UNIVERSIDADE FEDERAL DO RIO DE JANEIRO
INSTITUTO DE COMPUTAÇÃO
CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

PEDRO PAULO MOUSSA DE MEDEIROS

SAFECHANNEL: UMA ABORDAGEM PARA AUMENTAR A CONFIABILIDADE DE CANAIS EM GO

RIO DE JANEIRO
2025

PEDRO PAULO MOUSSA DE MEDEIROS

SAFECHANNEL: UMA ABORDAGEM PARA AUMENTAR A CONFIABILIDADE DE CANAIS EM GO

Trabalho de conclusão de curso de graduação apresentado ao Instituto de Computação da Universidade Federal do Rio de Janeiro como parte dos requisitos para obtenção do grau de Bacharel em Ciência da Computação.

Orientador: Profa. Silvana Rossetto

RIO DE JANEIRO

2025

CIP - Catalogação na Publicação

M488s

Medeiros, Pedro Paulo Moussa de
SafeChannel: uma abordagem para aumentar a confiabilidade de canais em Go / Pedro Paulo Moussa de Medeiros. -- Rio de Janeiro, 2025.
77 f.

Orientadora: Silvana Rossetto.
Trabalho de conclusão de curso (graduação) - Universidade Federal do Rio de Janeiro, Instituto de Computação, Bacharel em Ciência da Computação, 2025.

1. Programação concorrente. 2. Bugs de concorrência. 3. Channel. 4. Go. I. Rossetto, Silvana, orient. II. Título.

Elaborado pelo Sistema de Geração Automática da UFRJ com os dados fornecidos pelo(a) autor(a), sob a responsabilidade de Miguel Romeu Amorim Neto - CRB-7/6283.

PEDRO PAULO MOUSSA DE MEDEIROS

SAFECHANNEL: UMA ABORDAGEM PARA AUMENTAR A CONFIABILIDADE DE CANAIS EM GO

Trabalho de conclusão de curso de graduação apresentado ao Instituto de Computação da Universidade Federal do Rio de Janeiro como parte dos requisitos para obtenção do grau de Bacharel em Ciência da Computação.

Aprovado em 10 de Março de 2025

BANCA EXAMINADORA:

gov.br Documento assinado digitalmente
SILVANA ROSSETTO
Data: 11/03/2025 12:42:33-0300
Verifique em https://validar.iti.gov.br

Silvana Rossetto
D.Sc. (Instituto de Computação - UFRJ)

gov.br Documento assinado digitalmente
HUGO MUSSO GUALANDI
Data: 11/03/2025 12:51:34-0300
Verifique em https://validar.iti.gov.br

Hugo Musso Gualandi
D.Sc. (Instituto de Computação - UFRJ)

gov.br Documento assinado digitalmente
ROGERS REICHE DE MENDONCA
Data: 11/03/2025 13:25:03-0300
Verifique em https://validar.iti.gov.br

Rogers Reiche de Mendonça
M.Sc. (Instituto de Computação - UFRJ)

AGRADECIMENTOS

Agradeço a todo o corpo docente do Instituto de Ciência da Computação da UFRJ, em especial à professora Silvana Rossetto, pela contribuição para o desenvolvimento deste trabalho.

RESUMO

A comunicação entre fluxos de execução concorrentes baseada em troca de mensagens, por meio de *channels*, é um dos principais recursos de concorrência da linguagem Go. Entretanto, diversos estudos indicam que erros no uso de *channels* são uma das principais causas de *bugs* de concorrência em sistemas escritos na linguagem. Esses erros podem ser difíceis de detectar e depurar, impactando a confiabilidade das aplicações. Este trabalho propõe o `SafeChannel`, uma abstração para *channels* em Go, que adiciona mecanismos de segurança e um sistema de notificações para facilitar a detecção de problemas em tempo de execução. A implementação do `SafeChannel` é avaliada por meio de testes funcionais e de desempenho, demonstrando que, embora apresente uma sobrecarga em relação aos *channels* nativos, oferece uma interface mais segura e previsível para a troca de mensagens em sistemas concorrentes.

Palavras-chave: programação concorrente; *bugs* de concorrência; *channel*; Go.

ABSTRACT

Message-passing concurrency, implemented through channels, is one of the core concurrency resources of the Go programming language. However, several studies indicate that errors in the use of channels are among the leading causes of concurrency bugs in Go-based systems. These errors can be difficult to detect and debug, impacting application reliability. This work proposes `SafeChannel`, an abstraction for Go channels that introduces safety mechanisms and a notification system to facilitate runtime issue detection. The implementation of SafeChannel is evaluated through functional and performance testing, demonstrating that, although it introduces overhead compared to native channels, it provides a safer and more predictable interface for message passing in concurrent systems.

Keywords: concurrent programming; concurrency bugs; channel; Go.

LISTA DE ILUSTRAÇÕES

LISTA DE CÓDIGOS

LISTA DE ABREVIATURAS E SIGLAS

CSP	Communicating Sequential Processes
FIFO	First in, First out
SC	SafeChannel

SUMÁRIO

1 INTRODUÇÃO

Computação concorrente, ou computação de fluxos concorrentes, é um campo de estudo que surgiu da necessidade de permitir a execução de múltiplas tarefas de forma simultânea, otimizando o uso de recursos de *hardware* e aumentando a eficiência e capacidade de resposta das aplicações computacionais. A concorrência permite que diversas tarefas compartilhem os recursos limitados de uma máquina, tornando possível o processamento eficiente de várias operações simultaneamente, como mover dados entre a memória e dispositivos de armazenamento enquanto o processador realiza cálculos complexos.

A importância da concorrência se manifesta em vários contextos: desde sistemas operacionais que precisam gerenciar múltiplos processos e *threads* simultaneamente, até sistemas de tempo real e servidores web que atendem diversas requisições de usuários de forma paralela (HERLIHY; SHAVIT, 2008). Além disso, a concorrência se estende a sistemas distribuídos e aplicativos modernos, como sistemas embarcados, onde múltiplas operações precisam ocorrer em paralelo para garantir desempenho e eficiência (GOETZ et al., 2006).

A concorrência desempenha um papel importante na criação de sistemas modernos, mas seu uso também traz desafios significativos para desenvolvedores. Como explica Ben-Ari (2005), a interação entre processos concorrentes torna o desenvolvimento de programas corretos extremamente difícil, mesmo para os problemas mais simples. O autor aponta que a programação concorrente exige novas ferramentas para especificação, desenvolvimento e verificação de programas, pois programadores acostumados a escrever e testar programas sequenciais frequentemente irão se deparar com comportamentos inesperados e complexos ao implementar concorrência, devido às interações entre fluxos de execução concorrentes.

A concorrência introduz complexidades que não existem em programas sequenciais, como problemas de *race conditions* e *deadlocks*, que podem levar a falhas difíceis de reproduzir e diagnosticar. Segundo Herlihy e Shavit (2008), a corretude em programas concorrentes é mais complexa, pois, além de garantir propriedades de segurança, como evitar condições indevidas, é necessário também assegurar propriedades de vivacidade (*liveness*), o que não tem equivalente direto na programação sequencial.

A dificuldade em resolver *bugs* de concorrência é agravada pelo comportamento não determinístico dessas aplicações. Como a execução de aplicações concorrentes pode variar dependendo de fatores como carga de trabalho e ordem de escalonamento, reproduzir e identificar erros específicos torna-se um desafio técnico que muitas vezes exige ferramentas e estratégias de depuração especializadas (HERLIHY; SHAVIT, 2008).

Go, uma linguagem de programação criada pelo Google, foi projetada com foco na concorrência, oferecendo recursos como *goroutines* e *channels* para facilitar o desenvolvimento de *software multi-threaded* de forma eficiente e segura. *Goroutines* são funções que podem ser executadas concorrentemente, representando as unidades de execução em

um programa Go. Já os *channels*, inspirados no modelo CSP (*Communicating Sequential Processes*) de Tony Hoare, atuam como mecanismos de comunicação entre *goroutines*, permitindo a troca de dados e a sincronização de forma segura. Esses mecanismos tornam o Go particularmente adequado para aplicações concorrentes, como servidores web, sistemas distribuídos e programas de processamento paralelo (DONOVAN; KERNIGHAN, 2015).

Embora os *channels* sejam projetados para facilitar a comunicação segura entre *goroutines*, a utilização inadequada desses recursos pode levar a *bugs* de concorrência, como *deadlocks* e *races conditions*. Essas falhas podem ser difíceis de detectar e depurar, especialmente em programas complexos com múltiplas *goroutines* interagindo por meio de *channels* (TU et al., 2019).

Nesse contexto, garantir a segurança e a corretude de operações concorrentes é essencial para o desenvolvimento de sistemas confiáveis. Portanto, encontrar abordagens para evitar esses *bugs* e tornar o processo de desenvolvimento mais seguro e intuitivo é um objetivo importante na área de computação concorrente.

1.1 OBJETIVOS

Este trabalho apresenta um novo pacote desenvolvido em Go para tornar operações sobre *channels* mais seguras, com o objetivo de diminuir a ocorrência de *bugs* de concorrência. A introdução do pacote busca reduzir a necessidade de conhecimento aprofundado sobre os detalhes de baixo nível dos *channels* e oferecer uma interface mais intuitiva e segura para o desenvolvedor. Por meio de um conjunto de funções e estruturas de dados, o *package* visa minimizar as chances de erros comuns na manipulação de *channels*, como o envio de mensagens para um *channel* fechado ou o bloqueio indefinido de uma *goroutine*.

1.2 JUSTIFICATIVA

O desenvolvimento deste *package* é justificado pela crescente popularidade de Go em aplicações que exigem alto desempenho e concorrência, como sistemas distribuídos, servidores *web* e microsserviços. A crescente complexidade dessas aplicações torna a programação concorrente mais desafiadora, aumentando a necessidade de ferramentas e recursos que auxiliem na escrita de código livre de erros.

A programação concorrente em Go tem sido alvo de diversos estudos que analisam as vantagens e desafios do modelo de troca de mensagens e *channels*. Diferentes trabalhos exploram tanto os benefícios quanto as dificuldades inerentes ao uso dessas primitivas e propõem melhorias ou abordagens alternativas para tornar a concorrência mais segura e eficiente.

O estudo "*An Empirical Study of Concurrent Feature Usage in Go*", Tipirneni (2022), analisa o uso de primitivas concorrentes em projetos reais escritos na linguagem, identifi-

cando padrões e dificuldades enfrentadas por desenvolvedores, de modo similar ao que é feito no artigo "*An Empirical Study of Message Passing Concurrency in Go Projects*", de Dilley e Lange (2019). Os estudos revelam que, embora os *channels* sejam amplamente adotados, práticas inadequadas e dificuldades de depuração levam muitos programadores a recorrerem a mecanismos tradicionais de sincronização, como o `mutex` da biblioteca `sync`. Entre os problemas identificados, destacam-se erros decorrentes de *channels* fechados, bloqueios inesperados e dificuldades em coordenar múltiplos *channels* dentro de seleções não determinísticas (através da instrução `select`). Essa observação reforça a necessidade de uma interface que facilite o uso correto de *channels*, reduzindo a propensão a erros, e que simplifique a depuração dessas aplicações.

O artigo "*Jump over Golang Channels*", Najafizadeh e Javadi (2023), propõe uma implementação alternativa de *channels* que permite priorizar as mensagens em vez de seguir a ordem padrão *first-in first-out* (FIFO). Com esse mecanismo, mensagens de alta prioridade podem pular mensagens de menor prioridade na fila, garantindo um processamento mais eficiente para determinados cenários. Essa abordagem contrasta com a do `SafeChannel`, que mantém a semântica original de *channels* do Go, mas adiciona camadas de segurança para evitar erros comuns, como *panics* e bloqueios. Enquanto o *Jump over Golang Channels* busca melhorar a ordenação e o agendamento das mensagens, o `SafeChannel` foca na previsibilidade e confiabilidade da comunicação concorrente, evitando condições de corrida e bloqueios inesperados. Ambos os trabalhos demonstram como a infraestrutura de *channels* pode ser expandida para atender às necessidades específicas de diferentes aplicações.

Por fim, o artigo de opinião "*Go Channels Are Bad and You Should Feel Bad*", Olio (2016), apresenta uma visão crítica sobre o uso de *channels* em Go, argumentando que, embora o modelo de troca de mensagens seja elegante, sua implementação na prática pode levar a dificuldades de manutenção e *bugs* difíceis de depurar. O autor sugere que, em muitos casos, o uso de `mutex` e outras primitivas de sincronização tradicionais pode ser mais previsível e eficiente. Essa crítica reforça a importância de ferramentas como o *SafeChannel*, que não elimina o uso de *channels*, mas adiciona verificações que tornam sua utilização mais segura e previsível para desenvolvedores.

Esses trabalhos ilustram a complexidade da programação concorrente em Go e a necessidade de aprimoramentos nos mecanismos disponíveis. O *SafeChannel* se insere nesse contexto como uma solução que preserva a filosofia de troca de mensagens, ao mesmo tempo em que melhora a segurança e a experiência do desenvolvedor, tornando o uso de *channels* mais confiável e menos propenso a erros.

1.3 ORGANIZAÇÃO DO TEXTO

Este trabalho está estruturado em cinco capítulos, que abordam desde os conceitos fundamentais de concorrência na linguagem Go até a apresentação e análise da solução proposta. No Capítulo 2, são explorados os fundamentos da programação concorrente na linguagem Go, detalhando seus principais mecanismos, como *goroutines* e *channels*, e discutindo as primitivas de sincronização e comunicação. Em seguida, abordaremos os *bugs* de concorrência mais comuns, incluindo suas causas, impactos e exemplos práticos de situações em que esses problemas ocorrem. No Capítulo 3, é apresentada a proposta de solução: o *Safechannel*. Este capítulo explica a estrutura, funcionalidades e mecanismos internos da implementação que permitem seu funcionamento seguro. Serão destacados os principais componentes do projeto, suas interações e o porquê das escolhas técnicas. No Capítulo 4, uma avaliação do *package* é apresentada, com os casos de teste criados e uma comparação de desempenho entre os *channels* nativos e o *Safechannel*. Por fim, no Capítulo 5, são apresentadas as conclusões do trabalho, onde serão discutidos os resultados obtidos, as limitações encontradas e as possibilidades de melhorias e expansões futuras para a solução.

2 PROGRAMAÇÃO CONCORRENTE COM A LINGUAGEM GO

Segundo o livro *The Go Programming Language*, de Donovan e Kernighan (2015), Go é uma linguagem de programação criada por Robert Griesemer, Rob Pike e Ken Thompson na Google, projetada para oferecer uma combinação de simplicidade, eficiência e confiabilidade. Apresentada ao público em 2009, Go foi desenvolvida com o objetivo de proporcionar uma sintaxe expressiva e acessível, mantendo o foco na construção de programas eficientes, tanto em termos de desempenho em tempo de execução quanto na velocidade de compilação, reduzindo a sobrecarga de recompilação e otimizando o gerenciamento de dependências.

Embora Go compartilhe semelhanças com C em termos de sintaxe básica, ela incorpora conceitos de diversas linguagens modernas, eliminando características que podem gerar complexidade ou comprometer a estabilidade do código. Um de seus diferenciais é o suporte nativo à concorrência, por meio de primitivas como *goroutines* e *channels*, que permitem o desenvolvimento de programas concorrentes de modo um pouco mais simples e direto do que linguagens predecessoras como C. Go oferece também gerenciamento automático de memória (coleta de lixo), o que reduz a necessidade de controle manual por parte do programador.

Go foi concebida para resolver problemas comuns, como gerenciar *threads* e *locks* explicitamente. A concorrência está no núcleo da filosofia de design de Go, possibilitando que múltiplas tarefas sejam executadas simultaneamente com eficiência. Em Go, a concorrência é tratada como uma funcionalidade central, e não como uma biblioteca externa. Os principais mecanismos que suportam isso são as *goroutines* e os *channels*. O modelo CSP serviu como base para a implementação da concorrência em Go. Ele propõe que processos concorrentes devem se comunicar exclusivamente por troca de mensagens, ao invés de compartilhar diretamente a memória. Em Go, as *goroutines* representam os fluxos de execução independentes, enquanto os *channels* são o meio de comunicação entre eles. Essa abordagem tem como objetivo reduzir o risco de erros associados ao acesso à memória compartilhada, como *race conditions*.

2.1 MECANISMOS DE CONCORRÊNCIA EM GO

Nesta seção, serão abordadas as principais ferramentas fornecidas pela linguagem Go para o desenvolvimento concorrente, incluindo as *goroutines*, que permitem a execução de funções em paralelo, e o modelo de troca de mensagens por meio de *channels*, que facilita a comunicação entre *goroutines*. Serão discutidos também os axiomas que regem o comportamento dos *channels*, bem como a instrução *select*. Por fim, será feita uma menção aos mecanismos de sincronização baseados em memória compartilhada, que complementam o

modelo de concorrência da linguagem.

2.1.1 Goroutines

Segundo Soares (2019), as *goroutines* são a unidade básica de execução concorrente no Go. Criadas e gerenciadas pelo escalonador do *runtime* da linguagem, elas são projetadas para serem leves e eficientes. Uma *goroutine* pode ser iniciada precedendo uma chamada de função com a diretiva `go`. Esse mecanismo abstrai a complexidade associada à criação e ao gerenciamento manual de *threads*, oferecendo ao programador uma ferramenta simples e poderosa para paralelizar tarefas.

Uma das características que tornam as *goroutines* mais leves do que as *threads* tradicionais é o tamanho inicial reduzido de sua pilha, que cresce dinamicamente conforme necessário. Isso permite que centenas ou até milhares de *goroutines* coexistam em um único programa, enquanto o escalonador do Go distribui sua execução de maneira eficiente entre os núcleos disponíveis.

Por serem valores de primeira classe em Go, funções podem ser passadas como argumentos ou atribuídas a variáveis. Isso permite criar *goroutines* a partir de funções anônimas, como ilustrado no código abaixo:

Código 1 – Exemplo de função anônima concorrente

```
func processData() {
  // do something
}

func main() {
  go func() {
    for {
      fmt.Println(" working ... ")
      time.Sleep(30 * time.Second)
    }
  }()
  processData()
}
```

No código 1, uma *goroutine* é criada para executar uma função anônima que imprime uma mensagem a cada 30 segundos. Essa *goroutine* continuará em execução enquanto o programa principal estiver ativo, mas será finalizada assim que o fluxo principal da função `main` for encerrado. Esse comportamento reflete o modelo de execução em que todas as *goroutines* associadas a um programa terminam junto com o processo principal.

O uso de *goroutines* simplifica a programação concorrente, eliminando a necessidade de

gerenciar diretamente *threads*, enquanto o *runtime* da linguagem se encarrega de escalonar e otimizar sua execução.

2.1.2 Channels

No modelo de troca de mensagens do Go, as *goroutines* comunicam-se e sincronizam-se umas com as outras por meio de *channels*. Os *channels* atuam como condutores que permitem o envio e o recebimento de dados entre as *goroutines* de maneira segura, reduzindo os riscos de inconsistências e problemas como *race conditions*. Existem dois tipos de *channels*, buferizados e não buferizados, sendo os não buferizados bloqueantes e os buferizados não bloqueantes (enquanto houver espaço no *buffer* para o envio, ou se tiver algo preenchido no *buffer* para recebimento).

Channels são criados utilizando o comando `make`, que aloca um determinado tipo de dado na memória e retorna a sua referência. *Channels* oferecem dois tipos de operações: envios e recebimentos (semelhante a uma comunicação interprocessos). Utiliza-se o operador `<-` para executar estas operações (SOARES, 2019).

Os *channels* em Go possuem propriedades fundamentais (axiomas) que definem seu comportamento no contexto de sincronização entre *goroutines*. Essas propriedades servem como diretrizes importantes para o uso correto dessas primitivas (CHENEY, 2014):

- **Enviar em um *channel* não buferizado bloqueia até que outra *goroutine* esteja pronta para receber.** O envio em um *channel* não buferizado só é concluído quando o dado é consumido por outra *goroutine*.
- **Enviar em um *channel* buferizado cheio bloqueia até que outra *goroutine* esteja pronta para receber.** O envio em um *channel* buferizado cujo *buffer* está cheio só é concluído quando uma mensagem é consumida por outra *goroutine*.
- **Receber de um *channel* bloqueia até que outra *goroutine* esteja pronta para enviar.** Similarmente, o recebimento é suspenso até que haja um dado disponível no *channel*.
- **Enviar para um *channel* nulo bloqueia indefinidamente.** Um *channel* nulo (`nil`) não possui capacidade de envio ou recebimento, e qualquer operação sobre ele resulta em bloqueio eterno.
- **Receber de um *channel* nulo bloqueia indefinidamente.** Assim como no caso anterior, o recebimento de um *channel* nulo também resulta em bloqueio eterno.
- **Enviar para um *channel* fechado resulta em *panic*.** Após um *channel* ser fechado com a função `close()`, tentativas de enviar dados a ele provocam um erro fatal na execução do programa.

- **Receber de um *channel* fechado retorna imediatamente o valor zero do tipo.** Após o fechamento, as operações de recebimento não bloqueiam, mas retornam o valor padrão (*zero value*) do tipo de dado associado ao *channel*.

Esses axiomas garantem previsibilidade e segurança no uso dos *channels*.
A simplicidade do uso de *channels* em Go é ilustrada no exemplo a seguir:

Código 2 – Exemplo de comunicação entre goroutines

```
package foo

import "fmt"

func foo() {
  ch := make(chan int)

  go func() {
    ch <- 42
  }()

  value := <-ch
  fmt.Println(value)
}
```

No código 2, uma *goroutine* anônima envia o valor `42` para o *channel*, enquanto a *goroutine* principal o consome. O uso do *channel* não-buferizado garante que as operações sejam sincronizadas: a *goroutine* emissora é bloqueada até que a *goroutine* receptora consuma o valor.

2.1.3 Select

Go também introduz a instrução `select`, que é um mecanismo de controle de fluxos concorrentes, sendo essencial para lidar com múltiplos *channels* simultaneamente. O comportamento é similar ao de um `switch` (em C), que permite um fluxo condicional de múltiplas opções. A diferença é que, no `select`, as condições são operações de escrita ou de leituras em um *channel* qualquer (SOARES, 2019). O `select` permite que uma *goroutine* monitore vários *channels* e execute operações em qualquer um que esteja pronto, de forma bloqueante ou não. Um exemplo é o seguinte:

Código 3 – Exemplo de uso do select

```
package foo

import "fmt"

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

  go func() { ch1 <- "Mensagem do canal 1" }()
  go func() { ch2 <- "Mensagem do canal 2" }()

  select {
  case msg1 := <-ch1:
    fmt.Println(msg1)
  case msg2 := <-ch2:
    fmt.Println(msg2)
  }
}
```

No código 3, o `select` aguarda até que um dos *channels* esteja pronto para enviar dados, processando a mensagem correspondente. Esse mecanismo é útil para cenários onde múltiplas fontes de dados podem ser acessadas simultaneamente, mas com tempos de resposta imprevisíveis.

Como também no `switch`, é possível escolher um comportamento padrão, a partir do caso especial `default`, sendo escolhido se as operações de todos os casos resultarem num bloqueio. No caso de canal buferizado, significa que não possui nenhum valor no *buffer* em caso de leitura, ou não possui espaço em caso de escrita. Para o canal não buferizado, significa que não há nenhuma outra *goroutine* para receber o dado caso a condição seja um envio, ou nenhuma outra *goroutine* para enviar o dado caso a condição seja um recebimento.

2.1.4 Mutex, RWMutex, WaitGroup e Cond

Go também suporta o modelo de concorrência baseado em memória compartilhada. Nesse paradigma, múltiplas *goroutines* podem se comunicar lendo e escrevendo em variáveis compartilhadas. Contudo, para evitar inconsistências e garantir a segurança dos dados, são necessários mecanismos de sincronização, como os oferecidos pela biblioteca `sync`, conforme documentado na bilbioteca *sync* (SYNC PACKAGE, 2025). Embora eficiente em algumas situações, esse modelo exige maior atenção para evitar erros, como *deadlocks* ou *race conditions*, que podem ocorrer se os mecanismos de sincronização não

forem usados corretamente.

Entre os mecanismos mais relevantes estão:

- **Mutex e RWMutex.** O `Mutex` é a primitiva mais simples para sincronização em memória compartilhada. Ele permite que apenas uma *goroutine* tenha acesso exclusivo ao recurso bloqueado. Já o `RWMutex` é uma extensão que diferencia entre bloqueios de leitura e escrita, permitindo concorrência em operações de leitura. No entanto, a prioridade dada a bloqueios de escrita sobre os de leitura pode causar *deadlocks* em certas situações.

- **WaitGroup.** O `WaitGroup` é uma estrutura que facilita a sincronização de múltiplas *goroutines*, aguardando que todas concluam suas tarefas antes de prosseguir. Sua correta utilização requer que o número de *goroutines* seja definido com `Add()` antes de iniciar as operações assíncronas e que a chamada `Wait()` ocorra apenas após todas terem sido adicionadas.

- **Cond.** O `Cond` é utilizado para coordenar a execução entre *goroutines*, permitindo que uma *goroutine* notifique outras quando determinadas condições forem atendidas. Sua implementação requer o uso de um `Mutex` subjacente para garantir a segurança nas notificações e no acesso aos dados compartilhados.

Essas primitivas requerem cuidado em sua utilização, pois erros, como bloqueios duplos, ordem inadequada de *locks* ou sinais ausentes, podem levar a problemas como *deadlocks*, *starvation* e *data race*.

A escolha entre mecanismos de comunicação via troca de mensagens ou memória compartilhada depende das necessidades específicas da aplicação. Enquanto a troca de mensagens, com *goroutines* e *channels*, oferece maior segurança e simplicidade, a memória compartilhada pode ser mais eficiente em cenários que exigem compartilhamento de grande volume de dados entre *goroutines*.

2.2 BUGS DE CONCORRÊNCIA

Esta seção apresenta alguns dos principais *bugs* de concorrência e suas implicações no desenvolvimento de sistemas. Inicialmente, são abordados conceitos fundamentais relacionados a problemas de sincronização e comunicação entre fluxos concorrentes. Na sequência, é discutido um estudo empírico sobre *bugs* de concorrência na linguagem Go, analisando sua categorização e impactos. Ao final, são exploradas possíveis soluções e estratégias para mitigar esses problemas.

2.2.1 Data Race

Data race, ou corrida de dados, ocorre quando vários fluxos de execução acessam e manipulam os mesmos dados simultaneamente, e o resultado da execução depende da ordem específica em que esses acessos ocorrem (SILBERSCHATZ; GALVIN; GAGNE, 2018; NETZER; MILLER, 1994).

Código 4 – Exemplo de corrida de dados em Go

```go
package main

import (
  "fmt"
  "sync"
)

func main() {
  var counter int
  var wg sync.WaitGroup

  wg.Add(2)

  go func() {
    defer wg.Done()
    for i := 0; i < 5; i++ {
      counter++ // Write access
    }
  }()

  go func() {
    defer wg.Done()
    for i := 0; i < 5; i++ {
      fmt.Println("Counter:", counter) // Read access
    }
  }()

  wg.Wait()
}
```

No código 4, temos duas *goroutines* acessando simultaneamente a variável compartilhada `counter`. Uma delas incrementa o valor dessa variável, enquanto a outra lê e imprime o valor atual. No entanto, não há nenhuma sincronização entre essas operações. Isso resulta em uma corrida de dados, já que a execução não garante que as operações de leitura e escrita sejam atômicas ou ordenadas.

Por exemplo, uma leitura pode ocorrer enquanto outra *goroutine* está no meio de uma operação de escrita. Para corrigir esse problema, seria necessário usar mecanismos de sincronização para garantir que as leituras e escritas sejam realizadas de forma completa e na ordem esperada.

2.2.2 Order Violation

Order violation, ou violação de ordem, ocorre quando a sequência de execução esperada entre duas operações não é respeitada. Por exemplo, em um fluxo de execução, o acesso A deve sempre preceder o acesso B para manter a consistência dos dados. No entanto, em sistemas concorrentes, essa ordem nem sempre é garantida, especialmente quando múltiplos fluxos estão interagindo com os mesmos recursos sem mecanismos adequados de sincronização (LU et al., 2008).

Esse problema pode resultar em estados inesperados ou mesmo inconsistentes na aplicação, já que a sequência de operações observada em tempo de execução pode diferir daquela projetada pelo desenvolvedor.

Código 5 – Exemplo de violação de ordem

```
package main

import (
  "fmt"
  "sync"
)

var data string

func main() {
  var wg sync.WaitGroup
  wg.Add(2)

  go func() {
    defer wg.Done()
    fmt.Println("Sending data...")
    data = "Important information"
  }()

  go func() {
    defer wg.Done()
    // Assumes data has already been assigned
    fmt.Println("Data received:", data)
  }()

  wg.Wait()
}
```

No código 5, não há garantia de que a *goroutine* responsável pela inicialização dos dados será executada antes da *goroutine* que os utiliza. Isso caracteriza uma violação de ordem, onde a sequência esperada de inicialização e uso não é respeitada.

Como resultado, a segunda *goroutine* pode tentar acessar a variável `data` antes que seu valor seja definido. Para corrigir esse problema, seria necessário um mecanismo de sincronização, como uma variável de condição (`sync.Cond`) ou um *channel*, para assegurar que a inicialização seja concluída antes do acesso aos dados.

2.2.3 Atomicity Violation

Atomicity violation, ou violação de atomicidade, é uma propriedade fundamental para a consistência de estados em sistemas concorrentes. Ela assegura que, quando um fluxo modifica o estado da aplicação, outros fluxos só consigam observar o estado antes ou

depois da modificação, nunca estados intermediários.

Uma violação de atomicidade (*atomicity violation*) ocorre quando a garantia dessa propriedade falha, permitindo que um outro fluxo veja modificações parciais ou estados inconsistentes durante a execução. Essas violações geralmente surgem em operações que deveriam ser tratadas como uma unidade indivisível, mas que são executadas de forma fragmentada por múltiplos fluxos (LU et al., 2008; ARPACI-DUSSEAU; ARPACI-DUSSEAU, 2018).

Código 6 – Exemplo de violação de atomicidade

```go
package main

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

var balance int
var mutex sync.Mutex

func main() {
  var wg sync.WaitGroup

  balance = 100
  wg.Add(2)

  go func() {
    defer wg.Done()
    withdraw(50) // Expected to withdraw $50
  }()

  go func() {
    defer wg.Done()
    withdraw(70) // Expected to withdraw $70
  }()

  wg.Wait()
  fmt.Println("Final Balance:", balance)
}

func withdraw(amount int) {
  if balance >= amount {
    mutex.Lock()
    balance -= amount
    mutex.Unlock()
    fmt.Println("Withdrawal successful:", amount)
  } else {
    fmt.Println("Insufficient funds for:", amount)
  }
}
```

No código 6, a verificação (`if balance >= amount`) e a atualização (`balance -=`

`amount`) não ocorrem de forma atômica, pois o bloqueio do *mutex* é usado apenas para proteger a operação de escrita. Como consequência, duas goroutines podem ler o mesmo valor de saldo e concluir que a transação é válida, resultando em múltiplos saques além do saldo real disponível. Logo, ambas conseguem subtrair os valores, resultando em um saldo incorreto ou até negativo.

Por exemplo, se o saldo inicial for $100, uma *goroutine* pode tentar sacar $50 e outra $70 ao mesmo tempo. Ambas passam na verificação, pois ambas leem o saldo como $100 antes que qualquer atualização ocorra. Para corrigir esse problema, seria necessário garantir que tanto a verificação quanto a atualização ocorressem dentro do escopo protegido pelo *mutex*, impedindo que outra *goroutine* altere o saldo até que a operação anterior seja totalmente concluída.

2.2.4 Deadlock & Starvation

Um *deadlock* ocorre quando um conjunto de fluxos de execução fica preso em um estado onde todos estão aguardando eventos que só podem ser desencadeados por outro fluxo de execução do mesmo conjunto. Esse comportamento depende do escalonamento dos fluxos de execução pelo sistema operacional, o que torna a detecção e a reprodução de *deadlocks* difíceis durante o desenvolvimento e teste (SILBERSCHATZ; GALVIN; GAGNE, 2018).

Já *starvation* ocorre quando um fluxo de execução é constantemente impedido de progredir devido à priorização de outras tarefas. Embora menos prejudicial que o *deadlock*, a *starvation* pode levar a degradações significativas de desempenho.

2.3 ESTUDO SOBRE BUGS DE CONCORRÊNCIA EM GO

O artigo "*Understanding Real-World Concurrency Bugs in Go*", (TU et al., 2019), realiza um estudo empírico abrangente sobre 171 *bugs* de concorrência encontrados em seis aplicativos de código aberto escritos em Go, incluindo Docker, Kubernetes e gRPC. Este estudo categoriza os *bugs* em duas dimensões principais: comportamento e causa. No que diz respeito ao comportamento, os *bugs* podem ser bloqueantes (*blocking bugs*), quando uma ou mais *goroutines* ficam travadas aguardando recursos indisponíveis, ou não bloqueantes (*non-blocking bugs*), quando as *goroutines* completam suas tarefas, mas produzem resultados incorretos. Em termos de causa, os *bugs* decorrem de erros na proteção de memória compartilhada ou na troca de mensagens.

O estudo mostra que a ocorrência de *bugs* de concorrência em Go por meio de troca de mensagens é tão frequente quanto — e, em alguns casos, até mais problemática — do que com o uso de memória compartilhada. Por exemplo, aproximadamente 58% dos *bugs* bloqueantes analisados têm origem em problemas relacionados à troca de mensagens.

Um dos fatores que contribuem para essa alta incidência de *bugs* é a relativa novidade e especificidade dos mecanismos de troca de mensagens em Go. A criação de *goroutines* com

funções anônimas, o uso de *channels* buferizados e não buferizados, e o não determinismo introduzido pela instrução `select` são algumas das características que, apesar de projetadas para facilitar a programação concorrente, acabam se tornando fontes frequentes de erros devido ao uso inadequado ou à compreensão insuficiente de suas implicações.

Devido a isso, embora a troca de mensagens forneça um modelo simples e seguro para comunicação entre *goroutines*, sua eficácia depende de um entendimento profundo por parte dos programadores, não apenas dos mecanismos de troca de mensagens, mas também das demais ferramentas de sincronização oferecidas por Go.

Os exemplos de *bugs* (códigos 7, 8, 9, 10, 11) que serão descritos a seguir foram retirados do trabalho de Tu et al.

2.3.1 Bugs Bloqueantes

Os *bugs* bloqueantes representam 85 dos casos analisados no estudo e ocorrem quando operações realizadas por *goroutines* ficam permanentemente travadas, devido à indisponibilidade dos recursos necessários para sua continuidade. Apesar da crença de que a passagem de mensagens é menos suscetível a erros, o estudo revelou que 58% dos *bugs* bloqueantes são causados por problemas nesse mecanismo, superando os 42% atribuídos à memória compartilhada (TU et al., 2019).

Entre os erros mais comuns no uso de memória compartilhada estão:

- Uso inadequado de `Mutex`, como bloqueio duplo, esquecimento de desbloquear ou aquisição de bloqueios em ordens conflitantes;
- Problemas específicos da implementação do Go em `RWMutex`, onde a prioridade dada aos bloqueios de escrita sobre os de leitura pode resultar em *deadlocks* mútuos;
- Mau uso de variáveis `WaitGroup` e `Cond`, onde *goroutines* ficam presas em chamadas `Wait()` sem receber os sinais necessários para prosseguir.

No caso da passagem de mensagens, os erros mais frequentes incluem a falta de envio ou recebimento em *channels*, fechamento inadequado de *channels* e a combinação de *channels* com outras primitivas de bloqueio, como `Mutex`. Esses problemas são exemplificados no Kubernetes, onde *goroutines* permanecem bloqueadas em *channels* não consumidos após o término da *goroutine* principal (TU et al., 2019).

Código 7 – Exemplo de bug bloqueante (memória compartilhada)

```
var group sync.WaitGroup

group.Add(len(pm.plugins))

for _, p := range pm.plugins {
  go func(p *plugin) {
    defer group.Done()
  }
  group.Wait() // blocks
}
```

No código 7, o *bug* ocorre no uso da variável compartilhada do tipo `WaitGroup`. O `Wait()` só pode ser desbloqueado quando `Done()` for chamado `len(pm.plugins)` vezes, já que `len(pm.plugins)` é usado como parâmetro na chamada do `Add()`. Contudo, o `Wait()` é chamado dentro do *loop*, de modo que bloqueia a criação da *goroutine* em iterações futuras, e bloqueia a invocação do `Done()` dentro de cada *goroutine* criada.

Código 8 – Exemplo de bug bloqueante (troca de mensagem)

```
func goroutine1() {
  m.Lock()
  ch <- request //blocks
  m.Unlock()
}

func goroutine2() {
  for {
    m.Lock() //blocks
    m.Unlock()
    request <- ch
  }
}
```

No código 8, a `goroutine1` é bloqueada no envio da requisição ao *channel* `ch`, enquanto a `goroutine2` é bloqueada no `m.Lock()`.

2.3.2 Bugs não Bloqueantes

Os *bugs* não bloqueantes representam 86 casos, dos quais 80% decorrem de falhas na (ou falta de) proteção ao acesso de memória compartilhada, e ocorrem quando as *gorou-*

tines concluem sua execução, mas deixam o programa em estados inconsistentes. Esses *bugs* são frequentemente associados à violação de atomicidade, erros na ordem de execução e corrida de dados. O uso inadequado de funções anônimas, por exemplo, contribui significativamente para esses problemas. Como essas funções têm acesso a variáveis locais compartilhadas, podem gerar corrida de dados entre *goroutines* criadas a partir da mesma função (TU et al., 2019).

Outro exemplo notável de *bug* não bloqueante é o uso incorreto de `WaitGroup`, quando a função `Add()` é chamada após `Wait()`, quebrando a sincronização entre as *goroutines*. Além disso, o uso inadequado de *channels* em combinação com `select`, devido à natureza não determinística dessa construção, também é uma causa comum.

Código 9 – Exemplo de condição de corrida (memória compartilhada)

```
for i := 17; i <= 21; i++ { // write
  go func() { /* Create a new goroutine */
    apiVersion := fmt.Sprintf("v1.%d", i) // read
  // ...
  }()
}
```

No código 9, a variável local `i` é compartilhada entre a *goroutine* pai e as *goroutines* que ela cria. A intenção do programador era que cada *goroutine* filha usaria um valor `i` distinto para inicializar a *string* `apiVersion`. Porém, os valores de `apiVersion` são não-determinísticos no programa. Por exemplo, se as *goroutines* filhas iniciam após o final do *loop* da *goroutine* pai, todos valores de `apiVersion` serão iguais a 'v1.21'. O programa produzirá o resultado desejado apenas se cada *goroutine* filha inicializar a *string* `apiVersion` imediatamente após sua criação, e antes da variável compartilhada `i` receber um novo valor.

Código 10 – Exemplo de bug não bloqueante (troca de mensagem)

```
select {
  case <- c.closed:
  default:
    close(c.closed)
}
```

Causado pela violação da regra na qual um *channel* pode ser fechado apenas uma vez, o *bug* presente no código 10 ocorre quando o caso `default` é executado por mais de uma das *goroutines* que estão executando o código.

Código 11 – Exemplo de bug não bloqueante (troca de mensagem)

```
ticker := time.NewTicker()

for {
  f()
  select {
    case <- stopCh:
      return
    case <- ticker:
  }
}
```

Outro tipo de *bug* de concorrência acontece durante a utilização de *channel* e `select` em conjunto. Em Go, quando múltiplas mensagens são recebidas por um `select`, não existe garantia de qual será processada primeiro. No código 11, o *loop* monitora dois eventos: a execução de uma função pesada `f()` sempre que o *ticker* dispara (caso 2), e a interrupção do *loop* ao receber uma mensagem pelo canal `stopCh` (caso 1).

O problema ocorre quando mensagens de ambos os casos chegam simultaneamente. O `select` pode escolher qualquer um dos dois casos, devido à sua natureza não determinística. Se o `select` optar por executar o caso 2, a função `f()` será chamada desnecessariamente mais uma vez, mesmo que o *loop* devesse ser interrompido por causa da mensagem recebida em `stopCh`.

2.3.3 Impacto e Soluções para Bugs de Concorrência

Os impactos dos *bugs* de concorrência em sistemas escritos em Go podem ser severos. Bloqueios (*deadlocks*) podem paralisar serviços, enquanto erros de corrida de dados podem corromper estados críticos, levando a falhas graves nos sistemas finais. O estudo sugere que soluções automatizadas para detecção e correção de *bugs* podem ser viáveis, dado que muitas correções seguem padrões previsíveis, como ajustes em operações de sincronização ou uso de primitivas adequadas, como `Mutex` ou *channels* (TU et al., 2019).

Por fim, ferramentas de detecção existentes, como o *deadlock detector* embutido no *runtime* do Go e o detector de *data races*, mostraram-se limitadas na identificação de muitos dos *bugs* analisados. Isso reforça a necessidade de desenvolver técnicas mais eficazes, combinando análises estáticas e dinâmicas para lidar com os desafios únicos da concorrência em Go (TU et al., 2019).

A abordagem deste trabalho, no entanto, foca em aumentar a robustez e a confiabilidade do mecanismo de troca de mensagens, visando reduzir a ocorrência de *bugs* de concorrência. Para isso, propõe-se uma extensão da primitiva de comunicação *channel*, com mecanismos adicionais de validação e segurança. Essa solução busca minimizar os

riscos associados ao uso incorreto dessas primitivas, visto que grande parte dos erros é causado pela falta de familiaridade dos desenvolvedores com os mecanismos de troca de mensagens oferecidos pela linguagem, oferecendo uma interface mais segura e intuitiva para desenvolvedores.

3 SAFECHANNEL

Nesta seção, é apresentada a solução desenvolvida para aprimorar a segurança e a previsibilidade na comunicação por meio de *channels* em Go. Inicialmente, são discutidos os desafios na troca de mensagens e a abordagem adotada na proposta. Em seguida, é descrita a estrutura interna do `SafeChannel` e suas principais funcionalidades, assim como os mecanismos de notificações e as estratégias empregadas para prevenir erros comuns. É apresentado também um exemplo prático para ilustrar a aplicação da solução.

3.1 DEFINIÇÃO DO PROBLEMA

O uso de *channels* no Go, embora poderoso e essencial para a concorrência baseada em troca de mensagens, pode introduzir riscos significativos quando utilizado de maneira inadequada. Problemas como *panics* ao enviar dados para *channels* fechados, bloqueios indefinidos em operações sobre *channels* nulos, fechamento duplo de *channels*, e dificuldades em gerenciar o estado dos *channels* em sistemas complexos são desafios recorrentes. Esses erros, quando não tratados, podem levar a falhas críticas, impactando a estabilidade e a confiabilidade das aplicações.

Além disso, a detecção de *bugs* em sistemas concorrentes que utilizam *channels* é particularmente desafiadora. Ferramentas existentes, como o detector de *deadlocks* integrado no *runtime* do Go, mostram-se limitadas, pois identificam apenas situações onde todas as *goroutines* estão bloqueadas, ignorando cenários mais complexos, como bloqueios mútuos envolvendo *channels* e outras primitivas de sincronização. De forma semelhante, o detector de *data races* depende de *interleavings* específicos de execução para detectar *race conditions*, o que pode deixar erros críticos sem identificação durante o desenvolvimento. *Interleavings* referem-se às diferentes ordens possíveis de execução das operações concorrentes, ou seja, às formas como as *goroutines* podem ser intercaladas ao acessar recursos compartilhados. Como a execução não é determinística, alguns *interleavings* podem expor *race conditions*, enquanto outros podem ocultá-las, tornando a detecção inconsistente (TU et al., 2019).

Essas limitações são agravadas pela complexidade inerente à interação entre *channels* e outras primitivas, como `Mutex` e `WaitGroup`, e pelo uso de bibliotecas que integram *channels* implicitamente. Um exemplo comum é a combinação de `select` com múltiplos casos, que introduz comportamento não determinístico, dificultando ainda mais a análise e a depuração de erros. Assim, torna-se evidente a necessidade de uma abordagem que melhore a segurança e a robustez do uso de *channels*, reduzindo a ocorrência e o impacto desses *bugs*, e facilitando a detecção e depuração dos mesmos caso ocorram.

3.2 ABORDAGEM

A solução proposta neste trabalho é a criação de um pacote chamado `SafeChannel`, que atua como um *wrapper*, ou seja, uma camada de encapsulamento para as primitivas nativas de *channels* do Go. A principal ideia é encapsular a lógica de criação, manipulação e encerramento de *channels* em uma interface segura, capaz de prevenir erros como operações sobre *channels* fechados ou operações sobre *channels* nulos. Para isso, o `SafeChannel` incorpora verificações internas e mecanismos de controle que visam reduzir a ocorrência de *panics* e *deadlocks*. A implementação completa do pacote está disponível no repositório oficial do projeto no GitHub (MOUSSA, 2024).

O `SafeChannel` também busca mitigar as dificuldades associadas à detecção de *bugs* de concorrência em *channels*. Isso é feito por meio de verificações internas que ajudam a identificar e tratar situações problemáticas, como tentativas de envio para *channels* já fechados, ou bloqueios devido ao uso de *channels* nulos. Além disso, o pacote implementa um sistema de notificações opcional para relatar eventos críticos em tempo de execução, como falhas de envio e recebimento, oferecendo maior visibilidade ao desenvolvedor.

3.2.1 Fluxo de Operações do SafeChannel

O `SafeChannel` organiza as operações de envio e recebimento em *channels* segundo um fluxo controlado, podendo ser interpretado como uma máquina de estados. Após ser inicializado pela função `MakeSafechannel`, dois *channels* internos são criados: um para envio (`sendCh`) e outro para recebimento (`recvCh`). Essa separação permite ao `SafeChannel` controlar as mensagens de forma independente e garantir que todas as operações sejam realizadas de acordo com as condições apropriadas.

No envio de mensagens, a função `Send` executa verificações internas antes de encaminhar valores ao `sendCh`. Caso o *channel* esteja fechado, a função interrompe a operação, retorna um erro e, se habilitado, emite uma notificação para o desenvolvedor. Em *channels* não buferizados, a função `Send` bloqueia até que outra *goroutine* esteja pronta para consumir a mensagem. Em *channels* buferizados, o bloqueio ocorre apenas quando o *buffer* atinge sua capacidade máxima, momento em que o pacote sinaliza que o *buffer* está cheio, aguardando espaço para continuar.

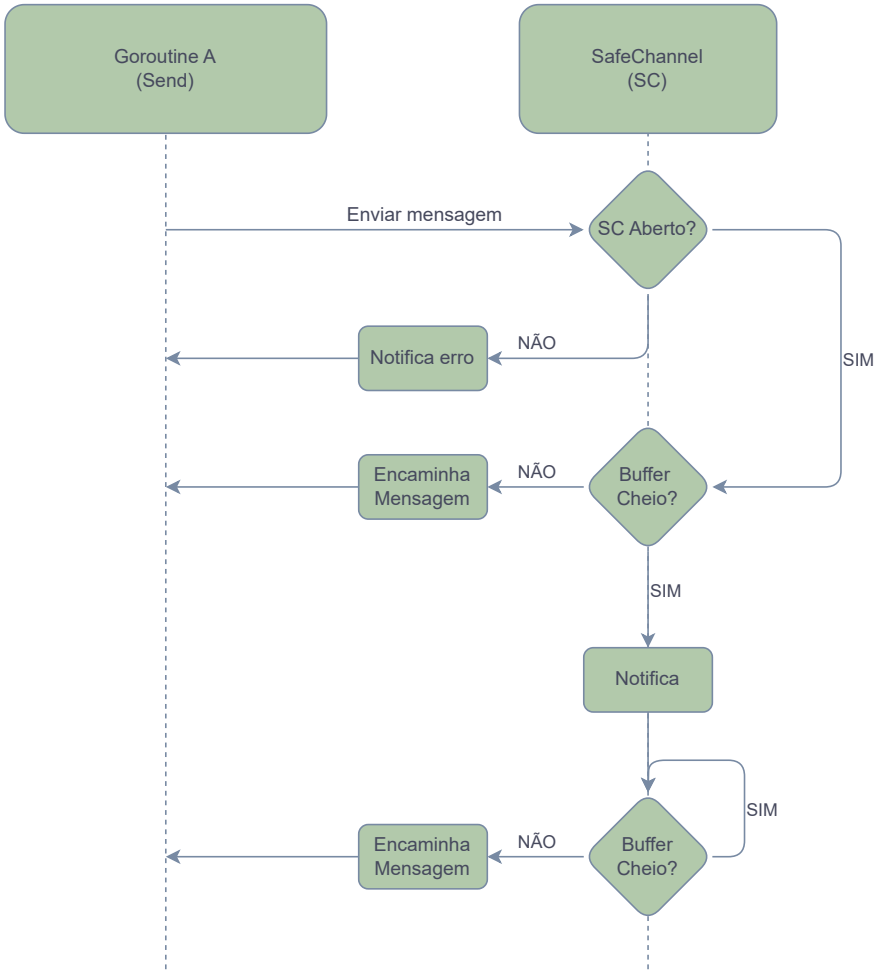

Figura 1 – Fluxo de Envio de Mensagem

Na figura 1, a `Goroutine A` envia uma mensagem através do `SafeChannel`, ficando bloqueada. Em seguida, o canal verifica se ele está aberto e, caso esteja fechado, cria uma notificação e retorna um erro de volta para `Goroutine A`, que será desbloqueada. No cenário onde o canal está aberto, é feita outra verificação, agora a respeito do *buffer*. Se o mesmo está com espaço, a mensagem é armazenada no *buffer*, é criada uma notificação, e a `Goroutine A` é desbloqueada. Em contrapartida, caso o *buffer* esteja cheio, ou o canal seja não buferizado, é criada uma notificação sobre a tentativa de envio e a `Goroutine A` ficará bloqueada até que seja realizada uma operação de leitura (`Receive`).

A função `Receive`, por sua vez, opera aguardando até que mensagens estejam disponíveis no `recvCh`. Para evitar bloqueios indefinidos, o `SafeChannel` utiliza um gerenciador de mensagens, o `forwardMessages`, que recebe um sinal quando há *goroutines* prontas para consumir dados. Esse gerenciador desempenha um papel essencial, encaminhando valores do `sendCh` para o `recvCh` sob demanda. Em *channels* buferizados, o

`forwardMessages` aproveita a oportunidade de `Receive` para transferir todas as mensagens disponíveis no `sendCh` para o `recvCh`, otimizando o fluxo de mensagens.

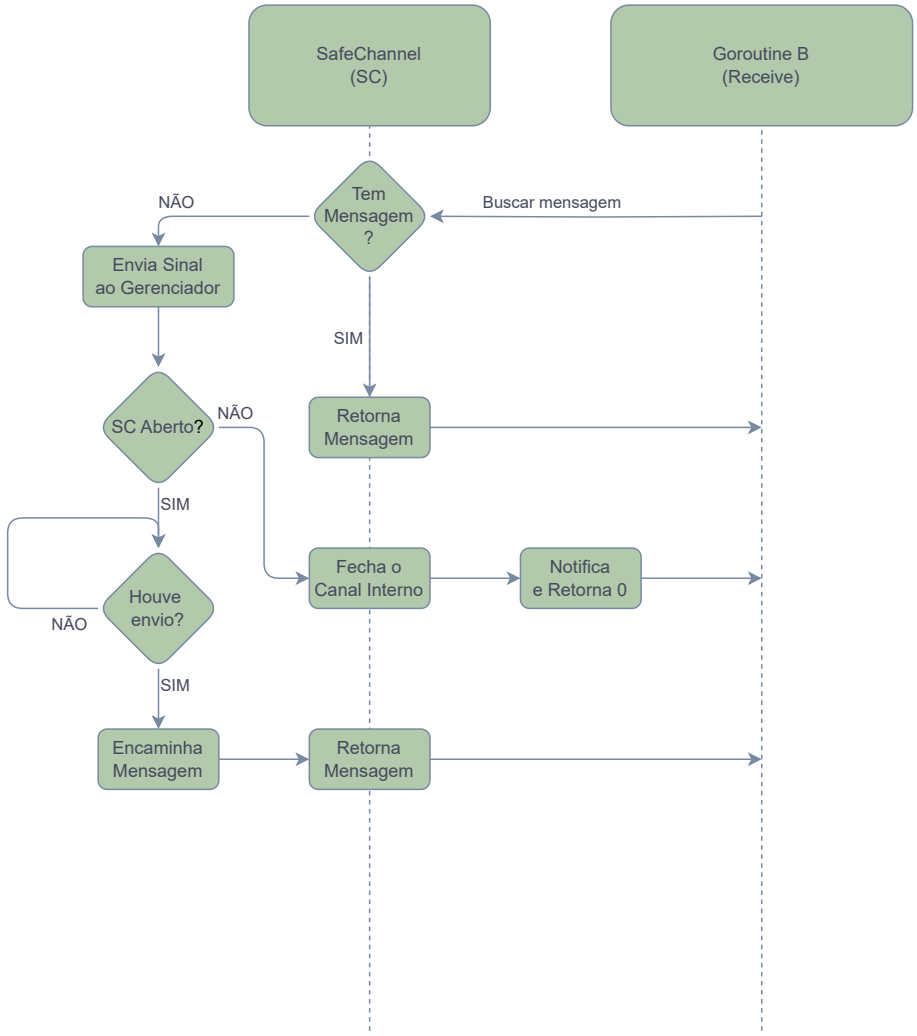

Figura 2 – Fluxo de Leitura de Mensagem

Na figura 2, a `Goroutine B` tenta escutar do `SafeChannel`, ficando bloqueada. Em seguida, o canal verifica se há mensagem no *channel* interno (`recvCh`). Se existir, a mensagem é retornada e a `Goroutine B` retoma sua execução. Caso não haja mensagem naquele momento, é enviado um sinal para o gerenciador de mensagens (`forwardMessages`) que irá buscar mensagens do *channel* interno `sendCh` e encaminhar para o `recvCh`, verificando se ocorreu o fechamento do primeiro. Se o *channel* foi fechado, então é criada uma notificação e retornado o valor padrão (*zero value*) do tipo de dado associado ao *channel*. Caso o gerenciador encontre mensagens no *buffer* do `sendCh`, todas serão encaminhadas para o *buffer* do *recvCh*, e poderão ser consumidas pelo `Receive` (uma de cada vez).

O `SafeChannel` também implementa mecanismos para gerenciar estados críticos, como o fechamento de *channels*. A função `Close` garante que, uma vez chamado o encerramento,

nenhuma nova mensagem seja enviada, embora permita que as mensagens ainda pendentes no *buffer* sejam consumidas. Essa abordagem previne *panics* e facilita um encerramento seguro para as *goroutines* que dependem do *channel*, proporcionando maior previsibilidade e robustez ao sistema.

3.3 ESTRUTURA DA SOLUÇÃO

O pacote `SafeChannel` foi projetado para encapsular as operações de *channels* nativos do Go, adicionando camadas de segurança e melhorias que facilitam o uso e ajudam a prevenir erros comuns e diagnosticar em *runtime* o funcionamento do `SafeChannel`. Esta seção detalha a estrutura da implementação e como as principais funcionalidades são realizadas.

O `SafeChannel` utiliza duas instâncias internas de *channels*, uma para envio (`sendCh`) e outra para recebimento (`recvCh`). Essa separação permite maior controle sobre o fluxo de mensagens e facilita a aplicação de mecanismos de verificação e tratamento de erros. A estrutura completa da implementação é descrita abaixo (código 12):

Código 12 – Estrutura do SafeChannel

```
type SafeChannel[T any] struct {
  sendCh chan T
  recvCh chan T
  notifyCh chan Notification[T]
  mu sync.Mutex
  cond *sync.Cond
  closed bool
  pendingReceivers int
  messagesInBuffer int64
}
```

Além dos *channels* internos de envio e recebimento, a estrutura inclui:

- `notifyCh`: Um *channel* opcional usado para emitir notificações sobre eventos relevantes, como falhas de envio ou recepção.
- `mu`: Um `sync.Mutex` que garante segurança em operações concorrentes, como envio e fechamento do `SafeChannel`.
- `cond`: Um `*sync.Cond` que sincroniza o encaminhamento de mensagens entre os *channels* internos.
- `closed`: Um indicador do estado atual do `SafeChannel`, usado para evitar operações inválidas.

- `pendingReceivers`: Um contador que é incrementado sempre que uma *goroutine* tenta escutar de um `SafeChannel`, atuando junto ao `cond` na sincronização.
- `messagesInBuffer`: Um contador atômico que rastreia o número de mensagens presentes no *buffer* do *channel*.

A instanciação de um `SafeChannel` é feita através da função `MakeSafechannel`, que encapsula o processo de inicialização. Essa função aceita um parâmetro opcional (`bufferSize`) para definir o tamanho do *buffer* dos *channels* internos. Caso nenhum tamanho seja especificado, os *channels* serão não buferizados (assim como ocorre na criação de *channels* com o `make`). A implementação é mostrada a seguir:

Código 13 – MakeSafechannel

```
func MakeSafechannel[T any](bufferSize ...int) *SafeChannel[T] {
  var sendCh, recvCh chan T
  var size = getBufferSize(bufferSize)

  sendCh = make(chan T, size)
  recvCh = make(chan T, size)

  sc := &SafeChannel[T]{
    sendCh: sendCh,
    recvCh: recvCh,
    closed: false,
  }
  sc.cond = sync.NewCond(&sc.mu)

  go sc.forwardMessages()
  return sc
}
```

Na função `MakeSafechannel`, descrita no código 13, os *channels* `sendCh` e `recvCh` são inicializados com o tamanho do *buffer* especificado, garantindo que ambos possuam a mesma capacidade. Em seguida:

1. A estrutura do `SafeChannel` é criada e inicializada, com os canais internos devidamente associados.
2. O estado inicial do `SafeChannel` é configurado, com o campo `closed` definido como `false`, indicando que o *channel* está aberto e pronto para uso.
3. É inicializada a variável `cond`, que será usada para sincronização entre o encaminhamento e consumo de mensagens.

4. A função inicia a execução da rotina `forwardMessages`, responsável por encaminhar mensagens do `sendCh` para o `recvCh`. Essa separação de *channels* permite maior flexibilidade no controle de fluxos e facilita o rastreamento de eventos durante a execução.

A `forwardMessages` gerencia o fluxo de mensagens entre os *channels* internos, garantindo que mensagens enviadas para o `sendCh` sejam corretamente transferidas para o `recvCh`. Essa funcionalidade reforça a robustez do `SafeChannel` ao prevenir perdas de mensagens ou estados inconsistentes.

3.4 FUNCIONALIDADES PRINCIPAIS

A seguir, serão apresentadas as principais funcionalidades do `SafeChannel`, detalhando como cada uma contribui para uma comunicação mais segura e previsível entre *goroutines*. Serão abordados os mecanismos de envio e recebimento de mensagens, o gerenciamento do encerramento do canal, a lógica interna de encaminhamento de mensagens e a implementação do `Select`. Ao fim, também é apresentada uma função de consulta de *buffer* disponibilizada pelo pacote.

3.4.1 Envio de Mensagens (`Send`)

A função `Send` (código 14) permite o envio de mensagens ao `SafeChannel`. Inicialmente, a função adquire um *lock* utilizando `sc.mu.Lock()` para verificar se o `SafeChannel` já foi fechado. Caso esteja fechado, a função dispara uma notificação registrando a tentativa de envio para um canal encerrado e retorna um erro, prevenindo um *panic*. Se o `SafeChannel` estiver aberto, o *lock* é liberado (sc.mu.Unlock()), permitindo que a execução continue normalmente.

Código 14 – Send

```
func (sc *SafeChannel[T]) Send(value T) error {
  sc.mu.Lock()

  if sc.closed {
    sc.notify(Notification[T]{
      ReturnValue: -1,
      Message: "send on closed channel",
      Value: value
    })
    sc.mu.Unlock()
    return errors.New("send on closed channel")
  }
  sc.mu.Unlock()

  select {
  case sc.sendCh <- value:
    sc.notify(Notification[T]{
      ReturnValue: 0,
      Message: "sent successfully",
      Value: value
    })
    atomic.AddInt64(&sc.messagesInBuffer, 1)
    return nil
  default:
    sc.notify(Notification[T]{
      ReturnValue: -1,
      Message: "channel buffer full",
      Value: value
    })
    sc.sendCh <- value
    atomic.AddInt64(&sc.messagesInBuffer, 1)
    sc.notify(Notification[T]{
      ReturnValue: 0,
      Message: "sent after waiting",
      Value: value
    })
    return nil
  }
}
```

A função utiliza um `select` para tentar enviar a mensagem ao canal interno `sendCh`. O `select` permite lidar com os seguintes cenários:

- **Envio imediato bem-sucedido:** Se houver espaço disponível no *buffer*, a mensagem é enviada diretamente para `sendCh`. Nesse caso, a função registra o sucesso da operação na estrutura de notificações e incrementa o contador `messagesInBuffer`, que mantém o rastreamento da quantidade de mensagens atualmente armazenadas.
- ***Buffer* cheio ou canal não-buferizado:** Se `sendCh` estiver cheio (no caso de um `SafeChannel` buferizado) ou se o `SafeChannel` for não-buferizado, o `select` segue para o caso `default`. Em seguida, uma notificação é gerada para indicar que o canal estava cheio, e a função executa uma nova tentativa de envio, que será bloqueante até que o espaço seja liberado no *buffer* ou uma *goroutine* esteja pronta para consumir a mensagem. Após o envio, a função emite outra notificação informando que a mensagem foi enviada após aguardar.

Essa abordagem permite que `Send` funcione corretamente tanto para `SafeChannels` buferizados quanto não-buferizados, garantindo que as mensagens sejam enviadas de maneira segura, sem risco de *panic*, e que os estados de bloqueio sejam corretamente gerenciados.

3.4.2 Recebimento de Mensagens (`Receive`)

A função `Receive` (código 15) é responsável por retirar mensagens do `SafeChannel`. Seu retorno inclui um valor do tipo armazenado no canal e um erro, caso a operação não possa ser concluída. Se o `SafeChannel` estiver fechado, a função retorna um valor padrão (*zero value*) e um erro.

Código 15 – Receive

```
func (sc *SafeChannel[T]) Receive() (T, error) {
  var zero T

  for {
    select {
    case value, ok := <-sc.recvCh:
      if !ok {
        sc.notify(Notification[T]{
          ReturnValue: -1,
          Message: "receive on closed channel"
        })
        return zero, errors.New("receive on closed channel")
      }
      atomic.AddInt64(&sc.messagesInBuffer, -1)
      return value, nil
    default:
      sc.mu.Lock()
      sc.pendingReceivers++
      sc.cond.Signal()
      sc.mu.Unlock()
    }
  }
}
```

Para garantir que a goroutine permaneça bloqueada até que uma mensagem esteja disponível, a função é estruturada com um *loop* contínuo que executa um `select` sobre o `recvCh`. Quando há uma mensagem disponível, o valor é lido e retornado, além de ser feita a atualização do contador `messagesInBuffer`, que mantém o rastreamento das mensagens armazenadas. Se o `recvCh` estiver fechado, um erro é retornado e uma notificação é gerada. Caso contrário, se nenhuma mensagem estiver imediatamente disponível, a função incrementa o contador `pendingReceivers` para indicar que há uma *goroutine* aguardando a chegada de dados. Em seguida, um sinal é enviado para a condição que controla a execução da `forwardMessages`, permitindo que mensagens sejam transferidas do `sendCh` para `recvCh`. Após esse processo, o *loop* reinicia e a função volta a verificar a existência de novas mensagens.

3.4.3 Encerramento do Canal (`Close`)

A função `Close`, código 16, é responsável por encerrar o `SafeChannel` de maneira segura, garantindo que novas operações não sejam mais permitidas. Para isso, a função adquire um *lock* com `mu.Lock()` antes de verificar o estado interno do canal. Caso o

`SafeChannel` já esteja fechado, o *lock* é liberado imediatamente e a função retorna um erro, além de gerar uma notificação para informar que o fechamento já havia sido realizado anteriormente. Se o `SafeChannel` ainda estiver aberto, a função prossegue com o encerramento ao fechar o `sendCh`. Em seguida, a variável `closed` é atualizada para refletir o novo estado do canal, e o *lock* é liberado.

Código 16 – Close

```
func (sc *SafeChannel[T]) Close() error {
  sc.mu.Lock()
  if sc.closed {
    sc.mu.Unlock()
    sc.notify(Notification[T]{
      ReturnValue: -1,
      Message: "channel already closed"
    })
    return errors.New("channel already closed")
  }

  close(sc.sendCh)
  sc.closed = true
  sc.mu.Unlock()

  return nil
}
```

3.4.4 Encaminhamento de Mensagens (`forwardMessages`)

A função `forwardMessages`, código 17, é responsável por intermediar a comunicação entre `sendCh` e `recvCh`, garantindo que as mensagens enviadas ao `SafeChannel` sejam devidamente entregues às *goroutines* receptoras. Ela roda como uma *goroutine* separada, e o encaminhamento só ocorre quando há pelo menos uma *goroutine* esperando para receber, o que é sinalizado pelo incremento de `pendingReceivers`.

Código 17 – forwardMessages

```
func (sc *SafeChannel[T]) forwardMessages() {
  for {
    sc.mu.Lock()
    for sc.pendingReceivers == 0 && !sc.closed {
      sc.cond.Wait()
    }
    sc.mu.Unlock()
    for {
      select {
      case value, ok := <-sc.sendCh:
        if !ok {
          close(sc.recvCh)
          return
        }
        select {
        case sc.recvCh <- value:
          sc.notify(Notification[T]{
            ReturnValue: 0,
            Message: "message forwarded successfully", Value: value
          })
        default:
          sc.notify(Notification[T]{
            ReturnValue: -1,
            Message: "channel buffer full", Value: value
          })
          sc.recvCh <- value
          sc.notify(Notification[T]{
            ReturnValue: 0,
            Message: "forwarded after waiting", Value: value
          })
          sc.mu.Lock()
          sc.pendingReceivers--
          sc.mu.Unlock()
          goto StopForwarding
        }
      default:
        sc.mu.Lock()
        sc.pendingReceivers--
        sc.mu.Unlock()
        goto StopForwarding
      }
    }
  StopForwarding:
    continue
  }
}
```

A função inicia bloqueando a execução com `cond.Wait()` sempre que `pendingReceivers` for zero, ou seja, quando não houver *goroutines* aguardando mensagens. Assim que um `Receive` é chamado, `pendingReceivers` é incrementado e um sinal é enviado, despertando `forwardMessages`, que entra em um *loop* interno para encaminhar as mensagens.

Se o `SafeChannel` for buferizado, todas as mensagens disponíveis em `sendCh` são transferidas para `recvCh` em sequência, permitindo que futuras chamadas a `Receive` possam consumi-las sem necessidade de reativar `forwardMessages`. Já em um `SafeChannel` não-buferizado, apenas uma mensagem é encaminhada por vez, e o `forwardMessages` bloqueia novamente até que uma nova *goroutine* esteja pronta para receber.

O código trata também o fechamento do `SafeChannel`. Se `sendCh` for fechado, `recvCh` também é fechado, garantindo que nenhuma *goroutine* fique bloqueada indefinidamente esperando por mensagens que nunca chegarão. O uso de notificações permite registrar eventos como o encaminhamento bem-sucedido de mensagens e situações em que o *buffer* está cheio, facilitando a depuração e o monitoramento do comportamento do canal.

3.4.5 Implementação do `Select`

A linguagem Go possui uma estrutura de controle chamada `select`, projetada para monitorar múltiplas operações sobre *channels* simultaneamente e executá-las conforme as condições são satisfeitas. No entanto, essa instrução não pode ser utilizada diretamente com o `SafeChannel`, já que as primitivas internas de envio e recebimento são encapsuladas, e a linguagem não suporta *override*. Para resolver essa limitação, o pacote `SafeChannel` inclui uma implementação própria do comportamento do `select`.

A implementação é baseada na função principal `Select`, que implementa uma versão personalizada da instrução `select` do Go, permitindo a escolha dinâmica entre múltiplos casos de envio e recebimento. Ela recebe uma lista de funções do tipo `SelectCaseFunc`, onde cada função representa um possível caso de operação concorrente, retornando se a operação foi bem-sucedida, seu índice, um valor associado e um possível erro.

Código 18 – Select

```
type SelectCaseFunc func() (bool, int, interface{}, error)

func Select(cases ...SelectCaseFunc) (int, interface{}, error) {
  var defaultIdx int = -1
  for i, c := range cases {
      ok, idx, value, err := c()
      if ok {
          return idx, value, err
      }
      if idx == -1 {
          defaultIdx = i
      }
  }

  if defaultIdx != -1 {
      _, idx, value, err := cases[defaultIdx]()
      return idx, value, err
  }

  return -1, nil, nil
}
```

O funcionamento da função `Select`, descrita acima no código 18, baseia-se em uma iteração sobre os casos fornecidos. Para cada caso, a função correspondente é executada e, se um deles estiver pronto para ser processado, o `Select` retorna imediatamente o índice do caso, o valor resultante e possíveis erros. Se nenhum caso estiver pronto para execução, a função verifica se há um `default` case presente na lista, garantindo que ele seja acionado para evitar bloqueios indesejados.

Os casos de envio e recebimento no `select` personalizado são implementados por meio das funções `CaseSend` e `CaseReceive`, que encapsulam operações específicas de `sendCh` e `recvCh`, respectivamente.

A função `CaseReceive` (código 19) implementa a lógica necessária para que uma operação de recebimento seja utilizada dentro do `Select`. Ela verifica se há mensagens disponíveis no `recvCh` e, caso haja, retorna imediatamente o valor lido. Se o `SafeChannel` estiver fechado, a função notifica o erro e retorna um valor padrão.

Caso nenhuma mensagem esteja disponível no momento da chamada, a função verifica o contador `messagesInBuffer`. Se houver mensagens no *buffer*, significa que elas ainda não foram encaminhadas do `sendCh` para o `recvCh`, então o `pendingReceivers` é incrementado e um sinal é enviado para o `forwardMessages`, que realizará o encaminhamento. Esse processo garante que mensagens pendentes sejam transferidas corretamente antes

da função retornar ao `Select`. Caso o *buffer* esteja realmente vazio, a função retorna false, indicando que não há mensagens disponíveis no momento, permitindo que o `Select` considere outras opções, como um *default case*.

Código 19 – CaseReceive

```
func CaseReceive[T any](sc *SafeChannel[T], onReceive func(T))
    SelectCaseFunc {
  return func() (bool, int, interface{}, error) {
    for {
      select {
      case msg, ok := <-sc.recvCh:
        if !ok {
          sc.notify(Notification[T]{
            ReturnValue: -1,
            Message: "receive on closed channel",
            Value: msg
          })
          return true, -1, nil, errors.New("receive on closed channel")
        }
        if onReceive != nil {
          onReceive(msg)
        }
        atomic.AddInt64(&sc.messagesInBuffer, -1)
        return true, 0, msg, nil
      default:
        if atomic.LoadInt64(&sc.messagesInBuffer) > 0 {
          sc.mu.Lock()
          sc.pendingReceivers++
          sc.cond.Signal()
          sc.mu.Unlock()
          continue
        }
        return false, 0, nil, nil
      }
    }
  }
}
```

De forma similar, a função `CaseSend` (código 20) verifica se o *channel* está disponível para envio. Caso o *buffer* esteja cheio ou o *channel* esteja fechado, a função retorna as notificações apropriadas.

A função `CaseSend` (código 20) permite que uma operação de envio seja utilizada

dentro do `Select`. Antes de tentar enviar a mensagem, a função verifica se o `SafeChannel` já foi fechado. Caso esteja fechado, uma notificação é registrada e a função retorna imediatamente.

Se o `SafeChannel` ainda estiver ativo, a função tenta enviar a mensagem para o `sendCh`. Se houver espaço disponível no *buffer* ou se o `SafeChannel` for não-buferizado e houver uma *goroutine* aguardando para receber, o envio ocorre com sucesso e a função retorna um indicativo de sucesso. Caso contrário, se o *buffer* estiver cheio, a função notifica a falha e retorna um erro, permitindo que o `Select` continue avaliando outros casos disponíveis.

Código 20 – CaseSend

```
func CaseSend[T any](sc *SafeChannel[T], msg T) SelectCaseFunc {
  return func() (bool, int, interface{}, error) {
    if sc.closed {
      sc.notify(Notification[T]{
        ReturnValue: -1,
        Message: "send on closed channel",
        Value: msg
      })
      return true, -1, nil,
      errors.New("send on closed channel")
    }
    select {
    case sc.sendCh <- msg:
      return true, 0, nil, nil
    default:
      sc.notify(Notification[T]{
        ReturnValue: -1,
        Message: "channel buffer full",
        Value: msg
      })
      return true, -1, nil,
      errors.New("channel buffer full")
    }
  }
}
```

A função `CaseDefault` (código 21) define um comportamento alternativo para o `Select`, garantindo que, caso nenhuma das operações de envio ou recebimento esteja pronta, uma ação predefinida seja executada. Essa funcionalidade é especialmente útil para evitar bloqueios quando há a necessidade de seguir com a execução do programa mesmo que nenhuma operação no `SafeChannel` possa ser realizada no momento.

Se uma função de ação for fornecida (`action func()`), `CaseDefault` a executa antes

de retornar um indicativo de sucesso. O índice retornado é -1, sinalizando que o caso padrão foi acionado. Dessa forma, o `Select` pode continuar seu fluxo sem ficar preso aguardando um envio ou recebimento.

Código 21 – CaseDefault

```
func CaseDefault(action func()) SelectCaseFunc {
  return func() (bool, int, interface{}, error) {
      if action != nil {
          action()
      }
      return true, -1, nil, nil
  }
}
```

3.4.6 Mensagens no Buffer

Ao utilizar os *channels* para troca de mensagens, é possível definir o tamanho do *buffer* interno do mesmo, e cada mensagem enviada ocupará um espaço desse *buffer*. No entanto, Go não permite a consulta de quantos espaços restam, ou quantos estão ocupados.

Por conta disto, foi adicionada ao pacote `SafeChannel` uma função que permite realizar essa consulta, retornando a quantidade de mensagens que estão presentes no *buffer*.

Código 22 – GetMessageCount

```
func (sc *SafeChannel[T]) GetMessageCount() int64 {
  return atomic.LoadInt64(&sc.messagesInBuffer)
}
```

No código 22, é retornado o contador `messagesInBuffer` daquela instância do `SafeChannel`. Como é utilizado o pacote `atomic` para a criação e manipulação desse contador, não é necessário o uso de outras primitivas de controle, como o `mutex`.

3.4.7 Notificações

O `SafeChannel` inclui um canal opcional de notificações (`notifyCh`), projetado para fornecer informações detalhadas sobre eventos internos, como falhas em operações de envio e recebimento. Esse mecanismo pode ser ativado dinamicamente por meio da função `EnableNotifications`, permitindo que os desenvolvedores acompanhem o compor-

tamento do `SafeChannel` em tempo de execução e identifiquem possíveis problemas com maior facilidade.

As notificações são armazenadas em uma estrutura FIFO (*first in, first out*), garantindo que os eventos mais antigos sejam descartados à medida que novas mensagens chegam, caso o canal tenha sido criado com um *buffer*. Cada notificação contém informações detalhadas, incluindo a mensagem do evento, o valor envolvido e dados de rastreamento, como o nome da função, o arquivo de origem e a linha de código onde a operação ocorreu. Isso permite que o `SafeChannel` não apenas forneça informações úteis para depuração, mas também ajude na identificação de padrões de uso incorretos ou potenciais falhas na comunicação entre *goroutines*.

Código 23 – Notification

```
type Notification[T any] struct {
  ReturnValue int
  Message     string
  Value       T
  FuncName    string
  File        string
  Line        int
}

func (sc *SafeChannel[T]) EnableNotifications(bufferSize ...int) {
  var size = getBufferSize(bufferSize)
  sc.notifyCh = make(chan Notification[T], size)
}

func (sc *SafeChannel[T]) ReadNotification() (Notification[T], error) {
  if sc.notifyCh == nil {
      return Notification[T]{},
      errors.New("notifications not enabled")
  }
  notification, ok := <-sc.notifyCh
  if !ok {
      return Notification[T]{},
      errors.New("notification channel closed")
  }
  return notification, nil
}

func (sc *SafeChannel[T]) notify(notification Notification[T]) {
  if sc.notifyCh == nil {
    return
  }

  pc, file, line, ok := runtime.Caller(2)
  if ok {
    notification.FuncName = runtime.FuncForPC(pc).Name()
    notification.File = file
    notification.Line = line
  }
  select {
  case sc.notifyCh <- notification:
  default:
    <- sc.notifyCh
    sc.notifyCh <- notification
  }
}
```

A função `notify` é utilizada internamente para gerar notificações, populando o canal `notifyCh` com os eventos relevantes. Caso o canal esteja desativado, a função retorna imediatamente sem realizar nenhuma operação. Para evitar bloqueios, as notificações são inseridas no *buffer* quando há espaço disponível e, em caso de *buffer* cheio, a notificação mais antiga é descartada para garantir que sempre a mensagem mais recente esteja acessível.

A função `ReadNotification` permite que as *goroutines* consumam as notificações geradas. Se o canal de notificações não estiver ativado ou tiver sido fechado, a função retorna um erro.

3.5 PREVENÇÃO DE BUGS E DEPURAÇÃO

O `SafeChannel` foi projetado para prevenir situações comuns que podem levar a *bugs* de concorrência, além de oferecer ferramentas para facilitar a depuração durante a execução da aplicação. Sua implementação encapsula as operações sobre os *channels* internos, restringindo acessos diretos e incorporando verificações para evitar comportamentos inseguros. Os principais mecanismos são:

- **Prevenção de Operações em *Channels* Nulos:** Como os *channels* internos do `SafeChannel` não são acessíveis diretamente, qualquer tentativa de envio ou recebimento em canais não inicializados é automaticamente impedida, evitando bloqueios indefinidos.

- **Prevenção de Fechamento Duplo:** O `SafeChannel` mantém o estado do *channel* internamente e verifica se ele já foi fechado antes de permitir qualquer operação de encerramento. Isso elimina erros comuns de fechamento duplo, que poderiam causar *panics*.

- **Envio para *Channels* Fechados:** A função `Send` inclui verificações explícitas para impedir tentativas de envio em *channels* que já foram fechados. Nesses casos, o `SafeChannel` emite uma notificação detalhada e retorna um erro, em vez de permitir que o programa entre em um estado de falha crítica (*panic*).

- **Canal de Notificações para Depuração:** O `SafeChannel` inclui um recurso opcional de notificações em tempo de execução, que relata eventos relevantes, como tentativas de envio para *channels* fechados ou mensagens enviadas com sucesso. No caso específico de envio via `Send`, a lógica interna do `select` gera notificações para lidar com situações onde o *buffer* do *channel* está cheio.

Código 24 – Exemplo de Notificação

```
select {
  case sc.sendCh <- value:
    sc.notify(Notification[T]{
      ReturnValue: 0,
      Message: "sent successfully",
      Value: value
    })
  default:
    sc.notify(Notification[T]{
      ReturnValue: -1,
      Message: "channel buffer full",
      Value: value
    })

    sc.sendCh <- value

    sc.notify(Notification[T]{
      ReturnValue: 0,
      Message: "sent after waiting",
      Value: value
    })
}
```

No código 24, o caso `default` é acionado quando o *buffer* do *channel* está cheio, gerando uma notificação antes de tentar o envio novamente, seguido de uma mensagem de confirmação após o mesmo ser concluído. Esse comportamento ajuda a monitorar estados críticos do *channel* e facilita a depuração em tempo de execução.

- **Sincronização Segura:** O uso de `sync.Mutex` garante que as operações concorrentes no `SafeChannel`, como `Send` e `Close`, sejam realizadas de forma controlada, prevenindo *data race* e estados inconsistentes.

3.6 EXEMPLO DE USO

A fim de demonstrar o funcionamento do `SafeChannel` na prática, foi desenvolvido um exemplo baseado no clássico problema do Produtor-Consumidor, um dos exemplos mais comuns em programação concorrente. No cenário escolhido, um restaurante possui um garçom, responsável por receber pedidos dos clientes e encaminhá-los para a cozinha, e um cozinheiro, que processa esses pedidos e os finaliza. O `SafeChannel` é utilizado para gerenciar a troca de mensagens entre as *goroutines* que representam o garçom e a cozinha,

garantindo que os pedidos sejam enviados e recebidos corretamente, prevenindo possíveis erros, como tentativas de envio para um canal fechado ou fechamento duplo do canal.

Além disso, um sistema de monitoramento é implementado para capturar notificações geradas pelo `SafeChannel`, registrando os eventos que ocorrem. Esse sistema de notificações permite depurar o fluxo de execução e visualizar como o `SafeChannel` gerencia a comunicação entre as *goroutines*.

Código 25 – Exemplo de Uso - Cenário de um Restaurante

```
package main

import (
  "fmt"
  "time"
  "biblioteca/safechannel"
)

// Pedido em um restaurante
type Pedido struct {
  ID      int
  Cliente string
}

func main() {
  sc := safechannel.MakeSafechannel[Pedido](2)
  sc.EnableNotifications(5)

  go Garcom(sc)
  go Cozinha(sc)
  monitorarNotificacoes(sc)

  time.Sleep(15 * time.Second)
  fmt.Println("Restaurante fechado!")
}
```

O código 25 define a estrutura de dados `Pedido`, que representa um pedido feito no restaurante. Cada pedido contém um identificador (`ID`) e o nome do cliente (`Cliente`). Na função `main`, um `SafeChannel` com *buffer* de tamanho 2 é criado para armazenar os pedidos. Em seguida, três *goroutines* são iniciadas: uma para o garçom, que envia os pedidos para a cozinha; outra para a cozinha, que consome os pedidos e os prepara; e uma terceira para monitorar notificações geradas pelo `SafeChannel`, garantindo que erros e eventos sejam registrados.

Código 26 – Garcom (Produtor)

```
// Garcom recebe pedidos e os envia para a cozinha pelo SafeChannel
func Garcom(sc *safechannel.SafeChannel[Pedido]) {
  for i := 1; i <= 5; i++ {
    pedido := Pedido{ID: i, Cliente: fmt.Sprintf("Cliente %d", i)}
    err := sc.Send(pedido)
    if err != nil {
      fmt.Println("Erro ao enviar pedido:", err)
    } else {
      fmt.Printf("Garcom: Pedido %d recebido\n", pedido.ID)
    }
    time.Sleep(1 * time.Second) // Simula tempo entre pedidos
  }
  sc.Close() // Fecha o canal apos todos os pedidos serem enviados
}
```

A função `Garcom` (código 26) representa o papel do produtor no problema do Produtor-Consumidor. Ela simula a chegada de pedidos, gerando um novo pedido a cada ciclo e enviando-o ao `SafeChannel` usando a função `Send`. Antes de enviar, o `SafeChannel` realiza verificações internas para garantir que o envio seja seguro. Se o canal estiver fechado, um erro será retornado e registrado no monitoramento. Caso o canal esteja cheio, a operação será bloqueada até que haja espaço disponível. Após o envio de um número fixo de pedidos, o garçom fecha o canal, indicando que não há mais pedidos a serem processados.

Código 27 – Cozinha (Consumidor)

```
// Cozinha recebe pedidos e os processa
func Cozinha(sc *safechannel.SafeChannel[Pedido]) {
  for {
    pedido, err := sc.Receive()
    if err != nil {
      fmt.Println("Cozinha: Nenhum pedido restante, fechando.")
      sc.Close()
      return
    }
    fmt.Printf("Cozinha: Preparando pedido %d para %s\n",
                pedido.ID,
                pedido.Cliente
              )

    time.Sleep(2 * time.Second) // Simula tempo de preparo
    fmt.Printf("Cozinha: Pedido %d pronto!\n", pedido.ID)
  }
}
```

A função `Cozinha` (código 27) representa o consumidor no problema do Produtor-Consumidor. Ela entra em um laço infinito aguardando a chegada de pedidos pelo `SafeChannel`. A cada iteração, tenta receber um pedido usando a função `Receive`, que garante que a operação ocorra de forma segura. Após receber um pedido, a cozinha simula seu processamento antes de sinalizar sua conclusão.

Caso o canal seja fechado e não haja mais mensagens, a função tenta novamente fechar o canal. Esse trecho foi incluído para demonstrar que a tentativa de fechar duas vezes o canal não causa um *panic*, como ocorre com os *channels*, e que esse evento também é registrado no canal de notificações.

Código 28 – Monitoramento

```
// Monitoramento do canal de notificacoes
func monitorarNotificacoes(sc *safechannel.SafeChannel[Pedido]) {
  go func() {
    for {
      notificacao, err := sc.ReadNotification()
      if err != nil {
        break
      }
      fmt.Printf("LOG: %s | Funcao: %s | Arquivo: %s | Linha: %d\n",
        notificacao.Message,
        notificacao.FuncName,
        notificacao.File,
        notificacao.Line
      )
    }
  }()
}
```

A função `monitorarNotificacoes` (código 28) permite capturar e registrar eventos gerados pelo `SafeChannel`. Esse sistema de notificações auxilia na depuração, fornecendo informações detalhadas sobre o fluxo de execução do programa. Sempre que um evento ocorre (como um envio para um canal fechado ou a tentativa de receber de um canal vazio), uma mensagem é registrada, contendo detalhes como a função e o arquivo onde o evento ocorreu. Essa abordagem facilita a identificação e correção de problemas relacionados à troca de mensagens.

A implementação desse exemplo destaca como o `SafeChannel` facilita a comunicação concorrente. Ele previne erros comuns ao uso de *channels*, proporcionando um fluxo de execução mais confiável para sistemas concorrentes que dependem da troca assíncrona de informações.

4 AVALIAÇÃO FUNCIONAL E DE DESEMPENHO

Esta seção apresenta a avaliação do `SafeChannel` por meio de testes funcionais e de desempenho. São descritos diversos casos de teste que analisam o comportamento da implementação em diferentes cenários, e uma análise de desempenho, comparando a eficiência do `SafeChannel` em relação aos *channels* do Go, destacando suas vantagens e desvantagens.

4.1 CASOS DE TESTE

A validação do `SafeChannel` foi realizada por meio de testes que cobrem diversos cenários de uso. Esses testes foram criados com o objetivo de verificar se a implementação do `SafeChannel` é capaz de atuar na troca de mensagens entre *goroutines*, prevenindo erros comuns de concorrência e comportamentos inesperados. Para isso, buscou-se alcançar uma cobertura abrangente, contemplando casos normais de operação, situações limite e condições de erro.

Os testes foram desenvolvidos utilizando o *package* `testing`, fornecido pela linguagem Go. Este *package* é uma ferramenta amplamente utilizada na comunidade para a criação de testes unitários e de integração, que permite a execução de funções específicas de teste e oferece suporte para paralelismo e medições de desempenho. Adicionalmente, o uso de funções auxiliares, como `t.Errorf()` e `t.Fatal()`, possibilita a captura de erros e a interrupção de testes quando necessário, facilitando a obtenção de diagnósticos durante o processo de validação.

Por meio desses testes, foi possível verificar o comportamento do `SafeChannel` em diferentes contextos, assegurando que o mesmo é capaz de cumprir com a proposta do pacote. A seguir, são apresentados os cenários de teste desenvolvidos.

4.1.1 Operação Normal

O objetivo deste teste (código 29) é verificar se o `SafeChannel` funciona corretamente em condições normais de uso, isto é, se ele permite o envio e o recebimento de mensagens sem interrupções ou erros.

Código 29 – Operação Normal

```
func TestSafeChannel_NormalOperation(t *testing.T) {
  sc := MakeSafechannel[int](1)

  err := sc.Send(42)
  if err != nil {
    t.Errorf("Unexpected error on send: %v", err)
  }

  value, err := sc.Receive()
  if err != nil {
    t.Errorf("Unexpected error on receive: %v", err)
  }
  if value != 42 {
    t.Errorf("Expected 42, got %v", value)
  }
}
```

Este teste cria um `SafeChannel` com um *buffer* de tamanho 1. Uma mensagem é enviada utilizando a função `Send()`, e, em seguida, recebida utilizando `Receive()`. O teste garante que o valor enviado e o valor recebido são iguais, verificando a integridade do fluxo de mensagens no `SafeChannel`.

4.1.2 Buffer Cheio

Este teste (código 30) verifica o comportamento do `SafeChannel` ao lidar com *buffers* cheios. O objetivo é garantir que mensagens adicionais não sejam perdidas e que notificações adequadas sejam enviadas.

Código 30 – Buffer Cheio

```
func TestSafeChannel_FullBuffer(t *testing.T) {
  sc := MakeSafechannel[int](1)
  sc.EnableNotifications(10)

  go func() {
    err := sc.Send(42)
    if err != nil {
      t.Errorf("Unexpected error on send: %v", err)
    }
    err = sc.Send(99)
    if err != nil {
      t.Errorf("Unexpected error on send: %v", err)
    }
  }()

  go func() {
    var messages [2]string
    index := 0

    for {
      notification, err := sc.ReadNotification()
      messages[index] = notification.Message
      index++
      if index == len(messages) {
        if messages[index-1] != "channel buffer full" || err != nil {
          t.Errorf("Expected 'channel buffer full' error, got %v", err)
        }
        break
      }
    }
  }()
}
```

Este teste configura um `SafeChannel` com um *buffer* de tamanho 1 e habilita o canal de notificações. Duas mensagens são enviadas consecutivamente: a primeira é aceita, enquanto a segunda gera uma notificação de *buffer full*. O teste garante que o `SafeChannel` lida corretamente com mensagens além da capacidade do *buffer*.

4.1.3 Envio Após Fechamento do Canal

Este teste (código 31) avalia se o `SafeChannel` impede corretamente o envio de mensagens após ser fechado, retornando um erro apropriado.

Código 31 – Send após Close

```
func TestSafeChannel_SendToClosed(t *testing.T) {
  sc := MakeSafechannel[int](0)

  sc.Close()

  err := sc.Send(42)
  if err == nil || err.Error() != "send on closed channel" {
    t.Errorf("Expected 'send on closed channel' error, got %v", err)
  }
}
```

Após fechar o `SafeChannel` com a função `Close()`, a função de teste tenta enviar uma mensagem utilizando `Send()`. Como esperado, um erro é retornado, confirmando que o `SafeChannel` protege contra envios para canais fechados.

4.1.4 Recebimento Após Fechamento do Canal

Este teste (código 32) verifica se o `SafeChannel` permite que mensagens enviadas antes do fechamento do canal sejam recebidas normalmente e se impede recebimentos após as mensagens disponíveis serem consumidas.

Código 32 – Receive após Close

```
func TestSafeChannel_ReceiveFromClosed(t *testing.T) {
  sc := MakeSafechannel[int](1)

  err := sc.Send(42)
  if err != nil {
    t.Errorf("Unexpected error on send: %v", err)
  }

  sc.Close()

  value, err := sc.Receive()
  if err != nil {
    t.Errorf("Unexpected error on receive: %v", err)
  }
  if value != 42 {
    t.Errorf("Expected 42, got %v", value)
  }

  _, err = sc.Receive()
  if err == nil || err.Error() != "receive on closed channel" {
    t.Errorf("Expected 'receive on closed channel' error, got %v", err)
  }
}
```

Este teste garante que o `SafeChannel` preserva mensagens enviadas antes do fechamento do canal, enquanto bloqueia operações de recebimento em um canal fechado e retorna um erro apropriado.

4.1.5 Fechamento do `SafeChannel` Duas Vezes

Este teste (código 33) valida que o `SafeChannel` lida corretamente com tentativas de fechamento múltiplas, retornando um erro apropriado quando uma segunda chamada para `Close()` é feita.

Código 33 – Múltiplos Close

```
func TestSafeChannel_CloseTwice(t *testing.T) {
  sc := MakeSafechannel[int](0)

  err := sc.Close()
  if err != nil {
    t.Errorf("Unexpected error on first close: %v", err)
  }

  err = sc.Close()
  if err == nil || err.Error() != "channel already closed" {
    t.Errorf("Expected 'channel already closed' error, got %v", err)
  }
}
```

Este teste garante que o `SafeChannel` sinaliza corretamente que já foi fechado e impede operações redundantes de fechamento, mantendo a consistência do estado interno.

4.1.6 Concorrência Entre `Send()` e `Receive()`

O objetivo deste teste (código 34) é assegurar que o `SafeChannel` funcione corretamente em cenários concorrentes, onde o envio e o recebimento de mensagens ocorrem simultaneamente.

Código 34 – Send após Receive

```
func TestSafeChannel_Concurrent(t *testing.T) {
  sc := MakeSafechannel[int]()

  go func() {
    time.Sleep(50 * time.Millisecond)
    err := sc.Send(99)
    if err != nil {
      t.Errorf("Unexpected error on send: %v", err)
    }
  }()

  value, err := sc.Receive()
  if err != nil {
    t.Errorf("Unexpected error on receive: %v", err)
  }
  if value != 99 {
    t.Errorf("Expected 99, got %v", value)
  }
}
```

Esse teste simula o envio de uma mensagem por uma *goroutine* enquanto outra realiza a operação de recebimento. Ele garante que o `SafeChannel` mantém a sincronização correta entre `Send()` e `Receive()`.

4.1.7 Bloqueio em `Send()` Quando Nenhum `Receive()` é Chamado

Este teste (código 35) avalia o comportamento de bloqueio do `SafeChannel` quando nenhuma operação de `Receive()` está disponível para consumir a mensagem enviada.

Código 35 – Send sem Receive

```
func TestSafeChannel_SendWithoutReceive(t *testing.T) {
  sc := MakeSafechannel[int](1)
  done := make(chan error)

  err := sc.Send(42)
  if err != nil {
    t.Errorf("Unexpected error on send: %v", err)
  }

  go func() {
    err = sc.Send(99)
    done <- err
  }()

  select {
  case <-done:
    t.Fatal("Test failed: Send should block until a receive occurs")
  case <-time.After(500 * time.Millisecond):
    t.Log("Send is correctly blocked in unbuffered SafeChannel")
  }

  go func() {
    _, _ = sc.Receive()
  }()
  time.Sleep(200 * time.Millisecond)

  select {
  case err := <-done:
    if err != nil {
      t.Errorf("Unexpected error on send: %v", err)
    } else {
      t.Log("Send completed successfully after receive")
    }
  case <-time.After(1 * time.Second):
    t.Fatal("Test failed: Send did not unblock after a receive")
  }
}
```

Este teste assegura que o `SafeChannel` segue o comportamento esperado de bloqueio em canais sem *buffer* até que uma operação de recebimento esteja disponível.

4.1.8 Envio Bloqueado em `SafeChannel` Não Buferizado

O objetivo deste teste (código 36)é verificar que o `SafeChannel` bloqueia corretamente o envio em um canal não buferizado até que uma operação de recebimento esteja disponível.

Código 36 – Send sem Receive caso Não Buferizado

```
func TestSafeChannel_SendWithoutReceiveUnbuffered(t *testing.T) {
  sc := MakeSafechannel[int]()
  sc.EnableNotifications(4)
  done := make(chan error)

  go func() {
    err := sc.Send(42)
    done <- err
  }()
  go func() {
    var messages [4]string
    index := 0
    for {
      notification, _ := sc.ReadNotification()
      t.Log(notification.Message)
      index++
      if index == len(messages) {
        break
      }
    }
  }()

  select {
  case <-done:
    t.Fatal("Test failed: Send should block until a receive occurs")
  case <-time.After(500 * time.Millisecond):
    t.Log("Send is correctly blocked in unbuffered SafeChannel")
  }

  go func() {
    _, _ = sc.Receive()
  }()

  time.Sleep(200 * time.Millisecond)
  select {
  case err := <-done:
    if err != nil {
      t.Errorf("Unexpected error on send: %v", err)
    } else {
      t.Log("Send completed successfully after receive")
    }
  case <-time.After(1 * time.Second):
    t.Fatal("Test failed: Send did not unblock after a receive")
  }
}
```

Este teste garante que o `SafeChannel` respeita o comportamento esperado de bloqueio em *channels* não buferizados, desbloqueando o envio apenas quando uma operação de recebimento for chamada.

4.1.9 Operação de Select

O objetivo deste teste (código 37) é assegurar o funcionamento do `Select` em um cenário simples e sem erros, garantindo que a função é capaz de executar e retornar os valores esperados.

Código 37 – Operação de Select

```
func TestSafeChannel_CustomSelect(t *testing.T) {
  sc := MakeSafechannel[int](1)

  err := sc.Send(42)
  if err != nil {
    t.Errorf("Unexpected error on send: %v", err)
  }

  idx, value, err := Select(
    CaseReceive(sc, nil),
  )

  if err != nil {
    t.Errorf("Unexpected error on select: %v", err)
  }
  if idx != 0 {
    t.Errorf("Expected case index 0, got %v", idx)
  }
  if value.(int) != 42 {
    t.Errorf("Expected value 42, got %v", value)
  }
}
```

Na função de teste acima (código 37), é feita uma operação de `Send` ao canal `SafeChannel`, que será não bloqueante já que há espaço no *buffer*, e em seguida é chamada a função `Select` com um `CaseReceive` para realizar a leitura da mensagem enviada àquele canal.

4.1.10 Operação de Select com Caso Default

O objetivo deste teste (código 38) é validar o funcionamento da função de `Select` do `SafeChannel` quando há um caso padrão (`CaseDefault`), verificando se o mesmo será corretamente executado quando nenhuma mensagem estiver disponível.

Código 38 – Operação de Select com Caso Default

```
func TestSafeChannel_CustomSelectDefault(t *testing.T) {
  sc := MakeSafechannel[int](1)

  go func() {
    time.After(50 * time.Millisecond)
    sc.Send(42)
  }()

  idx, value, err := Select(
    CaseReceive(sc, func(msg int) {
      t.Errorf("CaseReceive executed unexpectedly with message %v", msg)
    }),
    CaseDefault(func() {
      t.Log("Default case executed")
    }),
  )

  if idx != -1 || value != nil || err != nil {
    t.Errorf("Expected default case execution
              (idx=-1, nil value, nil error),
              got idx: %v, value: %v, err: %v",
              idx, value, err
            )
  }
}
```

Este teste faz uma chamada ao `Select` com um `CaseReceive` e um `CaseDefault` antes de ocorrer uma operação de envio no `SafeChannel`. Como há o caso padrão, a tentativa de receber a mensagem não irá bloquear e o caso padrão será executado.

4.1.11 Notificações do `SafeChannel`

Este teste (código 39) avalia se as notificações fornecidas pelo `SafeChannel` funcionam corretamente, garantindo que mensagens relevantes sejam emitidas durante operações como envio e recebimento.

Código 39 – Notificações

```
func TestSafeChannel_Notifications(t *testing.T) {
  sc := MakeSafechannel[int](1)
  sc.EnableNotifications(10)

  go func() {
    err := sc.Send(42)
    if err != nil {
      t.Errorf("Unexpected error on send: %v", err)
    }
  }()

  notification, err := sc.ReadNotification()
  if err != nil {
    t.Errorf("Failed to read notification: %v", err)
  }
  if notification.Message != "sent successfully" {
    t.Errorf("Expected 'sent successfully', got %v", notification.
       Message)
  }
}
```

O teste valida a funcionalidade do canal de notificações, verificando se mensagens sobre o sucesso do envio são emitidas corretamente.

4.2 DESEMPENHO

Nesta seção, o desempenho do `SafeChannel` é analisado em comparação aos *channels* nativos do Go. Foram realizados testes de *benchmark*, simulando um grande número de operações de envio e recebimento de mensagens. O objetivo foi avaliar o impacto do `SafeChannel` em relação à solução nativa, considerando diferentes tamanhos de *buffer*.

Os testes foram implementados utilizando o pacote `testing`. A função `b.N`, fornecida pelo pacote, é usada para determinar o número de iterações realizadas durante os *benchmarks*.

As especificações da máquina utilizada para os testes são:

- Sistema operacional: Windows (`goos: windows`);
- Arquitetura: 32 bits (`goarch: 386`);
- Processador: Intel(R) Core(TM) i5-7600K CPU @ 3.80GHz.

O código de *benchmark* utilizado é apresentado a seguir:

Código 40 – Código de Benchmark - Channel

```
import (
  "testing"
)

const numOperations, bufferSize = 1000000, 100

func BenchmarkNativeChannel(b *testing.B) {
  for i := 0; i < b.N; i++ {
    ch := make(chan int, bufferSize)
    b.StartTimer()
    go func() {
      for j := 0; j < numOperations; j++ {
        ch <- j
      }
      close(ch)
    }()
    for v:= range ch {
      _ = v
    }
    b.StopTimer()
  }
}

func BenchmarkSafechannel(b *testing.B) {
  for i := 0; i < b.N; i++ {
    sc := MakeSafechannel[int](bufferSize)
    b.StartTimer()
    go func() {
      for j := 0; j < numOperations; j++ {
        sc.Send(j)
      }
      sc.Close()
    }()
    for {
      _, err := sc.Receive()
      if err != nil {
        break
      }
    }
    b.StopTimer()
  }
}
```

O código 40 realiza um *benchmark* comparativo entre os *channels* nativos e o `SafeChannel`. Ambos os cenários executam 1000000 operações de envio e recebimento, com *buffer* de tamanho 100. No caso do `SafeChannel`, as operações utilizam as funções `Send` e `Receive`, enquanto nos *channels* nativos, é usado o operador padrão do Go (`<-`).

Os testes de *benchmark* geram as seguintes métricas principais:

- Iterações (`N`): Representam o número de vezes que o *benchmark* foi executado pelo pacote `testing`. O número de iterações é ajustado dinamicamente para garantir resultados confiáveis e consistentes, considerando o tempo necessário para realizar as operações testadas.
- Nanossegundos por operação (`ns/op`): Indica o tempo médio, em nanossegundos, necessário para executar uma única operação. Essa métrica é calculada dividindo o tempo total do teste pelo número de iterações e reflete a eficiência temporal da implementação.
- Bytes por operação (`B/op`): Representa a quantidade média de memória alocada (em bytes) por operação. Essa métrica é útil para avaliar o impacto da implementação no uso de memória, especialmente em sistemas que lidam com muitas operações simultâneas.
- Alocações por operação (`allocs/op`): Indica o número médio de alocações de memória realizadas durante cada iteração do teste. Essa métrica ajuda a identificar o custo das operações em termos de alocações dinâmicas de memória, que podem afetar o desempenho geral do sistema.

Os resultados obtidos estão descritos abaixo:

- `Channel`: `14 iterações, 78652043 ns/op, 181 B/op, 2 alocações/op.`
- `SafeChannel`: `3 iterações, 359795600 ns/op, 602 B/op, 5 alocações/op.`

Pode-se observar que, em um mesmo período de tempo, o teste com o *channel* executou 14 iterações, enquanto o teste com o `SafeChannel` executou apenas 3. O tempo médio por operação foi, aproximadamente, 4.6 vezes maior para as operações sobre `SafeChannel`. As operações realizadas com o `SafeChannel` consumiram, em média, 602 *bytes* por operação, comparados aos 181 *bytes* por operação do *channel*, refletindo um aumento substancial no uso de memória. O número de alocações por operação também foi maior, com 5 alocações em média para o `SafeChannel` em comparação com 2 para o *channel*.

Os resultados demonstram que o `SafeChannel` apresenta uma sobrecarga considerável em relação aos *channels* nativos. A lógica adicional implementada, como sincronização por meio de `Mutex`, gerenciamento de mensagens entre os *channels* internos, e tratamento de estados como fechamento de *channels*, contribui para esse impacto.

Embora a solução nativa de Go seja mais eficiente em termos de desempenho, o `SafeChannel` oferece maior segurança e robustez, além de ferramentas para depuração do código. Ele previne *bugs* relacionados ao uso de *channels*, como envio para *channels* fechados ou bloqueios devido a *channels* nulos, justificando seu uso em sistemas onde a confiabilidade é uma prioridade.

5 CONCLUSÃO

O uso de concorrência é um elemento importante no desenvolvimento de aplicações eficientes e escaláveis, e a linguagem Go se destaca por oferecer um modelo baseado na comunicação entre *goroutines* por meio de *channels*. No entanto, apesar de sua simplicidade aparente, a manipulação incorreta dessas primitivas, proveniente em partes da falta de familiaridade dos desenvolvedores com o modelo de troca de mensagens, pode resultar em diversos *bugs* concorrentes, como *deadlocks*, *starvation*, *data race*, entre outros. Diante desses desafios, este trabalho propôs a criação do `SafeChannel`, um *wrapper* para *channels*, projetado para oferecer maior segurança e confiabilidade no uso dessas estruturas por meio de uma interface simples que abstrai do programador os processos internos de verificação e controle, garantindo o funcionamento correto da comunicação entre *goroutines* e prevenindo a ocorrência de erros concorrentes.

A implementação do `SafeChannel` seguiu uma abordagem que encapsula a criação, envio e recebimento de mensagens, adicionando verificações internas para evitar cenários propensos a falhas. Foram desenvolvidos mecanismos de controle para impedir o fechamento indevido de *channels*, assim como operações sobre *channels* que já foram encerrados. Outro diferencial foi a inclusão de um sistema de notificações, permitindo que eventos sejam registrados e analisados em tempo de execução, auxiliando na depuração do programa.

Os testes realizados demonstraram que o `SafeChannel` consegue mitigar diversos problemas comuns no uso de *channels*, aumentando a previsibilidade e a segurança das operações concorrentes. Contudo, os *benchmarks* indicaram que essa robustez tem um custo de desempenho em comparação com *channels* nativos. O `SafeChannel` apresentou tempos de execução superiores aos *channels* convencionais, especialmente em cenários com *buffers* menores. Apesar disso, sua utilização pode ser justificada em aplicações onde a confiabilidade é prioritária, como em sistemas que exigem alta segurança na troca de mensagens entre *goroutines*, ou sistemas onde o alto desempenho não é essencial.

Como trabalhos futuros, há diversas oportunidades para aprimorar o `SafeChannel`. Uma das principais direções é a otimização do desempenho, reduzindo a sobrecarga introduzida pelos mecanismos de controle e sincronização. Talvez seja possível alcançar o mesmo resultado sem a criação de dois *channels* internos, ou encontrar uma lógica mais eficiente para a troca de mensagens entre eles. Outra possibilidade seria ampliar o escopo do pacote para lidar com um conjunto mais abrangente de *bugs* de concorrência, seja prevenindo sua ocorrência ou fornecendo formas mais eficazes de tratá-los. Pode ser explorada também a expansão do sistema de notificações, tornando-o mais informativo e flexível, permitindo que os desenvolvedores personalizem os tipos de eventos monitorados e suas respectivas ações.

Dessa forma, este trabalho contribui para o aprimoramento do modelo de concorrência baseado em troca de mensagens em Go, oferecendo uma alternativa mais segura para o uso de *channels* e possibilitando futuras pesquisas e melhorias nessa área.

REFERÊNCIAS

ARPACI-DUSSEAU, R. H.; ARPACI-DUSSEAU, A. C. **Operating Systems: Three Easy Pieces**. Madison, WI: Arpaci-Dusseau Books, 2018.

BEN-ARI, M. **Principles of Concurrent and Distributed Programming**. 2. ed. [S.l.]: Addison-Wesley Professional, 2005.

CHENEY, D. **Channel Axioms**. 2014. Último acesso em 07 de janeiro de 2025. Disponível em: https://dave.cheney.net/2014/03/19/channel-axioms.

DILLEY, N.; LANGE, J. An empirical study of messaging passing concurrency in go projects. In: **2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)**. [S.l.: s.n.], 2019. p. 377–387.

DONOVAN, A. A. A.; KERNIGHAN, B. **The Go Programming Language**. [S.l.]: Addison-Wesley, 2015.

GOETZ, B. et al. **Java Concurrency in Practice**. [S.l.]: Pearson Education, 2006.

HERLIHY, M.; SHAVIT, N. **The Art of Multiprocessor Programming**. [S.l.]: Morgan Kaufmann, 2008.

LU, S. et al. Learning from mistakes – a comprehensive study on real world concurrency bug characteristics. In: **Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation (PLDI)**. Tucson, Arizona, USA: ACM, 2008. p. 329–339.

MOUSSA, P. **SafeChannel**. 2024. Último acesso em 19 de fevereiro de 2025. Disponível em: https://github.com/pedromoussa/tcc.

NAJAFIZADEH, A.; JAVADI, S. A. **Jump over Golang channels**. 2023.

NETZER, R. H. B.; MILLER, B. P. What are race conditions? some issues and formalization. **ACM SIGPLAN Notices**, ACM, New York, NY, USA, v. 29, n. 4, p. 66–77, 1994.

OLIO, J. **Go Channels Are Bad and You Should Feel Bad**. 2016. Blog post, last edited on March 4, 2019. Disponível em: https://www.jtolio.com/2016/03/go-channels-are-bad-and-you-should-feel-bad/.

SILBERSCHATZ, A.; GALVIN, P. B.; GAGNE, G. **Operating System Concepts**. 10th. ed. Hoboken, NJ: John Wiley & Sons, 2018.

SOARES, J. L. da S. G. **Um Estudo Sobre os Mecanismos de Concorrência da Linguagem Go**. Monografia (Trabalho de Conclusão de Curso - Graduação) — Universidade Federal do Rio de Janeiro, Instituto de Matemática, Curso de Bacharelado em Ciência da Computação, Rio de Janeiro, Brasil, 2019.

SYNC PACKAGE. **sync package - Go Documentation**. 2025. Documentação online do pacote sync da linguagem de programação Go. Última atualização: 16 de janeiro de 2025. Disponível em: https://pkg.go.dev/sync.

TIPIRNENI, D. S. **An Empirical Study of Concurrent Feature Usage in Go**. Dissertação (Mestrado) — East Carolina University, December 2022. Master's Thesis. Disponível em: http://hdl.handle.net/10342/12295.

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**. New York, NY, USA: Association for Computing Machinery, 2019. (ASPLOS '19), p. 865–878. ISBN 9781450362405. Disponível em: https://doi.org/10.1145/3297858.3304069.