Citado 12 vezes nas páginas 19, 21, 22, 23, 24, 27, 29, 30, 33, 34, 35 e 36.

DOCKER. *Docker - Build, Ship, and Run Any App, Anywhere*. 2023. Disponível em: <https://www.docker.com/>. Acesso em: 15 dez 2023. Citado 3 vezes nas páginas 16, 17 e 37.

DONOVAN, A. A.; KERNIGHAN, B. W. *The Go programming language*. [S.l.]: Addison-Wesley Professional, 2015. Citado 5 vezes nas páginas 21, 23, 27, 28 e 36.

GITHUB. *The State of Open Source and AI*. 2023. Disponível em: <https://github.blog/2023-11-08-the-state-of-open-source-and-ai/>. Acesso em: 22 dez 2023. Citado na página 37.

GITHUB. *Repositorios Go*. 2024. Disponível em: <https://github.com/search?q=go+language%3AGo&type=repositories&l=Go>. Acesso em: 22 dez 2023. Citado na página 37.

GO. *Effective Go*. 2023. Disponível em: <https://go.dev/doc/effective_go/>. Acesso em: 15 dez 2023. Citado na página 16.

Golang Team. *Add a Partial Deadlock Detector For Go*. 2023. Disponível em: <https://github.com/golang/go/issues/13759>. Acesso em: 21 dez 2023. Citado na página 23.

GOOGLE. *A high performance, open source, general RPC framework that puts mobile and HTTP/2 first*. 2023. Disponível em: <https://github.com/grpc/grpc-go>. Acesso em: 15 dez 2023. Citado na página 16.

HASHICORP. *Terraform*. 2023. Disponível em: <https://github.com/hashicorp/terraform>. Acesso em: 21 dez 2023. Citado na página 37.

HECTANE. *Lightweight SMTP client written in Go*. 2023. Disponível em: <https://github.com/hectane/smtp>. Acesso em: 15 dez 2023. Citado na página 16.

ISO. *ISO/IEC 14882:2011 Information technology — Programming languages — C++*. 3. ed. [S.l.], 2011. Disponível em: <<https://www.iso.org/standard/50372.html>>. Citado na página 16.

LABS, C. *CockroachDB - Distributed SQL for your applications*. 2023. Disponível em: <https://www.cockroachlabs.com/>. Acesso em: 15 dez 2023. Citado na página 17.

LABS, C. *CockroachDB v2.2.3*. 2023. Disponível em: <https://github.com/cockroachdb/cockroach>. Acesso em: 21 dez 2023. Citado na página 37.

LAMPORT, L. Time, clocks, and the ordering of events in a distributed system. In: *Concurrency: the Works of Leslie Lamport*. [S.l.: s.n.], 2019. p. 179–196. Citado na página 19.

LU, S. et al. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In: *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*. [S.l.: s.n.], 2008. p. 329–339. Citado 2 vezes nas páginas 15 e 16.

ORACLE. *JDK 1.1 for Solaris Developer's Guide*. [S.l.]: Sun Microsystems, 2000. Citado na página 16.

PIKE, R. *Go at Google: Language Design in the Service of Software Engineering*. 2012. Disponível em: <https://go.dev/talks/2012/splash.article#TOC_13>. Acesso em: 21 dez 2023. Citado na página 27.

PIKE, R. e. a. *Documentation*. 2023. Disponível em: <https://go.dev/doc/>. Acesso em: 21 dez 2023. Citado na página 26.

PIKE, R. e. a. *Frequently Asked Questions (FAQ)*. 2023. Disponível em: <https://go.dev/doc/faq>. Acesso em: 21 dez 2023. Citado na página 26.

PIKE, R. e. a. *Frequently Asked Questions (FAQ): Does Go have a runtime?* 2023. Disponível em: <https://go.dev/doc/faq#runtime>. Acesso em: 21 dez 2023. Citado na página 26.

PIKE, R. e. a. *Frequently Asked Questions (FAQ): What is the history of the project?* 2023. Disponível em: <https://go.dev/doc/faq#history>. Acesso em: 21 dez 2023. Citado na página 26.

PIKE, R. e. a. *Frequently Asked Questions (FAQ): Why build concurrency on the ideas of CSP?* 2023. Disponível em: <https://go.dev/doc/faq#csp>. Acesso em: 21 dez 2023. Citado na página 26.

PIKE, R. e. a. *Frequently Asked Questions (FAQ): Why goroutines instead of threads?* 2023. Disponível em: <https://go.dev/doc/faq#goroutines>. Acesso em: 21 dez 2023. Citado na página 27.

PIKE, R. e. a. *The Go Programming Language Specification*. 2023. Disponível em: <https://go.dev/ref/spec>. Acesso em: 21 dez 2023. Citado na página 25.

PIKE, R. e. a. *sync package*. 2023. Disponível em: <https://pkg.go.dev/sync@go1.21.5>. Acesso em: 21 dez 2023. Citado 7 vezes nas páginas 26, 29, 30, 31, 33, 34 e 35.

PRANSKEVICHUS, E.; SELIVANOV, Y. *What's New in Python 3.5*. 2015. Disponível em: <<URL>>. Acesso em: 15 dez 2023. Citado na página 16.

STOICA, I. et al. The evolution of multicore processors: A survey of hardware trends and software challenges. *ACM Computing Surveys (CSUR)*, v. 47, p. 1–38, 2014. Citado na página 15.

TANENBAUM, A.; FILHO, N. M. Sistemas operacionais modernos (vol. 4). *Único,(Copyright, 1992)*, p. 152, 2016. Citado 3 vezes nas páginas 20, 21 e 25.

TERRAFORM. *Terraform - Infrastructure as Code*. 2023. Disponível em: <https://www.terraform.io/>. Acesso em: 15 dez 2023. Citado na página 17.

TU, T. et al. Understanding real-world concurrency bugs in go. In: *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. [S.l.: s.n.], 2019. p. 865–878. Citado 6 vezes nas páginas 16, 17, 18, 24, 25 e 35.

YIN, Z. et al. How do fixes become bugs? In: *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. [S.l.: s.n.], 2011. p. 26–36. Citado na página 16.

Apêndices

APÊNDICE A – Quisque libero justo

Texto texto texto

APÊNDICE B – Texto

texto texto texto

Anexos

ANEXO A – Morbi ultrices rutrum lorem.

Texto texto texto

ANEXO B – Cras non urna sed feugiat cum sociis natoque penatibus et magnis dis parturient montes nascetur ridiculus mus

Texto texto texto

ANEXO C – Fusce facilisis lacinia dui

Texto texto texto