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

Alonso L. Fritz[1], **Guilherme Galante**[1], **Marcio Oyamada**[1]

[1]Centro de Ciëncias Exatas e Tecnológicas – Universidade Estadual do Oeste do Paraná (UNIOESTE) Cascavel – PR – Brasil

alonso.fritz, guilherme.galante, marcio.oyamada @{unioeste.br}

Resumo. *Com a predominância de hardware multi-core, a demanda por softwares concorrentes tem aumentado. Porém escrever programas concorrentes corretos é uma tarefa pouco trivial, e lidar com esse desafio requer ainda hoje avanços em várias direções, incluindo a análise de programas concorrentes, a detecção de bugs de concorrência, padrões de correções de bugs, natureza da manifestação de bugs e outros. Este trabalho apresenta um estudo sobre as características de bugs de concorrência em aplicações escritas na linguagem Go. Foram analisados os padrões de bugs de concorrência, as causas raízes, as características referentes a manifestação, como a quantidade de goroutines e primitivas envolvidas, além das estratégias de correção de* 90 *bugs de concorrência selecionadas aleatoriamente de três aplicações:* `Docker`*,* `Terraform` *e* `CockroachDB`*. Os resultados indicam que* 72% *dos bugs registrados utilizam comunicação por memória compartilhada; e somente* 15% *dos bugs não-bloqueantes foram relacionados a passagem de mensagem. Esse estudo visa fornecer uma melhor compreensão dos modelos de concorrência da linguagem Go auxiliando no desenvolvimento mais confiável e seguro, e de ferramentas de diagnóstico para Go e linguagens para programação concorrente.*

1. Introdução

Os ganhos de desempenho mononúcleo em processadores está praticamente estagnado [Hennessy and Patterson 2011]. Nesse contexto, a introdução de processadores com múltiplos núcleos tem se destacado como uma estratégia para obter desempenho, contudo, para extrair o desempenho em tais processadores é necessário que aplicação seja desenvolvida para execução concorrente. No contexto de sistemas de propósito geral, escrever programas concorrentes não é trivial e essa complexidade tem atingido a comunidade de desenvolvimento de *software*, pois escrever programas concorrentes de boa qualidade e corretos tornou-se importante [Stoica et al. 2014].

A natureza não determinística dos programas concorrentes torna difícil para os desenvolvedores detectarem, diagnosticarem e corrigirem *bugs*. Estes chamados *bugs* de concorrência, são causados por problemas de sincronização em *software multithread* e podem causar bilhões de dólares em perdas econômicas, tornando-se uma ameaça à confiabilidade do uso de *softwares multithread* [Lu et al. 2008].

Os estudos de Lu et al. (2008), sobre *softwares* de código aberto, mostraram que geralmente leva vários meses para os desenvolvedores corrigirem corretamente os *bugs* de concorrência. Além disso, conforme descrito por Yin et al.(2001) os *bugs* de concorrência são os mais difíceis de corrigir corretamente entre os tipos de *bugs* comuns, geralmente

com muitos *patches* incorretos lançados. Portanto, desenvolver programas concorrentes de forma correta é um desafio que ainda necessita de conhecimento e avanços em diversos cenários, incluindo entre eles a detecção de *bugs* de concorrência, teste e verificação de modelo de programas concorrentes e o projeto de uma linguagem de programação concorrente.

A linguagem Go, originalmente desenvolvida pela Google em 2009 [GO 2023], tem como objetivo aprimorar a programação concorrente em comparação com as linguagens tradicionais *multithreaded*. Para alcançar esse objetivo, a linguagem Go concentra seu projeto de concorrência em dois princípios essenciais: primeiramente tornar as *threads*, chamadas *goroutines*, leves e de fácil criação; e em segundo adotar o uso de mensagens explícitas, conhecidas como canais ou *channels*, para facilitar a comunicação entre essas *threads*. No estudo realizado por Tu et al. (2019) sobre *bugs* de concorrência na linguagem Go, para as seguintes aplicações *open-source*: `Docker`, `Kubernetes`, `gRPC`, `etcd`, `CockroachDB` e `BoltDB`, o autor avaliou que, existe uma incerteza se os mecanismos de concorrência realmente tornam a programação em Go mais acessível e menos propensa a erros de concorrência quando comparada às abordagens tradicionais.

Este trabalho tem como objetivo fazer uma busca nos repositórios *open-source* de três aplicações populares em Go verificando a ocorrência de *bugs* de concorrência. Assim como realizado por Tu et al. (2019) e Lu et al. (2008) , o objetivo desse trabalho não é corrigir *bugs* de concorrência, mas sim estudar e classificar quais os fatores que ocasionaram os *bugs*, quais as características necessárias para sua manifestação e como ele foi corrigido. Será empregada uma metodologia própria para a classificação dos *bugs*, desenvolvida a partir das metodologias utilizadas por Tu et al. (2019) e Lu et al. (2008) buscando entender os *bugs* de concorrência em Go.

2. Fundamentação Teórica e Trabalhos relacionados

Go (2023) é uma linguagem de propósito geral explicitamente criada com a programação de sistemas em mente. Desenvolvido como um projeto de código aberto, a linguagem Go tem como proposta aprimorar a produtividade do programador, oferecendo uma linguagem expressiva, concisa e eficiente, com forte suporte para programação concorrente. A linguagem é fortemente tipada e possui coleta de lixo.

Goroutines são uma das unidades mais básicas de organização em um programa Go e simplificam o uso de concorrência pois são mapeadas para funções que operam de maneira independente ou como *coroutines*, multiplexadas em um conjunto de *threads* em um Sistema Operacional. O sistema de tempo de execução, conforme destacado por [Pike 2023], gerencia automaticamente situações bloqueantes, movendo *goroutines* entre *threads*, com a proposta de garantir eficiência na utilização dos recursos do sistema.

A singularidade das *goroutines* no Go está relacionada à sua profunda integração com o ambiente de execução da linguagem. Ao contrário de outras implementações, as *goroutines* não especificam seus próprios pontos de suspensão ou reentrada. O ambiente de execução do Go monitora dinamicamente o comportamento em tempo de execução das *goroutines*, suspendendo-as automaticamente quando bloqueadas e retomando-as quando desbloqueadas.

Os *channels* na linguagem Go são primitivas de sincronização inspiradas no CSP de Hoare (1978), são empregados não apenas para sincronizar acesso à memória, mas

principalmente para facilitar a comunicação entre *goroutines*, conforme destacado por Cox-Buday (2018). Funcionando como condutores de fluxo de informações, os canais permitem a transmissão de valores entre partes distintas do programa, proporcionando uma forma eficiente de compor funcionalidades em programas de qualquer tamanho.

2.1. Trabalhos relacionados

Estudo realizado por Lu et al. (2008) abrange as características reais de *bugs* de concorrência, em aplicações em linguagem C, examinando padrões de *bugs* de concorrência, manifestações e estratégias de correção de 105 *bugs* de concorrência em aplicações de código aberto representativas tais como `MySQL`, `Apache`, `Mozilla` e `OpenOffice`. O processo de coleta de *bugs* foi realizado utilizando um conjunto de palavras-chave como *"race(s)", "deadlock(s)", "synchronization(s)", "concurrency", "lock(s)", "mutex(es)", "atomic", "compete(s)"*, e suas variações, filtrando assim meio milhão de relatórios de *bug*. Destes, foram escolhidos aleatoriamente 500 relatórios com descrições de causa raiz, códigos-fonte e informações de correção de *bugs*. Depois de verificado manualmente para garantir que os *bugs* fossem realmente causados por suposições erradas dos programadores sobre a execução concorrente, dessa forma chegou-se a 105 *bugs* de concorrência.

Nesse estudo os *bugs* foram separados em dois tipos sendo eles de *deadlock*, que ocorrem quando duas ou mais *threads* esperam circularmente uma pela outra para liberar o recurso adquirido; e de não *deadlock*, que ocorrem essencialmente quando as intenções de sincronização são violadas. Lu et al. (2008) adota também a subclassificação com base em suas causas raízes, que podem ser *bugs* de violação de ordem e *bugs* de violação de atomicidade. Dos 105 *bugs*, 74 são *bugs* de concorrência não relacionados a *deadlock* enquanto 31 são *bugs* de *deadlock*. Portanto, grande parte dos *bugs* não causam um bloqueio na execução da aplicação, sendo detectados pelo comportamento não esperado na resposta.

No trabalho de Tu et al. (2019), um estudo empírico sobre *bugs* de concorrência em aplicações implementadas com a linguagem Go é apresentada. Foram estudados 171 *bugs* de concorrência em seis aplicações de código aberto dentre eles `Docker`, `Kubernetes`, `etcd`, `gRPC`, `CockroachDB` e `BoltDB`. Os autores buscam definir qual mecanismo de comunicação entre *threads* (passagem de mensagem ou memória compartilhada) resulta em um número menor de *bugs*. A linguagem Go oferece estruturas para ambos os modelos, porém incentiva o uso de passagem de mensagem, utilizando a primitiva *channels* da linguagem, com a crença de que a passagem explícita de mensagens é menos propensa a erros. No processo de coleta de *bugs* foi realizado uma busca nos históricos de *commits* do Github, nos repositórios das aplicações definidas no estudo, buscando em seus logs pelas seguintes palavras-chave: *"race", "deadlock", "synchronization", "concurrency", "lock", "mutex", "atomic", "compete", "context", "once"* e *"goroutine leak"*. O autor descreve que foram encontrados 3211 *commits* distintos correspondentes aos critérios de busca, uma amostragem aleatória foi retirada e foram estudados 171 *bugs*. O estudo classifica os *bugs* por duas visões, uma relacionada ao comportamento e outra a causa. A dimensão baseada no comportamento separa os *bugs* entre *bloqueantes* e *não-bloqueantes*, sendo os *bloqueantes* aqueles onde uma ou mais *goroutines* ficam involuntariamente presas em sua execução e não conseguem avançar e os *não-bloqueantes* aqueles que em vez disso, todas as goroutines podem concluir suas tarefas, mas o resultado não é o esperado. A dimensão da causa separa os *bugs* em aque-

les causados pelo uso inadequado de memória compartilhada e aqueles causados pelo uso inadequado de passagem de mensagens. Constatou-se que dos 171 *bugs*, existem um total de 85 *bugs bloqueantes* e 86 *bugs não-bloqueantes*, e há um total de 105 *bugs* causados por proteção incorreta de memória compartilhada e 66 *bugs* causados por passagem incorreta de mensagens.

Este trabalho é similar ao proposto por Tu et al. (2019), e realizada a busca por *bugs* de concorrência em repositórios de aplicações *open-source* escritas em linguagem Go. No entanto, a busca é realizada em um período mais recente (de 2020 a 2023), e avalia se a tendência dos tipos de *bugs* ainda se mantém. Como diferencial ao trabalho de Tu et al. (2019), o trabalho além de classificar o *bugs*, busca avaliar a estratégia adotada na correção e como os *bugs* foram detectados.

3. Metodologia

Foram selecionadas três aplicações de código aberto, sendo um sistema de contêineres `Docker` (2023) , um sistema de banco de dados chamado `CockroachDB` (2023) e a ferramenta `Terraform` (2023) , utilizada para provisionar e gerenciar a infraestrutura de nuvem em *IaC* (Infraestrutura como Código).

Para coletar *bugs* de concorrência, foi filtrado os históricos de *commits* dos repositórios no Github das três aplicações, buscando em seus registros por palavras-chave relacionadas a concorrência: *"race"*, *"deadlock"*, *"synchronization"*, *"concurrency"*, *"lock"*, *"mutex"*, *"atomic"*, *"compete"*, *"context"*, *"once"* e *"goroutine leak"*. Essas palavras chave são as mesmas utilizadas em estudos de *bugs* de concorrência de Lu et al. (2008) e também no trabalho de Tu et al. (2019) , o qual adiciona as novas palavras a busca como *"context"* e *"once"* pois estão relacionadas a novas primitivas ou bibliotecas de concorrência introduzidas pela linguagem Go e *"goroutine leak"*, que está relacionada a um problema específico durante a execução de aplicações Go, onde a não finalização de *goroutines* pode levar a problemas de estouro de memória.

Além das palavras chaves, foram filtrados *logs* de *commits* que tenham sido criados a partir de 2020 e que já tenham sido fechados antes de 2024 (2020-2023) para as aplicações `Docker` e `CockroachDB`, isso porque o trabalho realizado por Tu et al. (2019) contempla os *bugs* registrados nessas aplicações até 2019, portanto analisando um período diferente. Para o caso da aplicação `Terraform`, além de ser relativamente nova e menor se comparada com as outras duas, também não possuía nenhum estudo prévio quanto aos seus *bugs* antigos, dessa maneira não houve restrição de data para os *bugs* selecionados para o `Terraform`. Foram encontrados no total 31049 resultados nas pesquisas realizadas para as três aplicações que correspondem com o critério de busca estabelecido. A Tabela 1 apresenta a quantidade de resultados encontrados para cada aplicação.

Tabela 1. Quantidade total de resultados para os critérios de busca

Aplicação	**# de *logs* de *commits***
`Docker`	857
`Terraform`	8647
`CoackroachDB`	21545
Total	31049

Considerando a quantidade total de resultados retornados pela pesquisa, foram pré-selecionados manualmente 5 *bugs* para cada palavra-chave em cada aplicação, totalizando 165 *bugs* . Para o estudo, dos 165 *bugs* foram selecionados aleatoriamente 30 *bugs* de cada aplicação. Adicionalmente, eles foram identificados e verificados manualmente para garantir que os *bugs* sejam realmente causados por suposições erradas dos programadores sobre a execução concorrente ou por problemas na utilização de primitivas e paradigmas concorrentes do projeto da linguagem. Foram assim selecionados um total de 90 *bugs* com detalhes sobre a causa, códigos-fonte e informações sobre sua correção.

3.1. Classificação

Para realizar a classificação dos *bugs* utilizou-se uma mescla entre os métodos propostos por Lu et al. (2008) e Tu et al. (2019). Dessa forma a análise é realizada a partir dos aspectos das características de *bugs* de concorrência: padrão de *bug*, causa e estratégia de correção, conforme apresentado na Figura 1.

Na análise utiliza-se a definição da dimensão de comportamento adotada por Tu et al. (2019), separando em *bugs* bloqueantes, quando uma ou mais *goroutines* ficam involuntariamente presas em sua execução e não conseguem avançar, sendo uma definição mais ampla do que *deadlocks* e inclui situações em que não há espera circular; e *bugs não-bloqueantes* que ocorrem sempre que todas as *goroutines* podem concluir suas tarefas, mas seus comportamentos ou resultados não são os corretos.

No aspecto do padrão de *bug*, eles são classificados em duas categorias utilizando a segunda dimensão de classificação proposta por Tu et al. (2019), aqueles ocasionados na comunicação utilizando o paradigma de memória compartilhada, ou e aqueles causados por uso indevido de passagem de mensagem.

<table>
<tr><td colspan="10">Bugs</td></tr>
<tr><td colspan="6">Bloqueantes</td><td colspan="4">Não-Bloqueantes</td></tr>
<tr><td colspan="3">Memória Compartilhada</td><td colspan="3">Passagem de Mensagem</td><td colspan="3">Memória Compartilhada</td><td colspan="1">Passagem de Mensagem</td></tr>
<tr><td>Mutex</td><td>RWMutex</td><td>Wait</td><td>Channel</td><td>Channel c/</td><td>Lib</td><td>Tradicional</td><td>WaitGroup</td><td>Lib</td><td>Channel</td><td>Lib</td></tr>
</table>

Figura 1. Diagrama de classificação de *bugs*

E por último, para a estratégia de correção de *bugs* estuda-se tanto a estratégia de correção do *patch* final, quanto os erros em *patches* intermediários. Também analisa-se a vida útil dos *bugs*, ou seja, o tempo desde a adição do código com *bug* no *software* até sua correção no *software*. Como observado a maioria dos *bugs* estudados por Tu et al. (2019) possui uma longa vida útil, ou seja, demoram para serem detectados desde sua adição ao código, porém assim que são identificados são rapidamente corrigidos, demonstrando a dificuldade de detectar esses tipos de erros.

4. Resultados e discussões

Para entender melhor os padrões de usos de *goroutines* em aplicações Go de mundo real, utilizando o mesmo método de Tu et al. (2019) realizou-se a coleta de dados referente a quantidade de locais de criação de *goroutines*, mais especificamente dividindo-as entre

goroutines que usam funções normais e aquelas que usam funções anônimas. Os dados referentes a quantidade de linhas de código por aplicação, foram coletados utilizando o *VScode* em conjunto com a extensão *VSCode Counter*. A Tabela 2, apresenta os resultados em termos de quantidade de criações de *goroutines* com funções normais e com funções anônimas, quantidade total de criação de *goroutines* e a média por mil linhas de código.

Tabela 2. Utilização de goroutines

Aplicação	Normal Func	Anon. Func	Total	Por 1000 linhas
`Terraform`	41 (40%)	62 (60%)	103	0.638
`Docker`	249 (36%)	448 (64%)	696	0.513
`CockroachDB`	48 (19%)	205 (81%)	253	0.176

As três aplicações usam mais funções anônimas do que funções normais, as proporções de uso entre as aplicações é semelhante para a aplicação `Terraform` e `Docker`, porém para `CockroachDB` podemos notar uma proporção maior de funções anônimas utilizadas, representando 81% do total das *goroutines* criadas na aplicação.

A aplicação `Docker`, quando comparadas as estatísticas dos padrões de usos de *goroutines* coletadas por [Tu et al. 2019] em 2019, com as coletadas em 2024 para este estudo, é notável o crescimento na quantidade de chamadas para criação de *goroutines* . A Tabela 3, apresenta uma comparação entre os valores coletados em ambos os trabalhos. A quantidade total de chamadas de criação de *goroutines* aumentou aproximadamente 379% de 2019 para 2024, o aumento de *goroutines* com funções normais foi de aproximadamente 654%, enquanto que o aumento de *goroutines* com funções anônimas foi de 300%, indicando o aumento do uso de primitivas de programação concorrente nesta aplicação.

Tabela 3. Comparativo do uso de goroutines no Docker em 2019 [Tu et al. 2019] e estatísticas coletadas em 2024

Docker	Normal Func	Anon. Func	Total	Por 1000 linhas
2019	33 (23%)	112 (77%)	145	0.180
2024	249 (36%)	448 (64%)	696	0.513

.

4.1. Estudo de padrões dos *bugs*

Para melhor entender os *bugs* de concorrência, os mesmos foram divididos em duas categorias principais: os *bloqueantes*, que incluem problemas em que um conjunto de *goroutines* não pode avançar devido a um tipo de bloqueio; e *não-bloqueantes*, que incluem problemas que não envolvem bloqueio de *goroutines*. A Tabela 4 apresenta essa classificação.

Dos 90 *bugs* selecionados, 49 são bloqueantes, enquanto 41 são identificados como *não-bloqueantes*. Analisando por aplicação específica, para o `Docker`, os *bloqueantes* representam 66% dos *bugs* selecionados; `CockroachDB` registrou um total de 60% de *bugs* bloqueantes; enquanto para o `Terraform` os *bugs bloqueantes* representam 37% da amostragem selecionada.

A análise da distribuição dos *bugs* pode ser realizada a partir do aspecto padrão de *bug*, analisando como as *goroutines* se comunicam entre si, seja por meio de memória

Tabela 4. Total de *bugs*. Categorizados entre *bloqueantes* e *não-bloqueantes*.

Aplicação	*Bugs*		Total
	Bloqueantes	***Não-bloqueantes***	
`Docker`	20	10	30
`Terraform`	11	19	30
`CockroachDB`	18	12	30
Total	49	41	90

compartilhada ou passagem de mensagem. Do total analisado, 72% (65 de 90) *bugs*, utilizam o paradigma de compartilhamento de memória para comunicação, enquanto 28% (25 dos 90) tinham como padrão a passagem de mensagem. Esses dados demonstram que *bugs* como comunicação utilizando o paradigma de memória compartilhada tendem a ser mais comuns, sendo a grande parte dos *bugs* analisados. Podemos considerar que o compartilhamento de memória é mais propenso a erros devido à complexidade de implementação para garantir as intenções do programador. Além disso, a frequência na utilização de memória compartilhada pode ser um indício de que mesmo com a possibilidade da passagem de mensagens, os desenvolvedores encontrem certas dificuldades de codificar dessa maneira e acabam optando por continuar utilizando estruturas de memória compartilhada tradicionais.

A Tabela 5 traz uma visão ampla da divisão dos *bugs* a partir de seu aspecto categoria e padrão. Os *bugs bloqueantes* representam somente 45% (29 de 65) daqueles que se comunicam a partir de memória compartilhada. Por outro lado, aproximadamente 88% dos *bugs não-bloqueantes* estão relacionados com o padrão de memória compartilhada. Os resultados obtidos são próximos aos observados pelo estudo de [Tu et al. 2019], onde 45% dos *bugs* que tenham como padrão a memória compartilhada são bloqueantes, enquanto que 80% dos *bugs* por passagem de mensagem são bloqueantes.

Tabela 5. Total de *bugs* em Aplicações classificados por Categoria e Padrão de *bug*.

Aplicação	*Bugs*			
	Bloqueantes		***Não-Bloqueantes***	
	Memória Compartilhada	**Passagem de Mensagem**	**Memória Compartilhada**	**Passagem de Mensagem**
`Docker`	13	7	7	3
`Terraform`	4	7	17	2
`CockroachDB`	12	6	11	1
Total	29	20	36	5

Para detectar e evitar *bugs* bloqueantes, é importante entender suas causas. A partir do aspecto de causa, é examinado qual operação bloqueia uma *goroutine* e porque a operação não é desbloqueada por outras *goroutines*. A Tabela 6 representa como os *bugs* foram classificados, com relação as causas.

Os acessos à memória compartilhada são notoriamente difíceis de programar corretamente e sempre foram um dos principais focos de pesquisa sobre *deadlock* e continuam sendo a causa raiz de *bugs bloqueantes* em aplicações Go. Aproximadamente 90%

Tabela 6. Aspecto Causa: *Bugs bloqueantes*

Bugs bloqueantes					
49					
Memória Compartilhada			**Passagem de Mensagem**		
29			20		
Mutex	**RWMutex**	**Wait**	**Channel**	**Channel c/**	**Lib**
24	2	3	13	5	2

(26 de 29) dos *bugs bloqueantes* relacionados a memória compartilhada são causados pelo uso indevido de bloqueios, *Mutex* ou *RWMutex*, incluindo dupla aplicação de bloqueios e desbloqueios, aplicação de bloqueios em ordens conflitantes, esquecimento de bloqueios e falta de liberação de bloqueios.

O uso incorreto de *channels* na passagem de mensagens entre *goroutines* foi responsável por aproximadamente 37% das causas de *bugs* bloqueantes. Dos 20 *bugs bloqueantes* por passagem de mensagem, 18 deles tem suas causas diretamente ligadas a utilização de *channels*. A maioria desses *bugs* são ocasionados pela falta de envio ou recebimento para um canal, ou pelo fechamento de um canal, resultando assim no bloqueio de uma *goroutine* que está aguardando para receber deste canal ou enviar para este canal.

Os *bugs não-bloqueantes* ocorrem quando todas as *goroutines* podem concluir suas tarefas, mas seus comportamentos não são desejados, ou os resultados não são os esperados. Na Tabela 7 a maioria dos *bugs não-bloqueantes*, quase 88%, se comunicam a partir de memória compartilhada, enquanto que aproximadamente apenas 12% dos *bugs não-bloqueantes* se comunicam utilizando a passagem de mensagens.

Tabela 7. Aspecto Causa: *Bugs não-bloqueantes*

Bugs não-bloqueantes				
41				
Memória Compartilhada			**Passagem de Mensagem**	
35			6	
Tradicional	**WaiGroup**	**Lib**	**Chan**	**Lib**
31	3	1	4	2

Pode-se notar na Tabela 7 que ao menos 85% (35 de 41) dos *bugs não-bloqueantes* coletados são causados por problemas tradicionais que também ocorrem em linguagens clássicas, como violação de atomicidade, violação de ordem e principalmente condições de corrida que costumam ocorrer devido a acessos não protegidos, ou protegidos de maneira incorreta.

No entanto, nem todos os *bugs não-bloqueantes* compartilham das mesmas causas e soluções que seus semelhantes em linguagens tradicionais. Foi encontrado ao menos um *bug* que suas causas raízes são tradicionais, mas são causadas pela falta de compreensão por parte dos programadores dos novos recursos introduzidos pela linguagem Go. Como ocorre na *issue* 112117, presente em [Terraform 2023], onde métodos que se utilizavam de *Mutex* para proteger a atualização do estado de erro da estrutura, foram substituídos por um canal para permitir que *goroutines* concorrentes notifiquem a *goroutine* principal,

eliminando assim a necessidade de usar *Mutex* complexos para proteger o acesso ao estado de erro concorrentemente.

Os *bugs* relacionados a comunicação por passagem de mensagem representam apenas 15% (6 de 41) dos *bugs não-bloqueantes* coletados. Sendo registrados três *bugs* causados pelo uso inadequado de *channels*, além de dois *bugs* relacionados a utilização de primitivas *context* e *select* com *channels*. Quando empregada de forma adequada, o paradigma de passagem de mensagens pode apresentar uma menor incidência de *bugs não-bloqueantes* em comparação com os acessos à memória compartilhada.

Com relação ao tempo de vida dos *bugs*, analisando as Tabelas 8 e 9, grande parte dos *bugs* são solucionados antes de completarem 30 dias de vida, e aproximadamente 50% são corrigidos nos 5 primeiros dias.

Tabela 8. Vida útil de *bugs bloqueantes*.

Aplicação	**Vida Util - *Bloqueantes***					
	< 1 dia	**1 a 5 dias**	**5 a 10 dias**	**10 a 30 dias**	**> 30 dias**	**Total**
`Docker`	1	6	2	2	9	20
`Terraform`	3	6	1	1	0	11
`CockroachDB`	5	7	3	1	2	18
Total	9	19	6	4	11	49

Tabela 9. Vida útil de *bugs não-bloqueantes*

Aplicação	**Vida Util - *Não-bloqueantes***					
	< 1 dia	**1 a 5 dias**	**5 a 10 dias**	**10 a 30 dias**	**> 30 dias**	**Total**
`Docker`	1	3	4	1	1	10
`Terraform`	4	7	3	2	3	19
`CockroachDB`	4	2	0	3	3	12
Total	9	12	7	6	7	41

4.2. Aspecto: Estratégias de correção

Uma abordagem geral para corrigir *bugs bloqueantes* é eliminar a causa que leva à espera indefinida de uma *goroutine*, o que, por sua vez, desbloqueará a execução do programa. Para alcançar esse objetivo, os programadores da linguagem Go frequentemente realizam ajustes nas operações de sincronização, ajustes esses que podem incluir a adição de operações ausentes, o movimento ou modificação de operações mal colocadas ou mal utilizadas, e a remoção de operações extras que não contribuem para o funcionamento correto do programa. Essas medidas visam garantir que as operações de sincronização sejam realizadas de maneira apropriada e eficiente, evitando situações de bloqueio desnecessário.

A Tabela 10 apresenta as estratégias de correção de *bugs bloqueantes*. Na memória compartilhada 26 dos 29 *bugs* bloqueantes, foram corrigidos por métodos semelhantes às correções tradicionais. Desses 26 *bugs* relacionados a *Mutex* e *RWMutex*, 18 foram corrigidos adicionando *locks* ou *unlocks* ausentes, 2 foram corrigidos movendo operações de *lock* e *unlock* para os locais corretos, 5 foram corrigidos alterando lógicas

Tabela 10. Estratégias de Correção: *Bugs bloqueantes*

	Adiciona	Move	Altera	Remove
Memória Compartilhada				
Mutex	17	2	4	1
Wait	1	0	2	0
RWMutex	1	0	1	0
Passagem de Mensagem				
Chan	7	0	1	0
Chan c/	3	1	5	1
Lib	2	0	0	0
Total	31	3	13	2

de bloqueio e intenção na utilização de *Mutex* e *RWMutex* e 1 *bug* foi corrigido removendo uma operação de *unlock*. Uma outra estratégia adotada para corrigir problemas de memória compartilhada é alterar a comunicação para *channels*, visando passar valores entre duas *goroutines* e substituir variáveis compartilhadas para evitar condições de corrida, além de ser utilizado para impor a ordem entre duas operações em *goroutines* diferentes. Os *channels* foram usados não apenas para corrigir *bugs* relacionados a passagem de mensagens, mas também *bugs* causados por sincronização tradicional de memória compartilhada.

A Tabela 11 apresenta as estratégias de correção dos *bugs não-bloqueantes*. Dezenove *bugs não-bloqueantes* foram corrigidos adicionando operações relacionadas a *Mutex*, como *locks()* e *unlocks()*, para garantir a exclusão mútua de variáveis compartilhadas. Quatro *bugs não-bloqueantes* foram corrigidos movendo *unlocks()* para locais corretos. Sete foram corrigidos ou copiando uma variável compartilhada ou movendo a variável compartilhada para o local correto dentro da estrutura, atentando para que esses *bugs* são todos relacionados ao compartilhamento de memória e implementações já tradicionais.

Tabela 11. Estratégias de correção: *Bugs não-bloqueantes*.

	Adiciona	Move	Altera	Remove
Memória Compartilhada				
Tradicional	19	4	7	1
Wait	2	0	0	1
Lib	1	0	0	0
Passagem de Mensagem				
Channel	3	1	1	0
Lib	1	0	0	0
Total	26	5	8	2

Assim como nos *bugs* bloqueantes, algumas correções em comportamentos *não-bloqueantes* utilizam *channels* para correção. A suspeita é que isso ocorra ou porque os programadores que utilizam a linguagem Go, após algum tempo de uso, tem visto a passagem de mensagens como uma maneira mais confiável ou mais fácil de realizar comunicação entre *goroutines* ou então porque a abstração na utilização de *channels* torna a tentativa de correção menos complexa.

A Tabela 12 apresenta como os *bugs* foram detectados. A maioria dos *bugs* fo-

ram reportados de forma manual, ou seja, capturados a partir de análise e/ou situações extraordinárias onde os *bugs* ocorreram e foram reportados para discussão.

Tabela 12. Formas de Deteção de *Bugs*. Formas de detecção de *bugs*

Forma de Detecção	***Bloqueantes***		***Não-bloqueantes***	
	Memória Compartilhada	**Passagem de Mensagem**	**Memória Compartilhada**	**Passagem de Mensagem**
Manual	28	16	24	5
Detector Go	1	1	4	0
Teste Unitário	0	3	7	1
Total	29	20	35	6

Portanto 73 dos 90 *bugs* foram reconhecidos de forma manual, sem a utilização de ferramentas para auxílio. Enquanto isso, 6 condições de corrida foram detectadas pelo próprio detector do Go, e 11 *bugs* relatados após terem falhado ao passar em testes. Esses números demonstram que os detectores próprios do compilador e ambiente de execução Go tem suas dificuldades e limitações para capturar *bugs* em aplicações, que envolvem muita complexidade quanto a regra de negócios e implementação. Outro ponto é que apesar dos testes unitários aparentemente capturarem bem a situação dos *bugs*, melhor inclusive que o detector da linguagem, ainda se viu poucos *bugs* capturados por eles, se comparados com os relatados por análise manual, sendo um indício de que não é trivial escrever testes unitários para aplicações concorrentes, pois envolvem cenários não determinísticos difíceis de serem modelados nos testes.

5. Conclusões e Trabalhos Futuros

O trabalho aborda os problemas de concorrência em programas reais, investigando suas características, padrões de ocorrência, métodos de correção e outras propriedades relevantes na linguagem Go. A pesquisa se baseia na análise de 90 *bugs* de concorrência encontrados em três projetos de código aberto: `Docker`, `Terraform` e `CockroachDB`. Os resultados obtidos fornecem implicações e observações sobre a natureza e a distribuição desses *bugs*, com possíveis implicações para o desenvolvimento de ferramentas de detecção de falhas.

A linguagem Go promove a facilidade e leveza na criação de threads, além de favorecer a passagem de mensagens em vez de memória compartilhada para a comunicação entre elas. Nosso estudo revela que, quando não utilizadas corretamente, essas práticas de programação podem potencialmente resultar em *bugs* de concorrência. Dos 90 *bugs* identificados, 49 são bloqueantes, enquanto 41 são *não-bloqueantes*. Entre os *bugs* bloqueantes, 29 estão relacionados à memória compartilhada e 20 à passagem de mensagem, enquanto 36 dos 41 *bugs não-bloqueantes* estão ligados à memória compartilhada e 5 à passagem de mensagem.

Ao comparar os resultados obtidos por Tu et al. (2019) com os obtidos neste estudo, os *não-bloqueantes* tendem a diminuir conforme a aplicação evolui. Podemos notar que, em 2019, para o `Docker` e o `CockroachDB`, havia predominância de *bugs não-bloqueantes*, o que mudou com a amostragem retirada entre 2020 e 2024. Já para a

aplicação `Terraform`, que teve *bugs* catalogados desde o início da aplicação, a amostragem demonstrou uma predominância de *bugs não-bloqueantes* implicando que podemos associar a maturidade da aplicação aos tipos de *bugs* que costumam ser causados, detectados e corrigidos.

Como trabalhos futuros, podemos citar a avaliação de outras aplicações *open-source* escritas em linguagem Go, ampliando e avaliando se os dados obtidos nesse estudo se mantém em outras aplicões. Os dados levantados apontam que os detectores de *bugs* podem se beneficiar principalmente dos aspectos de causas e correções, podendo, de certa forma, inferir quais são os principais problemas e soluções no desenvolvimento de aplicações concorrentes. No caso da memória compartilhada, poderia ser analisado se o acesso está protegido, visto que grande parte das correções envolve a adição de algum método de sincronização, ou mesmo a utilização de canais para comunicação e sincronização.

Referências

Cockroach (2023). CockroachDB - distributed SQL for your applications. https://www.cockroachlabs.com/.

Cox-Buday, K. (2017). *Concurrency in Go: Tools and Techniques for Developers*. "O'Reilly Media, Inc.".

Docker (2023). Docker - build, ship, and run any app, anywhere. https://www.docker.com/.

GO (2023). Effective Go. https://go.dev/doc/.

Hennessy, J. L. and Patterson, D. A. (2011). *Computer architecture: a quantitative approach*. Elsevier.

Hoare, C. A. R. (1978). Communicating sequential processes. *Commun. ACM*, 21(8):666–677.

Lu, S., Park, S., Seo, E., and Zhou, Y. (2008). Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, pages 329–339.

Pike, R. e. a. (2023). Frequently asked questions (FAQ): Why goroutines instead of threads? https://go.dev/doc/faq#goroutines.

Stoica, I., Stanculescu, M., Marinescu, D. C., and Popescu, T. (2014). The evolution of multicore processors: A survey of hardware trends and software challenges. *ACM Computing Surveys (CSUR)*, 47:1–38.

Terraform (2023). Terraform - infrastructure as code. https://www.terraform.io/.

Tu, T., Liu, X., Song, L., and Zhang, Y. (2019). Understanding real-world concurrency bugs in go. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 865–878.

Yin, Z., Yuan, D., Zhou, Y., Pasupathy, S., and Bairavasundaram, L. (2011). How do fixes become bugs? In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 26–36.