UNIVERSIDADE FEDERAL DE CAMPINA GRANDE
CENTRO DE ENGENHARIA ELÉTRICA E INFORMÁTICA
UNIDADE ACADÊMICA DE SISTEMAS E COMPUTAÇÃO
CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

LUCAS HENRIQUE DE LIMA E SILVA

IMPLEMENTAÇÃO DO GARBAGE COLLECTOR CONTROL INTERCEPTOR PARA .NET CORE COMMON LANGUAGE RUNTIME

CAMPINA GRANDE - PB

2020

LUCAS HENRIQUE DE LIMA E SILVA

IMPLEMENTAÇÃO DO GARBAGE COLLECTOR CONTROL INTERCEPTOR PARA .NET CORE COMMON LANGUAGE RUNTIME

Trabalho de Conclusão Curso apresentado ao Curso Bacharelado em Ciência da Computação do Centro de Engenharia Elétrica e Informática da Universidade Federal de Campina Grande, como requisito parcial para obtenção do título de Bacharel em Ciência da Computação.

Orientador: Professor Dr. Thiago Emmanuel Pereira da Cunha Silva.

CAMPINA GRANDE - PB

2020

S586i Silva, Lucas Henrique de Lima.
Implementação do garbage collector control interface para .NET core common language runtime. / Lucas Henrique de Lima Silva. – 2020.

9 f.

Orientador: Prof. Dr. Thiago Emmanuel Pereira da Cunha Silva.
Trabalho de Conclusão de Curso - Artigo (Curso de Bacharelado em Ciência da Computação) - Universidade Federal de Campina Grande; Centro de Engenharia Elétrica e Informática.

1. Computação em nuvem. 2. Coletor de lixo. 3. Dados – coleta e análise. 4. Tempo de execução de linguagem comum .NET. 5. Linguagem de programação e tempo de execução. 6. Garbage collector control interceptor. 7. ASP.NET. 8/ Processador de requisição em C#. I. Silva, Thiago Emmanuel Pereira da Cunha. II. Título.

CDU:004(045)

Elaboração da Ficha Catalográfica:

Johnny Rodrigues Barbosa
Bibliotecário-Documentalista
CRB-15/626

LUCAS HENRIQUE DE LIMA E SILVA

IMPLEMENTAÇÃO DO GARBAGE COLLECTOR CONTROL INTERCEPTOR PARA .NET CORE COMMON LANGUAGE RUNTIME

Trabalho de Conclusão Curso apresentado ao Curso Bacharelado em Ciência da Computação do Centro de Engenharia Elétrica e Informática da Universidade Federal de Campina Grande, como requisito parcial para obtenção do título de Bacharel em Ciência da Computação.

BANCA EXAMINADORA:

Professor Dr. Thiago Emmanuel Pereira da Cunha Silva
Orientador – UASC/CEEI/UFCG

Professor Dr. Francisco Vilar Brasileiro
Examinador – UASC/CEEI/UFCG

Professor Dr. Tiago Lima Massoni
Disciplina TCC – UASC/CEEI/UFCG

Trabalho aprovado em: 2020.

CAMPINA GRANDE - PB

Implementação do Garbage Collector Control Interceptor para .NET Core Common Language Runtime

Lucas H. de Lima e Silva (Aluno), Thiago Emmanuel Pereira (Orientador)
Universidade Federal de Campina Grande - UFCG
Departamento de Sistemas e Computação
lucas.silva@ccc.ufcg.edu.br, temmanuel@computacao.ufcg.edu.br

ABSTRACT

HTTP services that run in environments supported by runtime environments, such as Java, Go and Ruby, are very popular. These environments offer security, portability, ease of integration and automatic memory management. However, the garbage collection mechanism of these runtimes can considerably increase the service time, as it takes control of the processor. To mitigate this problem, a mechanism called Garbage Collector Control Interceptor (GCI) was created. It is an interceptor that monitors and controls the garbage collector, in order to avoid serving requests at the moment the mechanism executes, thus reducing the impact on service time. Published studies show that GCI works and significantly reduces service time in cases where it would normally be higher than normal [1]. However, there is still no application of the technique for Common Language Runtime, the runtime environment of the .NET Core framework. This work aimed to implement the GCI for .NET according to its specification and to evaluate its functioning experimentally. The results showed that the GCI works in ASP.NET applications, reducing the impact of garbage collection and service time on 4-node services, without penalizing throughput.

Keywords

Cloud computing, garbage collector, .NET common language runtime.

RESUMO

Serviços HTTP que executam em ambientes apoiados por ambientes de execução *(runtimes)*, como em Java, Go e Ruby, são muito populares. Estes ambientes oferecem segurança, portabilidade, facilidade de integração e gerenciamento automático de memória. No entanto, o mecanismo de coleta de lixo destas runtimes pode aumentar consideravelmente o tempo de serviço, uma vez que toma o controle do processador. Para atenuar esse problema, criou-se o interceptador de requisições chamado Garbage Collector Control Interceptor (GCI). Trata-se de um interceptador que monitora e controla o coletor de lixo, de maneira a evitar atender requisições no momento em que o mecanismo executa, diminuindo, assim, o impacto no tempo de serviço. Estudos anteriores mostram que o GCI funciona e reduz consideravelmente o tempo de serviço em casos que ele normalmente seria mais alto que o normal [1]. No entanto, ainda não há uma aplicação da técnica para Common Language Runtime, o ambiente de tempo de execução do framework .NET Core. O objetivo deste trabalho foi implementar o GCI para .NET de acordo com sua especificação e avaliar seu funcionamento de maneira experimental. Os resultados mostraram que o GCI funciona em aplicações ASP.NET, diminuindo o impacto da coleta de lixo e o tempo de serviço em serviços de 4 nós, sem penalizar a taxa de requisições atendidas.

Palavras-chave

Computação em nuvem, coletor de lixo, tempo de execução de linguagem comum .NET.

1. INTRODUÇÃO

A maioria das linguagens de programação fornecem um ambiente de tempo de execução, onde os programas rodam. Esses ambientes podem tratar uma série de questões, incluindo o gerenciamento da memória, como o programa acessa variáveis, mecanismos para passar parâmetros entre procedimentos, interface com o sistema operacional e outros. Normalmente, esse conjunto de características reduz a preocupação dos programadores com aspectos de mais baixo nível, como configurar e gerenciar a pilha de execução e a *heap*, e pode incluir recursos como coleta de lixo, *threads* ou outros recursos dinâmicos integrados à linguagem [8].

É comum que essa camada adicional não traga preocupações em estágios iniciais de uma aplicação. No entanto, tais benefícios podem trazer algumas penalidades. Vários estudos mostram que o coletor de lixo pode introduzir sobrecargas de desempenho quando comparado com o gerenciamento de memória manual [9, 10], afetando negativamente o tempo de serviço em aplicações, principalmente aquelas que lidam com estado. Isso acontece porque quando o coletor de lixo executa, ele toma o controle do processador. As coletas acontecem, na maioria das vezes, de forma não determinística. Os mecanismos de coleta de lixo atuam, tipicamente, em dois modos, intercambiáveis durante a execução de uma mesma aplicação. No primeiro modo, o procedimento de coleta executa de maneira concorrente à aplicação. No segundo modo, a aplicação é pausada enquanto o mecanismo de coleta executa. Muitos fatores influenciam o uso de um ou outro modo, sendo o principal deles a quantidade de memória alocada pela aplicação. Em ambos os modos, pode haver impacto no desempenho da aplicação, embora o segundo apresente maior impacto. A solução típica para aliviar estes problemas envolve modificar as aplicações para utilizar a memória de maneira mais adequada (por exemplo, evitando a execução do coletor que exige pausas na aplicação). Esta solução exige maior conhecimento em sobre os mecanismos de gerência da memória da linguagem escolhida, o que não é trivial. Portanto, ela aumenta o tempo de desenvolvimento das aplicações e contradiz as motivações para a escolha do uso de runtimes.

Como tentativa de melhorar esse cenário, foi desenvolvido o Garbage Collector Control Interceptor (GCI). Este mecanismo consiste em um interceptador de requisições que gerencia melhor a coleta de lixo na runtime em serviços HTTP. O GCI foi

implementado nas linguagens Java, Go e Ruby. Ele funciona através da interceptação de requisições, do monitoramento e do controle da runtime, para evitar que requisições sejam atendidas em uma instância do serviço, no momento que seu coletor de lixo esteja executando. Em outras palavras, ao invés de tentar minimizar o impacto de eventuais intervenções, o GCI transforma essas intervenções em falhas temporárias [1]. Nesse caso, é feita uma negação de serviço, e a requisição é encaminhada para outras réplicas pelo balanceador de carga. Como existem poucas diferenças significativas entre o funcionamento do coletor de lixo na Máquina Virtual Java (JVM) e em .NET CLR [7], e dados os ótimos resultados obtidos nos experimentos realizados com o GCI em Java [1, 2], pretende-se expandir o seu suporte para outras runtimes, incluindo .NET Common Language Runtime. O principal objetivo deste trabalho é, portanto, conduzir um estudo sobre a técnica do GCI e desenvolver uma aplicação da técnica para Common Language Runtime, o ambiente de tempo de execução do framework .NET Core de acordo com sua especificação original e avaliar seu funcionamento de maneira experimental.

2. FUNDAMENTAÇÃO TEÓRICA

O GCI em sua versão inicial foi avaliado em um serviço de única instância [2]. Os resultados preliminares demonstraram a eficácia da técnica e motivaram a expansão para diferentes tipos de serviços, inclusive com múltiplas réplicas e gerenciadores de carga. Os novos requisitos trouxeram mudanças significativas no design e na implementação do GCI [1], cuja versão foi usada como base neste estudo. O GCI possui duas camadas principais: i) um interceptador de requisições, responsável por expor uma interface de controle para a coleta de lixo no ambiente de tempo de execução e ii) um *proxy*, responsável por decidir se requisições podem ser atendidas por uma instância do serviço, monitorar o uso da *heap* e disparar coletas de lixo no ambiente de tempo de execução.

2.1 .NET Core CLR e o Coletor de Lixo

.NET Core possui uma máquina virtual chamada Common Language Runtime, ou Ambiente de Tempo de Execução Comum. Ela suporta programas escritos em diversas linguagens, entre elas C#, usada para a implementação dos mecanismos descritos nas seções subsequentes. Em .NET Core CLR, a coleta de lixo ocorre quando uma das seguintes condições é verdadeira: i) o sistema está com pouca memória física, ii) A memória usada por objetos alocados no heap gerenciado ultrapassa um limite aceitável ou iii) o método *GC.Collect()* é chamado. Antes de uma coleta de lixo ser iniciada, todas as *threads* gerenciadas são suspensas, exceto aquela que acionou a coleta de lixo [10]. A Figura 1 ilustra esse comportamento.

Figura 1. Em .NET Core CLR, uma thread que aciona uma coleta de lixo faz com que as outras threads sejam suspensas [8], o que causa picos de latência em requisições atendidas no momento em que o coletor de lixo está executando.

Além do método *GC.Collect()*, existe outro método importante para o funcionamento do GCI, o *GC.GetTotalMemory()*. Ele será usado para fornecer o uso atual da *heap* para o *GCI Proxy*, cujo valor é dado em *bytes* e é usado para determinar o melhor momento de fazer uma coleta de lixo, como descrito na seção 2.1.1.

2.2 Garbage Collector Control Interceptor (GCI)

Como a execução do coletor de lixo acontece de forma não determinística, Fireman et al. propuseram e implementaram uma técnica fácil de usar chamada Garbage Collector Control Interceptor (GCI) [1]. Entre as vantagens de se usar o GCI ao invés de tentar manipular o coletor de lixo ou mudar o código está o agnosticismo em termos de lógica de negócio e as adaptações transparentes às mudanças de carga. A especificação do GCI é independente de ambiente de execução e protocolo de comunicação. Sendo assim, é um mecanismo fácil de usar e não requer configuração específica ou entendimento dos aspectos internos do ambiente de execução.

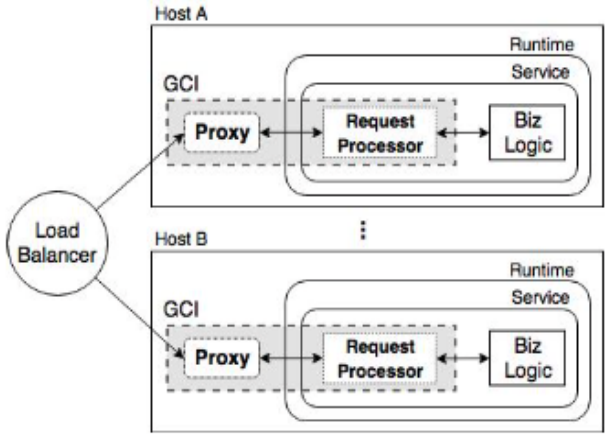

Figura 2. Arquitetura do GCI [1].

A Figura 2 ilustra a arquitetura do GCI [1]. Ela consiste em um servidor de uma aplicação que possui vários nós (*hosts*) atrás de um balanceador de carga. Em cada *host*, existe uma instância do *GCI Proxy*, que é agnóstico em termos de ambiente de execução e responsável por controlar o coletor de lixo, além de um processador de requisições acoplado ao serviço e, no caso de .NET, em forma de *Middleware* [4], responsável por executar dois comandos específicos do ambiente de execução, i.e, checar o estado atual da heap e executar uma coleta de lixo.

2.1.1 GCI Proxy

A Figura 3 representa o fluxo de uma requisição feita a uma instância que roda o GCI. Se o serviço estiver indisponível, o que significa que o coletor de lixo já está rodando, o *GCI Proxy* deixa de atender a requisição, gerando uma indisponibilidade de nó temporária (p. ex., serviço http indisponível). Isso permite que o balanceador de carga encaminhe a requisição para outra instância do serviço. Se o serviço estiver disponível e for hora de checar o uso da heap, o *GCI Proxy* delega essa ação para o processador de requisições. Após a checagem da heap, se ainda não for a hora de coletar o lixo, a requisição é atendida. Caso contrário, o *GCI Proxy* marca o nó como indisponível, deixa de atender a requisição, atende as requisições recebidas até então e delega uma

coleta de lixo para o processador de requisições. As requisições recebidas durante uma indisponibilidade de nó serão negadas e também passarão para outra instância do serviço. Quando o processo de coleta finalizar, o nó ficará disponível novamente.

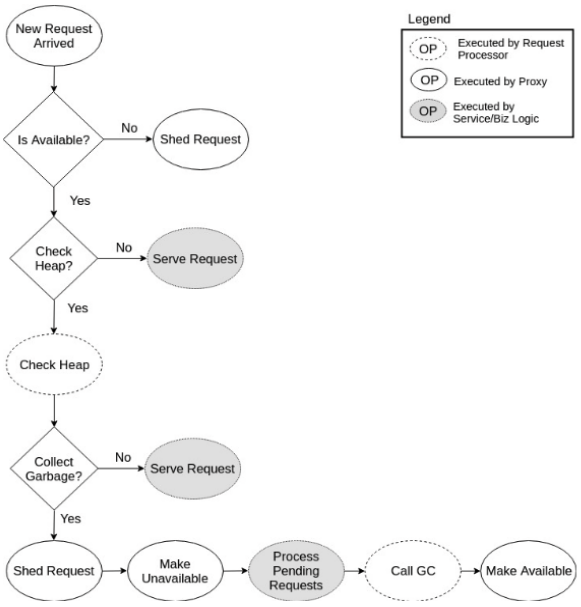

Figura 3. Fluxo de requisições no *GCI Proxy* [1].

Fireman et al. descrevem dois pontos como sendo não triviais na elaboração do GCI [1]. O primeiro deles é decidir quando checar o uso da heap. O GCI checa o uso da heap a cada **N** requisições. O valor de N é recalculado a medida que novas requisições são atendidas, começando por um valor conservador (p. ex., 10) e variando pouco a pouco durante o tempo de vida do serviço. O segundo é decidir quando fazer uma coleta de lixo. O GCI estabelece um limiar **T** e, quando **N** atinge **T**, o GCI se prepara para coletar lixo. Assim como **N**, **T** também começa com um valor conservador (p. ex., 30% da heap) e aumenta após cada coleta com um teto, para evitar indisponibilidade de todos os nós ao mesmo tempo. Os valores de **T** variam de acordo com pequenos valores aleatórios. Mais detalhes sobre o algoritmo do *GCI Proxy* estão disponíveis no trabalho original de Fireman et al. e seu código está disponível em https://github.com/gcinterceptor/gci-proxy.

2.1.2 GCI Request Processor

O Processador de Requisições do GCI é uma camada acoplada a cada instância do serviço que executa dois comandos específicos do ambiente de execução: i) checar o uso da heap e ii) realizar uma coleta de lixo.

2.1.2.1 Especificação original

A especificação original do GCI consiste em um interceptador que acessa um cabeçalho "GCI" em uma chamada HTTP para verificar se ela consiste em uma ação solicitada pelo *GCI Proxy,* como descrito na seção 2.1.1. As ações podem ser i) coletar o lixo através do método *GC.Collect()* ou ii) verificar o uso da *heap* através do método *GC.GetTotalMemory()*. As informações necessárias do *GCI Proxy* são enviadas de volta através de cabeçalhos.

O pseudo-código a seguir detalha o algoritmo:

```
switch RequestHeaders.Get("GCI")
case "GC":
Runtime.ForceGC()
case "ALLOC":
Alloc ⇐ Runtime.GetHeap().GetUsage()
ResponseHeaders.Put("GCI−ALLOC", Alloc)
end switch
```

Quadro 1. Especificação do interceptador de requisições do GCI [1].

3. METODOLOGIA

A maior parte do GCI não depende do ambiente de execução. No entanto, como citado anteriormente, existe uma camada leve acoplada ao serviço, o processador de requisições. Sendo assim, este trabalho se deu em duas etapas: i) implementar o processador de requisições do GCI para .NET CLR e ii) realizar experimentos com o GCI para comparar o tempo de serviço dos cenários com e sem GCI. A partir dos experimentos, queremos analisar se o GCI consegue atenuar o impacto do coletor de lixo, descritos nas seções anteriores, sem penalizar a taxa de requisições atendidas. Mais detalhes da experimentação serão descritos na seção 3.3.

3.1 Implementação do Processador de Requisições em C# para ASP.NET

Este componente, a parte dependente de ambiente de execução do GCI, foi implementado em forma de *Middleware* do framework ASP.NET.

```
public class GCIMiddleware
{ ...
  public async Task Invoke(HttpContext context)
  {
    if (context.Request.Path.Equals("/__gci"))
    {
      string header = context.Request.Headers["gci"];

      switch (header)
      {
        case "ch":
          await context
            .Response
            .WriteAsync(GC.GetTotalMemory(false).ToString());
          break;
        case "gc":
          GC.Collect();
          break;
      }
      await context.Response.CompleteAsync();
    }
    else
    {
      await _next(context);
    }
  }
}
```

Quadro 2. Implementação em C# do algoritmo do processador de requisições do GCI em forma de Middleware para ASP.NET[1].

Em ASP.NET, *middleware* é um montado em um *pipeline* de aplicação para manipular solicitações e respostas [4], utilizado

para desenvolver aplicações *web* em .NET O código foi escrito em C#. O Middleware cria um endpoint e se baseia na presença de um cabeçalho de requisição HTTP que contém o código da ação a ser executada. O resultado das operações são enviados em uma resposta HTTP. Em outras palavras, quando o GCI quer checar o uso da *heap*, ele faz uma requisição para o *endpoint* "/__gci" com o cabeçalho "gci" de valor "ch". A resposta dessa requisição contém o valor total em *bytes* de memória que a *heap* está utilizando. Quando o GCI quer realizar uma coleta de lixo, ele faz uma requisição para o mesmo endpoint anterior com o cabeçalho "gci" com valor "gc". Em ambas as requisições, o status HTTP 200 indica que a ação foi realizada com sucesso. O código foi exportado para um arquivo de extensão .dll, que pode ser importado e utilizado em todas as linguagens compatíveis com .NET Core e ASP.NET. Algumas pequenas modificações foram feitas (p. ex., nomes de cabeçalhos) para se adequar ao protocolo do *GCI Proxy* em sua última versão disponível.

3.2 Implementação do Gerador de Carga

Para testar o GCI em uma simulação do mundo real, foi necessário criar um gerador de carga. Na prática, este componente serve como um *mock* de um serviço HTTP, que tem o processador de requisições do GCI acoplado em forma de *Middleware*, a fim de se tornar compatível com o *GCI Proxy*. A implementação deste componente também foi em C#. Ele consiste em uma aplicação simples que utiliza o framework ASP.NET e expõe um *endpoint* que, quando recebe uma requisição, gera lixo na aplicação. Numa aplicação ASP.NET, existe um *pipeline* de requisições, composto por *middlewares*. O Quadro 3 mostra como a *GCIMiddleware*, classe que contém a implementação do interceptador de requisições para .NET Core CLR, pode ser adicionada a um projeto ASP.NET.

```
namespace GarbageGenerator
{
   public class Startup
   {
      ...
      public void Configure(
         IApplicationBuilder app,
         IWebHostEnvironment env)
      {
         ...
         app.UseMiddleware<GCIMiddleware>();
      }
   }
}
```

Quadro 3. Código de configuração da aplicação geradora de carga. A classe *GCIMiddleware* é usada como *middleware* no *pipeline* de requisições.

Para a geração de carga optou-se por utilizar a mesma abordagem utilizada nos experimentos em Java, que consiste em um *endpoint* que aloca uma mensagem de tamanho fixo e calcula 5000 números primos [1]. Essa decisão se deu pelo fato de ser mais fácil controlar as variáveis independentes em experimentos, que estão diretamente ligadas ao gerador de carga implementado, já que se sabe exatamente quanto de memória cada requisição consome. Isso é possível através de uma variável de ambiente da aplicação, e permite uma melhor manipulação desses valores para tentar reproduzir diferentes cenários do mundo real. O Quadro 4 mostra detalhes da implementação do *endpoint*.

```
namespace GarbageGenerator.Controllers
{
   [ApiController]
   [Route("garbage")]
   public class GarbageGeneratorController : ControllerBase
   {
      ...
      [HttpGet]
      public IActionResult Get()
      {
         byte[] byteArray = new byte[MSG_SIZE];

         for (int i = 0; i < MSG_SIZE; i++)
         {
            byteArray[i] = (byte)i;
         }

         if (WINDOW_SIZE > 0)
         {
            buffer[msgCount++ % WINDOW_SIZE] = byteArray;
         }

         long max = 5000;
         long count = 0;
         for (long i = 3; i <= max; i++)
         {
            bool isPrime = true;
            for (long j = 2; j <= i / 2 && isPrime; j++)
            {
               isPrime = i % j > 0;
            }
            if (isPrime)
            {
               count++;
            }
         }

         return Ok();
      }
   }
}
```

Quadro 4. Código do *controller* do gerador de carga, expondo um *endpoint* "/garbage", que gera lixo na aplicação e realiza o cálculo de números primos.

3.3 Experimentação

Para validar a implementação do processador de requisições GCI para ASP.NET e o funcionamento do GCI em ambientes .NET Core, foi realizado um experimento com um servidor HTTP que fornece um serviço, cujo funcionamento e implementação foram descritos na seção anterior. O principal objetivo deste experimento foi responder a seguinte pergunta: **A técnica do GCI diminui o tempo de serviço sem penalizar seu rendimento em aplicações .NET?** As variáveis dependentes consideradas no experimento foram i) o tempo de serviço e ii) o número de requisições atendidas com sucesso. As variáveis independentes consideradas foram i) a escala do serviço (i.e., número de nós), ii) o status do GCI (ligado/desligado), iii) a carga de requisições, iv) o tamanho da heap e v) o tamanho da mensagem. O experimento inicial considerou um serviço de 4 nós com 2GB de RAM e 2 vCPUs cada. O *GCI Proxy* foi configurado com uma heap de 256MB.

[1] Disponível em https://github.com/lucashsilva/garbage-generator

Para cada status do GCI (ligado/desligado), foi gerada uma carga de 40 requisições por segundo durante 10 minutos, totalizando 24000 requisições. O tamanho de mensagem considerado para cada requisição foi de 64KB. Todas as requisições foram registradas em um *log*, bem como informações relevantes para análise do funcionamento do experimento (p. ex.,. *logs* de saída e erro da aplicação e uso de CPU). As 5000 primeiras requisições de cada rodada foram descartadas, pois foram consideradas como fase de *warm up* do sistema. Cada rodada foi executada 15 vezes para cada status do GCI.

3.4 Validação dos resultados preliminares

Em busca de uma validação estatística para nortear os estudos dos resultados obtidos de forma preliminar, foi realizado o teste de t-Student para os cenários com e sem GCI. Aos níveis de significância 0,1, 0,01 e 0,001, todos os testes foram favoráveis para a rejeição da hipótese nula. Sendo assim, havia grandes indícios de que o tempo de serviço com o GCI seria inferior àquele sem a implementação ou, em outras palavras, de que o GCI reduziu o tempo de serviço nos testes preliminares em .NET Core CLR. A partir disso, procuramos melhorar alguns aspectos de implementação do processador de requisições e do gerador de carga, abordados com mais detalhes na seção 5.

4. RESULTADOS

As figuras 4 e 5 mostram que, embora a média do tempo de serviço esteja muito próxima para ambos os cenários, há menos variação no cenário com o GCI. O cenário com GCI tem ainda menos *outliers* que o cenário sem GCI, indicando uma melhor previsibilidade no tempo de serviço e menos requisições com picos no tempo de serviço. Isso confirma que o *proxy* está conseguindo evitar que requisições sejam atendidas quando há iminência de coleta de lixo. Além disso, fica evidente que a eliminação de checagens não determinísticas internas de .NET Core CLR, relacionadas ao mecanismo de coleta de lixo, diminui o tempo de serviço. Estes resultados nos permitem responder a questão inicial e mostra que o GCI reduz a cauda da distribuição do tempo de serviço, principalmente nos percentis de interesse (99,9° em diante). Além disso, a Tabela 1 mostra uma diminuição bastante considerável, principalmente a partir do percentil 99,99°, aumentando a previsibilidade do tempo de serviço, ou seja, tornando a distribuição mais uniforme. De forma similar aos resultados dos experimentos realizados em Java [1], é possível observar que ativar o GCI para um serviço pequeno com estado melhora o tempo de serviço. Os benefícios são mais evidentes à medida que se move em torno do fim da distribuição, com uma melhora de 51,4% no percentil 99,999.

Percentil	Com GCI	Sem GCI
99°	50,0	50,0
99,9°	60,00	61,00
99,99°	72,00	92,00
99,999°	85,32	175,68

Tabela 1. Tempo de serviço (ms) nos cenários com e sem GCI e percentis de interesse.

Portanto, dada a melhora significativa na distribuição do tempo de serviço, principalmente em sua cauda, conclui-se que a técnica do GCI diminui o tempo de serviço sem penalizar seu rendimento em aplicações .NET Core.

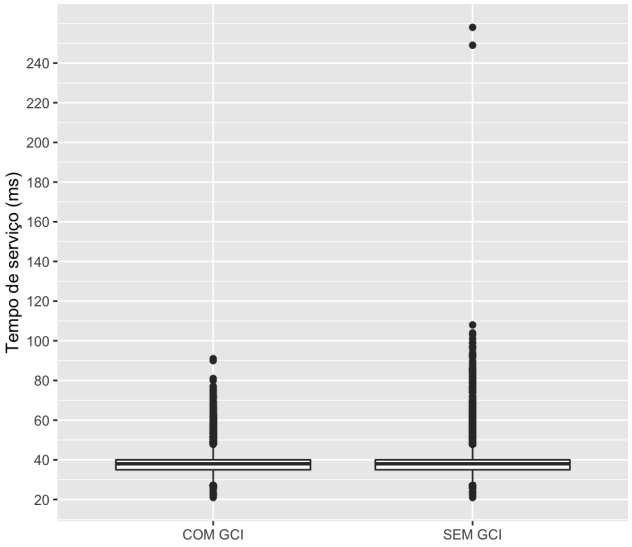

Fig 4. Boxplot do tempo de serviço nos cenários com e sem GCI.

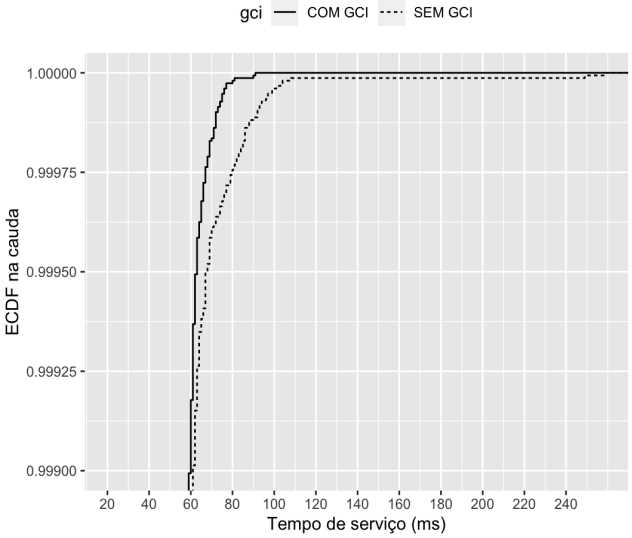

Fig 5. ECDF na cauda da distribuição do tempo de serviço nos cenários com e sem GCI.

5. LIMITAÇÕES E TRABALHOS FUTUROS

5.1 Controle da coleta de lixo

Um grande desafio ao lidar com o coletor de lixo em .NET Core CLR é tomar o total controle de quando as coletas serão realizadas. O *GCI Proxy* foi implementado de forma que pudesse detectar quando uma coleta de lixo acontecer de forma não programada ou inesperada, tendo sido iniciada pelo ambiente de execução. Nos estágios iniciais dos experimentos deste trabalho, deixamos que o ambiente de execução lidasse com as coletas de lixo, baseando-se nessa vantagem do *GCI Proxy*. Porém, a medida que aspectos relacionados ao uso do coletor de lixo foram sendo estudados com mais profundidade, surgiu a ideia de bloquear totalmente a coleta de lixo utilizando-se de um método oferecido pela própria runtime. Trata-se do método

GC.TryStartNoGCRegion(), presente na classe *GC*, que oferece uma API para manipulação do coletor de lixo em .NET Core. Esse método deve ser utilizado para desabilitar a coleta de lixo durante a execução de um caminho crítico [5]. Como o *GCI Proxy* faz o trabalho de checagem de heap e coleta de lixo, utilizou-se esse método para bloquear a execução do coletor de lixo por completo. O método não é chamado novamente até que esse limite tenha sido atingido e precisa ser chamado novamente caso isso aconteça. Por esse motivo, o gerador de carga precisa verificar constantemente se é necessário entrar na zona crítica (i.e., chamar o método novamente). Ele também permite estabelecer um limite via parâmetro para que uma coleta de lixo aconteça em uma situação não prevista pelo *GCI Proxy* (p. ex., falta de memória). O uso desse método funcionou perfeitamente bem no cenário do experimento realizado e teve resultados significamente melhores.

5.2 Cenário de experimentação

O trabalho atual limita-se ao cenário de um serviço com 4 instâncias e sem escalabilidade dinâmica. Considerando a grandeza das aplicações modernas, é interessante que o experimento seja feito em cenários mais complexos, assim como aconteceu nos experimentos de Java, com serviços com mais nós. Essa tarefa poderia sugerir pontos nos quais o GCI poderia melhorar.

6. AGRADECIMENTOS

Este artigo foi escrito como trabalho de conclusão do curso de Ciência da Computação na Universidade Federal de Campina Grande, sob a orientação de Thiago E. Pereira e com apoio do Laboratório de Sistemas Distribuídos, que ofereceu a infraestrutura em *cloud* necessária para a execução dos experimentos. Agradecimentos a Daniel Fireman et al. pela excelente base de conhecimento gerada para permitir a expansão da técnica do GCI para .NET, aos meus Professores que durante toda a minha jornada acadêmica sempre me ajudaram, aos meus colegas de curso e à minha família que sempre me apoiou nas minhas decisões e me motivou a focar nos meus estudos apesar de todas as dificuldades. Enfim, a todos os envolvidos direta e indiretamente, o meu eterno reconhecimento e agradecimento pela contribuição essencial para a realização deste trabalho.

7. REFERÊNCIAS

[1] FIREMAN, Daniel, BRUNET, João, LOPES, Raquel, QUARESMA, David, PEREIRA, Thiago E. Improving Tail Latency of Stateful Cloud Services via GC Control and Load Shedding. Campina Grande, PB, BR.

[2] FIREMAN, Daniel, BRUNET, João, LOPES, Raquel. Using Load Shedding to Fight Tail-Latency on Runtime-Based Services. Campina Grande, PB, BR.

[3] Richter, Jeffrey. CLR via C# Fourth Edition, Microsoft Press (2012).

[4] ANDERSON, Rick, SMITH, Steve. ASP.NET Core Middleware. 2020. (https://docs.microsoft.com/pt-br/aspnet/core/fundamentals/middleware/?view=aspnetcore-3.1)

[5] Preventing .NET Garbage Collections with the TryStartNoGCRegion API. 2016. (https://mattwarren.org/2016/08/16/Preventing-dotNET-Garbage-Collections-with-the-TryStartNoGCRegion-API/)

[6] PARKINSON, VASWANI, COSTA, DELIGIANNIS, BLANKSTEIN, MCDERMOTT, BALKIND, VYTINIOTIS. Project Snowflake: Non-blocking safe manual memory management in .NET. 2017.

[7] Singer. JVM versus CLR: a comparative study. University of Cambridge. 2003.

[8] Fundamentals of garbage collection. 2019. (https://docs.microsoft.com/pt-br/dotnet/standard/garbage-collection/fundamentals)

[9] M. Hertz and E. D. Berger. Quantifying the performance of garbage collection vs. explicit memory management. OOPSLA, 2005.

[10] B. G. Zorn. The measured cost of conservative garbage collection. Software – Practice and Experience. 1993.

Sobre o autor:

Lucas Henrique de Lima e Silva é graduando do curso de Ciência da Computação na Universidade Federal de Campina Grande. Atualmente trabalha como Desenvolvedor Full Stack na empresa IT4process GmbH, onde desenvolve aplicações móveis e *web*. Previamente, trabalhou em projeto de pesquisa e desenvolvimento com foco em software como serviço e *cloud* no Laboratório de Sistemas Distribuídos, em parceria com o Laboratório de Práticas de Software, ambos na Universidade Federal de Campina Grande, e com as empresas Lenovo e Red Hat. No seu tempo livre, contribui para projetos open source através do Github.

https://github.com/lucashsilva