Universidade Federal de Campina Grande

Centro de Engenharia Elétrica e Informática

Coordenação de Pós-Graduação em Ciência da Computação

Daniel Lacet de Faria Fireman

Improving Tail Latency of Interactive Cloud Microservices through Management of Background Tasks

Campina Grande-PB

2021

Universidade Federal de Campina Grande

Centro de Engenharia Elétrica e Informática

Coordenação de Pós-Graduação em Ciência da Computação

Improving Tail Latency of Interactive Cloud Microservices through Management of Background Tasks

Daniel Lacet de Faria Fireman

Tese submetida à Coordenação do Curso de Pós-Graduação em Ciência da Computação da Universidade Federal de Campina Grande - Campus I como parte dos requisitos necessários para obtenção do grau de Doutor em Ciência da Computação.

Área de Concentração: Ciência da Computação

Linha de Pesquisa: Redes de Computadores e Sistemas Distribuídos

Raquel Lopes e João Arthur Brunet

Campina Grande, Paraíba, Brasil

F523i Fireman, Daniel Lacet de Faria.

Improving tail latency of interactive cloud microservices through management of background tasks / Daniel Lacet de Faria Fireman. – Campina Grande, 2021.

109 f. : il. color.

Tese (Doutorado em Ciência da Computação) – Universidade Federal de Campina Grande, Centro de Engenharia Elétrica e Informática, 2021.

"Orientação: Profa. Dra. Raquel Lopes, Prof. Dr. João Arthur Brunet".

Referências.

1. Computação em Nuvem. 2. Redes de Computadores. 3. Sistemas Distribuídos. 4. Modelagem Estocástica. 5. Avaliação de Desempenho. I. Lopes, Raquel. II. Brunet, João Arthur. III. Título.

CDU 004.738.5(043)

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECÁRIA SEVERINA SUELI DA SILVA OLIVEIRA CRB-15/225

MINISTÉRIO DA EDUCAÇÃO
UNIVERSIDADE FEDERAL DE CAMPINA GRANDE
POS-GRADUACAO CIENCIAS DA COMPUTACAO
Rua Aprigio Veloso, 882, - Bairro Universitario, Campina Grande/PB, CEP 58429-900

FOLHA DE ASSINATURA PARA TESES E DISSERTAÇÕES

DANIEL LACET DE FARIA FIREMAN

IMPROVING TAIL LATENCY OF INTERACTIVE CLOUD MICROSERVICES THROUGH MANAGEMENT OF BACKGROUND TASKS

Tese apresentada ao Programa de Pós-Graduação em Ciência da Computação como pré-requisito para obtenção do título de Doutor em Ciência da Computação.

Aprovada em: 30/07/2021

Profa. Dra. RAQUEL VIGOLVINO LOPES, UFPB, Orientadora

Prof. Dr. JOÃO ARTHUR BRUNET MONTEIRO, UFCG, Orientador

Prof. Dr. FRANCISCO VILAR BRASILEIRO, UFCG, Examinador Interno

Prof. Dr. MARCUS WILLIAMS AQUINO DE CARVALHO, UFPB, Examinador Interno

Prof. Dr. NABOR DAS CHAGAS MENDONÇA, UNIFOR, Examinador Externo

Prof. Dr. LUIZ FERNANDO BITTENCOURT, UNICAMP, Examinador Externo

Documento assinado eletronicamente por **RAQUEL VIGOLVINO LOPES**, **Usuário Externo**, em

Referência: Processo nº 23096.044490/2021-72 SEI nº 1649731

Resumo

O aumento nos casos de uso com requisitos rígidos de latência, por exemplo, e-commerce e gerenciamento de dados em tempo real, tem pressionado para um maior controle sobre a variabilidade de desempenho. Isso ocorre porque mesmo picos infrequentes podem ser inaceitáveis, pois podem dominar o tempo que a leva resposta para chegar ao usuário final em implantações de nuvem multicamadas modernas. Neste contexto, a execução descoordenada de tarefas de manutenção, por exemplo, coleta de lixo e compactações de log, pode levar a uma degradação de desempenho inaceitável. Mesmo que pesquisadores e profissionais tenham trabalhado muito para melhorar o impacto não determinístico dessas tarefas no desempenho dos serviços em nuvem, este impacto ainda não é aceitável quando garantias de desempenho estritas são exigidas. Além disso, as soluções propostas são específicas a sistemas, aplicações, carga ou tarefas. O objetivo desta pesquisa é eliminar o impacto negativo de uma classe representativa de tarefas no desempenho de sistemas de nuvem. Para tanto, começamos propondo uma taxonomia para essas tarefas com base em seu gatilho e viabilidade de controle. Em seguida, definimos formalmente a ampla classe de tarefas interativas, compostas de atividades controláveis acionadas pelo processamento de requisições. Além disso, propomos o Controlador de Tarefas de Segundo Plano (BTC), uma solução totalmente distribuída para eliminar o impacto negativo de tarefas interativas em microsserviços em nuvem. O BTC atinge o objetivo evitando que o processamento de requisições enquanto executa tarefas. Realizamos simulações e experimentos de medição para avaliar a eficácia do BTC para lidar com o impacto do coletor automático de lixo, uma causa bem conhecida da degradação do desempenho dos serviços de nuvem. Os resultados mostram que o BTC elimina efetivamente o impacto do coletor de lixo em cargas de trabalho reais e sintéticas; e microsserviços de produção e fictícios. Além disso, a utilização do BTC leva a uma perda de capacidade proporcional à frequência e duração das tarefas, o que permite estimar o aumento de capacidade necessário para lidar com essa perda.

Palavras-chave: Computação em nuvem. Modelagem estocástica. Avaliação de desempenho.

Abstract

The increase in use cases with strict latency requirements, e.g., e-commerce and real-time data management, has been pushing towards greater control over performance variability. That is because even infrequent spikes might be unacceptable, as they could dominate the time it takes the response to reach the end-user in modern multi-layered cloud deployments. In this context, the uncoordinated execution of maintenance tasks, e.g., garbage collection and log compactions, might lead to unacceptable performance degradation. Even though researchers and practitioners have been working hard to improve the non-deterministic impact of those tasks on cloud services' performance, that still unacceptable when strict performance guarantees are required. Furthermore, the proposed solutions are system, application, load, or task-specific. This research goal is to eliminate the negative performance impact of a representative class of tasks. To do so, we start by proposing a taxonomy of those tasks based on their trigger and feasibility to control. Then we formally define the broad class of interactive background tasks composed of controllable activities triggered by request handling. Furthermore, we propose the Background Tasks Controller (BTC), a fully distributed approach to eliminate the negative impact of interactive tasks on cloud microservices. The BTC does so by transparently evicting request handling while executing background tasks. We performed simulated and measurement experiments to thoroughly evaluate BTC efficacy to deal with the automatic garbage collector's impact, a well-known cause of cloud services' performance degradation. The results show that BTC effectively eliminates the impact of the garbage collector in real and synthetic workloads; and production and dummy microservices. Furthermore, it leads to a loss in capacity proportional to the frequency and duration of the background tasks, which allows estimating the needed capacity increase to deal with this loss.

Keywords: Cloud Computing. Stochastic Modelling. Performance Evaluation.

Dedicatória

Dedico este trabalho a meu avós José Felinto (*in memorian*) e Maria Vilani, pelos muitos anos de cuidado, carinho e apoio.

Agradecimentos

Agradeço a minha mãe, Sandra Faria, e minha companheira Lillian Oliveira, por estarem na linha de frente dessa grande rede de apoio que tive durante todos esses anos. As brasileiras(os) que financiaram, mesmo sem saber, parte desta pesquisa, via afastamento remunerado concedido pelo IFAL. Aos meus orientadores Raquel Lopes e João Arthur Brunet por terem embarcado nessa jornada. Não posso deixar de mencionar a importância das bancas de qualificação e tese para minha formação como pesquisador. Muito obrigado, Francisco Brasileiro, Patrícia Endo, Luiz Bittencourt, Marcus Carvalho, Nabor Mendonça, Dalton Guerreiro.

Agradeço também a Dartagnan Fireman e Rita Oliveira, aos meus irmãos Camila e Vitor Faria e Elder Fireman, a Alan Alcântara, Jailson Carlos e Ilza Cavalcante por todo apoio e torcida. Também sou muito grato pelo apoio de as minhas tias(os), primas(os), em particular, a Alexandre pela ajuda com a matemática e apoio, Elis, Davyd e Alessandra pelos papos e paciência. Não posso deixar de agradecer a Rita Miranda e George Michael pelo carinho.

Dado que foram muitas pessoas especiais, infelizmente não conseguirei listar todos os nomes. Desde já, peço desculpas. Sou muito grato a todos os meus amigos e amigas pela paciência, por ouvir minhas frustrações, dificuldades e propostas, por discutir soluções e por torcer e vibrar comigo pelos pequenos progressos realizados. Muito obrigado David Quaresma, Felipe Alencar, Nazareno Andrade, Thiago "Manel" Pereira, "Ricardinho" Oliveira pelo apoio técnico e acadêmico. Em ordem alfabética, não posso deixar de mencionar: Arthur Lustosa, Aurélio Buarque, Auriane, Bruno Dias, Charles Gonçalves, Dayse Gama, Elthon Oliveira, Fred Carvalho, Gibran, Igor Natanael, Jefferson Neves, Jonatas Xavier, Júlio Vianna, Leandro Sales, Leonardo "Barroca", Lorena Martins, Luccas Augusto, Marco Rosner, Norah Gamarra, Rebeca Oliveira, Rodolfo Viana, Taciano Morais, Talita "Talishow", Thiago Macambira, Thomás Camargo e Viviane Tenório.

Por fim, este trabalho é resultado da ação de muitas pessoas que apoiam o ensino público, universal e gratuito de qualidade, pilar básicos para construção de um Brasil melhor. Assim, assumo o compromisso de continuar me esforçando para que mais pessoas tenham momentos mágicos em contato com educação, pesquisa, ensino e extensão.

Contents

Acronyms

ANOVA Analysis of Variance. xiii, 50, 108, 109

API Application Programming Interface. 64

AWS Amazon Web Services. 68

BT Background Task. xi, 11–13, 15, 18, 19, 22–24, 26, 31, 37, 39, 40, 49–61, 70, 84, 85, 88, 89, 91–94, 108, 109

BTC Background Tasks Controller. x, xii–xiv, 8–10, 21–24, 26, 28–33, 36–43, 46, 48, 49, 52, 54–57, 59–64, 67–83, 91–94, 108, 109

CDF Cumulative Distribution Function. 19, 20, 85

CLR Common Language Runtime. 2

COA Capacity-Oriented Availability. 33

COUA Capacity-Oriented Unavailability. xi, xiii, 33, 36, 38, 44, 45, 49, 50, 52–55, 92, 109

CTMC Continuous Time Markov Chain. 15–17

DSPN Discrete Stochastic Petri Nets. 85

EC2 Elastic Cloud Computing. 68

ECDF Empirical Cumulative Distribution Function. x, xii, 2, 31, 32, 44, 47, 69, 71

FaaS Function as a Service. 9

G1GC Garbage First Garbage Collector. 46, 79

GC Garbage Collector. x, xiii, 2–6, 9, 12–15, 19, 20, 26, 27, 49, 56, 63–65, 69, 70, 73–76, 78–82, 84, 87–90, 93, 94

GCI Garbage Collector control Interceptor. xii, 9, 63–68, 70, 75–77, 79, 81, 82

GSPN Generalized Stochastic Petri Nets. 16, 17

HTTP Hypertext Transfer Prototocol. 30, 64, 87

JRE Java Runtime Environment. 7–9, 80

JVM Java Virtual Machine. 2, 3, 46, 62, 64, 74–77, 79, 80, 82, 87, 88, 90, 92

JVMTI JVM Tool Interface. 77

KS Kolmogorov-Smirnov. xiii, 45, 47, 48

PN Petri Net. 16

RDMA Remote Direct Memory Access. 87

RT Response Time. 3, 4, 32, 41, 44

SA Sensitivity Assessment. 48, 49

SPN Stochastic Petri Nets. 15, 16, 84, 85

SRN Stochastic Reward Nets. x, xiii, 16–18, 20, 31, 38–42, 84, 85, 91, 92

VM Virtual Machine. 22, 63

YCSB Yahoo! Cloud Services Benchmark. 2, 80, 83

List of Figures

List of Tables

Listings

Chapter 1

Introduction

Cloud computing - or simply, cloud - is a term used to describe a category of computing services delivered on-demand, which were initially offered by commercial providers like Amazon, Google, and Microsoft. The term denotes a model in which other companies and users can access computational infrastructure through the Internet. This model's principle is to offer computing, storage, and software as a service (BUYYA et al., 2009). Despite the numerous definitions, there is consensus on some characteristics that a cloud must have: i) pay-per-use, ii) elastic capacity, iii) self-service, and iv) resource virtualization. In addition to these features, cloud providers offer a wide range of software services and include development tools that simplify the construction of scalable applications (BUYYA; BROBERG; GOSCINSKI, 2011).

The easy access to an elastic infrastructure brought by cloud computing has impacted software architecture. The microservices architectural style has been extensively used and consists of factoring applications in a set of services designed around specific business needs, which run in independent processes or containers launched into production independently through automated deployment processes (PAHL; JAMSHIDI, 2016). As a result of splitting service responsibilities, each unit executes potentially fewer and more cohesive functions, thus having a more predictable resource consumption and performance (LEWIS; FOWLER, 2014). Furthermore, a request to a distributed application could potentially go through many layers of replicated microservices. As an example of using this architectural style, in 2017, the Netflix video-on-demand service had its functionality distributed among hundreds of microservices running on tens of thousands of virtual machines and containers (NAIR, 2017).

Following the productivity and agility offered by combining the on-demand cloud infrastructure and microservices architectural style, managed programming languages, such as Java, Python, Ruby, and C#, became prevalent in cloud deployments (MEYEROVICH; RABKIN, 2013). The "managed" adjective comes from the fact that programs created using these languages require the execution management support of a runtime environment, for example, the Java's Runtime Environment and .NET's Common Language Runtime (CLR), which brings several benefits for software and operation engineers (GREGORY, 2013; BLACKBURN et al., 2008). For example, runtimes typically provide automatic memory management, which removes from developers the need to explicitly allocate and deallocate variables, avoiding entire classes of errors. Another benefit commonly offered is multiplatform execution, which allows the execution of the same code in different operating systems and architectures (DEGENBAEV; LIPPAUTZ; PAYER, 2019; CABALLERO et al., 2012; COWAN et al., 2000).

Due to these benefits, large companies run most of their applications in cloud deployments and write them as microservices using managed languages (NAIR, 2017; HUMBLE, 2011; VERLAGUET; MENGHRAJANI, 2014). Furthermore, commercial cloud providers have focused on native support for languages that need runtime support (KRISHNAN; GONZALEZ, 2015; LI, 2009). Despite the advantages of managed languages, the execution of runtime's internal routines could negatively impact microservices' performance (MAAS; ASANOVIC; KUBIATOWICZ, 2017; CAO et al., 2012a). Automatic memory management using garbage collectors (GCs) is one of those internal routines that affect the response time the most, and due to this reason, its impact on performance has been studied extensively (TENNAGE et al., 2019; MAAS et al., 2016; XIAN; SRISA-AN; JIANG, 2008).

To illustrate the impact of the GC on microservice's performance, we executed a measurement experiment using the widely-used Hazelcast in-memory cache (SALHI et al., 2017) and Yahoo! Cloud Services Benchmark (YCSB), an industry-standard load generator (COOPER et al., 2010). We detail the methodology and experiment results in Section 5.3. Figure 1.1 presents last percentile (i.e., tail) of the response time ECDF (VAART, 1998) for both cases, i.e., with and without garbage collections. A quick look at the $99.99^{th}-$percentile vertical lines shows a $\approx 3x$ performance degradation caused by the Java Virtual Machine (JVM)'s automatic garbage collection.

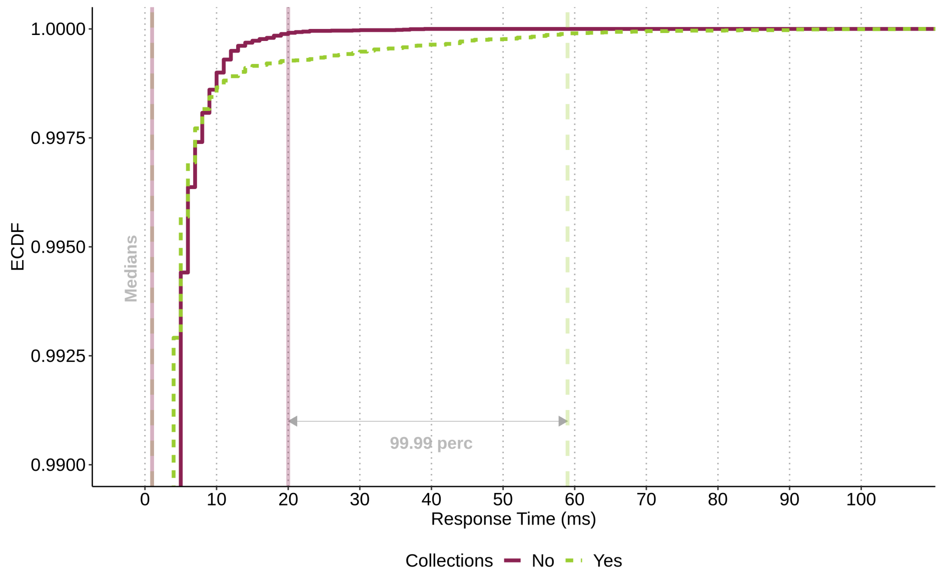


Figure 1.1: Last percentile (tail) of the response time ECDF of measurement experiments evaluating two replicated Hazelcast deployments: with and without the execution of automatic garbage collections.

Figure 1.1 shows that when there are no garbage collections, the 99.99^{th} percentile of the Response Time (RT) is approximately $20ms$, that is, 0.001% of the requests took more than $20ms$ to complete. When the JVM's automatic GC follows the default configuration, the 99.99^{th} percentile of the RT grows to $60ms$ ($\approx 3X$). The only change between these two runs is the activity of the GC. Once we have isolated this variable, we can assume that the increase in the tail of the RT is due to peaks caused by the runtime's memory cleanup happening during request handling. These non-deterministic spikes make it difficult to predict performance, affecting capacity planning and service level agreements.

GC is one of the many examples of background tasks. Log compactions in storage systems (BALMAU et al., 2019) and data reconstruction in distributed file systems (HAO et al., 2016) are both background tasks that could heavily impact the performance of cloud microservices. All those background tasks are management procedures needed for the system's correct and efficient operation and are executed concurrently on the same virtual machine or container of the microservice replicas. The non-deterministic impact of background tasks occurs mainly for two reasons: i) competition for resources happening during request handling and ii) application execution pauses, which can happen due to many reasons, for instance, garbage collection or blocking I/O.

The negative impact of background tasks is amplified in the cloud since user-facing interactive services are typically structured in layers of replicated microservices. Thus, a single request from the user might need to wait for all microservices to respond as part of a potentially large fan-out (DEAN; BARROSO, 2013). For example, a query to Facebook's real-time data management system results in calls to hundreds of services (ABRAHAM et al., 2013) and some stages of a query to Bing can reach thousands of servers in parallel (JALAPARTI et al., 2013).

In the context of interactive cloud services that need fluid responsiveness like e-commerce, real-time analytics, or web search, temporary spikes on the response time are unacceptable (DEAN; BARROSO, 2013). More strict performance deadlines are needed because RT spikes could dominate the time the response takes to reach the end-user (OUSTERHOUT et al., 2011; RUMBLE et al., 2011a). Dean and Barroso illustrate this problem by showing the effect of a large fan-out on the latency distribution of a replicated Google service structured in layers (DEAN; BARROSO, 2013). Their results show that waiting for the slowest 5% requests to finish is responsible for 50% of the 99%-percentile latency, thus focus on these slow outliers can significantly improve overall service performance.

Researchers and practitioners have been working hard to improve the non-deterministic impact of background tasks on cloud services' performance. For instance, the performance of GCs in managed languages, and the flush of transaction logs and merging segments in search engines have undergone significant advances over the years (YU et al., 2016; AKDOGAN, 2015; DETLEFS et al., 2004). However, since those tasks are optimized for a broad set of use cases, the negative impact is still not acceptable in some scenarios with strict performance requirements (MAAS et al., 2016; RUMBLE et al., 2011b).

It is also possible to argue that careful application optimizations and parameter adjustment can reduce the negative impact of background tasks. Nonetheless, this task is challenging and can become impracticable as it depends on the application configuration, code, and load, and those are changing very frequently on modern cloud deployments (JAYASENA et al., 2015; GHEORGHE; HINMAN; RUSSO, 2015). Some solutions like switching to manual memory management might even result in a productivity drop, requiring adaptations or code rewriting (DEGENBAEV; LIPPAUTZ; PAYER, 2019; CABALLERO et al., 2012; COWAN et al., 2000).

There are also approaches to mitigate the impact of non-deterministic response time spikes that are not specific to background tasks. An example is to issue a request to one replica and fall back on sending a subsequent request after some brief delay, canceling the remaining outstanding requests once the first result is received. Dean and Barroso call this technique hedged requests. They suggest deferring sending the subsequent request until the first request has been outstanding for more than the 95^{th}-percentile expected response time for that particular class of requests. Authors argue that this approach limits the additional load to approximately 5% while substantially shortening the latency tail (DEAN; BARROSO, 2013).

The threshold value determines the effectiveness of the hedged requests approach described above, and its choice heavily depends on the characteristics of the cloud service and the interference source. A high value may not be efficient due to infrequent or late hedging. Alternatively, a low value could lead to many duplicates, overloading the system (MISRA et al., 2019). We performed a simulated experiment to illustrate the hedge requests technique's performance when the impact of background tasks increases. For instance, that can represent new deployments triggering garbage collections more frequently due to a change in the memory consumption patterns. Section 4.7 details the methodology and results.

The experiment results show that the more requests are affected by background tasks (i.e., the more frequent a GC run), the less efficient the hedged requests technique is in mitigating this impact, delivering worse performance. A possible solution to this problem would be fine-tuning the threshold parameter, but that can be challenging, as the performance interference's origin is difficult to spot. For instance, it could be due to some implementation detail, which is dependent on the load. The difficulty increases with the adoption of multiple programming languages, continuous delivery, and global-scale deployments.

Thus, although suitable for some contexts, the solutions to deal with the impact of background tasks' interference are system, application, load, or task-specific. Some of those solutions can be impracticable, considering the typical complexity of current polyglot cloud deployments. Others might even result in a productivity drop, as to require adaptations or code rewriting. We believe that an approach to eliminate background tasks' impact on microservice performance is still relevant for cloud deployments with strict latency requirements.

1.1 Research Objectives

We investigate methods to eliminate background tasks' (or simply tasks) impact on cloud microservices, leading to better and more predictable performance for users. Tasks can be routines, procedures, or computer programs that could execute concurrently with the microservice replica's request handling — for instance, the GC in managed languages and the log compression in some distributed data stores.

This work aims at interactive online services with strict performance requirements and we choose this problem because background tasks are essential to a plethora of cloud applications, and due to that, there has been much work trying to mitigate their impact. That bulk of work focuses on specific tasks, specific applications, or specific load. There are also solutions trying to solve the problem generally, but they do not eliminate the impact. Furthermore, this is key due to the cascade effect caused by the massive fan-out structure found in cloud deployments nowadays.

In other to achieve that primary goal, we define the following objectives:

1. To formally define background tasks and model its impact on microservice's performance;

2. To propose a taxonomy of background tasks, grouping them by relevant properties;

3. To propose a practical approach that eliminates the impact of the most common or harmful background tasks on microservice performance.

1.2 Research Questions and Evaluation

We evaluate the impact of the proposed solution on cloud microservice's performance and reliability (performability), to answer the following research questions:

1. Does the solution affect microservice's performability?

2. Does the solution perform better than other practical approaches?

The evaluation focuses on microservices with dynamic scalability turned off, i.e., a static set of replicas. This scenario is prevalent in cloud services where reconfiguring the replica placement is costly—for instance, sharding-based data stores.

To be valuable and practical, we show our approach's applicability and draw our motivation from a representative use-case: the performance impact of Java Runtime Environment (JRE)'s automatic garbage collector in interactive cloud microservices. We adopt analytical formalism to describe the problem and the solution. We use both measurement and simulated experiments to perform a thorough evaluation. We use factorial design to drive our comparisons. We compare the proposed approach with the optimal case (no impact), with the baseline case (either default or production settings), and a practical alternative solution used at Google.

We execute measurement experiments with either a copy of the production environment or the latest stable version of the microservices. To facilitate reproducibility, we share our detailed setup configuration and use virtual machines from a cloud environment. That allows creating a clean, isolated environment containing only the minimum necessary to execute the experiment. We choose representative load and microservices for our evaluation by either using those that often appear in related research or mimicking production environments.

Finally, we use a simulated model to perform a sensitivity assessment of the solution parameters. To increase confidence in the simulated results, we verify the simulator model by comparing its results with the analytical model results. We also validate the simulated model by comparing its results with measurement experiments executed using a proof-of-concept implementation.

1.3 Thesis Contributions

The main contributions of this thesis are listed below:

- A taxonomy of background tasks based on their trigger and feasibility of being controlled. We also propose a formal description and a probabilistic model to describe background tasks' impact on the performance of interactive services. We show that many well-documented causes of response time spikes fit in this description. As an example, we formally describe and calculate the impact of the JRE garbage collector in the response time of a cloud microservice;

- The Background Tasks Controller (BTC), a fully distributed approach to eliminate the negative impact of controllable background tasks on interactive cloud microservices. The BTC does so by transparently evicting request handling while executing background tasks. We apply the approach to the JRE and evaluate its efficacy to deal with the automatic garbage collector's impact, a well-known cause of performance degradation;

- Analytical and simulated performability models of a replicated cloud microservice using BTC. We verify the simulated model by comparing its results to the results calculated with the analytical model. We validate the simulated model by comparing its results to the results obtained from measurement experiments;

- A thorough evaluation of the solution using simulated and measurement experiments. The results show that BTC effectively eliminates the impact of the garbage collector in representative and synthetic workloads, applied to production and dummy microservices. Furthermore, it leads to a loss in capacity proportional to the frequency and duration of the background tasks, which allows estimating the needed capacity increase to deal with this loss.

1.3.1 Publications

Results presented in this thesis have been published as papers in conference proceedings as follows:

- FIREMAN, D.; BRUNET, J.; LOPES, R.; QUARESMA, D.; PEREIRA, T. E. Improving tail latency of stateful cloud services via GC control and load shedding. In: Proceedings of the 10th IEEE International Conference on Cloud Computing Technology and Science (CloudCom), Nicosia, Cyprus, December 10-13, 2018. p. 121–128.

- FIREMAN, D.; LOPES, R.; BRUNET, J. Using load shedding to fight tail-latency on runtime-based services. In: Proceedings of the XXXV Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos. Porto Alegre, RS, Brasil: SBC, 2017.

1.3.2 Other Contributions

Other related contributions, which are not included in this thesis because other researchers drove them, are as follows:

- An evaluation of the impact of the automatic GC on the performance of Function as a Service (FaaS) applications. Unlike long-running cloud microservices, FaaS applications are typically composed of ephemeral stateless replicas and have a much different heap usage and GC activity patterns. The results show that GC also impacts FaaS functions leading to an increase of the latency tail;
- An approach to integrate BTC to a FaaS provider autoscaler. The evaluation results show that using GCI in FaaS effectively reduces the long tail latency caused by automatic collections and does not lead to an increase in resource usage;
- An evaluation of how the JRE setup contributes to the startup time and performance of FaaS applications;
- Prebaking, an approach to improve the performance of FaaS applications by reducing the function startup and warmup times. The proposed mechanism is based on the checkpoint-restore process technique and runs on out-of-the-box Linux. We evaluate the Prebaking technique using three functions running on the JRE: do-nothing, markdown renderer, and the image resizer. Our results show that it can improve the startup time up to 71% for the Image Resizer function compared to the state-of-the-practice mechanism to start processes.

These contributions appear in publications as follows:

- QUARESMA, D.; FIREMAN, D.; PEREIRA, T. E. Controlling Garbage Collection and Request Admission to Improve Performance of FaaS Applications. In: 2020 32nd International Symposium on Computer Architecture and High Performance Computing, Virtual, September 8-11, 2020.
- Silva, P.; FIREMAN, D.; PEREIRA, T. E. Prebaking Functions to Warm the Serverless Cold Start. In: 2020 21st ACM/IFIP International Middleware Conference, Virtual, December 7-11, 2020.

1.4 Thesis Organization

The remainder of this document is structured as follows. Chapter 2, provides the necessary background to understand the problem, a taxonomy of background tasks, and formally describes the large class of tasks that are the focus of this work: the ones triggered directly or indirectly by request processing. Furthermore, we propose a stochastic model to describe those tasks' impact on response time. Chapter 6 summarizes the work related to performance and availability modeling, and techniques to decrease the latency variability in cloud services.

In Chapter 3, we introduce the BTC, a distributed approach to eliminate the negative impact of background tasks on interactive cloud microservices' response time. Furthermore, this chapter also presents an analytical model of BTC's impact. Chapter 4 performs a thorough performability evaluation of the proposed solution and compares it to hedged requests, an approach used at Google (DEAN; BARROSO, 2013). Chapter 5 presents a proof of concept of the solution and an evaluation of the performance of BTC using measurement experiments with production and synthetic load. We summarize the work, conclude, and present our future work in Chapter 7.

Some passages and figures in this thesis have been quoted verbatim from the following sources FIREMAN; LOPES; BRUNET 2017 and FIREMAN et al., 2018.

Chapter 2

Background Tasks

This Chapter aims to present the target problem of this thesis: the negative impact of Background Tasks (BT) on interactive cloud microservices' response time.

As the concept of background tasks is broad (i.e., any routine executed concurrently with the microservice's request handling), we first present a taxonomy, defining and grouping tasks by two essential characteristics: the feasibility of control and its triggers. Then we formally describe the broad class of tasks that we focus on in this work: controllable tasks triggered directly or indirectly by the request handling.

We finish this chapter by proposing a stochastic model to describe those tasks' impact on interactive cloud microservices' response time and presenting an example of the model resolution.

2.1 Definition and Taxonomy

We define Background Task (BT) as any routine, procedure, or computer program executed concurrently with the microservice's request handling. Those tasks could have many sources, such as the microservice's business logic, the cloud provider, the runtime environment, or the operating system. We can cite automatic memory management in managed languages or log compression in some distributed data stores as examples of such tasks. Even though the execution of these tasks is essential to these systems, these routines can impact their performance in several ways, such as competition for resources and pauses in execution.

Regarding the control over their execution, tasks can be classified into two types:

- **Controllable**: those are tasks in which it is possible to control the start of its execution. For example, a backup routine of on-premise databases or the Java and Node.JS' automatic GC. In the latter case, even though the runtime does not allow the GC to be switched off, it can still be controlled through a combination of configuration and management APIs;

- **Uncontrollable**: these are tasks whose start time is not predictable, or there is no way to control it. As an example of those tasks, we can mention performance changes triggered by the virtual machine's energy-saving system. Eliminate the impact of uncontrollable tasks is outside the scope of this work.

Concerning what drives its execution, controllable tasks can be of two types:

- **Request oriented (interactive)**: tasks in which handling requests impact the moment of execution. In interactive systems, the system state changes when a request arrives and is processed. Thus, this handling might trigger a BT when a particular condition related to this state is satisfied. For example, runtime-based web servers could trigger the automatic garbage collection when the heap utilization reaches a certain threshold. Another example is the record compaction, typical of storage systems like Bigtable (CHANG et al., 2006). Interactive background tasks are the focus of this thesis;

- **Time-oriented (periodic)**: these are tasks in which the moment of execution depends solely on the time of the day. For example, a backup routine that takes place daily at a specific time. Controlling periodic tasks is outside the scope of this work.

Formally, an interactive BT is a tuple $T = (I, \zeta, A)$, where I is a start command, belonging to the abstract set of all possible commands and is triggered when it is the moment to execute T. We consider that all interactive tasks have a monitor associated, which provides measurements to drive its start. So, we define $\zeta : \mathbb{R} \to \mathbb{R}$, a function that represents a measurement of the monitor associated with T performed in an instant of time. For simplicity, let us define $\zeta = \zeta(now)$, which results in the monitor's measurement for the current time. When necessary, we will use the full notation - $\zeta(t)$ - to specify a time different from the current one. Finally, the tuple T also contains the activation interval $A = \{a \in \zeta(\mathbb{R}) \,|\, A^{min} \leq a \leq A^{max}\}$, which represents the subset of ζ measurements ranging from A^{min} to A^{max} that should lead to the start of T. The following paragraph shows an example of how the model can be used to define interactive tasks.

To exemplify the formal description of an interactive BT, let us consider T_{GC}, representing the automatic GC present in managed languages. Popular runtimes in the cloud, e.g., C#, Java, Node.js, Ruby, and PHP, have specific commands for forcing a memory cleanup to happen and retrieving the heap and GC statuses. That way, we can define I_{GC} as the command that triggers a garbage collection. Supposing that we are modeling a runtime configured to use a maximum heap 16GB and has a way to obtain the amount of memory used in bytes[1], we have $\zeta_{GC} : \mathbb{R} \rightarrow \{s \in \mathbb{R} : 0 \leq s < 16777216\}$. Finally, assuming the developer would like to keep the heap usage between 2 and 2.5 GB, we could define the activation interval $A_{GC} = \{a \in \mathbb{R} : 2097152 \leq a < 2621440\}$. With that, one could manage the execution of T_{GC} by executing I_{GC} every time $\zeta_{GC} \in A_{GC}$.

2.1.1 Garbage Collection and Other Maintenance Activities

The acceleration provided by the cloud triggered a push for productivity, which popularized the use of managed programming languages such as Java, Python, and Ruby in cloud deployments. The execution management support of a runtime environment brings several benefits for engineers, for example, dynamic typing, class resolution at runtime, reflection, and GC (BLACKBURN et al., 2008). The latter is essential since automatic memory management shields developers from explicitly managing pointers and reduces the risk of errors (CABALLERO et al., 2012; COWAN et al., 2000).

Due to these benefits, many widely-used distributed systems are written in managed languages (JOHNS, 2015; HUNT et al., 2010; SHVACHKO et al., 2010). Furthermore, companies like Netflix, Twitter, and Facebook write most of their applications in Java, Scala, and PHP (NAIR, 2017; HUMBLE, 2011; VERLAGUET; MENGHRAJANI, 2014). Simultaneously, commercial cloud computing providers like Google, Amazon, and Microsoft have been focusing on native support for languages that need runtime support like Go, Java, Python, and Node.js (KRISHNAN; GONZALEZ, 2015; LI, 2009).

Unfortunately, the advantages of managed languages come at a performance cost (CAO et al., 2012b; Anderson et al., 2011). The GC leads to the most significant overhead among

[1]In Java, this information can be obtained using the function MemoryMXBean.getHeapMemoryUsage.

runtime's maintenance routines, and due to that, many studies analyzed its impact (TENNAGE et al., 2019; YU et al., 2016; JONES; HOSKING; MOSS, 2011). Despite the tremendous efforts to improve the GC performance, its negative performance impact is still not acceptable in some scenarios with strict performance requirements (HUDSON, 2018; GIDRA et al., 2015). Furthermore, the existing solutions are either runtime, application, or load-specific (KURNIAWAN et al., 2020; MAAS et al., 2016; YU et al., 2016; TEREI; LEVY, 2015; GOG et al., 2015).

Other application-specific maintenance activities increase the latency tail of cloud microservices, such as backup, index defragmentation, and compactions. To make things more concrete, let us take a look at the issue opened by Dominic Kim in CouchDB's Github repository[2]. The issue describes a systematic performance degradation happening during a benchmark of the CouchDB API. The test consists of inserting batches composed of 500 documents, and the cluster used in the test was composed of 3 replicas. They found that CouchDB compactions and competition for resources slowed the server down during those intervals (CouchDB Team, 2020). When the compaction finished, the performance was restored.

Another widely-used distributed data store, Apache Cassandra, has a troubleshooting page dedicated to slow reads[3] suggesting tweaks on the compaction frequency and strategy. A similar performance impact might happen when search requests arrive into an Apache SOLR (LAMBERT, 2016) or Elasticsearch (GORMLEY; TONG, 2015) cluster, and a merge operation is happening (ELASTIC, 2020). Despite their impact, all those routines are essential to those software systems' effective operation.

Even though there has been much engineering effort to make those maintenance operations faster and consume fewer resources, the problem has not been solved. For instance, one can find recent posts suggesting tweaks on indexing refresh interval[4] to make search performance better. The main problem with those configuration recipes is that they are application and load-specific.

[2]Available at https://github.com/apache/couchdb/issues/1065
[3]Available at https://docs.datastax.com/en/dse-trblshoot/doc/troubleshooting/slowReads.html
[4]Available at https://www.elastic.co/blog/advanced-tuning-finding-and-fixing-slow-elasticsearch-queries

2.2 Impact Model

An interactive BT starts when enough requests have been processed, so the system state fulfills the task-specific trigger conditions. Processing a BT might degrade the microservice's performance on the same virtual machine or container. For example, assuming one or more requests are being handled while a garbage collection is happening, there would be competition for shared resources like CPU. Another source of performance degradation caused by GC execution is the pause of all application threads, leading to new requests waiting in a queue and no progress on requests in-flight. Proposing a solution to eliminating the performance impact of interactive background tasks is the primary goal of this thesis.

To better understand the problem, we propose a model to the microservice's replica degraded state of operation while executing background tasks. In that state, the microservice processing the request might be operating at a reduced capacity, leading to a performance penalty. To accommodate the evaluation of degradable-performance systems using model-based methods, Meyer introduced "performability" for combining performance and availability metrics (Meyer, 1980). In this case, the failure-repair model includes performance indicators to evaluate overall system behavior.

The interactive BT T_j starts when a measurement $\zeta_j(t) \in A_j$ for a certain time t. Note that only the current measurement is needed to calculate whether the task should be initiated at time t. That means the request-oriented background task interference is memoryless. When that is the case, a common approach for performability evaluation results in a Markov reward model (Smith; Trivedi; Ramesh, 1988). One major disadvantage of such models is the explosion of state-space even for small systems, resulting in tedious specifications (HAVERKORT; MARIE; TRIVEDI, 2001; TRIVEDI et al., 1993).

Stochastic Petri Nets (SPN) offers a concise representation of a system's behavior. Furthermore, its modeling power is the same as that of Markov reward models, as their representation is isomorphic to a Continuous Time Markov Chain (CTMC) (CIARDO et al., 1993). Thus, the evaluation of an SPN comprises generating and solving the underlying CTMC, and there are many tools for automating that (BIAGI et al., 2017; Couvillion et al., 1991; Ciardo; Muppala; Trivedi, 1989). In the following subsections, we will present an overview of SPNs and use this background to model the impact of BTs on microservice performability.

2.2.1 Stochastic Reward Nets (SRN)

Petri Net (PN) is a bipartite directed graph of places connected via transitions (PETERSON, 1981). Places can contain a non-negative number of tokens. The presence of tokens defines a marking $M = (\#(P_1), ..., \#(P_n))$, where $\#(P_i)$ is the number of tokens in place i, and n is the number of places in the net. Alternatively, $\#(P_1, M)$ gives the number of tokens at place P_1 in the marking M. Each distinct marking of the PN constitutes a state of the system.

A marking M_i is reachable from marking M_j if there exists a sequence of transitions from M_i that leads to M_j. The reachability graph of a PN is the set of markings that are reachable from the initial marking. A marking is vanishing if there is at least one immediate transition enabled. Otherwise, it is called tangible. The graphical view of a SRN presents places as circles, transitions as bars, and tokens as dots or integers inside places. Timed transitions are drawn as unfilled rectangles and immediate transitions as lines.

There are two types of directed arcs in the PN: input arcs, which connect places to transitions, and output arcs, which connect transitions to places. The net condition may enable some transitions to fire, which happens when each of its input places contains the number of tokens assigned to the input arc (i.e., arc multiplicity). This firing of an enabled transition moves tokens, which flow atomically from places connected to the enabled transition by an input arc to places connected via an output arc. This movement might lead the net to a new marking.

For what concerns performance evaluation, to obtain equivalence between a PN and a CTMC, it is necessary to introduce temporal specifications such that the model's evolution is memoryless. To this end, an extension of PN called Stochastic Petri Net (SPN) introduced markings with exponentially distributed firing times (MOLLOY, 1982). SPNs have been further generalized to improve the stiffness of their subjacent linear equations, i.e., making them easier to solve with an acceptable degree of accuracy through numerical techniques. That led to Generalized Stochastic Petri Nets (GSPN), which introduces the concept of immediate transitions, which are fired promptly (MARSAN; CONTE; BALBO, 1984).

GSPN also introduces ways to break ties between simultaneously enabled transitions. Priorities are non-negative integers, and the enabled transition with the greatest priority has precedence over all other transitions. Probabilities are weights of each transition, and differently from priorities, any enabled transition with positive probability may fire next. If two or more immediate transitions can be enabled, they must have either a priority or a probability set. Ties between exponentially distributed timed transitions can be broken by choosing the one with a minimum delay to fire next (i.e., race). Finally, GSPN also has the concept of inhibitor arc, which is an arc from a place to a transition that inhibits the firing of the transition when a token is present in the input place. Those features make possible to convert a GSPN to an equivalent CTMC and vice-versa (MARSAN; CONTE; BALBO, 1984), which enables the usage of numerical computation to calculate the steady-state, transient, cumulative, and sensitivity measures of the GSPN based on the subjacent CTMC.

Stochastic Reward Nets (SRN) are an extension of GSPNs proposed by Ciardo et al., which substantially increase the modeling power of GSPNs by adding many concepts, for instance, guard functions and general transition priorities (MUPPALA; TRIVEDI; WOOLET, 1991; Ciardo; Muppala; Trivedi, 1989). One important concept for this thesis is the reward rate, which can be generally defined as the numeric representation of tangible marking's worthiness. Those are usually expressed as real numbers and are specific to the situation to model. Reward rates can be used to obtain system performance/availability measures and combined measures of performance and availability (i.e., performability). For instance, the instantaneous bandwidth can be used to compute the bandwidth availability in the face of system degradation (TRIVEDI et al., 1992). Trivedi et al. present many practical examples of such combined analysis (TRIVEDI; ANDRADE; MACHIDA, 2012).

SRNs can be automatically converted into a Markov reward model (CIARDO et al., 1993), and thus several measures can be computed numerically (MUPPALA; CIARDO; TRIVEDI, 1994). This work has two measures of interest: the expected reward rate in steady-state and at a given time (instantaneous). The first measure of interest concerns the net - and the subjacent stochastic process - in the steady-state, where the transition probabilities do not change over time. Assuming that X is the random variable corresponding to the reward rate in steady-state, its expected value in steady-state $E[X]$ can be calculated as,

$$E[X] = \sum_{M_k \in \mathcal{T}} r_k \cdot \pi_k$$

where $\mathcal{T}$ is the set of tangible markings, r_k is the value of the reward rate in the marking k, and π_k is the steady-state probability of that marking.

In addition to the steady-state analysis, SRNs allow us to perform analyses based on the instant (transient) reward rate value. Let $X(t)$ be the random variable corresponding to the instantaneous value of the reward rate at time t, the expression for expected reward rate at time t is

$$E[X(t)] = \sum_{k \in \mathcal{T}} r_k \cdot P_k(t)$$

where $P_k(t)$ is the probability of the tangible marking M_k at time t.

It is important to note that the definition of reward rates is orthogonal to the target measure. Thus with the same reward definition, one can compute the steady-state expected reward rate and instantaneous reward rate at time t.

2.2.2 SRN Model

The SRN model presented in Figure 2.1 outlines the servicing of one request by a cloud microservice and explicitly represents the impact of background tasks. It starts at the place P_serv, which represents the start of the request handling. The stochastic transition T_serv represents the response time. After handling, the request could end, reaching the place P_fin through the transition T_no_impact. Alternatively, the token can go to the state P_impact with probability Ω through the immediate transition T_has_impact. Through that latter flow, before reaching P_fin, the token has to wait for the timed transition T_impact, which represents the impact of an interactive BT on the service time.

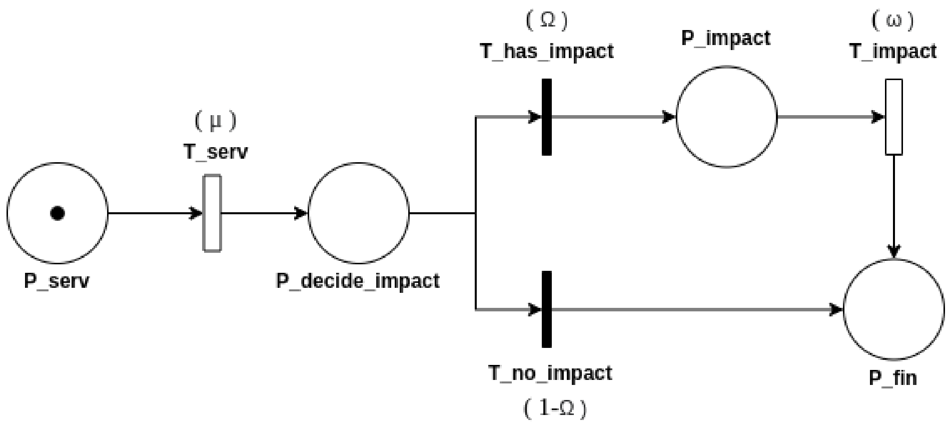


Figure 2.1: SRN model of the impact of an interactive background task on the cloud microservice's response time

In summary, the model has three parameters: i) μ - parameter of the exponential distribution describing the response time; ii) ω - parameter of the exponential distribution describing the impact of an interactive background task on the response time; iii) Ω - the probability of a BT interfere in a request.

This model allows us to compute the Cumulative Distribution Function (CDF) of the response time considering the BT impact. That comes from the fact that the probability of P_fin having tokens at time t gives us the probability the request has ended by t. That probability can be computed as the expected value of the reward rate $E[X(t)]$ at time t by attaching the reward rate st to tangible marking M_k as follows:

$$st = \begin{cases} 1, \text{if } \#(P\_fin, M_k) > 0, \\ 0, \text{otherwise}, \end{cases} \tag{2.1}$$

The reward function st assigns 1 to all markings where P_fin is not empty and 0 to all the others markings. Thus, the distribution of the response time can be obtained by calculating the value of $E[X(t)]$ for different values of t.

2.2.3 Example of the Model Resolution

To illustrate the model resolution, we analyze the impact of the automatic GC (background task) in the execution of a read-only workload in the distributed database Cassandra (LAKSHMAN; MALIK, 2010). The model parameters were extracted from measurement experiments presented by Maas et al (MAAS et al., 2016).

The average response time observed in the measurement experiments was $277\mu s$, which brings us to a rate parameter of $\mu \approx 1/277 \approx 0.0036$. Since the impact of the GC caused requests to have a response time of around 3 orders of magnitude above the average, we configured the exponential distribution parameter describing the impact of the background task to $\omega \approx 1/10^3 \approx 10^{-3}$. The example compares the modeled impact with a virtual case where the GC is not executed. The former scenario can be described by considering that the GC impacted $\approx 3 \cdot 10^3$ requests out of 10^6 requests processed, which leads to $\Omega \approx 3 \cdot 10^3/10^6 \approx 0.003$. The latter scenario, by making $\Omega = 0$. Table 2.1 summarizes the parameters used in this analysis.

Parameter	Description	Value	
		GC Imp.	No GC Imp.
μ	Response time rate without impact of GC	0.0036	0.0036
ω	Rate of the impact of GC on the response time	0.0001	0.0001
$\boldsymbol{\Omega}$	**Probability of GC impact**	**0.003**	**0**

Table 2.1: Parameters used to execute the SRN model and calculate the impact of GC on the microservice's response time.

We execute the SRN model described in Figure 2.1 using the ORIS tool (BIAGI et al., 2017) to calculate and compare the response time CDF using $E[X(t)], 0 \leq t \leq 20$. Figure 2.2 presents the results of the execution of the SRN model for both cases: with GC impact ($\Omega = 0.003$) and without ($\Omega = 0$). The vertical lines allow us to compare the 99.9^{th} percentile and the medians of both cases considered. The execution of the GC during request handling is well known to bring the highest percentiles far from the median of the response time distribution, i.e., the tail of the distribution (DEAN; BARROSO, 2013).

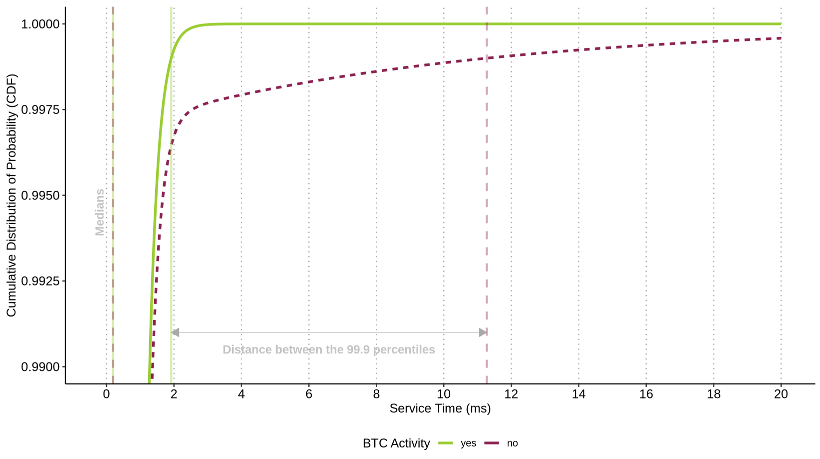


Figure 2.2: Modeled impact of the GC on the response time of a cloud microservice.

The impact of GC on the service response median is negligible. However, triggering the GC during request processing causes the 99.9^{th} percentile response time to rise from $1.9ms$ to $11.27ms$, which makes it ≈ 50 times more distant to the median. In addition to being close to the results presented by Maas et al. (MAAS et al., 2016), the long tail performance presented by the model is confirmed by other measurement experiment results, e.g., Sections 5.2, and 5.3.

Chapter 3

Background Tasks Controller (BTC)

This work focuses on interactive cloud applications with strict performance requirements and whose architecture is based on microservices, such as (near) real-time data search and management services (ABRAHAM et al., 2013; JALAPARTI et al., 2013). In such applications, running a background task without any coordination can lead to unacceptable performance degradation. Let us consider the case of a cloud microservice written using managed languages. An automatic garbage collection can happen during the processing of a request and affect its response time because of competition for CPU or freezing of the execution environment (MAAS et al., 2016; XIAN; SRISA-AN; JIANG, 2008).

In the following sections, we describe the Background Tasks Controller (BTC), a decentralized solution to eliminate the negative impact of background tasks on cloud microservices' response time. After providing an overview of the execution flow, we detail how BTC controls one replica. We then present how the solution avoids the impact of background tasks on a replicated microservice.

3.1 Execution Flow Overview

The main goal Background Tasks Controller (BTC) is to eliminate the negative impact of executing background tasks on cloud microservices' response time. BTC does it by avoiding processing requests while executing background tasks. To avoid that concurrent execution, BTC needs to control i) the execution of background tasks and ii) the microservice's request admission. The former allows the solution to avoid starting background tasks while requests are executing and the latter allows denying requests during these executions, i.e., replica unavailability periods. Each microservice replica is associated with an independent instance of the BTC running on the same Virtual Machine (VM) or container. Figure 3.1 presents the flowchart of the BTC operation on a single microservice replica.

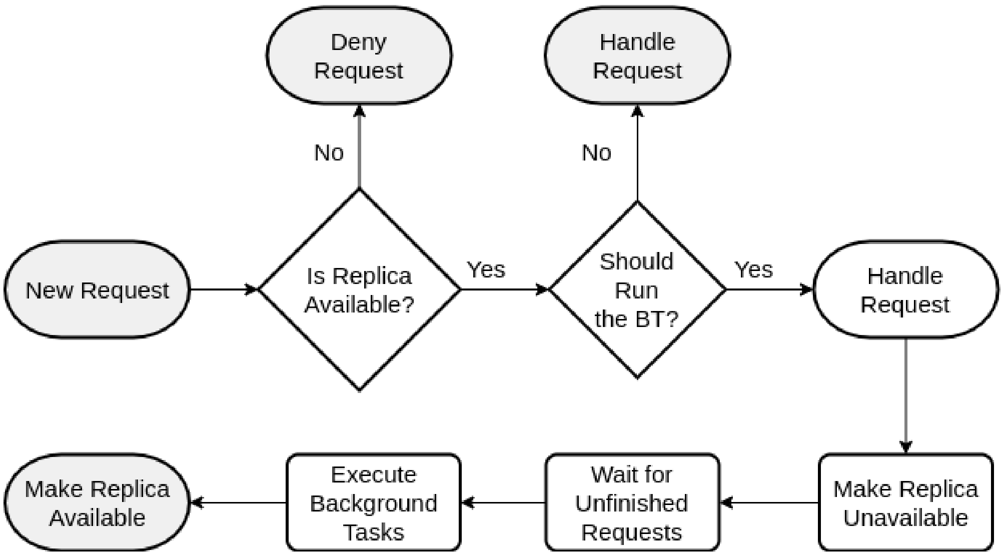


Figure 3.1: Flowchart describing the operation of a single BTC instance, which is associated to a microservice replica.

When a new request arrives, the first step is to decide whether the request should be admitted for execution. Controlling the request admission is crucial to avoid the concurrent execution of requests and background tasks. If the replica is unavailable, e.g., because a BT is already running, BTC denies the request. Denying a request means informing the load balancer about the replica's temporary unavailability. That allows the cloud load balancer to resend the request to a different service replica.

The BTC takes measurements from task monitors to decide when to start background tasks. For example, heap usage is relevant to decide when to trigger the garbage collector, and disk usage to decide when to execute log compaction. A task monitor represents any source of information, e.g., the container, the virtual machine, the runtime environment, the associated microservice, or any other source. Determining the right time to collect measurements from task monitors is not a trivial task. On the one hand, collecting too often might impact system performance. On the other hand, collecting too late might excessively delay the execution of background tasks. We will detail the problem and the solution in the next section.

If it is time run a BT, besides triggering the request processing, the BTC makes the replica unavailable, denying incoming requests until the replica is available again. In addition to that, the BTC tracks the completion of all pending requests. When the microservice is done processing all unfinished requests, the BTC triggers the appropriate BT. The replica becomes available when the BT finishes.

3.2 Controlling One Replica

To perform the execution control flow illustrated in Figure 3.1 each microservice replica is associated with an independent instance of the BTC running on the same Virtual Machine (VM) or container. Each BTC instance is composed of a controller, the admission control, and a set of notifiers. Figure 3.2 details the relationship between BTC components of a single microservice replica.

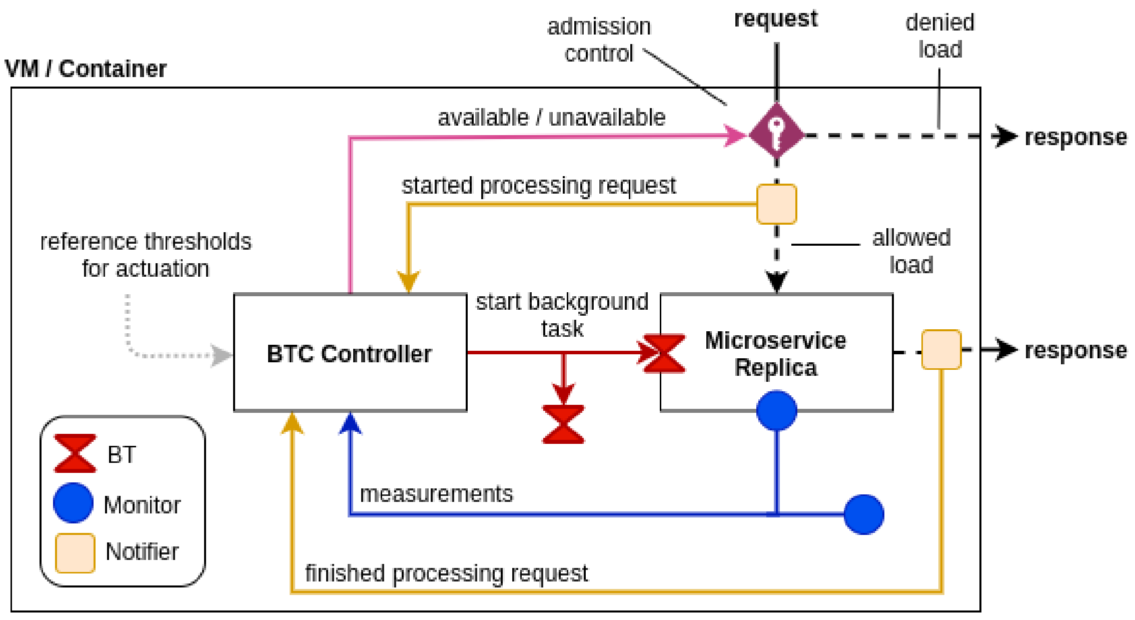


Figure 3.2: BTC internal components and operation.

Colored arrows represent BTC internal messages, which flow to or from the controller depending on the case. Dashed black arrows describe the request path, which starts at the admission control and may reach the microservice replica or be denied. Dotted gray arrows represent parameters specified by the BTC operator.

The load directed to a microservice replica goes through the admission control, which is managed by the BTC controller through available/unavailable messages. Those messages indicate whether the microservice replica should handle incoming requests or report unavailability. The admission control only needs a boolean state to determine whether to handle or reject the request and change that state based on controller messages.

Listing 3.1 presents that behavior assuming the $is\_available()$ function reflects the state updated by the last controller message. When $is\_available()$ returns $true$, the request flows to the microservice replica through the $handle(request)$ function. Otherwise, the admission control prevents the microservice replica from handling the request by informing the load balancer about the replica's temporary unavailability. This response allows the load balancer to resend the request to another replica of the microservice transparently to the client.

Listing 3.1: Algorithm of the admission control

```
def before_service(request):
    if is_available():
        handle(request)
    reject(request)
```

The controller receives a notification as soon as the microservice replica finishes handling each request. That notification flows concurrently with the response delivery, and each notification triggers the execution of Listing 3.2. First, the controller selects background tasks that need to be executed at that moment, considering previous measurements and configuration parameters. If there is no BT to execute, the flow skips to the end.

Suppose it is time to execute at least one background task. In that case, the controller starts the replica unavailability period by calling $make\_unavailable$ (Listing 3.2 line 4). That sends a message to the admission control, which updates its internal state so that $is\_available()$ - from Listing 3.1 - begins to return $false$ and thus, starts denying requests. While the replica is unavailable, the controller waits for all remaining requests. The controller does that using a counter, which is updated based on notifier messages. Only when all requests are finished execution, the controller starts the background tasks. The controller waits for the BT to finish (i.e., Listing 3.2 line 8) and sends the availability message to the admission control. That action makes the admission control start accepting new requests.

Listing 3.2: Algorithm executed by the controller after processing each request

```
def after_service():
    to_execute = select(tasks)
    if not to_execute.is_empty():
        make_unavailable()
        wait_unfinished_requests()
        for task in to_execute:
            task.execute()
        make_available()
```

So far, we have detailed almost all the controller's algorithms to coordinate the execution of background tasks and the request admission to avoid the impact of those tasks on the microservice's performance. The only missing part is to explain how the controller selects the background tasks to execute (function $select(tasks)$ in Listing 3.2), which is presented by Listing 3.3. This pseudocode's rationale is based on the definition of interactive background tasks provided in Section 2.1. There we define the activation interval $A_j = \{a \in Im(\zeta_j) \,|\, A_j^{min} \leq a \leq A_j^{max}\}$, which represents a set of measurements from task monitor ζ_j that should lead to the start of T_j. The controller chooses tasks in which the monitor measurement for the current moment belongs to the activation interval. If there is none, it returns an empty set.

Listing 3.3: $select(tasks)$: selects the set of background tasks to be executed at the current moment

```
def select(tasks):
    selected = set()
    for task in tasks:
        if task.monitor.measure() in task.activation_interval:
            selected.add(task)
    return selected
```

Estimating the Execution of the Next Backgrount Task

Although simple, executing the Algorithm 3.2 after each request has a potential performance problem, which could occur if the monitors's measurements are costly operations or when the microservice is under heavy load. For example, let us consider the methods to access information regarding memory usage exported by Java, Python, Ruby, and many other runtime environments. Even though these are generally inexpensive operations that return reasonable estimates (GORELICK; OZSVALD, 2014; SHAUGHNESSY, 2013), there are scenarios in which determining current memory usage are costly operations, e.g., when objects are not allocated contiguously in memory (ORACLE, 2018b).

Furthermore, the execution of the Algorithm 3.2 on different replicas might hurt the microservice availability due to concurrent BT executions. As the activation interval for each BT is the same across different replicas, and the load is balanced among replicas, those concurrent unavailability periods happen when the request processing impact on the task monitor measurements has not much variance among the replicas.

As we can neither optimize all possible task monitors' overhead nor want to make BTC usage more complicated, BTC avoids calling the $select(tasks)$ function after processing every request. A simple way to reduce the number of calls is to set a fixed $r > 1$ number of requests processed between checks. That approach has two main problems. First, even though a good value for r can be selected through extensive performance testing or by an expert operator, that value might change over time, as it depends on the microservice's code, setup, and load. Second, non-expert operators and expert operators that do not update r very often deal with a complex trade-off: on the one hand, checking too often can harm performance; on the other hand, if the number of requests processed between checks is very high, the controller loses effectiveness, and some interactive tasks might not be triggered when they should.

Another way to make $select(tasks)$ calls less frequent is to repeat the check after a time interval $\Delta t > 0$. Other studies used this method, setting a minimal Δt to check the Java GC (MAAS et al., 2016). Even though that might work for some cases, we do not consider this solution for the broad class of interactive background tasks because it leads to trade-offs similar to the fixed number of requests processed. Furthermore, the arrival rate can vary widely, and peaks or valleys can occur unexpectedly.

As neither fixing the number of requests nor a fixed period are good options to decrease the frequency of $select(tasks)$ calls, we propose a technique that adjusts the number of requests processed before checking monitors during runtime. That adjustment happens after each execution of a background task and must consider the variation of how processing requests impact background tasks. To do so, the controller needs to keep track of the number of requests processed since the last execution of each task[1], $P^{last} = \{P_j^{last} \in \mathbb{N}^0 \mid \forall T_j \in T\}$.

Furthermore, the controller accepts one optional parameter $P^{max} = \{P_j^{max} \in \mathbb{N}^0 \mid \forall T_j \in T\}$, which is a set containing upper bounds of the number of requests that should be processed between executions of each task $T_j \in T$. As P^{max} is configured by a specialist, we expect that, for every background task $T_j \in T$, if $P_j^{last} \leq P_j^{max}$, there is low probability of measurements go beyond the activation interval (i.e. $\zeta_j > A_j^{max}$). That is important because, in some scenarios like the Java GC, skipping an activation interval might lead to a performance drop due to uncontrolled background task execution. Suppose operators cannot provide P_j^{max} for any background task T_j. In that case, the controller falls back to checking task monitor measurements after each request processing and updates P_j^{max} to the number of requests processed by the time to trigger the first execution of T_j. We treat this as an exceptional warmup case that is completely decoupled from the remainder of the solution.

Finally, the technique only has one constraint, which should be valid for every background task $T_j \in T$: the measurements $\zeta_j : \mathbb{R} \to \mathbb{R}$ between two consecutive executions of T_j must be monotonic. That constraint ensures that if the controller allows processing enough requests, all task monitor measurements will eventually reach the activation interval, which leads to task executions. All controlled interactive background tasks presented in this research fit in this constraint.

Equation 3.1 defines est_j, which estimates how many requests should finish until the next execution of $T_j \in T$. The equation calculates how much the monitor measurement needs to increase to reach the activation interval (i.e., $X_j - \zeta_j$) and uses a simple average to estimate the impact of each request processing on monitor measurements. Furthermore, the result is bound by P_j^{max} for lowering the probability of skipping the activation interval.

$$est_j = \begin{cases} P_j^{max} & \text{if } P_j^{last} = 0 \\ min\left(P_j^{max}, \left\lceil \dfrac{X_j - \zeta_j}{\zeta_j / P_j^{last}} \right\rceil\right) & \text{otherwise} \end{cases} \tag{3.1}$$

Instead of using A_j^{max} or any other deterministic number to determine how much ζ_j should grow before it is time to execute T_j, Equation 3.1 uses $X_j = U(A_j^{min}, A_j^{max})$, that

[1]If it is the first execution of T_j, P_j^{last} is set to the number of requests processed since the start.

is a continuous uniformly random variable which draws numbers from the activation interval. This entropy bound to the activation interval is a fundamental condition for the solution's decentralization and has already been used to avoid synchronization of independent instances (FLOYD; JACOBSON, 1994). In the context of BTC, it decreases the probability of service outage caused by the concurrent execution of background tasks in different replicas of the microservice.

For better understanding the importance of randomness to the tecnique let us use the garbage collector T_{GC} example from Section 2.1. The example provides $\zeta_{GC} : \mathbb{R} \to \{s \in \mathbb{R} : 0 \leq s < 16GB\}$ and $A_{GC} = \{a \in \mathbb{R} : 2GB \leq a < 2.5GB\}$. Let us also consider that microservice has 2 replicas, each request consumes 1MB, the $P_{GC}^{max} = 1000$ requests, and the runtime starts with 0.5GB heap used. After P_{GC}^{max} requests, $\zeta \approx (1000 * 1MB) + 0.5GB \approx 1.5GB$. If we use Equation 3.1 to estimate how many requests to process before the next execution of T_{GC} in each replica we could have:

- Replica 1 ($X_{GC} = 2.3$GB): $\left\lceil \frac{2.3\text{GB} - 1.5\text{GB}}{1.5\text{GB}/1000} \right\rceil \approx \lceil 600\text{MB}/1.5\text{MB} \rceil = 400$
- Replica 2 ($X_{GC} = 2.5$GB): $\left\lceil \frac{2.5\text{GB} - 1.5\text{GB}}{1.5\text{GB}/1000} \right\rceil \approx \lceil 1000\text{MB}/1.5\text{MB} \rceil = 667$

So, the unavailability period of the microservice replicas would be 267 requests executions apart. Assuming that each controller replica starts with a different pseudorandom seed, X_{GC} would likely return a different value each time the estimation function is invoked, both in the same or in different replicas. That difference is essential for reducing the capacity loss caused by the unavailability periods while keeping the solution completely decentralized.

Those results allows us to use Equation 3.1 to modify Listing 3.2 and update the number of requests to wait before the subsequent task monitor verification. That new algorithm is presented by Listing 3.4, which is still being executed after each notification of a request termination. Those notifications happen concurrently with the response delivery, so the client does not wait for that internal procedure.

The functions $make\_available$ and $make\_unavailable$ are the same as in Listing 3.2 and control the replica availability. Requests are only accepted for processing when $is\_available()$ is $true$. The variable num_wait represent the number of requests that should be processed before the next round of checks, respectively. num_wait is only updated by the $after\_service()$ function.

Listing 3.4: New algorithm executed at the end of every request processing. It uses the result of Equation 3.1 to decrease the frequency of task monitor measurements.

```
def after_service(tasks):
    if num_wait > 0:
        num_wait = num_wait - 1
        return
    ### Executing selected tasks
    to_execute = select(tasks)
    if not to_execute.is_empty():
        make_unavailable()
        wait_unfinished_requests()
        for task in to_execute:
            task.execute()
        make_available()
    ### Updating num_wait
    num_wait_tasks = set()
    for task in tasks:
        num_wait_tasks.add(est(task))
    num_wait = min(num_wait_tasks)
```

When the controller starts, $num\_wait = 0$. This setup leads to a round of estimations (possibly executions). If it is time to check, the code proceeds and executes all the needed tasks. Finally, Algorithm 3.4 estimates, for each task, how many requests until the next execution. It then updates num_wait with the minimum number of requests estimated. In that way, the behavior of BTC remains simple and purely based on only two types of notifications: it executes Algorithm 3.1 at request arrival and Algorithm 3.4 after request termination.

3.3 Controlling Multiple Replicas

We have presented how the BTC technique works to control the admission and execution of competing tasks in each replica of the microservice. It is important to note that we designed the technique so that the control of each replica's background tasks is carried out independently, without the need for coordination between the controllers. Also, BTC operates without changing the microservice business logic or inspecting the request content.

Cloud microservices are usually composed of replicas and a load balancing system, which forwards requests to replicas according to a scheduling policy. Since BTC usage does not depend on changes in the microservice code or the infrastructure, this *plug and play* nature can lead to the denial of requests directed to replicas executing background tasks. Note that this is different from replicas being simultaneously unavailable due to the execution of tasks, which is effectively reduced by the randomness introduced by Equation 3.1.

There are several ways to prevent the temporary unavailability of a replica from leading to a request denial. One of the simplest and already used by the state of practice is to configure the load balancer to reschedule a request with a specific response (NGINX, 2020). One of the most common reschedule policies is trying all replicas and denying servicing service only when none are available. This policy comes out of the box in the load balancer Nginx (NGINX, 2020). A more complex policy is to keep track of unavailable replicas and avoid them when (re)scheduling. Suppose we use the Hypertext Transfer Protototocol (HTTP) protocol as an example (FIELDING; RESCHKE, 2014). In that case, that policy can be implemented using the status code 503 for blocklisting replicas and using the `Retry-After` response header to set a deadline and mark them back as available. Propose and evaluate (re)scheduling policies is out of the scope of this work.

Chapter 4

Evaluation of the BTC Impact on Performability

Performability models are powerful in describing such complex interactions, but the underlying analytical framework, i.e., Markov chains, limits transition rate generators to exponential distributions. As an example of such rates, we have the impact of background tasks. Even though other studies have used SRNs to evaluate distributed systems (ZHANG et al., 2018; ENTEZARI-MALEKI; TRIVEDI; MOVAGHAR, 2015), our experience shows that the exponential distribution might not be ideal to describe the BT duration in production workloads. For example, Figure 4.1 presents the garbage collection duration ECDF measured in experiments with a replicated production-grade microservice (more details at Section 5.3).

Some authors have been working hard on analytical methods to overcome this limitation of SRNs (A; M, 1998). However, unfortunately, the model description language, its requirements, and the tooling for solving such problems increase complexity. Another limitation of the Markovian models is the state space explosion. Using the BTC model as an example, the state space grows with the number of replicas. Thus, evaluating a big cluster might be unfeasible due to the long processing time.

Our work follows the suggestion of some studies and uses simulation to circumvent those limitations (TUFFIN et al., 2007). We limit the usage of the SRN models to formally describe the BTC behavior. The simulator enabled the usage of production workloads in our evaluation. Furthermore, it might achieve results faster than the analytical counterpart[1].

[1]The performance comparison with the analytical model is outside the scope of this work.

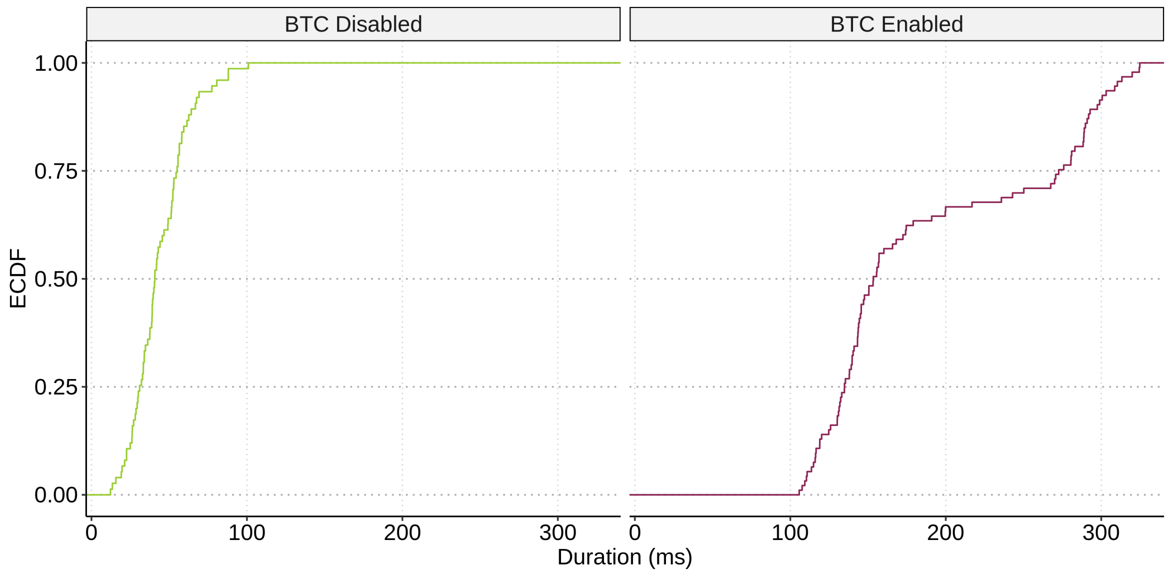


Figure 4.1: Garbage collector duration ECDF of a replicated production-graded microservice with BTC enabled and disabled.

The following sections detail the simulator used in the evaluation and its implementation. Furthermore, we present a sensitivity assessment of the simulated model, which aims to investigate the relations between the model input and outputs (NORTON, 2015). After understanding how model factors affect the performability metrics, we compare the proposed solution with other practical solutions for dealing with response time spikes. Finally, to increase confidence in the simulated results, we verify the simulator model by comparing its results with the analytical model results. We also validate the simulated model by comparing its results with results from measurement experiments executed using a proof-of-concept implementation.

4.1 Metrics

A key performance metric is response time (RT). For the sake of this work, RT is the interval between the load balancer receives the request from the client and its response from a microservice's replica. That definition does not contemplate the network time from the client to the load balancer. Furthermore, the response time considers the impact of background tasks on the microservice's container or virtual machine. The lower and less variable the RT, the better.

As the proposed solution trades a performance improvement for a temporary replica availability loss, another important aspect is the cloud microservice availability or the readiness for correct service (AVIZIENIS et al., 2004). For simplicity, this evaluation does not consider network partitions, hardware, or software errors. Thus, the cloud service is unavailable only when BTC induces the unavailability of all replicas simultaneously. Furthermore, microservices replicas execute independently, and their number does not change during the observed period. Let n be the number of replicas and $P(u_i)$ be the probability of $i \leq n$ replicas are simultaneously unavailable during the observation period. A service outage can be formally described as follows,

$$Unavailability = P(u_n) \tag{4.1}$$

It is also essential to consider the impact of of BTC on microservices partial failures (AVIZIENIS et al., 2004) or outages (BAUER; ADAMS, 2012). Heimann, Mittal, and Trivedi defined Capacity-Oriented Availability (COA), which measures how much service the system delivers, taking into account the relative amount of lost capacity due to internal failures (HEIMANN; MITTAL; TRIVEDI, 1990). As we are more interested in replica unavailability's potential impact, we use its complement, the Capacity-Oriented Service Unavailability (COUA). This metric allows estimating the average service capacity loss due to partial failures. We could formally define COUA as follows,

$$COUA = \sum_{i=1}^{n} P(u_i) \cdot \frac{i}{n} \tag{4.2}$$

Both unavailability and COUA are complementary proxies to evaluate how the solution might affect the costs to run the target system at a specified capacity. While unavailability allows us to assess service outages, an essential aspect of service quality, COUA enables the evaluation of the system's internal faults (AVIZIENIS et al., 2004), which affects the delivered capacity. A decrease in any of those metrics might lead to performance losses due to requests being denied or enqueued (DELIMITROU; KOZYRAKIS, 2018). Depending on the case, keep the desired performance targets might lead to an adjustment in the number of replicas. That adjustment might incur costs.

As an example of applying those metrics, let us suppose a highly loaded cloud microservice with replicas rA and rB observed for 10 minutes. Assume that both replicas run without any unavailability for the first 5 minutes. Suppose that rA starts being unavailable at the end of minute 5 and remains for 2 minutes. During the observation period, rB was unavailable for 1 minute, starting from minute 6. Figure 4.2 illustrates this scenario.

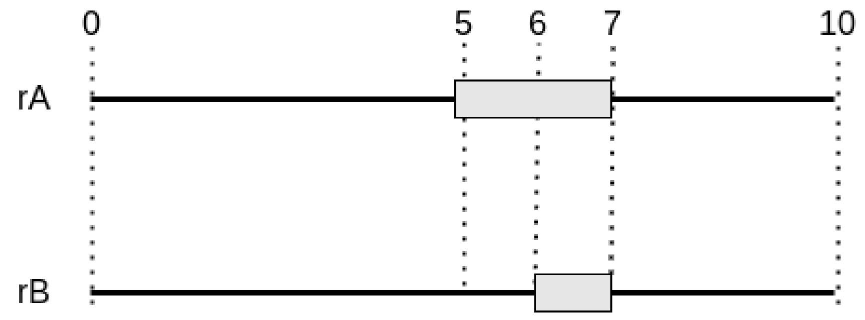


Figure 4.2: Availability timeline of a cloud microservice with two replicas. Replica unavailability is depicted as a gray rectangle over the timeline. The serving capacity is for one minute and there is an outage starts at minute six and ends at minute 7.

As both replicas have been simultaneously unavailable for 1 minute, we have that $Unavailability = P(u_2) = 1/10 = 10\%$. Thus, considering the observed period, the microservice in the example has a 10% chance of denying requests due to an outage. As one replica was available for one minute, $P(u_1) = 1/10$. That leads to $COUA = 0.1 \cdot \frac{1}{2} + 0.1 \cdot \frac{2}{2} = 0.15 = 15\%$, which means that the average service capacity decreased by 15% during the observed period. With that information, the system administrators might decide to increase the cluster size, for example.

4.2 BTC Simulated Model

Focusing on increasing the descriptive power while maintaining the model's simplicity, we implemented a discrete event simulator[2]. This simulator's main objective is to analyze scenarios closer to the state of practice, such as services running in production using traces or measurements. Furthermore, the model is extensible, allowing the simulation of other tail-tolerant techniques, such as duplicating requests.

[2]Source code available https://github.com/gcinterceptor/gci-simulator/tree/master/clustergo

Figure 4.3 presents an overview of the simulator internals. The load generator is responsible for sending requests to the load balancer. It simulates n concurrent client connections, where n is the number of replicas. The load balancer forwards those requests to available replicas using the round-robin policy. Each of the simulated replicas replays a service history, i.e., status code and duration, sequentially as it receives requests from the load balancer. Those service history files are the simulator input.

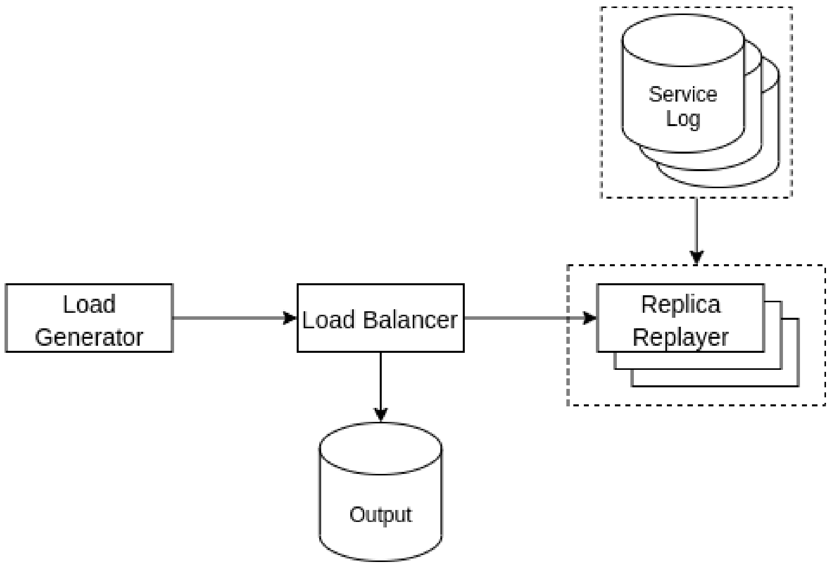


Figure 4.3: Summary of the simulator model operation.

The service history for each replica is stored in a different service log file. Each entry in each of these files represents how a replica handles a request. For instance, let us suppose the following entry is $(50, 200)$ in the history of replica 1. That leads to the subsequent request scheduled to replica 1 will have a simulated response time of $50ms$ and a 200 status code.

After sending the request, the load balancer marks the replica as busy and will only send requests again after the duration specified in service logs. A status code 503 indicates a period of replica unavailability. In the latter case, the replica is marked as unavailable for the duration indicated in the history. When the replica finishes the simulated processing of the request, it sends the response back to the load balancer. That action triggers the output of relevant metrics associated with the request, e.g., the start timestamp, duration, status code.

Bellow, we summarize the behavior of the three simulator components:

- **Load Generator**: responsible for sending requests to the load balancer. It simulates n persistent client connections, where n is the number of microservice replicas;

- **Load Balancer (LB)**: receives requests sent by the load generator and forwards them to available replicas using the round-robin policy. The LB processes requests according to the first-in-first-out policy;
- **Replica Replayer**: models the performance and availability of each replica. Rather than modeling these aspects separately, the simulated replicas reproduce each entry of its respective service log file, which can come from a production trace or a synthetical generator.

4.2.1 Obtaining Metrics

The response time distribution and the unavailability periods of each replica can be obtained from the simulator output. However, the same does not happen with the Unavailability and the Capacity-Oriented Unavailability (COUA).

COUA was defined in Section 4.1 as an average measure of how much service the microservice is unable to deliver. In the context of this thesis, that capacity loss happens due to replica unavailability periods induced by BTC while a background task is executing (i.e., internal failures). When all replicas are concurrently unavailable, we say the microservice is unavailable, which means it cannot attend requests. The Unavailability metric measures the probability of a request be denied due to microservice unavailability.

Let $n \in \mathbb{N}^+$ be the number of replicas of a microservice being simulated. For $i \in \mathbb{N}^+, i \leq n$, let $U_i = \{u | u \in \mathbb{R}\}$ be multiset containing the duration of each unavailability episode involving i replicas, and $SU_i = \sum_{u \in U_i} u$ the total amount of time. For instance, each element of U_2 represent a time interval where 2 replicas have been simultaneously unavailable. For an experiment during D time units, we calculate Unavailability as the following,

$$Unavailability = \frac{SU_n}{D} \tag{4.3}$$

Furthermore, we calculate COUA as the following,

$$COUA = \sum_{i=1}^{n} \frac{SU_i}{D} \cdot \frac{i}{n} \tag{4.4}$$

4.3 Synthetic Input Generator

We create a synthetic input generator to enable the simulator verification and a thorough evaluation of BTC's impact on the cloud microservice performability. The generator creates synthetic service time history, the only simulator input[3]. For instance, the input generator allows us to smoothly perform an experiment varying the probability of background task impact.

Listing 4.1 presents the input generation algorithm when BTC is not enabled. The $num\_replicas$ and num_reqs specify the number of service log files and the number of requests per file, respectively. The rt_set is a set of response times. The bt_prob and bt_impact_set specify the probability of background execution and its impact on response time.

Listing 4.1: Algorithm of the simulator's synthetic input generator when BTC is not active

```python
def gen_input(num_replicas, num_reqs, bt_prob, rt_set, bt_impact_set):
    for replica in range(num_replicas):
        for req in range(num_reqs):
            rt = sample(rt_set, 1)
            if rand() < bt_prob: # Should have BT impact?
                rt = rt + sample(bt_impact_set, 1)
            out(replica, req, 200, rt)
```

For each simulated replica, it generates num_req requests. As BTC is off, there is no induced replica unavailability. After randomly sampling rt_set, if it is time to introduce background task impact synthetically, it randomly samples bt_impact_set and increments the response time.

Listing 4.2 presents the input generation algorithm when BTC is active. It has very similar parameters to the case when BTC is disabled. The only difference is the $bt\_duration\_set$, which contains the duration of background task executions instead of its impact. It computes whether to execute the BT based on bt_prob. When is the time, it samples $bt\_duration\_set$ and returns a status code 503, which indicates the replica unavailability to the simulator. Otherwise, it samples the rt_set and returns a success status code.

[3]Available online at <https://github.com/gcinterceptor/gci-simulator/blob/master/clustergo/sa_inputgen>

Listing 4.2: Algorithm of the simulator's synthetic input generator when BTC is active

```
def gen_input_gci(num_replicas, num_reqs, bt_prob, rt_set,
    bt_duration_set):
    for replica in range(num_replicas):
        for req in range(num_reqs):
            if rand() < bt_prob: # Should execute a BT?
                out(replica, req, 503, sample(bt_duration_set, 1))
            else:
                out(replica, req, 200, sample(rt_set, 1))
```

The parameters rt_set, btc_prob, btc_impact_set, $btc\_duration\_set$, could be either synthetically generated or extracted from measurement experiments. The power of that flexibility will be clear in the next sections, as we present the simulator verification and validation, the sensitivity assessment, and the comparison with other solutions.

4.4 Model Verification

Verification is the process of determining whether a simulated model implementation and its associated data accurately represent the corresponding conceptual specification (SARGENT, 2005). This work uses SRN as the language to formally describe the system behavior[4]. So, the verification process presented in this Section consists of asserting whether the metrics derived from the simulated model are in an acceptable range of accuracy compared to the corresponding metrics calculated by the SRN model. More specifically, we are interested in the metrics presented at Section 4.1, i.e., the response time, COUA and unavailability.

4.4.1 SRN Model

Section 3.2 describes the operation of one instance of BTC, the technique proposed in this work, which aims to reduce the negative impact of running concurrent routines on the performance of replicated cloud microservices. Those algorithms are executed by independent controllers and define whether to accept requests based on background tasks' execution. As BTC operates by making replicas temporarily unavailable, it is imperative to describe and

[4]You can find more about the SRN model, its constraints, and parameters at Chapter 3

evaluate its combined impact on performance and availability (i.e., performability). Figure 4.4 presents the SRN model describing the behavior of a cloud microservice integrated with BTC.

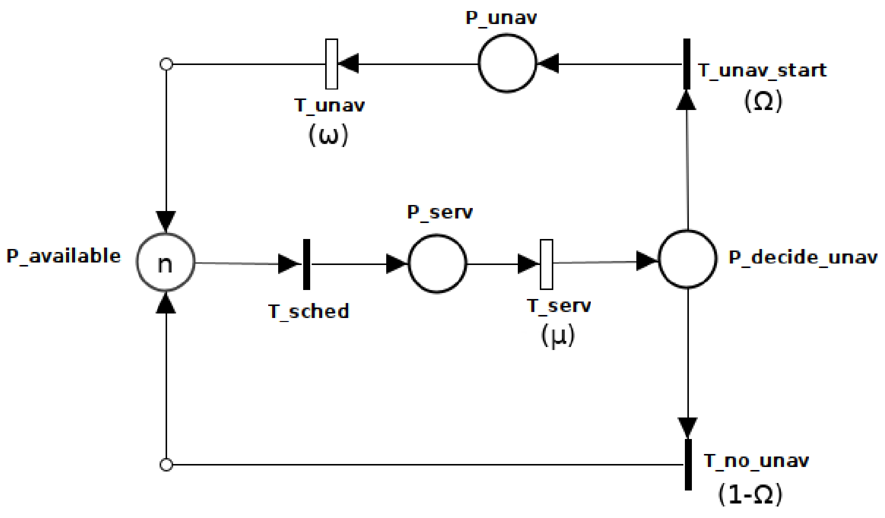


Figure 4.4: SRN model of a replicated microservice using BTC.

We define that the microservice is available when at least one (out of n) replica is available to process requests. The model does not describe the delivery of the client's response, which would happen concurrently after the execution of T_serv. Furthermore, the model restricts to one the number of requests processed by each replica at a time, i.e., no concurrent request handling in each replica. The decision to not model and evaluate the impact of per-replica concurrent request execution aims to simplify the model and isolate BTs as the only performance impact source. Furthermore, even though this work focuses on interactive background tasks, the model expressiveness is not restricted to its impact. One can adjust parameters to consider other performance impact sources, for instance, a backup routine.

The model parameters are summarized in Table 4.1. It is important to notice that SRN restricts the description of stochastic transitions to exponential distributions. Even though other studies have applied this approach to evaluate distributed systems (ZHANG et al., 2018; ENTEZARI-MALEKI; TRIVEDI; MOVAGHAR, 2015), we do not claim exponential rates represent production workloads. Instead, we limit the usage of the SRN model to describe the BTC behavior formally, and Chapter 4 presents the BTC simulator, built to remove this limitation and execute the model with production workloads.

Name	Description
n	Number of microservice replicas
μ	Exponential distribution parameter describing the request service time without BT interference
ω	Exponential distribution parameter describing the duration of BT executions
Ω	Probability of execution of BT

Table 4.1: Parameters of the BTC's SRN model.

Tokens in the place $P\_available$ are immediately transferred to P_serv, which represents the beginning of the request handling. The stochastic transition T_serv is activated according to an exponential distribution with parameter μ and represents the service time. After serving the request, the flow gets back to the controller, which decides if the replica may need to perform a background task; thus, becoming unavailable. The place P_decide_unav represents this verification, and the need to execute a BT happens with probability Ω. The latter is modeled as the weight of the immediate transitions T_unav_start and T_no_unav.

If the activation of T_unav_start occurs, the token goes to the place P_unav. The token remains at P_unav until the trigger of the stochastic transition T_unav, which is characterized by ω. The permanence of a token at P_unav represents the replica unavailability period. Both the activation of the stochastic transition T_unav and the immediate transition T_no_unav takes the token to the place $P\_available$, representing the replica to take new requests.

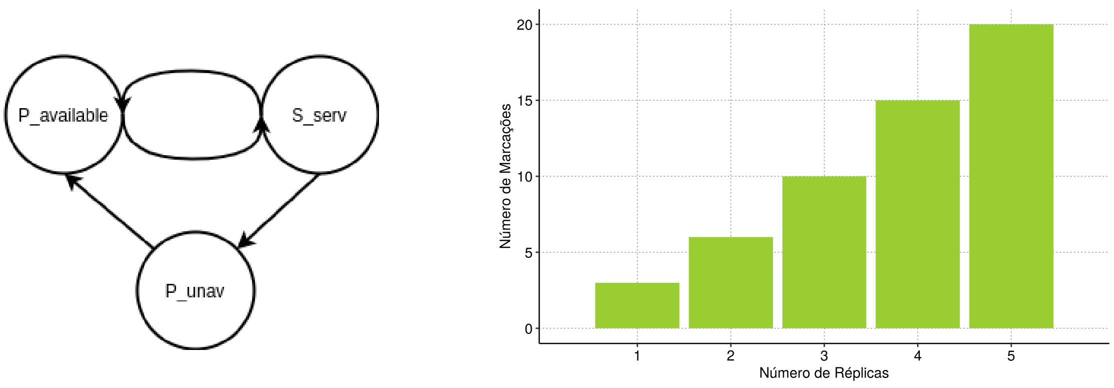
The Underlying Markov Chain

Modeling using SRNs combines the power and simplicity of Petri Nets to describe the system behavior and the computation of performability metrics through the automatic creation of the underlying Markov chain. That automatic generation is essential due to the exponential increase in the number of states in the Markov chain as the system being modeled becomes more complex. Figure 4.5a illustrates this relationship by showing the states and transitions of a Markov process generated from the SRN model presented by Figure 4.4, for $n = 1$. Note that the place P_decide_unav is not part of the Markov chain as it is used to represent a bifurcation (condition) in the SRN model and is only linked to instantaneous transitions.

Once in the place $P\_available$, the chain can only go to P_serv state, which matches the behavior of the SRN model. From that point, the Markov process could go back to the state

$P\_available$, which indicates the task termination without needing to execute a background task. Another option is to go to the state P_unav, representing the execution of a background task and the replica's temporary unavailability. After the task's execution, it can only return to the state $P\_available$, representing that the replica is available to process new requests.

Figure 4.5b illustrates the growth of the underlying Markov chain's tangible state space as the number of microservice replicas increases. This complexity is represented in the SRN model by the initial number of tokens n in the state $S\_available$ and we can see in Figure 4.5a. For example, when $n = 2$ we have 6 possible tangible markings, i.e., $\{(S\_available,S\_available)$, $(S\_available,S\_serv)$, $(S\_available,S\_unav)$, (S_serv,S_serv), (S_serv,S_unav), $(S\_unav,S\_unav)\}$, and for $n = 5$, the number of tangible marking goes up to 20.



(a) Underlying Markov chain (b) Complexity growth

Figure 4.5: Underlying Markov chain for $n = 1$ replicas and the complexity growth as the number of replicas increase

Response Time Distribution

We use the tagged customer model technique to derive the RT distribution (MUPPALA et al., 1994). This technique uses a modified version of the SRN model of interest focused on tracking the steps of a possible customer in the system. We want to evaluate the response time distribution when the BTC is active and not. When BTC is not active, the response time distribution of a request processing can be obtained using the model describing the problem, presented in Figure 2.1.

The base model for BTC activity is presented in Figure 4.4. In that case, background tasks do not impact the service time. Instead, the BTC controls and executes tasks only after request termination. As detailed in Section 3.2, this flow of checking and executing tasks happens concurrently with the delivery of the response. Hence, the client receives the response just after the request processing termination.

The tagged customer SRN model presented in Figure 4.6 represents the fulfillment of a request when BTC is active. It starts in the state P_serv because it is the beginning of a request's processing. As in the SRN model of interest shown in Figure 4.4, the tagged customer model assumes that the service times are described according to an exponential distribution. As the response time does not account for the response delivery, the arrival of the token in the absorbing state P_fin represents the end of the request processing. The model has only one parameter μ, extracted from the base model, which describes the service time.

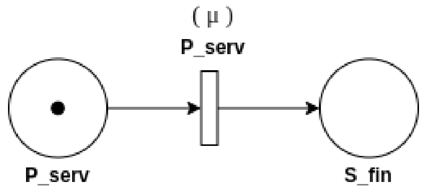


Figure 4.6: Tagged customer model of the request handling when BTC is active

The probability of the token reaches the state P_fin at time t is the probability of modeled request processing has finished at that time. We can obtain that probability by attaching the reward rate rt to the tagged customer model presented at Figure 4.6 and calculating the instantaneous expected value $E[X(t)]$:

$$rt = \begin{cases} 1, \text{ if } \#(P\_fin, M_k) > 0, \\ 0, \text{ otherwise} \end{cases} \tag{4.5}$$

Remembering the notation introduced at Section 2.2.1, $\#(P\_fin, M_k)$ represents the number of tokens at state S_fin in the tangible marking M_k. So, the reward function rt assigns 1 to markings where S_fin has the token and 0 to all the other markings. The response time distribution can be obtained by calculating $E[X(t)]$ for different values of t.

Unavailabiliy and Capacity-Oriented Unavailability

The unavailability is defined as the probability of all microservice replicas be unavailable at the same time due to BTC interference. Considering all n microservice replicas in steady-state, we can obtain the Unavailability metric by attaching the $unav_k$ reward rate to the base model shown in Figure 4.4, and calculating the expected value $E[X]$:

$$unav_k = \begin{cases} 1, \text{ if } \#(P\_unav, M_k) = n, \\ 0, \text{ otherwise,} \end{cases} \tag{4.6}$$

The reward rate $unav_k$ assumes the value one for every marking M_k where all n tokens are in the state P_unav. That means the situation where all replicas are unavailable because of background tasks execution. The reward rate is zero for all other markings because the microservice would either be waiting for or handling requests.

The Capacity-Oriented Unavailability (COUA) was defined at Section 4.1 and represents the average amount of service capacity lost due to replica unavailability. Considering all n microservice replicas in steady-state, we can obtain the COUA metric by attaching the $coua_k$ reward rate to the base model shown in Figure 4.4, and calculating the expected value $E[X]$:

$$coua_k = \#(P\_unav, M_k)/n \tag{4.7}$$

The $coua_k$ reward rate is defined for every marking M_k as the ratio between the number of tokens in the place P_unav and the total number of tokens. It only leads to values greater than zero when the marking represents at least one replica unavailable. The calculation of the $COUA$ metric based on the $coua_k$ reward function relies on the fact that $E[X] = \sum_{M_k \in \mathcal{T}} coua_k \cdot \pi_k$ (more at Equation 2.2.1). As π_k is the steady-state probability of the marking M_k, that expected value is equivalent to $COUA$ definition presented at Equation 4.2.

4.4.2 Methodology

The parameters used in this verification were extracted from the measurement experiments presented by Maas et al. (MAAS et al., 2016), which are explained in Section 2.2.3. The simulated service time, unavailability probability, and duration are be described by exponential distributions with rate parameters, $\mu = 0.0036$, $\Omega = 0.003$, and $\beta = 0.0001$, respectively. We execute the model using the ORIS tool (BIAGI et al., 2017).

To execute a comparable simulation, we need a service history according to the analytical model parameters. To do so, we use the Input Generator, described in Section 4.3. We set the btc_prob to 0.0001 and synthetically generate the rt_set parameter according to an exponential distribution with rate $\mu = 0.0036$. The btc_impact_set and $btc\_duration\_set$ are generated according an exponential distribution with rate $\Omega = 0.003$. With that configuration, we generate one service history file for each simulated replica. Each simulated run processes $\approx 280{,}000$ events as an attempt to approximate the steady-state. We repeated each treatment 20 times.

For the verification of the response time distribution we use the tagged customer model of the request handling presented at Figure 4.6. The verification experiment vary the model type, i.e., analytical and simulated. We obtain the RT distribution by attaching the reward rate rt (i.e., Equation 4.5) and calculating the instantaneous expected value $E[X(t)]$ for $0 < t < 20$.

We use the model depicted at Figure 4.4 to verify COUA and unavailability. We calculate those metrics at steady-state by attaching the rewards rates presented at Equations 4.6 and 4.7 to the model. We verify the results using a 2^2-factorial experiment, varying the number of replicas in addition to the model type.

4.4.3 Results

Figure 4.7 presents the response time ECDFs obtained from $20{,}000$ data points from the analytical model results and $1{,}000{,}000$ from the simulator output. It is essential to notice that the analytical model does not provide us with actual response times but with the probability of requests to finish at a certain time. Thus, we can not use that data to compute a statistical verification, for instance, a non-parametric Goodness-of-Fit test (FEITELSON, 2015). Both curves' matching confirms an acceptable response time distribution.

Figure 4.8 shows how the behavior of COUA and unavailability metrics change as we add more tokens to the analytical model and replicas to the simulated model. Even though we see an almost perfect match in the unavailability distribution, the COUA metric curves slightly differ in some points.

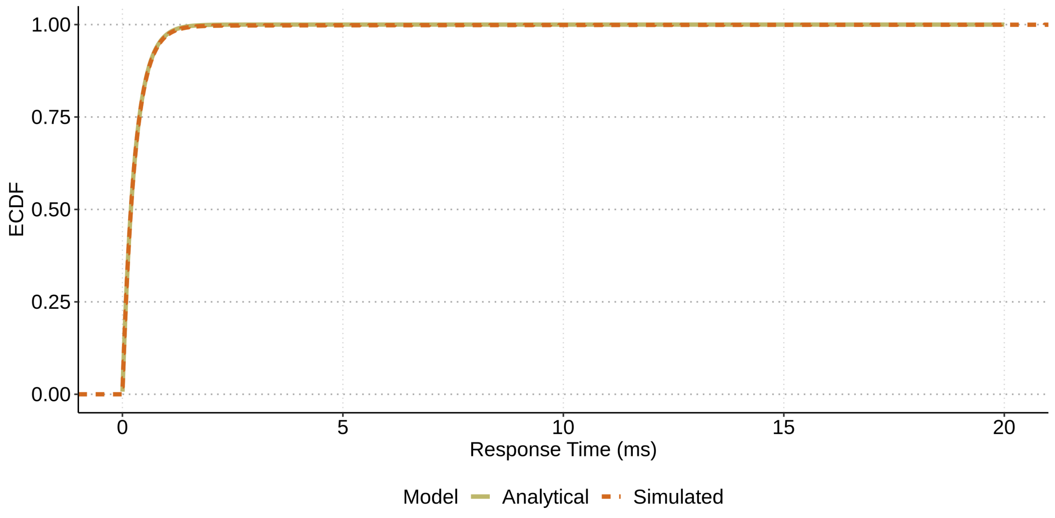


Figure 4.7: Response from analytical and simulated models.

To verify whether the simulated model results for COUA are accurate we use the two-sample KS test (MASSEY, 1951). This non-parametric Goodness-of-Fit test (FEITELSON, 2015), checks whether both samples come from populations with identical distributions. The test execution provides us with a p-value of 0.9904, which does not allow us to refute the null hypothesis. That test result allows us to say with 95% statistical confidence that the simulated model behaves according to its formal description, i.e., the analytical model.

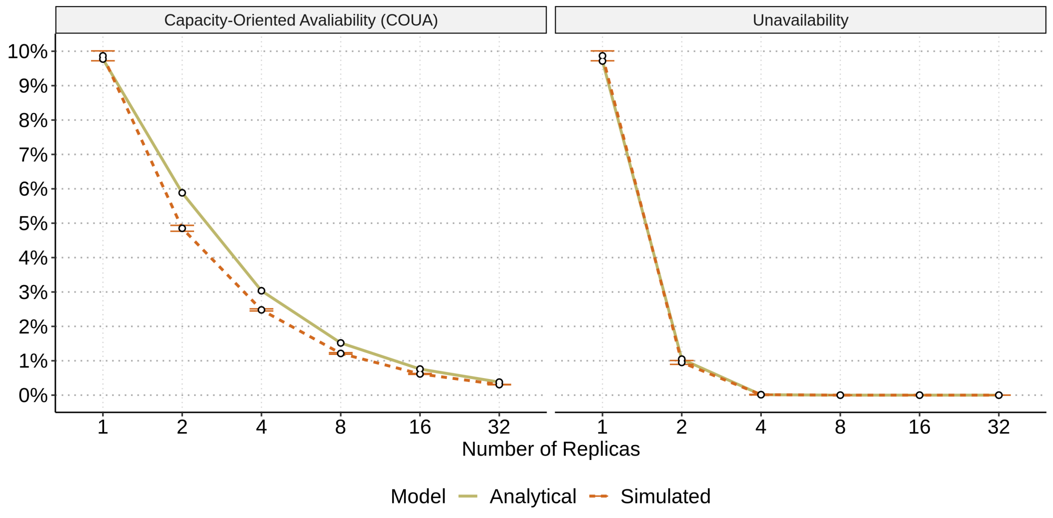


Figure 4.8: Performability metrics calculated by the analytical and simulated models.

4.5 Model Validation

We focus on the operational validation, which aims to determine whether the simulation model's output behavior has the accuracy required for the model's intended purpose over the domain of the model's intended applicability (SARGENT, 2005). We validate our simulated model by comparing its results with observations from measurement experiments. That is the most reliable and preferred way to validate a simulation model (JAIN, 1991).

4.5.1 Methodology and Experimental Setup

We vary two factors to perform the simulation validation: the BTC status – i.e., enabled and disabled – and the number of microservice replicas – i.e., 1, 2 and 4. The dependent variables considered are the response time and the unavailability. We calculate unavailability as the ratio of requests denied due to the BTC eviction. We replicated each treatment five times and compared the simulation results and the measurement experiments in the exact scenarios.

To carry out these measurement experiments we use the BTC implementation for the JVM's automatic garbage collector described in Section 5.1. We implement a toy stateful microservice whose requests allocate 256KB of memory, compute five thousand prime numbers[5], and update its state. We used Nginx as the load balancer and its default balancing algorithm, round-robin. The service was executed in a virtual machine with two cores and 1GB of RAM. The Nginx and the load generator ran in a separate virtual machine, configured with four cores and 2GB of RAM. The microservice runs on JVM version 10, which was configured to work in server mode, with a 512MB heap (50% of this heap is used to store the young generation) and to use the default garbage collector, i.e., the Garbage First Garbage Collector (G1GC) (DETLEFS et al., 2004). The state allocated by the service was fixed to 132MB, which is less than the heap space left to the tenured generation.

To narrow the causes of latency variance to the garbage collection, we chose to send the same request at a constant and low rate, which is 30 requests per second. Each run lasts around 10 minutes, which is enough to reach the steady-state after discarding the first 4 minutes of each test to minimize the effects of JVM warm-up (BLACKBURN et al., 2008). We restart all microservice replicas before each experiment run.

[5]https://github.com/gcinterceptor/java-experiments/tree/master/msgpush

Runs of a single-node measurement experiment generated the input of simulated experiments. Each run outputs a replay log with an entry for each request processed. The entry has a status code to indicate whether the request was served or denied and the response time, measured in the load balancer. For the simulation experiment, we did not use repeated logs. Thus, the simulation of a 4-replica microservice requires 4 measurement experiment executions, one for each simulated server.

4.5.2 Results

Figure 4.9 presents the response time ECDFs for each scenario considered in the simulator validation. Even though it provides us with an excellent visual hint of the simulation model validity regarding the response time, as ECDFs overlap in all cases, we would like to have statistical confidence about this similarity. We resort to the two-sample KS test (MASSEY, 1951) to check whether those results come from populations with identical distributions.

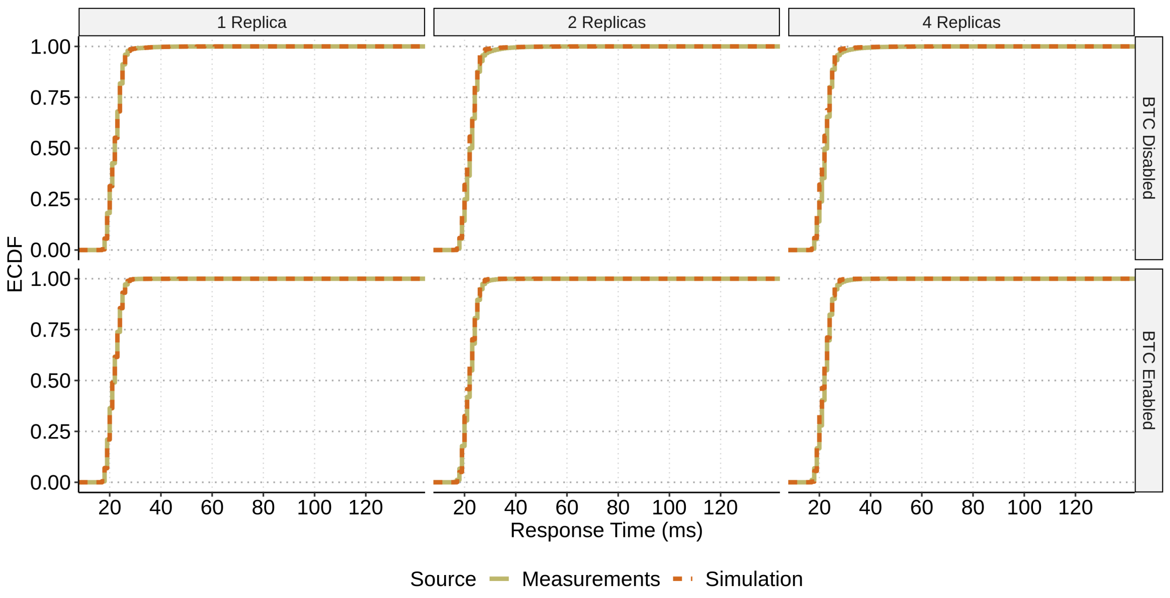


Figure 4.9: Comparison of simulation and measurement experimental results showing very close (overlapping) distributions. Each graph shows the ECDFs of the latency of the measurement (solid line) and the simulation experiments (dashed lines).

The challenge to perform the KS test using the obtained results is that the test is sensitive to large samples. As the number of requests processed during each experiment test is substantial, we applied an approach used in other studies to adapt the test to be used with large samples (CARVALHO; BRASILEIRO, 2012; JAVADI et al., 2011). We selected $1{,}000$ random samples of size 30 from each result dataset (i.e., from the simulation and measurement experiments). We run the KS test for each pair of samples and obtained the p-values.

Table 4.2 presents the average p-values rounded to the three most significant digits. As none of the p-values presented in Table 4.2 are small enough to reject the KS test's null hypothesis (e.g., 5%)[6], we conclude that the simulator model represents the response time accurately.

	1 Replica	2 Replicas	4 Replicas
BTC Disabled	0.453	0.388	0.329
BTC Enabled	0.439	0.394	0.295

Table 4.2: Result of KS tests comparing simulation and measurement experiments response times.

Table 4.3 compares the fraction of requests that failed due to BTC eviction, i.e., unavailability. All values are rounded to the three most significant digits. Like the response time, the simulated model shows good accuracy, deviating 0.03% in the worse case.

	1 Replica		2 Replicas		4 Replicas	
	Measurem.	Sim.	Measurem.	Sim.	Measurem.	Sim.
BTC Disabled	0.0	0.0	0.0	0.0	0.0	0.0
BTC Enabled	1.59%	1.56%	0.0111%	0.0	0.0	0.0

Table 4.3: Simulation and measurement experiments unavailability.

4.6 Sensitivity Assessment

Sensitivity Assessment (SA) investigates the relations between parameters and outputs of a simulation model. We follow Norton (NORTON, 2015) and call sensitivity assessment rather than analysis. The former is much more often done by looking at model run results than analyzing the model's equations.

Generally speaking, an "output" is a value computed by the model or any statistic extracted from it, such as peak or mean value. The assumption is that each parameter and

[6]From the original p-values, we observed that only 2% failed to reject the KS test null hypothesis

output can be described by a single number (NORTON, 2015). Thus, to perform the SA we rely on the input generator to generate the service log used as simulator input. The following subsections detail the methodology used in this SA and its results.

4.6.1 Methodology

As the analytical model formally describes the simulator behavior, we vary the model inputs, i.e., the response time, BT execution duration, BT execution probability, and the number of replicas. The dependent variables are the unavailability and COUA, explained in Section 4.2.1. The main question to answer with the SA is: when BTC is active, what is the effect of each factor variation on the observed metrics? Thus, we conducted the sensitivity assessment through a full factorial experiment design.

Intending to be more representative, we derive baseline-case parameters from the measurement experiment described in Section 5.3, which uses Hazelcast (JOHNS, 2015) to evaluate the impact of BTC on a replicated microservice's performance. The baseline case must follow the response time and garbage collection activity (i.e., target background task) measured in the experiments runs with BTC enabled. To do so, we pre-process the response time and GC logs to generate the rt_set and $bt\_duration\_set$. Furthermore, the BT execution frequency or probability is equal to 0.018%, which we call baseline BT frequency, for simplicity.

We selected the remainder levels as variations of the baseline case. For instance, the case "Baseline-50%" means that we subtract 50% from each sampled value. To implement those cases, we only need to change the pre-processing step to increment/decrement the baseline response time and BT duration sampled values based on the desired variation. For the BT frequency, we apply the variation to the parameter.

Below we summarize the SA parameters:

- **Number of replicas**: 1, 2, 4, 8, 16;
- **Response time**: Baseline-50%, Baseline, Baseline+50%;
- **BT Duration**: Baseline-50%, Baseline, Baseline+50%;
- **BT Frequency**: Baseline-50%, Baseline, Baseline+50%.

Each treatment simulates approximately 1.3M requests and is repeated ten times to increase the results' confidence.

4.6.2 Results

We started our analysis by running a factorial ANOVA (FARAWAY, 2002) and found out that all four factors, isolated and combined, impact both metrics, i.e., COUA and unavailability. The pre-condition assessment and the tests' detailed results can be found in Appendix A.

As we need to evaluate all possible scenarios, we break the discussion of the results of each metric into three parts: i) fixed the response time at the baseline level and varied the remainder factors, ii) fixed the BT duration at the baseline level and varied the remainder factors, iii) varied all factors at the same time. Regarding the following figures, each white dot is the result of a treatment, and the error bars represent intervals for a 95% of statistical confidence.

Unavailability

Figure 4.10 presents the probability of a system outage if we vary the number of replicas, BT duration, and frequency. In those treatments, we use the baseline response time. The first characteristic that caught our attention was that, regardless of the BT duration, changing the frequency of BT executions leads to a directly proportional impact on system unavailability. For instance, if we increase the frequency of BT executions in 50%, the chances of outages double on average.

A similar impact is observed when the duration of BT executions increase. For instance, if we pick the baseline frequency, increasing the BT duration in 50% increases the chances of system outage in $\approx 100\%$. Furthermore, those two factors, i.e., BT frequency and duration, have an additive impact on the system unavailability. Finally, the chances of a modeled system's outage decrease with the addition of more replicas. For the simulated service with two replicas, the worst-case unavailability was 0.15% and happened for a $+50\%$ BT duration and frequency.

As presented by Figure 4.11, the effect of the response time is inversely proportional to the unavailability. Fixing the BT duration and frequency to the baseline case, a shorter

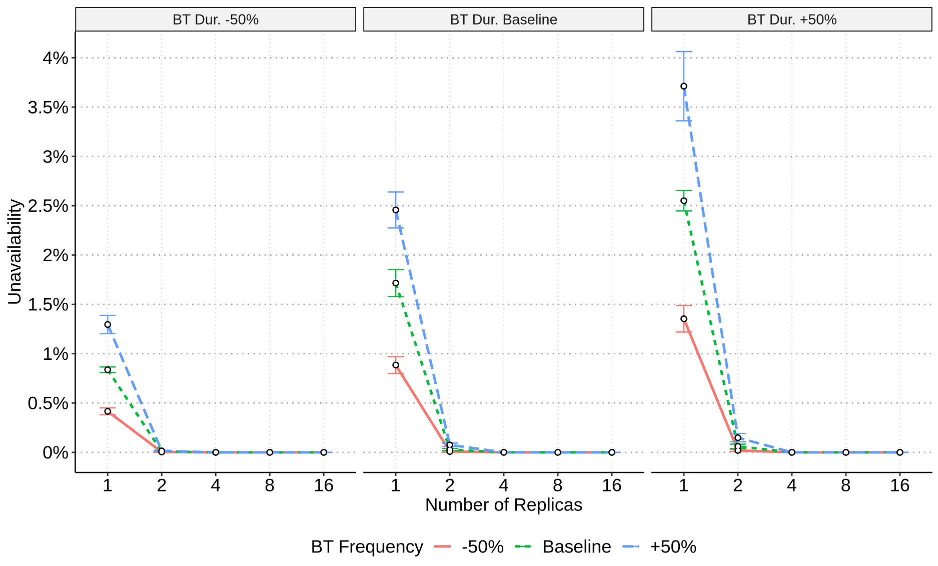


Figure 4.10: Effect of number of replicas, BT duration and frequency on the system unavailability.

response time leads to bigger unavailability. For instance, if we consider baseline BT frequency and one replica, the unavailability decreases by the same proportion of the response time increase, i.e., 50%. Regardless of the BT frequency and response time, the chances of evicting a request due to an outage decrease with the addition of more replicas.

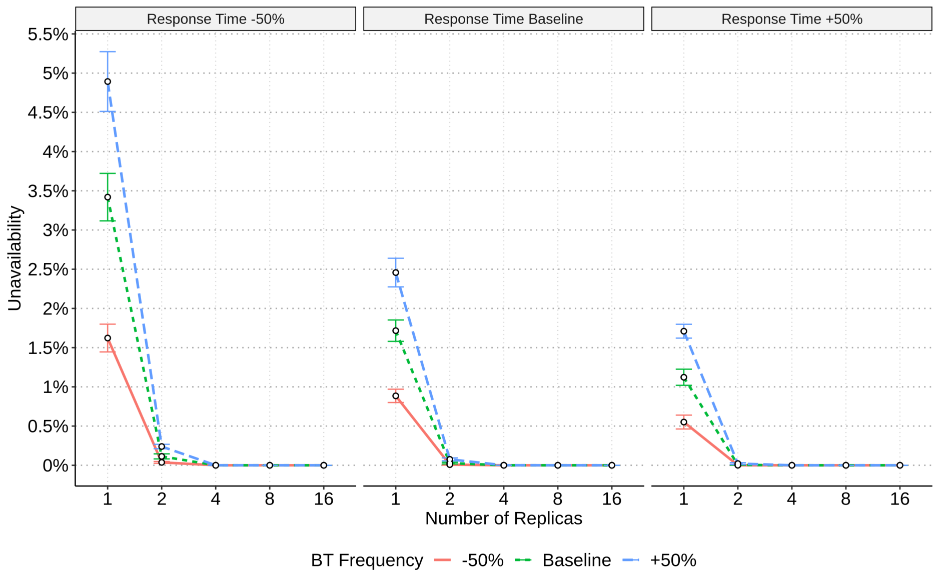


Figure 4.11: Effect of number of replicas, response time, and BT frequency on the system unavailability.

We can explain these results by revisiting the model and input generator operation. A shorter response time means more requests are processed, which increases the number of BT executions. That increases the chance of all replicas be unavailable at the same time. Furthermore, like the BT duration, the response also has an additive effect with BT frequency.

The proportional and additive effects allow us to understand the results of varying all four factors simultaneously. Figure 4.12 presents those results. For example, the worst throughput loss, i.e., $\approx 7.5\%$, happens for one replica's microservice with BT frequency and duration 50% bigger than the baseline and the response time 50% smaller. That is exactly the combination of the two worst-case scenarios we presented above, and the impact is additive. The rationale can be used to explain all the other cases. Regardless of the case, adding more processing replicas decreases the chances of system unavailability. Four replicas were enough to avoid denying requests in all considered scenarios.

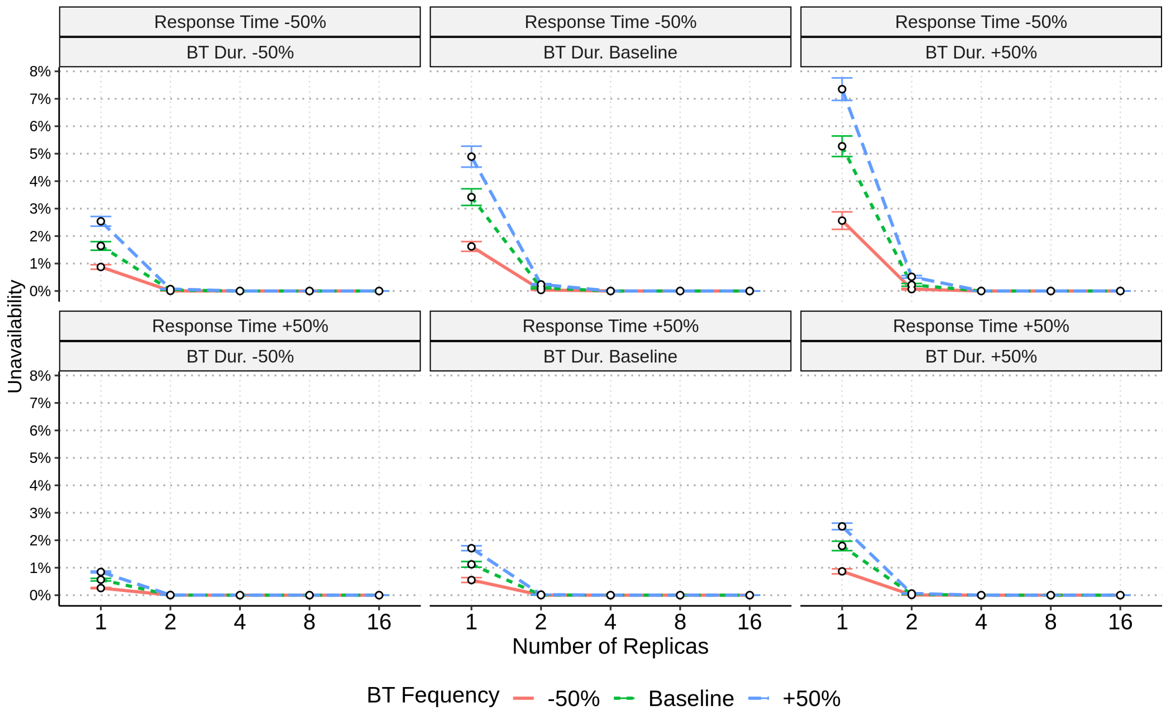


Figure 4.12: Effect of number of replicas, response time, and BT frequency and duration on the unavailability.

COUA

While the unavailability metric indicates how BTC impacts the probability of service denial, COUA tells us how BTC affects the average handling capacity. For the cases of microservices running with one replica, COUA and system unavailability are equal because the unavailability of that replica is effectively a service outage.

Figure 4.13 presents the results of the cases when the response time is fixed to the baseline case, and the remaining factors vary. One first thing to point is that the number of replicas impact is inversely proportional to COUA. Furthermore, the addictive effect also holds across all three factors. For example, if two microservices with statistically similar response times and two different background tasks whose duration and frequency differ in 50%, their average loss in capacity would differ by 100%.

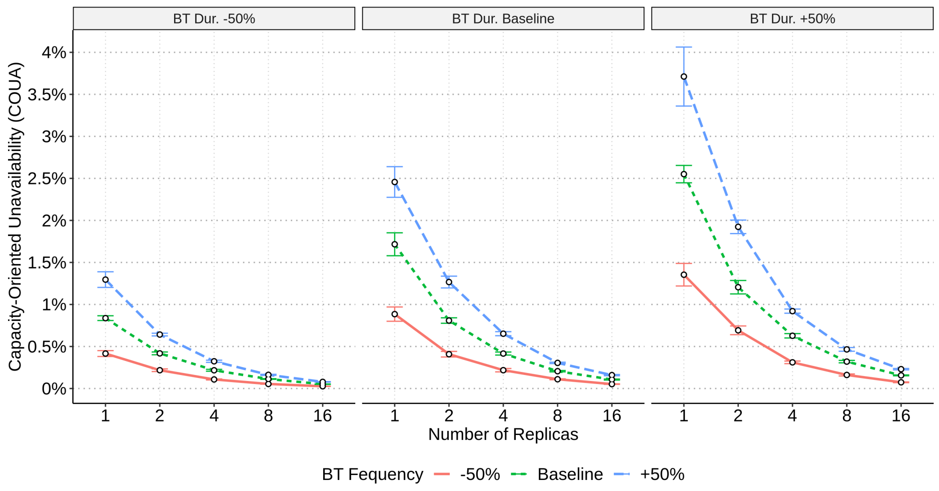


Figure 4.13: Effect of number of replicas, BT duration and frequency on COUA.

The worst result of 3.71% COUA happened for the modeled microservice deployed with one replica and with BT duration and frequency 50% bigger than the baseline. Considering the same BT duration and frequency, but 16 replicas, COUA decreases to 0.2% (i.e., 16x improvement). The average capacity loss reaches 0.079% if we consider 16 replicas, the smallest BT duration and frequency.

Figure 4.14 presents the results for the variation of the response time, and BT frequency when the BT duration is pinned to the baseline case. The variation of the response time has an inversely proportional effect on COUA. The reason for this effect is similar to the impact of the response time on the unavailability. For the same BT duration, a longer response time leads to fewer requests being attended, which leads to fewer chances of measuring task monitors, which translates into a smaller impact of BT executions. The additive effect also holds when we vary the response time.

Figure 4.15 presents the results of experiments varying all four factors simultaneously.

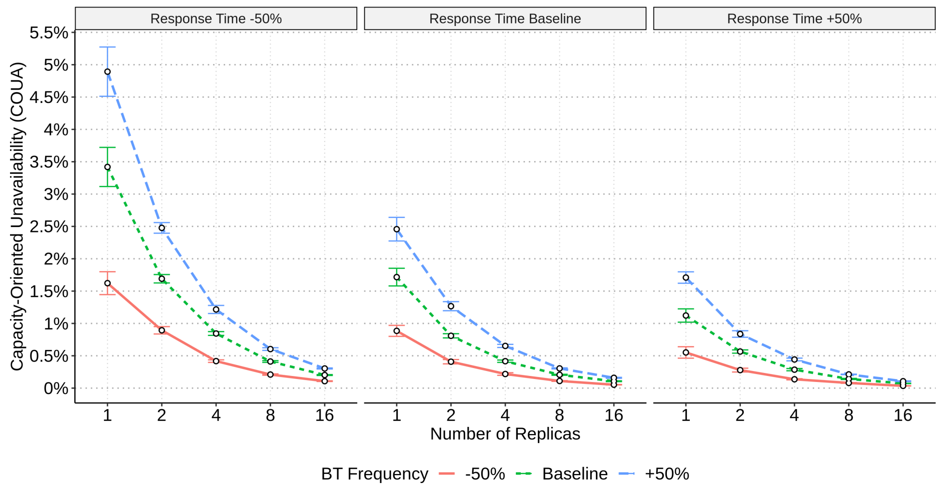


Figure 4.14: Effect of number of replicas, response time, and BT frequency on the COUA.

In the same way as the unavailability, the proportional and additive effects allow us to understand how COUA varies in the face of the variation of all four factors simultaneously. Again the worst case, i.e., $\approx 7.5\%$, happens for one replica's microservice with BT frequency and duration 50% bigger than the baseline and the response time 50% smaller. Finally, regardless of the case, adding more replicas leads to a decrease of COUA. That is a very desired cloud system characteristic and translates to the practice by allowing system operators to decrease the impact of BTC on capacity by scaling up the system horizontally.

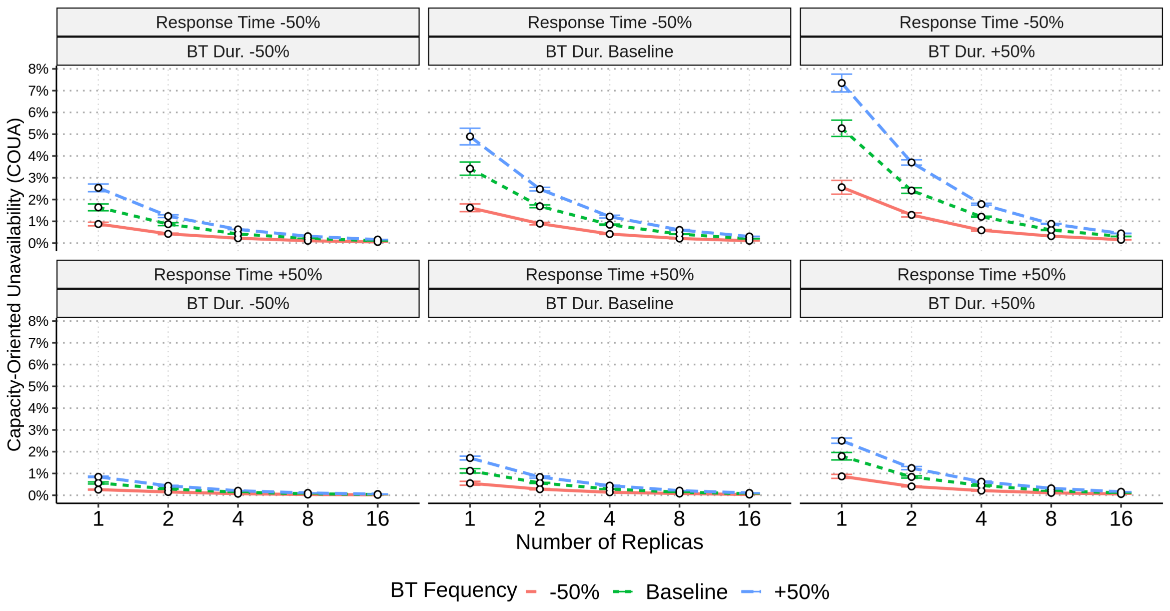


Figure 4.15: Effect of number of replicas, response time, and BT frequency on the COUA.

4.6.3 Summary

We performed a sensitivity assessment, and the results allowed us to understand better the impact of BTC on different cloud microservices with different characteristics. One important finding is that the chances of system unavailability and COUA decrease rapidly by adding replicas. Regardless of the case, the unavailability dropped to zero with four replicas. The BT duration and frequency are also critical factors. That is expected, as the BTC gains on performance, comes from exchanging the negative impact of those background tasks by a transient capacity decrease (i.e., replica unavailability).

4.7 Comparison with Other Tail-Tolerant Techniques

Dean and Barroso refer to systems that create a predictably responsive whole out of less predictable parts as tail-tolerant (DEAN; BARROSO, 2013). Those systems use the so-called tail-tolerant techniques to mitigate the impact of latency spikes on whole system performance to achieve that goal. Their work also shows that tail-tolerant techniques could take advantage of the fault-tolerant deployments, resulting in low additional cost. Furthermore, these techniques enable an increase in resource utilization without lengthening the latency tail.

This section compares BTC with two tail-tolerant techniques presented by Dean and Barroso (DEAN; BARROSO, 2013). The Request Duplication technique consists of sending each request twice and use the results from whichever replica responds first. Another tail-tolerant technique is the Hedge Requests, in which the client falls back on sending a secondary request after some brief delay. The paper suggests deferring sending a secondary request until the first request has been outstanding for more than the 95^{th}-percentile expected latency.

It is essential to note that the original proposal by Dean and Barroso mentions canceling the outstanding request as soon as the first arrives. We found that canceling a request is not practical and sometimes unfeasible. For instance, the HTTP/1.1 protocol (FIELDING et al., 1999), does not have support for that feature. Furthermore, the widely-used Envoy Proxy (KLEIN, 2017) supports request hedging but does not cancel late requests (Envoy Project Authors, 2021). As our goal is to compare with practical solutions, we simulate those techniques without canceling late requests.

4.7.1 Methodology

We modified the simulator by introducing the two tail-tolerant techniques, in addition to BTC: Request Hedging and Duplication. With those additions, we perform a full factorial simulated experiment to answer the following question: how does BTC compare to other practical tail-tolerant techniques on the response time and throughput?

As the BT duration and frequency impacts are similar, we choose only one of them to be a factor. Furthermore, as the simulated model restricts the sources of performance impact to the execution of background tasks, the case with zero BT executions provides us with the optimal case for the response time and throughput.

Similar to the sensitivity assessment, we extract the baseline-case parameters from the measurement experiment described in Section 5.3, which uses Hazelcast (JOHNS, 2015) to evaluate the impact of BTC on a replicated microservice's performance. The baseline BT execution frequency is the same observed on those experiments and is equal to 0.018% of the requests. The experiments' response time and GC logs were pre-processed to generate the input generator parameters rt_set and $bt\_duration\_set$.

The experiment factors are detailed below. The dependent variables are the average rate of requests successfully processed during the simulated experiment, i.e., average throughput and the response time.

- **Number of replicas**:1, 2, 4, 8, 16
- **BT execution frequency**: 0 (no executions), 0.18% (baseline), 1.8% (10x baseline) and 18% (100x baseline);
- **Method**: Default (without tail-tolerant technique), Request Hedge, Duplication and BTC;

We repeat each of the 80 treatments 5 times and sampled $50{,}000$ requests to perform the response time analysis.

4.7.2 Results

Regarding the response time presentation in this Section, Figures show the confidence intervals of distribution tail, i.e., 99^{th}, 99.9^{th} and, 99.99^{th} percentiles, for 95% of statistical confidence. The distribution is extremely biased towards the left, so the median and the 95^{th}-percentiles are equal (4.0,4.0), regardless of the case. The vertical axis scale is different to allow better visualization of each case. As the variation of the number of simulated replicas does not affect the response time, we do not present this factor.

Regarding the throughput presentation, the horizontal axis presents the number of replicas. Each point represents the average throughput of a treatment, and the error bars delimit the confidence interval around the median with 95% of statistical confidence.

Zero Background Task Executions (Optimal)

Figures 4.16 and 4.17 present results for the case when there is no BT execution. The response time percentiles presented by Figure 4.16 confirm that there is no need for tail-tolerant techniques when no BT executions occur as the response distributions are statistically similar.

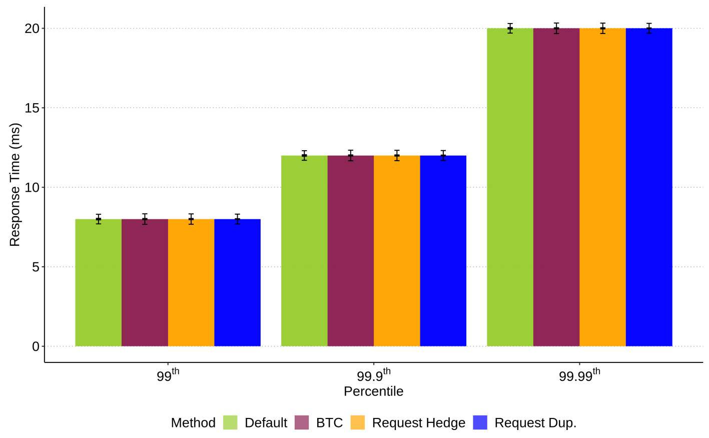


Figure 4.16: Response time when the background task executions do not affect requests (optimal case).

As presented by Figure 4.17, the throughput of the request hedging, default, and BTC techniques are the same when the number of replicas is one. As the number of replicas increases, the request hedging technique does not keep the throughput increase. For 16 replicas, the request hedging case's average throughput is 8% worse than the default and BTC cases.

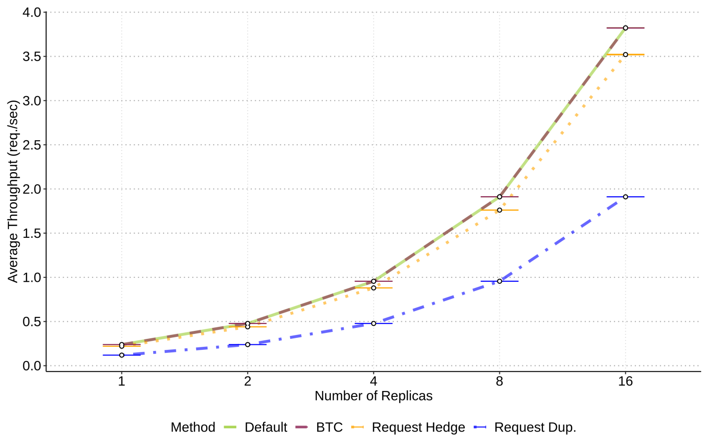


Figure 4.17: Throughput time when the background task executions do not affect requests (optimal case).

Baseline Background Task Executions (0.18%)

Figures 4.18 and 4.19 present the results when the BT impacts 0.18% of the requests. Even though the baseline BT frequency is low enough not to impact the 99^{th} and 99.9^{th} percentiles of the response time, it is enough to make the 99.99^{th} percentile $\approx 60\%$ bigger when no tail-tolerant technique is enabled. All three tail-tolerant techniques entirely avoid the BT impact on the baseline case's response time.

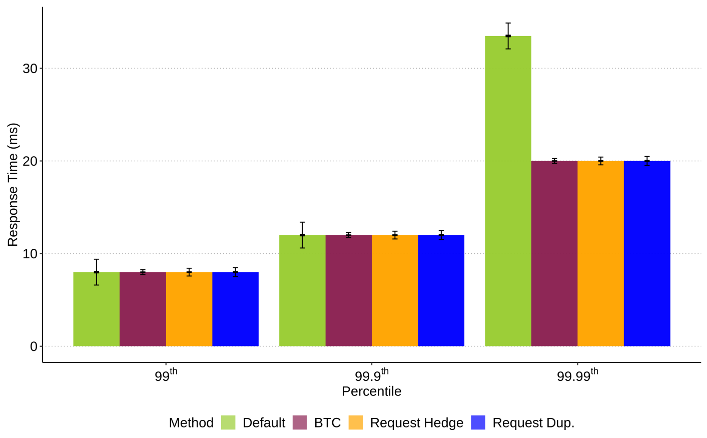


Figure 4.18: Response time for the baseline frequency of background task executions, i.e., affecting 0.18% of the requests.

As presented by Figure 4.19, the average throughput does not present any statistically significant change with the 0.18% increase in BT frequency.

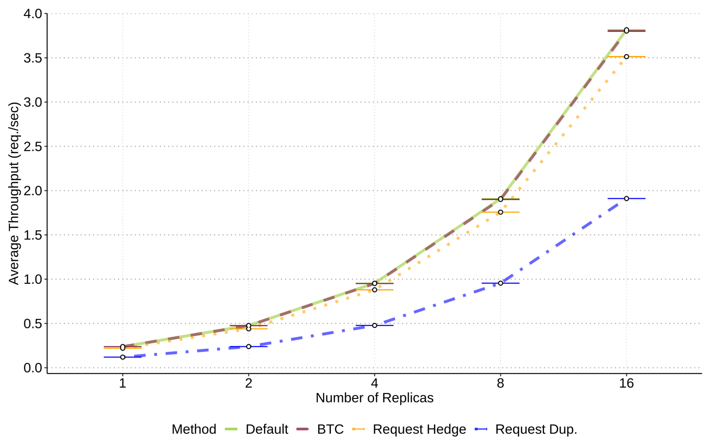


Figure 4.19: Throughput for the baseline frequency of background task executions, i.e., affecting 0.18% of the requests.

10x Baseline Background Task Executions (1.8%)

Figure 4.20 shows that when BT impacts 1.8% of the processed requests (10 times the baseline case), the response time increase starts at the 99.9^{th} percentile when no tail-tolerant technique is being used. Furthermore, the request hedging and duplication could not avoid that impact, and the performance drop by a factor of 2.33 and 2.18, respectively. When BTC is enabled, there is no statistically significant difference in response time between this and the optimal case.

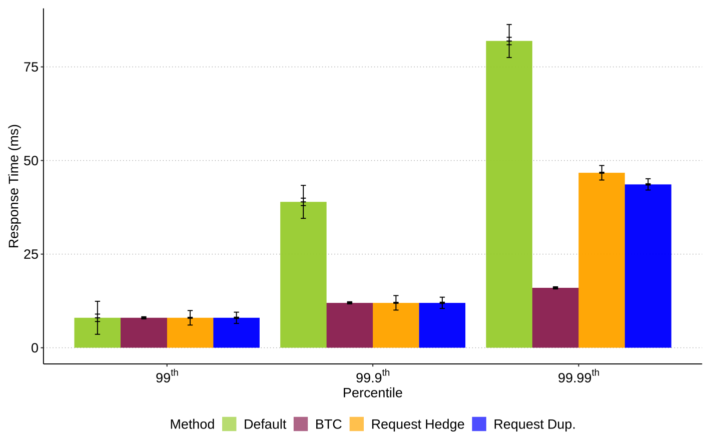


Figure 4.20: Response time for the 10x baseline frequency of background task executions, i.e., affecting 1.8% of the requests.

As presented by Figure 4.21, the 10x increase in BT frequency led to a slight decrease in the average throughput for all considered cases. The default case was the least impacted with a 1.3% throughput decrease. The BTC and request hedging techniques decrease was around

1.4%, making them 3.8% and 8.6% worse than the default case. The request duplication technique had the smallest change 1% and kept being the worst throughput.

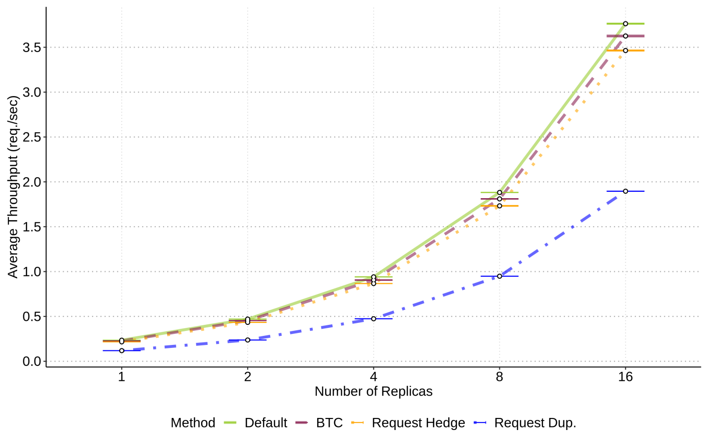


Figure 4.21: Throughput for the 10x baseline frequency of background task executions, i.e., affecting 1.8% of the requests.

100x Baseline Background Task Executions (18%)

Finally, Figures 4.22 and 4.23 present the response time and throughput when the background task executions affect 18% of the requests. In that scenario, the performance impact starts at the 99^{th} percentile. The request hedging and duplication started to perform worse than BTC at the 99.9^{th} percentile and hit a 4 times performance drop at the 99.99^{th} percentile. Once more, BTC was effective at eliminating the impact of BT executions on response time.

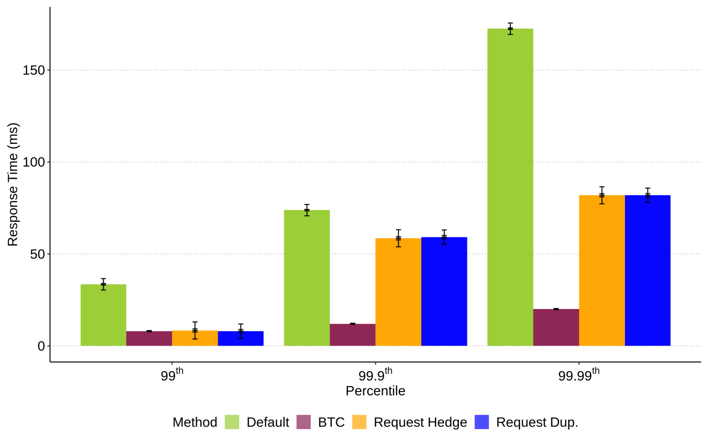


Figure 4.22: Response time for the 100x baseline frequency of background task executions, i.e., affecting 18% of the requests.

As presented by Figure 4.23, the BTC great results in response time came with a cost of 32.8% in throughput, which made it 26% worse than the best case, i.e., the default. The request hedging had a very similar average throughput to the default technique, being at most 0.1 requests per second slower. The request duplication was the worst technique, achieving 1.77 requests per second of average throughput for 16 replicas.

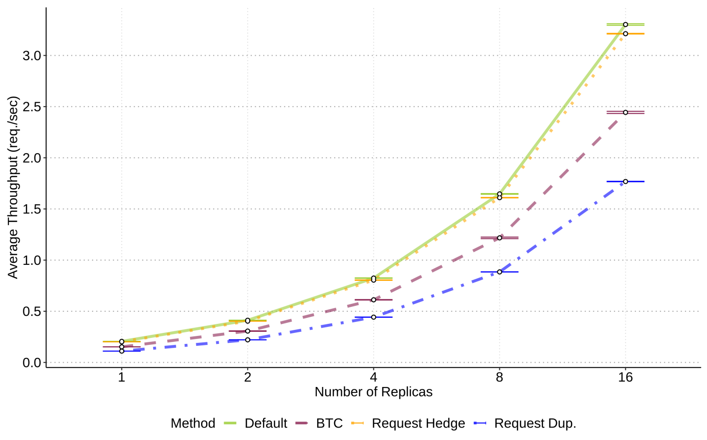


Figure 4.23: Throughput for the 100x baseline frequency of background task executions, i.e., affecting 18% of the requests.

4.7.3 Summary

This Section compared BTC with two practical tail-tolerant techniques, i.e., Request Hedge and Duplication. Regardless of the case, an increase in the BT frequency impacts the response time tail and the average throughput. Enabling BTC led to the best response time, and the best average throughput is achieved when no tail-tolerant technique is enabled. The request hedging and duplication response time tail were 4 times worse than BTC when the execution of background tasks hit 18% of the requests. That result came at the cost of 25% throughput decrease.

Chapter 5

Performance Evaluation of Representative Cloud Services using BTC

Even though the simulation model's verification and validation provide confidence in the results presented in Chapter 4, it still important to investigate whether BTC improves the performance of production cloud services. Therefore, this Chapter takes two widely used real-world services to run a series of measurement experiments aiming to verify whether the usage of BTC leads to performance improvements.

The evaluation presented in this Section focuses on controlling the automatic garbage collection of stateful JVM-based services running on a cloud environment, either standalone and in a replicated fashion. The rationale behind this choice is the importance and widespread adoption of distributed databases like Cassandra (HEWITT, 2016), Elasticsearc (GORMLEY; TONG, 2015), and distributed in-memory data stores like Hazelcast (JOHNS, 2015) and EVCache (MADAPPA et al., 2016). Furthermore, the state saved in memory leads to collections with a more significant performance impact (YU et al., 2016). Regarding the programming language, we chose Java because it is widely used to implement this type of service and for not allowing the automatic garbage collector to be disabled, which makes its control more difficult.

The remainder of this Chapter describes the Garbage Collector Control Interceptor (an implementation of BTC to control the JVM garbage collector), the experiments carried out, the results obtained, and discusses possible threats to validity.

5.1 Garbage Collector Control Interceptor (GCI)

We choose the automatic garbage collection as the background task to evaluate BTC. The reason for this choice is the wide adoption of managed languages in cloud deployments and the importance of the GC, attested by the bulk of work evaluating and proposing solutions for its performance impact (more on this on Chapter 6). This section describes an implementation of BTC called Garbage Collection Control Interceptor (GCI). GCI transparently controls replica garbage collectors' interventions and denies requests during these periods. Thus, resource competition or runtime pauses do not impact the replica's response time.

5.1.1 Operation

As described in Chapter 3, each microservice replica is associated with an independent instance of the BTC running on the same VM or container. Its architecture is based on the interceptor design pattern (DAIGNEAU, 2011), prevalent in cloud services. Being an input/output interceptor allows GCI to be transparently plugged into microservice replicas.

Figure 5.1 illustrates the architecture of GCI, which consists of two main components (highlighted in violet): i) the Proxy and ii) the GC Agent. The former implements all needed tasks that are not specific to GC: the controller, in-and-out notifications, the admission control, and all the logic that determines the right time to start the GC. The GC Agent implements the task monitor, which gathers information about the heap usage and the GC trigger. The remainder of this Section details all depicted components.

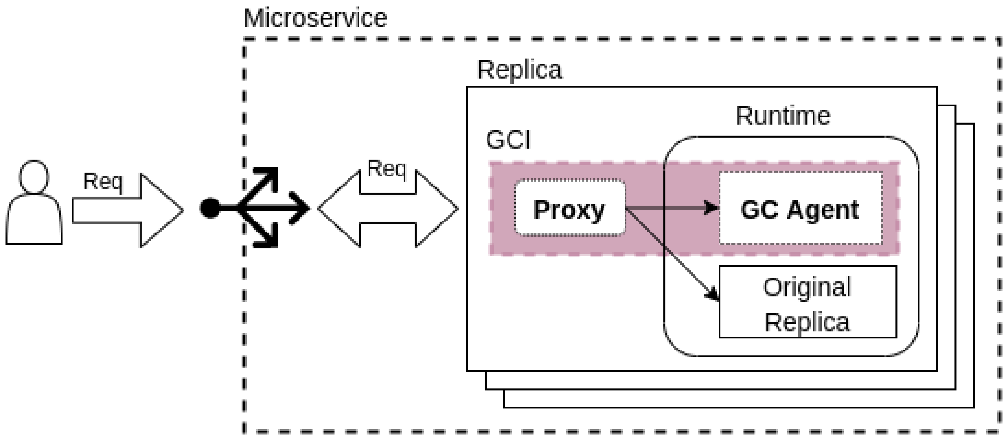


Figure 5.1: GCI components: the Proxy and the Agent.

The **Original Replica** is the microservice code, which handles the requests. The **GC Agent** is an Application Programming Interface (API) with two methods: i) returning the runtime heap usage and ii) triggering a garbage collection execution. That is the minimum needed by the Proxy to control any interactive background tasks.

The GC Agent runs on the same runtime of the original replica, and to make it as decoupled as possible from the Proxy, the access to the API occurs via the HTTP protocol. Furthermore, to decouple the Agent from the original microservice replica, it listens to a specific TCP/IP port, different from the one listened to by the original replica. We implement a version of the GC Agent for the JVM[1]. In that case, using GCI does not need microservice code changes. The JVM GC Agent's activation and the selection of the port it listens to are made through parameters passed to the command that initializes the Java runtime.

Besides the GC Agent, the usage of GCI requires an instance of the **GCI Proxy**[2] attached to each pair of original microservice replica and GC Agent (which run in the same runtime). The GCI Proxy is deployed as a sidecar (BURNS, 2018). Thus, the runtime and the GCI Proxy must execute in the same container or virtual machine. That allows the GCI Proxy to intercept all requests and responses from the microservice and is entirely transparent to the original replica. Furthermore, it is not coupled with a specific GC Agent implementation. The Proxy's only requirement is to call GC Agent's HTTP API, which could be implemented for any runtime.

5.1.2 GCI Request Processing Flow

Section 3 presented the request processing flow of a microservice replica controlled by BTC. Here we aim to describe that flow being specific to GCI and GC. In this context, the term request handling means sending the request to the Original Replica. When the GC needs to happen, the GCI Proxy denies the request, informing the load balancer about the replica's temporary unavailability (e.g., using the HTTP response status code 503). This response allows the load balancer to resend the request to another service replica.

Every replica controlled by GCI starts its activities in the available state. As soon as a request arrives, the GCI Proxy checks if the replica is available. If yes, the GCI Proxy

[1]Source code available https://github.com/gcinterceptor/gci-java
[2]Source code available https://github.com/gcinterceptor/gci-proxy

verifies its internal counters to determine whether it is time to call the GC Agent and check the heap usage (i.e., measure the task monitor). If the replica is not available, the GCI Proxy responds to the load balancer, letting it know about the replica's temporary unavailability. The Original Replica is entirely oblivious to request arrivals when marked unavailable by the GCI Proxy.

The next step is to determine whether to check the heap usage. The condition is based on the estimation described in Section 3.2 and aims to avoid the overhead of invoking the GC Agent at every request. If it is not time to check the heap usage, the GCI Proxy sends the request to the Original Replica for handling. Otherwise, it calls the GC Agent to retrieve the heap information and decides whether it is time to perform a garbage collection based on the heap usage. If it is still too early to reclaim heap space, the GCI Proxy triggers the request handling.

Finally, if the garbage collector's intervention is necessary, the GCI Proxy starts a concurrent flow while sending the Original Replica request for handling. The first action is to change the replica status to unavailable. After that, the Proxy waits to complete all pending requests and then calls the Agent to execute the garbage collection. When the background task finishes, the GCI Proxy updates the replica's status to available. It denies all requests received during the replica's unavailability period.

Example

To ease the understanding of the GCI's request handling flow, Figure 5.2 presents how Microservice A's replicas handle requests X and Y, which arrive concurrently. The load balancer sends each request to replicas 1 and 2, respectively. Following the service flow for request X, the GCI Proxy running on Replica 1 evaluates that it is time to check the heap usage and request this information. Based on the information received by the GC Agent, the Proxy calculates that it is not yet time to run the GC and forwards the request to the Original Replica. The request is processed, and the response is forwarded back to the client.

Request Y arrives at Replica 2 when it is unavailable. The Proxy denies the request, notifying the load balancer about Replica 2 temporary unavailability. The load balancer forwards the request Y to Replica 3. The Proxy running on Replica 3 evaluates that it is not the time to check heap usage and send the Original Replica request for service.

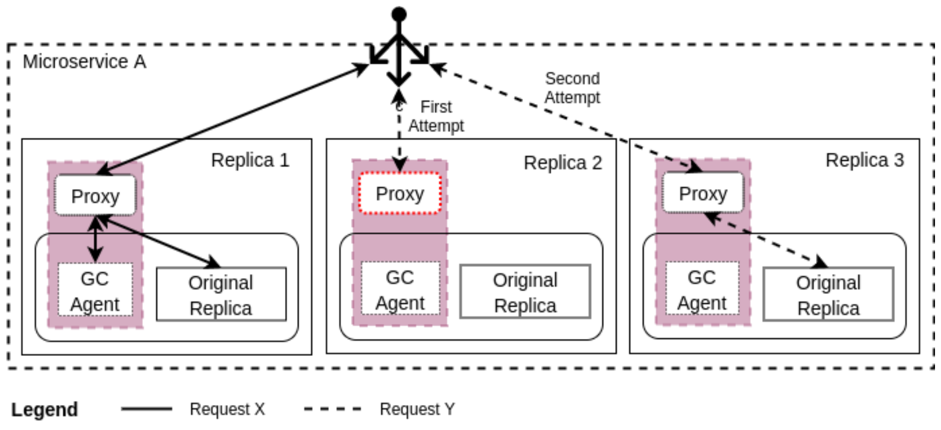


Figure 5.2: Example of GCI executing in a replicated microservice behind a load balancer.

5.1.3 Implementation Details

Avoiding automatic garbage collection Ideally, GCI should guarantee that no automatic garbage collection triggered by the runtime would occur. Languages like Python and Ruby have easy ways to disable automatic garbage collection entirely. However, for other languages, including Java, the same may not be possible. Hence, for such languages, we need to make GC interventions the least frequently possible. We accomplish this goal by configuring the runtime with the maximum allowed heap and young memory pool size.

GCI for other runtimes As the GC Agent API must run within the target runtime, its code is language-specific. To demonstrate the generality of this model, as well as enabling the evaluation of other runtimes, we also implement GC Agents for Ruby[3], Node.js[4], and Go[5] runtimes.

Resending denied requests to avoid user impact due to replica unavailability, the requests denied by the GCI Proxy must be immediately resent to another server. In practice, this is done transparently by the client APIs or load balancers. For instance, the widely used load balancer NGINX (REESE, 2008) has the `proxy_next_upstream` directive (NGINX, 2020), allowing users to specify in which cases a request should be forwarded to the next server.

[3]Source code available https://github.com/gcinterceptor/gci-ruby
[4]Source code available https://github.com/gcinterceptor/msgpush-nodejs/blob/master/gcinterceptor.js
[5]Source code available https://github.com/gcinterceptor/gci-go

5.2 Performance Evaluation of a Single Replica

The results presented by Fireman *et al.* (FIREMAN; LOPES; BRUNET, 2017) led to a collaboration with the Linx Impulse project, former Chaordic[6]. This collaboration's main goal was to increase the confidence in BTC efficiency by carrying out measurement experiments in a scenario as close as possible to their production environment. To this end, we used production virtual machines, data, and load to evaluate Elasticsearch (GORMLEY; TONG, 2015), a distributed search engine used by Linx Impulse and by other significant cloud services, such as Netflix and eBay (ELASTIC, 2014).

During our evaluation, we developed ESPerf – an open-source tool[7], capable of parsing the Elasticsearch slow logs (ELASTIC, 2017b) and replaying the workload, respecting the load content and shape recorded during log capture. Furthermore, ESPerf controls the back-off of requests during the unavailability period and the collect metrics used in this evaluation.

5.2.1 Methodology

Our goal with this set of experiments is to evaluate the impact of BTC on one replica of the Linx Impulse backend performance, i.e., Elasticsearch. More specifically, we would like to evaluate the service time and the throughput. Service time refers to the elapsed time to execute the query, including any queue waiting time. We define the replica throughput as the rate of queries successfully processed per unit of time. The assessment aims to answer the following research questions:

1. Does BTC impact the microservice response time?
2. Does BTC impact the microservice throughput?
3. Does the workload influence on BTC's impact?

To do so, we carry out two sets of experiments. The first set varies the load intensity and BTC presence in a 2^2 factorial evaluation. The evaluation considers cases with synthetic low (35 requests/second) and high (150 requests/second) loads and BTC enabled and disabled. The second set of experiments uses production load and varies only the BTC usage. We repeat each experiment 5 times, measuring the response time and throughput. The GCI Proxy and the runtime are restarted before each test.

[6]More at https://www.linx.com.br/transformacao-digital/linx-impulse/

[7]Available at https://github.com/danielfireman/esperf

5.2.2 Experimental Setup

Common setup The same data was loaded for all experiment runs. Thus, regardless of the experiment, a query would lead to the same resource usage and same results. The data came from three different production indexes (databases), summing 12GB. Each index is a fragment collected and anonymized and represents information about products. Query results on indexes like these are used in recommendations provided by Linx Impulse. These queries use product attributes, for example, category and brand.

We executed all experiments on Amazon Web Services (AWS)' Elastic Cloud Computing (EC2) platform. More specifically, on a virtual machine of type c3.2xlarge (8 vCPUs and 15GB RAM), which is very similar to the instances that are part of the Linx Impulse production environment. The service instance runs Ubuntu Linux 14.04 with core version $3.13.0 - 115$, Java version 8 (Oracle Hotspot 64 bits, build 25), Elasticsearch version 2.4. Furthermore, we execute the GCI Proxy and Agent to evaluate the impact of BTC.

Regardless of the status of BTC, the available memory size has been set to 10 GB (-Xms 10 g -Xmx 10 g). For the case where GCI is enabled, the Proxy and the JVM are configured to perform collections when the heap usage is 50%. The rest of the configuration followed the standard recommendations for running Elasticsearch in a production environment (ELASTIC, 2017a).

Experiments with synthetic load The evaluation of the impact of different load intensities on the service time and throughput was done by disabling cache and repeatedly submitting the same request at fixed low (35 reqs/sec) and high rates (150 reqs/sec). The request being submitted was randomly selected from the production workload after discarding queries that returned zero items. We chose the request rates for low and high rates based on the average CPU utilization of 20% and 70%, respectively.

Each run lasts around 10 minutes. That duration is enough for the microservice to reach the steady state and discard the first 4 minutes of each test to minimize the effects of JVM warm-up (BLACKBURN et al., 2008). More than $60{,}000$ and $200{,}000$ requests were processed across all five repetitions of the low and high load scenarios, respectively.

Experiments with production load We also evaluated how BTC performs when subjected to a production workload. This scenario differs from the previous one in two ways: i) the inter-arrival times vary throughout the experiment execution, and ii) the requests are not equal – thus, the resources used to process each request vary. This evaluation uses the ESPerf tool[8] to replay logs collected from 5 different Elasticsearch instances that ran in Linux Impulse production environment (ELASTIC, 2017b). ESPerf dispatches the requests extracted from the logs respecting their content and shape, i.e., query URL, body, and interarrival time.

Since collecting the request history negatively impacts Elasticsearch performance, we only recorded nineteen minutes of history per instance. We discarded the first four minutes of results from each experiment to minimize the effects of warming up the execution environment (BLACKBURN et al., 2008), which led to a total of 237,601 requests considered by our analysis.

5.2.3 Results

We observed that BTC successfully shortened the mean, median, and tail of the response time distribution for all the experiments carried out. Enabling BTC also improved the throughput of the production workload replay experiments. For the constant rate experiments, the gain in response time came at the expense of some throughput loss.

In the following subsections, we answer the research questions by detailing the impact of BTC on each dependent variable. Furthermore, we perform a thorough evaluation of the GC behavior during the experiments' execution.

Impact of BTC on Response Time

Figure 5.3 presents the service time's ECDF for experiments with synthetic low load. The vertical lines highlight some important statistics regarding BTC in disabled and enabled states. The distances between respective dashed and continuous lines denote the impact of BTC. The results show that enabling BTC improves the response time percentiles above 99^{th}. The 99^{th}-percentile goes down from 35ms to 8ms, an improvement of $\approx 77\%$. The 99.9^{th}-percentile fell off to 33ms, which is less than the 99^{th}-percentile when BTC is disabled.

[8]Tool developed by the author. Source code available at https://github.com/danielfireman/esperf

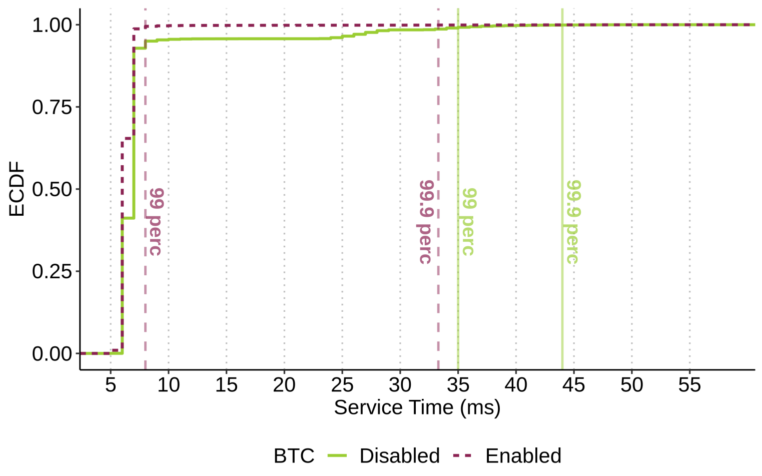


Figure 5.3: Response time's ECDF of an Elasticsearch replica exposed to synthetic low load.

Figure 5.4 shows that the substantial improvement remains even with a high load. Furthermore, as the impact of the GC increases with the load when BTC is disabled, it affects the center of the distribution: i.e., the average and median. BTC successfully avoids that impact, keeping the same median observed for the experiments with low load, i.e., 6ms. So, besides improving the response time percentiles above the 99^{th} (i.e., tail), enabling GCI enhances the average and the median when the load is high.

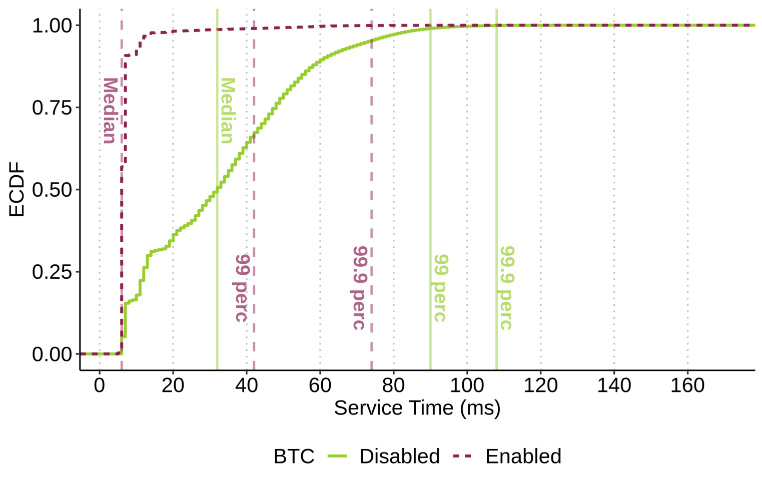


Figure 5.4: Response time's ECDF of an Elasticsearch replica exposed to synthetic high load.

Table 5.1 presents the confidence intervals for the main response time statistics. There is no intersection between intervals when BT is disabled and enabled for all statistics. Thus, with high statistical confidence, we can state that enabling BTC improved the response time for high and low load cases. For low load experiments, the improvement reaches $\approx 74\%$ at the 99^{th}-percentile. Regarding experiments with high load, the improvement approaches $\approx 80\%$ on the center of the distribution and $\approx 51\%$ at the 99^{th}-percentile. Those performance gains lead to less response time variability and, thus, to a more predictable response time profile.

	Low Load (ms)		High Load (ms)	
Response Time	GCI Disabled	GCI Enabled	GCI Disabled	GCI Enabled
Median	[7.00, 7.00]	[6.00, 6.00]	[32.00, 32.00]	[6.00, 6.00]
Average	[7.58, 7.66]	[6.40, 6.42]	[32.90, 33.00]	[7.45, 7.50]
99^{th} percentile	[34.00, 35.00]	[8.00, 8.00]	[90.00, 90.00]	[41.00, 44.00]
99.9^{th} percentile	[43.00, 45.00]	[31.00, 40.00]	[108.00, 109.00]	[72.00, 76.00]

Table 5.1: Synthetic and constant workload: median, mean and long tail confidence intervals. Ranges were calculated for 95% of statistical confidence using bootstrap resampling configured to generate 1000 samples (EFRON, 1982).

After evaluating the BTC response time under high and low load, we evaluate its performance on a production setup. Through a fragment of the ECDF, Figure 5.5 shows the tail of service time for experiments with production load, starting from the 99^{th}-percentile. As we can see, enabling BTC improves the center and the tail of the service time distribution. Those improvements translate into a better quality of service through faster and more predictable request processing.

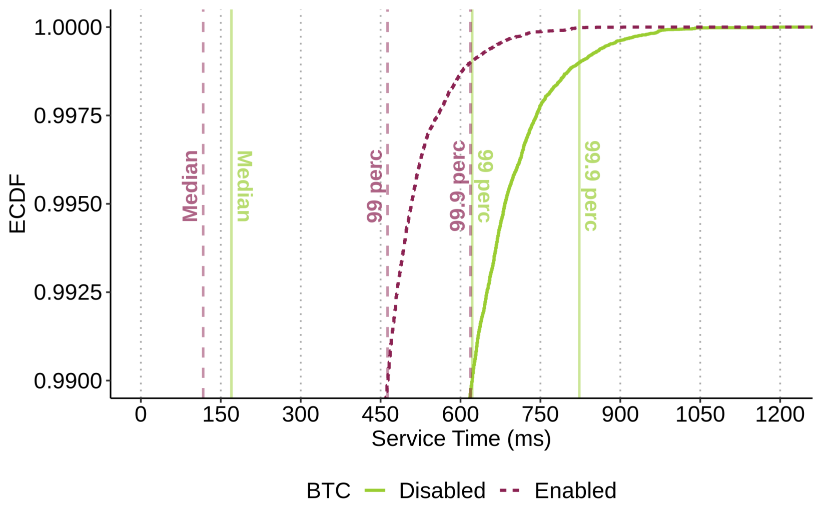


Figure 5.5: Service time ECDF above the 99^{th}-percentile for experiments with production environment, data and load.

Table 5.2 allows us to see more details and add statistical confidence about improvements in service time observed when BTC is enabled. We calculate the intervals according to the method proposed by Nyblom (NYBLOM, 1992) and represent 95% of statistical confidence. It shows that enabling BTC improves the overall distribution and makes the tail $\approx 18\%$ closer to the median.

	BTC Improvement (%)
Average	[26.3, 26.3]
Median	[30.8, 31.0]
99^{th}-perc.	[25.4, 25.4]
99.9^{th}-perc.	[24.7, 24.8]

Table 5.2: Service time statistics for experiments with production load.

Conclusion The presented results allow us to answer the first and third research questions presented in Section 5.2.1 with high statistical confidence: *Even though the benefits of using BTC differ with the load pattern, enabling BTC improved the Elasticsearch replica response time for all cases.*

Impact of BTC on throughput

Figure 5.6 shows service throughput for BTC enabled and disabled across synthetic low and high loads. The mean throughput loss was $[7.48\%, 8.89\%]$ when the load is low and $[17.6\%, 19.1\%]$ when the load is high - both intervals calculated with 95% of statistical confidence.

Although the instance throughput loss is more significant for the high utilization scenario, it does not increase proportionally to the workload. While the query rate increases around 4.2 times from the low to the high load levels, the throughput loss is 2.25 times greater when moving from the low to the high load levels.

The reason for this difference is that the higher the load, the higher the number of requests evicted by BTC during the unavailability period. As we measured the impact of BTC considering one replica, denied requests do not account for throughput calculation. Nonetheless, it is essential to point out that a load balancer would have transparently routed those requests to another service instance in a typical cloud production environment.

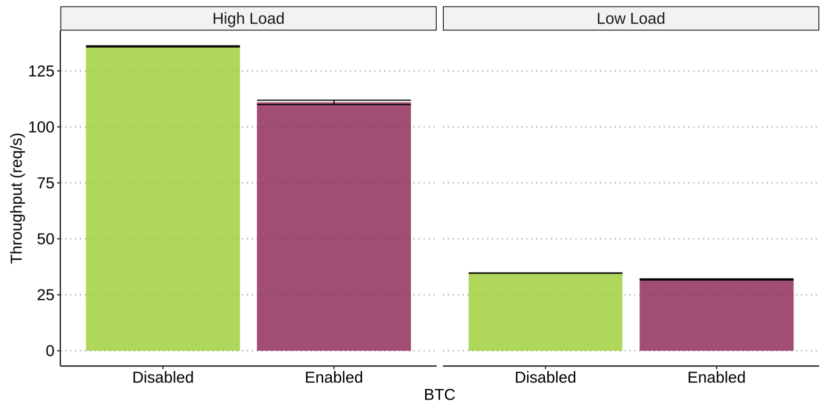


Figure 5.6: Replica throughput comparison for constant load experiments. Error bars indicate a 95% confidence interval computed using Bootstrap resampling (EFRON, 1982) configured to generate 1000 samples.

Looking at results from experiments with production load, when BTC is disabled, the CPU utilization average and standard deviation were 72.32% and 14.26%, respectively, meaning very few (or no) cores available most of the time. When the load is that high, the competition for CPU between request processing and the concurrent GC leads to the increase in response time presented in the last Section. Furthermore, Elasticsearch has a mechanism to cap the number of requests that can be processed concurrently. That limit is widespread in production deployment to avoid instance resource saturation.

When the performance drop combines with a maximum value in the number of requests that can be processed concurrently, the experiment takes more time to finish. That increase in the experiment duration leads to an average throughput decrease. So, enabling BTC and avoiding the concurrent GC execution led to the $\approx 8\%$ difference in throughput presented in Figure 5.7. The error bars present the confidence interval with 95% statistical confidence. The average throughput and CPU utilization differences are similar. The latter dropped $[8.27\%, 9.19\%]$.

Those results confirm that enabling BTC avoids the CPU competition between GC and request handling, which decrease the average CPU usage. Finally, the improvement brought by enabling BTC came with the cost of denying some requests. More precisely, 7% of requests would have been transparently routed to other service instances in a production cloud environment.

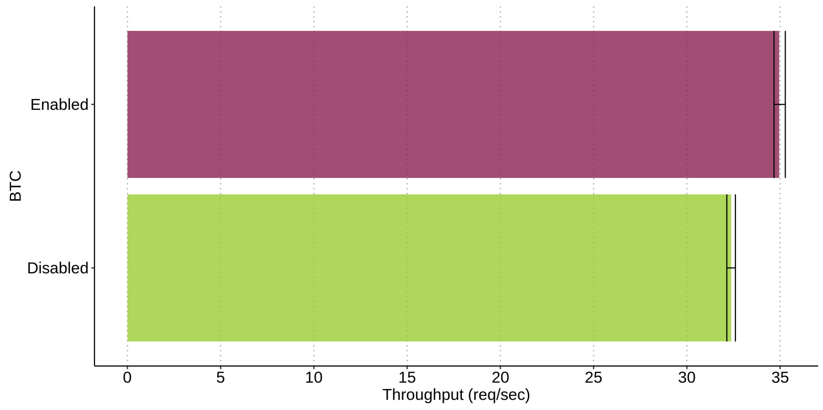


Figure 5.7: Replica throughput comparison for production replay experiments. Error bars indicate a 95% confidence interval for each metric computed using Bootstrap resampling (EFRON, 1982) configured to generate 1000 samples.

Conclusion The presented results allow us to answer the second and third research questions presented in Section 5.2.1 with high statistical confidence. The impact of BTC on throughput depends on the load and the service replica configurations. For experiments with the constant synthetic workload, Elasticsearch processed the queries quickly enough not to reach the parallelism limit. In those cases, enabling BTC leads to a throughput drop, which is less than 20% and is not proportional to the load.

The experiments with production workload had a much broader response time distribution and demanded higher concurrency, which exceeded the Elasticsearch limit. In that case, enabling BTC improved the response time and decreased the number of requests that need to be processed concurrently. The average rate of request processing improved by 2% at the cost of evicting 7% of the total number of requests.

GC Behavior

Besides answering the research questions, we also analyze how the target background task, i.e., the JVM GC, behaves during our experiments. It is essential to understand a bit of this task before we dive into the analysis. One important definition is that the JVM memory management is generational. A generational memory system divides the heap into partitions called generations. Its efficiency is based on the observation that most of the objects are short-lived, called the generational hypothesis. As these objects accumulate, a low memory condition occurs, forcing GC to take place (ORACLE, 2010).

The JVM heap space has two primary partitions, i.e., the young and the tenured generations. The young generation uses a fast copying garbage collector. Collecting the young generation is called minor collection. Objects that survive multiple young space collections are tenured, meaning they are copied to the tenured generation. The tenured generation is more extensive and fills up less quickly. So, it is garbage-collected less frequently; and each collection takes longer than a young space collection. Collecting the tenured space is also referred to as doing a full collection (ORACLE, 2010).

The JVM GC logs provide much information, including duration, collection type, and the amount of heap cleaned. Let us use the JVM log information to understand the GC behavior, its impact, and how good was the BTC implementation - i.e., GCI - on controlling it.

Minor collections Figure 5.8 shows boxplots of the duration of minor collections for all experiments. The setup enabled a concurrent collector, following the recommended configuration (ELASTIC, 2017a). This class of GCs is designed for shorter pauses at the expense of sharing processor resources with the GC while the application is running.

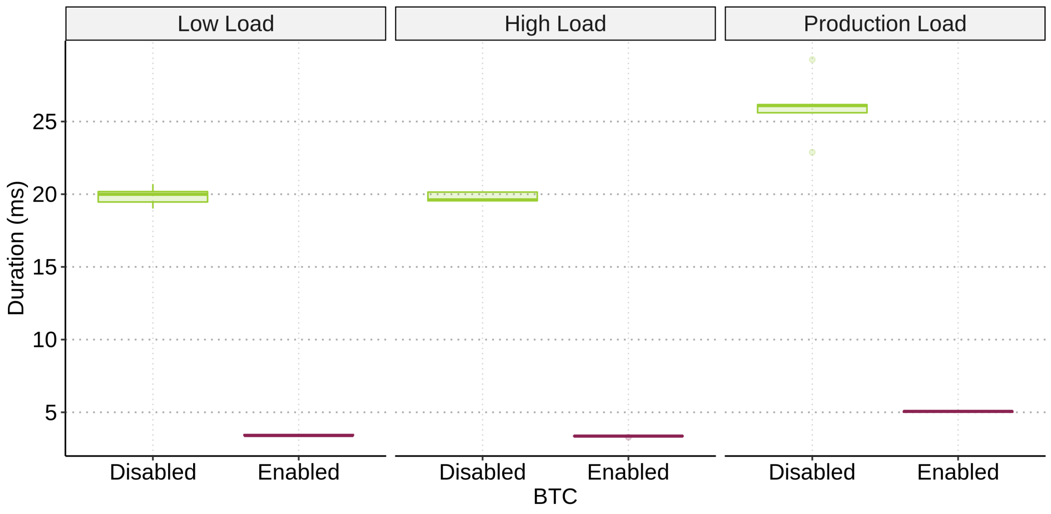


Figure 5.8: Duration of minor collections for experiments with synthetic load

For experiments with synthetic load, the increase did not impact the duration of the collections. The rationale for it is twofold: i) the requests did not change during the experiment, and ii) the service resources did not saturate. That combination of factors allows the GC to execute with a very low-performance variation. For experiments replaying production load, we have different requests with different heap usage patterns.

Table 5.3 provides more data. Regardless the type of load, enabling BTC decreased the average collection duration by $\approx 80\%$. Furthermore, enabling BTC also decreased the number of collections. This happened because we configured the BTC activation interval to collect less often. The experiments with synthetic load experienced a decrease of 86% and 90%, for the low and high load levels, respectively. The number of collections decreased 89% when the production load was replayed.

	Synt. Low		Synt. High		Production	
	BTC Off	BTC On	BTC Off	BTC On	BTC Off	BTC On
Average Dur. (ms)	19.9	3.43	19.8	3.35	26	5.06
Count	554	76.6	2750	270	2820	299
Exp. Dur. Impact (%)	3.04	0.0736	12.7	0.238	4.98	0.12

Table 5.3: Minor GC detailed metrics.

Assuming a very small or no idleness period during these experiments, the impact of these two metrics could be combined in the fraction of the experiment duration impacted by GC executions (column "Exp. Dur. Impact (%)"). Again, regardless of the load, enabling BTC decreased the garbage collection impact by a factor of ≈ 40x. That metric could also be used to explain the performance hurt caused by adding more load. Considering experiments with the synthetic load when BTC is off, even though the average duration did not change, the impact of minor GCs grew by a factor of 4.17, almost proportionally to the 4.28x load increase.

Finally, this analysis also exposed that GCI operation was not ideal. That is because there were still some minor collections executed automatically by the JVM when BTC was enabled, and those are concurrent collections. Fortunately, that affected less than 1% of the experiment duration. Those experiments' problem is that we did not know the system enough and misconfigured the activation interval. Unfortunately, the partnership with Linx Impulse was already over by the end of the analysis, and we did not have the opportunity to re-execute experiments.

Full Collections Figure 5.9 presents the the duration and count of full collections. The figure has only data for cases when BTC is enabled because that kind of collection has not been automatically triggered by the JVM in our experiments. That is aligned with the generational hypothesis described at the beginning of this Section, so only young collections happen when BTC is not enabled. Even though the experiment's median duration with production load appears smaller than the other cases, the difference is not statistically significant because all median intervals intersect for 95% statistical confidence.

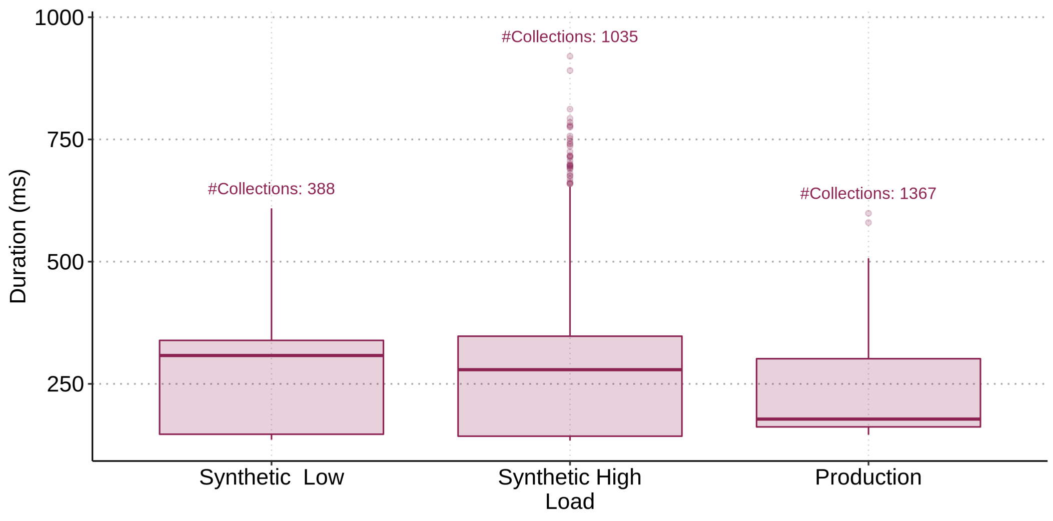


Figure 5.9: Duration of full (or major) collections, which only occurred in experiments with BTC enabled.

Those results show that enabling BTC leads to longer full collections, when compared to the minor collections that happen when BTC is not enabled. To explain this behavior, we need to dive into the GCI implementation and the API's mechanics exposed by the JVM to trigger collections. When BTC is enabled, the GCI Proxy control collections through the ForceGarbageCollection function, exposed by the JVM Tool Interface (JVMTI) call (ORACLE, 2007), which only allows to synchronously force the execution of full collections.

Even with the limitation imposed by the JVM of only invoking full collections, enabling BTC improves the response time. That is because those BTC manages the background task execution and denies requests while that execution is happening. Thus, improving the runtime to allow triggering young collections could improve the replica unavailability, which leads to fewer requests being redirected.

5.2.4 Summary

The first set of measurement experiments focused on being as close as possible to a production environment, using Linx Impulse's production virtual machines, data, and workload to evaluate a standalone Elasticsearch replica (GORMLEY; TONG, 2015). The results provide us with high statistical confidence to state that that enabling BTC improves 99.99^{th}-percentile by $\approx 20\%$, the mean by $\approx 26\%$ and median by $\approx 30\%$. Even though those service time improvements lead to better service quality, they come with a throughput drop: less than 20% in the worst case.

5.3 Performance Evaluation of a Replicated Service

The results obtained with the evaluation of a single Elasticsearch replica in a production environment led to further questions, mostly related to the setup we had access to at that moment. As we performed the evaluation using a single instance of Elasticsearch, the first question is about a replicated microservice's performance improvements. Even though Section 4.5 already explores a similar scenario with a toy service and constant load, now we are interested in real-world service and production-like load.

Another critical point is that we only replayed Elasticsearch search requests on the previous evaluation. We would like to understand whether the benefits of enabling BTC still apply if different endpoints – with different heap usage patterns – are invoked concurrently. For instance, would the performance still improve if the microservice is exposed to reads and writes?

Lastly, our production-load experiments were misconfigured (more about this at Section 5.2.3), and we would like to be confident about the performance benefits of using BTC when properly configured. To do so, we need to compare the results with the optimal case. In other words: considering a replicated environment and a diverse load, does BTC eliminates the negative performance imposed by the automatic GC?

5.3.1 Methodology

We want to evaluate the impact of BTC on a replicated microservice's performance. More specifically, we would like to answer the following research questions:

1. Does BTC improve the performance of a replicated microservice exposed to a production-like load that exercises one endpoint?

2. Does BTC improves the performance of a replicated microservice exposed to a production-like load that exercises multiple endpoints?

3. How does BTC performance compare to the execution without collections?

We use Hazelcast in this evaluation, a widely-used in-memory distributed data store (JOHNS, 2015). We carry out a full factorial experiment varying the mode of operation of the microservice – no garbage collector activity, BTC enabled and disabled – and the workload access pattern – i.e., read-heavy (95% reading and 5% writing) and read-only (100% reading). These workloads generate access patterns typical of user profile fetching and photo tagging, respectively (COOPER et al., 2010). The latter factor is crucial because it allows the verification of the BTC strategy when different microservice endpoints are invoked concurrently, leading to different heap usage patterns. The dependent variable measured is the response time of reading requests.

5.3.2 Experimental Setup

All measurement experiments use virtual machines with two cores and 4GB of RAM (Linux operating system version 4.15.0-39-generic). The Hazelcast deployment was composed of four replicas. We use the Nginx version 1.14.0 as the load balancer and configure it to transparently try another replica before relaying responses with HTTP status code 503 (indicator for replica unavailability). To enable BTC we use GCI, the proof of concept described in Section 5.1. All Hazelcast replicas, their respective GCI Proxies, and the load balancer were restarted between tests.

Each Hazelcast replica executed the OpenJDK version 10.0.1 2018-04-17 configured to use the G1GC (DETLEFS et al., 2004), the default GC. We set the JVM heap size to 1GB. To avoid the problem of uncontrolled collections when BTC enabled, we configured the JVM to perform collections only when the heap usage goes beying 50%. In addition to that, we configure the BTC activation interval upper bound to 500MB, so the collection triggered by BTC happens before the JVM one.

To run experiments without garbage collections, we configure the JVM to use 3.5GB as the maximum heap and run collections when the usage reached 90%. We selected those parameters after a series of trials certifying that the JRE did not trigger the GC automatically, and the operating system did not use the swap space. The experiment requires this configuration because the JRE does not allow turning off the automatic GC.

We used YCSB (COOPER et al., 2010), an industry-standard to generate the workload. The experiment begins with the store phase, leading to the insertion of $22{,}500$ records with a size of 50KB each. YCSB split the insertions evenly between the replicas, which led to ≈ 400MB of data stored per replica. The store phase was taken as the JVM warmup and, therefore, disregarded when analyzing the results.

After adequately storing the data and warming up the JVM, the load was generated by an instance of YCSB running on a virtual machine other than the replicas and the load balancer. We configured the YCSB instance to create three different generator instances – sending load from different TCP/IP connections – to emulate concurrent clients.

The generators create requests distributed according to a Zipf curve ($\rho = 0.99$), which is the access standard prescribed by YCSB. We use two types of workloads: read-heavy (95% reading and 5% writing) and read-only (100% reading). Each run of the experiment sent $75{,}000$ requests and was repeated 5 times.

5.3.3 Results

One important result is that, regardless of the load, enabling BTC does not lead to a service outage, directly impacting the client. This result confirm the efficacy of the BTC mechanism created to avoid unavailability presented Section 3.2. That means that, during the experiment run, Nginx could always transparently route evicted requests to a different replica. In other words, there was not the case when a request needed to be processed, and all Hazelcast replicas were simultaneously unavailable.

Figure 5.10 shows the response time of read-requests dispatched for read-only and read-heavy workloads. The error bars represent the range of the 99.99^{th}-percentile, for statistical confidence of 95%, calculated according to the method proposed by Nyblom (NYBLOM, 1992). We can visually answer the three research questions by analyzing this graphic. By comparing the error bars of the cases in which the BTC is disabled and enabled, we can conclude with high statistical confidence that enabling BTC reduced the response time's tail, regardless of the load.

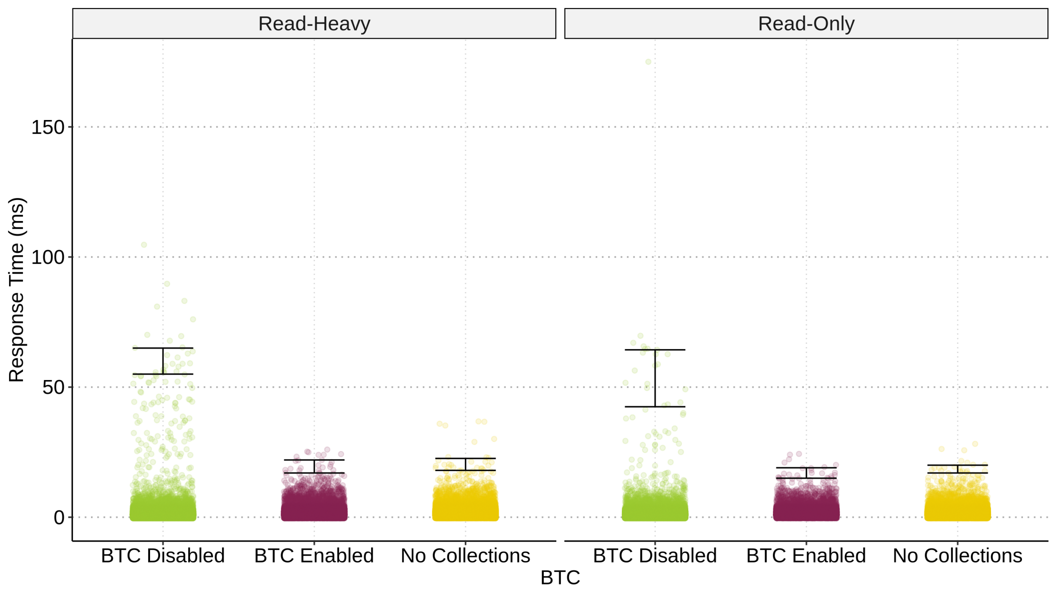


Figure 5.10: Response time of read requests dispatched for read-only and read-heavy workloads.

Furthermore, when comparing the case in which BTC is enabled and the case in which there are no collections, we notice that the tails of response times intersect. This intersection indicates that for the two workloads considered, enabling GCI leads to a performance equivalent to the case where there are no collections. That is, it eliminates the impact of the garbage collector's non-deterministic action on the response time.

We are well aware that visual confirmation is not enough. Table 5.4 details the experimental results showing the read requests' response time intervals (in milliseconds), for a statistical confidence of 95%. We can see that for the read-heavy workload, enabling BTC reduced the 99.99^{th}-percentile from $55m$ to $22ms$: an improvement of $\approx 2.5X$. In the read-only case, the improvement was $\approx 2.2X$. Besides, the distance from that percentile to the median has also been reduced by half. It is also important to note that enabling BTC did not harm other statistics, namely: average, median, 99^{th} and 99.9^{th} percentiles.

Results presented in Table 5.4 also provide us with high statistical confidence to state that enabling BTC leads to a distribution with characteristics similar to the case in which the garbage collector did not run. In other words, enabling BTC eliminated the negative impact of the garbage collector on the cloud microservice performance.

Finally, Figure 5.11 presents the results of the GC behavior analysis. The goal here is to evaluate the BTC configuration by checking if no minor GC happened when BTC is

	Read-Heavy			Read-Only		
	No Coll.	BTC Off	BTC On	No Coll.	BTC Off	BTC On
Average	(1.4 ; 1.4)	(1.2 ; 1.2)	(1.5 ; 1.5)	(1.3 ; 1.3)	(1.2 ; 1.3)	(1.5 ; 1.5)
Median	(1 ; 1)	(1 ; 1)	(1 ; 1)	(1 ; 1)	(1 ; 1)	(1 ; 1)
99^{th}-perc.	(4 ; 4)	(3 ; 4)	(4 ; 4)	(4 ; 4)	(3 ; 4)	(3 ; 4)
99.9^{th}-perc.	(9 ; 11)	(11 ; 15)	(10 ; 11)	(9 ; 10)	(9 ; 11)	(9 ; 10)
99.99^{th}-perc.	(18 ; 23)	(55 ; 65)	(17 ; 22)	(17 ; 20)	(42 ; 64)	(15 ; 19)

Table 5.4: Read requests' response time confidence intervals for read-only and ready-heavy workloads.

enabled. First, due to the generational hypothesis, the JVM does not automatically trigger full collections when BTC is disabled. Second, as we configured the activation interval correctly, there were no uncontrolled minor collections when BTC is enabled.

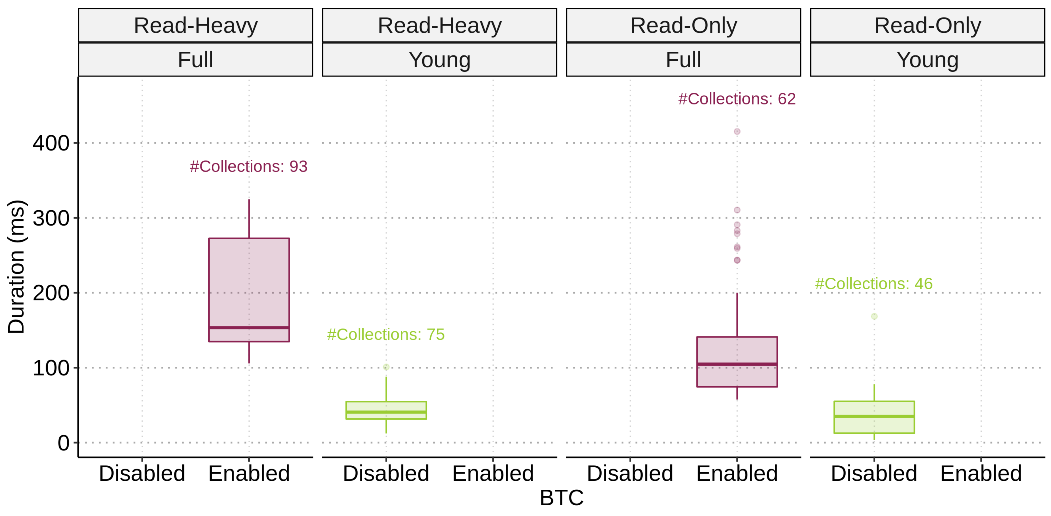


Figure 5.11: Duration and number of garbage collections for read-only and ready-heavy workloads.

5.4 Summary

Our second evaluation was focused on investigating whether the performance improves due to BTC usage in a replicate deployment. Another point explored was to understand whether the benefits of enabling BTC still apply if different endpoints – with different heap usage patterns – are invoked concurrently. Finally, we compared the BTC improvements with the optimal case, i.e., no collections. That allowed us to verify whether GCI eliminates the impact of GC on response time.

We used a four-replica deployment of Hazelcast (JOHNS, 2015) in this evaluation, a widely-used in-memory distributed datastore. The production-like workload was generated by YCSB and emulated access patterns typical of user profile fetching and photo tagging (COOPER et al., 2010). With high statistical confidence, the results show that regardless of the workload type, enabling BTC decreased the tail of the response time by a factor of $\approx 2X$ in both workload types. The performance compares to when no collections happen and do not lead to throughput loss.

Chapter 6

Related Work

This chapter summarizes the work related to performance modeling and reducing the tail of the response time tail of cloud (micro)services, the two main contributions of this thesis. Regarding the former, we focus on work that used probabilistic techniques, more specifically SPN and SRN.

As for reducing the response time tail, we present the work related to request reissuing and managing background interference. Furthermore, we also evaluate improvements on the application and background task, which aim to reduce the negative performance impact. Due to the popularity of managed languages in cloud data centers, the negative impact of automatic garbage collection has attracted much research attention. Thus, we use GC as the primary example of BT.

6.1 Performance and Availability Modeling

For online interactive cloud services, the mean values of performance measures, such as waiting time and response time, are usually considered when evaluating the user experience. Due to that importance, there was much work focusing on those statistics (URGAONKAR et al., 2005; ROY; GOKHALE; DOWDY, 2010). As the complexity of user-facing applications executing on cloud data centers increases, the attention shifted to the whole distribution, and specifically, the high percentiles (i.e., tail) has gained much attention.

Melamed and Yadin proposed a method for evaluating the response time distribution using discrete-state Markov queueing networks (MELAMED; YADIN, 1984). To solve the state-space explosion of Markov chains, even for moderately complex systems, Muppala *et. al.* designed a SRN tagged customer approach to derive the response CDF (MUPPALA et al., 1994). That approach models a closed queuing system with a fixed number of customers. Recent studies use SRN to analyze the experience availability, a metric that combines the traditional availability and tail latency for the first time into a single metric. The model takes into account partial failures and how those impact performance (CAI et al., 2017; CAO et al., 2018). Furthermore, some authors use a combination of techniques, for example, Bendechache *et. al.* used fault trees, SPN and simulation to analyze the dependability and throughput of a representative Elasticsearch application (BENDECHACHE et al., 2019a; BENDECHACHE et al., 2019b; BENDECHACHE et al., 2021).

There is also research focusing on specific aspects that affect performance. For instance, Souza *et. al.* investigated how the JBoss application server's pooling mechanism affects the response time (SOUZA et al., 2006). The authors used Discrete Stochastic Petri Nets (DSPN) to model the system and evaluate that impact. Regarding queuing, Zhang *et. al.* use SRN and the tagged customer technique to model and evaluate the impact of queuing in a multi-tier online cloud service(ZHANG et al., 2018). Furthermore, M/M/s queues (SAKUMA et al., 2011) and SRN (BRUNEO, 2014) have been used to model the impact of the waiting time on performance.

To the best of our knowledge, this thesis proposes the first taxonomy of background tasks based on their trigger. Furthermore, that taxonomy is followed by a sound formal definition of interactive BTs and a model of their impact. By making BTs a well-defined first-class citizen, we can think of better and more general solutions, as well as more accurate performance models and analysis.

This thesis proposes a solution that eliminates the negative impact of the whole class of interactive BTs. We also provide and validate a model that describes its behavior on the field: a replicated cloud microservice. That model allows assessing the solution's impact on performance and availability on a vast range of scenarios.

6.2 Tail-Tolerant Techniques

There has been quite a lot of research aiming to improve tail latency, especially in the cloud platform context. Dean and Barroso described some causes of latency spikes, the consequence of those spikes on interactive latency-critical systems, and techniques to tolerate those spikes, so-called tail-tolerant (DEAN; BARROSO, 2013).

This section describes the main tail-tolerant techniques by splitting them into three groups: i) techniques based on reissuing and routing requests, ii) techniques based on the management of the target background task, and iii) techniques that combine elements from both of the former groups. After describing all three groups, we compare them with the solution proposed in this thesis based on their main characteristics and limitations, justifying the contributions of this thesis.

6.2.1 Request Re-issuing and Routing

The Request Hedging technique consists of issuing the same request to multiple replicas and use the results from whichever replica responds first. Trying to avoid the additional load introduced by the duplication, the work suggests deferring sending of the second request until the first request has been outstanding for more than the 95^{th}-percentile expected latency. This thesis uses the term Request Duplication to refer to when copies of each request are issued simultaneously. We compare these two techniques with the solution proposed by this thesis in Section 4.7.

Bashir *et al.* present the duplicate-aware scheduling (DAS), which claims to make duplication safe and easy-to-use by leveraging prioritization and purging (BASHIR et al., 2019). By upgrading duplication to a first-class concept, DAS enables the mitigation of stragglers without overloading the system. Furthermore, this work also summarizes the differences between many different tail-tolerant mechanisms based on request re-issuing.

Another way to mitigate the increase in load brought by duplication is to select the replica to re-issue the request carefully. The main goal is to predict which replicas will serve the request in a more performant way. To make adoption easier, that prediction requires detailed latency monitoring and expensive computation for increasing accuracy, leading to higher costs (WU; YU; MADHYASTHA, 2015). There is also work focused on improving the performance of specific classes of systems, for instance, distributed storage systems (DATASTAX, 2021; SURESH et al., 2015).

Finally, Portillo-Dominguez et al. studied how to modify HTTP load-balancing algorithms to avoid the impact of major garbage collections on the response time of Java systems (Portillo-Dominguez et al., 2014; Portillo-Dominguez et al., 2015). Their approach uses runtime information to predict when a specific replica will collect and avoid routing requests to that replica. As there is no enforcement of when collections should happen, each major GC executed impacts at least one request (i.e., the GC trigger). Furthermore, mispredictions imply worse performance, and authors were not concerned with concurrent GC executions.

6.2.2 Background Task Management

Terei and Levy proposed BLADE (TEREI; LEVY, 2015), an API that allows developers to leverage existing failure recovery mechanisms in distributed systems to manage GC and bound response time. BLADE authors suggest using BLADE in interactive systems by explicitly communicating with the cluster load-balancer to remove replicas from the pool during collections. Aiming to manage the memory of replica set as a whole, Maas *et. al.* proposed Taurus (MAAS et al., 2016), a mechanism to reduce response time spikes by applying user-defined garbage collection coordination policies. The solution allows the implementation of different coordination strategies, which are applied across multiple nodes.

Cinnober proposes the combination of redundancy with JVM GC pause management to eliminate the negative effect of collection pauses on response time (CINNOBER, 2017). The TRADExpress trading system uses Remote Direct Memory Access (RDMA) to efficiently duplicate incoming requests among primary and standby nodes, which are used to increase availability. As the primary and standby servers are replicas and process the same requests, the orchestration mechanism does not need to be overly complex. The goal is that the primary server performs its JVM pause when the standby servers are actively serving incoming transactions and vice versa.

6.2.3 Request Re-issuing + BT Management

To address some of the problems described above, Kurniawan *et. al.* proposed MITMEM, a JVM replacement that provides support to improve the negative impact of GC on response time (KURNIAWAN et al., 2020). MITMEM is a drop-in replacement, requires no configuration, and can run off-the-shelf Java applications with a minimum modification (e.g., adding 120 lines of code to integrate with Cassandra).

6.2.4 Discussion

This Section presented many solutions proposed to improve the tail latency problem, which we group in three categories: i) request re-issuing and routing, which deal with requests at the client or load-balancer level; ii) background task management, which manages or orchestrate the target background task; and iii) a combination of both, i.e., "request re-issuing + BT management".

To make the comparison more accessible, we group their differences according to their specificity to the application and BT, and whether the solution eliminates background tasks' negative impact. Table 6.1 presents the main difference between the groups of work that aim to improve tail latency.

	App-specific?	BT-specific?	Eliminate BT impact?
Re-issue Requests and Routing	No	No	No
BT management	No	Yes	Yes
Re-issue req. + BT mgt.	Yes	No	No
Solution proposed in this thesis	**No**	**No**	**Yes**

Table 6.1: Comparison of the tail-tolerant techniques.

Although techniques based on re-issuing requests are not restricted to the negative impact of BTs on performance, they might not eliminate that impact. Another critical characteristic of re-issuing techniques is that they can only be used in systems in which requests are idempotent. The work proposed in this thesis is focused on eliminating the impact of interactive background tasks and is general regarding the request nature and can be used by non-idempotent cloud microservices. Finally, it can be combined with other tail-tolerant techniques, improving the coverage, and achieving even better performance gains.

We were heavily inspired by all the work on background task management, particularly the work focused on the GC. The contribution to this bulk of work is the generality. While the vast majority of the BT management solutions are either application, runtime, or BT-specific, this work is based on the formal definition of interactive BTs, enabling it to be applied to a wide range of problems.

Even though the combination of re-issuing requests and background task management is very promising, it still has the drawback of needing application adjustments. Furthermore, it relies on predicting the GC pause duration, and that is a challenging task, so they focused on a class of applications. Our solution does not rely on predictions and does not require application adjustments.

6.3 Per-Instance (Local) Improvements

Two tasks have been widely applied and studied to improve cloud systems' local performance affected by background tasks: improve the application code and the background task. This section focuses on the GC since it has been a widely-studied well-known Achiles' heel of complex interactive distributed applications running on cloud data centers (XIAN; SRISA-AN; JIANG, 2008).

Jones and Lins provide a comprehensive survey about GC techniques and improvements (JONES; HOSKING; MOSS, 2011). Even though there have always been incremental improvements, the community reaffirms its interest in that specific area when new application requirements arise, such as increasing e-commerce demand and big data. Some of those improvements are general purpose and support further application-specific optimizations through fine-tune settings (HUDSON, 2018; Azul Systems, 2015; CLICK; TENE; WOLF, 2005). There are also domain-specific GC algorithms, which use modern operating system or hardware capabilities to improve GC performance (LEE et al., 2020; GIDRA et al., 2015). Finally, there are attempts to automatically change or generate code and either avoid the GC executions or improve their performance (PARKINSON et al., 2017; GOG et al., 2015; NGUYEN et al., 2015).

Much work has been published to improve the performance by exploring the GC configuration space either manually (ORACLE, 2018a; SILVA; MARTINS; GOES, 2015) or automatically (JAYASENA et al., 2015; LENGAUER; MöSSENBöCK, 2014). Results confirm that properly adjusting the configuration to the code and load specific needs could increase throughput or response time by decreasing GC pressure, improving parallelism, or caching. However, the vast parameter space found in modern cloud deployments might make this problem intractable. For instance, some versions of the JVM allow the selection of over 600 flags. As the tunning process depends on the deployment bundle (code and configuration) and load, the issue is further amplified by polyglot microservices, the diversity of workloads, and continuous deployment.

Finally, it is also possible to perform application-level improvements, optimizing tail-prone areas with better computation and memory usage. An excellent example is the application of mechanical sympathy on software systems (Amazon Web Services, 2020). For example, the LMAX retail financial trading platform has improved its architecture and code to handle six million orders per second on a single JVM thread (FOWLER, 2011). Another example is SILK, a key-value datastore designed for reducing the impact of client writes, flushes, and compactions on latency (BALMAU et al., 2019; BALMAU et al., 2020).

Our work is complementary and benefits from the performance improvements presented in this Section. This thesis proposes an approach to deal with background task interventions, e.g., garbage collections in managed languages and log compactions in distributed datastores. As needed for the system to work, even if perfectly tuned, those tasks will hit the service performance when running concurrently with requests. Thus, there is still a need for a solution to avoid such concurrent processing. Furthermore, Chapter 3 shows that the longer and more frequent background tasks take executing, the more significant the impact of our solution. Thus, a cloud service using our solution would benefit from tasks' configuration tuning.

Chapter 7

Conclusion

A Background Task (BT) can be defined as any routine executed concurrently with the microservice's request handling. As this definition is too broad, we start by proposing a taxonomy, which groups BTs by the feasibility of control and triggers. We also formally define and use SRNs to describe the impact of controllable tasks triggered by the request handling, which we call interactive. That definition embraces important management tasks prevalent in cloud microservices, including the automatic garbage collector of managed languages and merge/optimize routines in distributed search engines. Eliminate the harmful performance effects of that class of BTs is the primary goal of this thesis.

With the problem defined and modeled, we reviewed the work related to performance and availability modeling. To the best of our knowledge, this is the first work that models the impact of interactive BTs on cloud microservices as a first-class citizen. Furthermore, we also reviewed proposals to mitigate latency spikes, i.e., tail-tolerant techniques, and found out that the proposed solutions are either: i) application or ii) BT-specific or iii) eliminate the BT impact. In other words, there is a need for a solution that is general regarding the BT and the microservice and eliminates the negative impact of BTs.

With that clear goal in mind, we designed the Background Tasks Controller (BTC), a decentralized mechanism to eliminate the impact of BT execution on interactive cloud microservices' response time. BTC trades partial availability by performance as it makes microservice replicas temporarily unavailable while triggering the executing background tasks. In addition to a formal description of the solution, we provided a performability model of BTC. That model allows the evaluation of the BTC impact on the microservice unavailability, capacity-oriented unavailability, and response time.

To avoid limitations imposed by the SRN model, i.e., only exponential rates describing parameters, we implemented a simulated model of BTC behavior. To increase the confidence in the simulator, we contrasted its results with the SRN model used to describe the solution behavior. Furthermore, we validated the simulator by comparing its results with measurement experiments.

We used the simulated model to thoroughly evaluate the BTC impact, starting with a sensitivity assessment. The assessment varied the duration and frequency of BTs, the response time, and the number of microservice replicas. One crucial finding is that the chances of system unavailability and COUA decrease rapidly by adding replicas and dropped to zero with four replicas. The BT duration and frequency are also critical factors. That is expected, as the BTC performance improvements come from exchanging the negative impact of those background tasks by a transient capacity decrease (i.e., replica unavailability).

The simulator extensibility allowed the comparison of BTC with two practical tail-tolerant techniques, i.e., Request Hedge and Duplication. Our results show that an increase in the background task frequency impacts the response time tail and the average throughput regardless of the case. Enabling BTC led to the best response time, and all tail-tolerant techniques evaluated impacted the throughput. The request hedging and duplication response time tail were 4 times worse than BTC when the execution of background tasks hit 18% of the requests. That result came at the cost of a 25% throughput decrease.

Finally, we evaluated the BTC impact with two real-world widely-used cloud microservices, i.e., Elasticsearch and Hazelcast, under different conditions. We ran measure experiments using a prototype of BTC implemented for controlling the JVM garbage collection mechanism. We varied the number of service replicas and used production and synthetic load. The results show that, regardless of the workload type, enabling BTC decreased the tail of the response time by a factor of ≈ 2. The performance compares to when no collections happen, confirming that BTC successfully eliminates the BT impact. Furthermore, it did not lead to throughput loss when the service is running with four replicas.

7.1 Results Generality

Even though we were heavily inspired by the work focused on mitigating the GC impact, we designed BTC to eliminate the impact of any interactive BT. Formally, an interactive BT is a tuple $T = (I, \zeta, A)$, where I is a start command, $\zeta : \mathbb{R} \rightarrow \mathbb{R}$, is a function that represents a measurement of the monitor associated with T, and $A = \{a \in Im(\zeta) \mid A^{min} \leq a \leq A^{max}\}$, is the subset of ζ measurements that should lead to the start of T. We are confident that this definition allows the description of a wide range of maintenance activities, which heavily contribute to increasing the latency tail of cloud and distributed services, such as backups, index defragmentation, and compactions.

The evaluation focused on Java microservices. The rationale behind that decision is the wide adoption of the language on cloud deployments. Besides the generality of the formal definition, we increased the confidence in the practicality of the solution by implementing many proofs-of-concept in many languages, i.e., Python, MRI (Ruby), Go, V8 (Javascript/Node.JS).

7.2 Future Work

Motivated by the promising results presented by Quaresma *et al.*, in the future, we would like to extend the proposed solution models to support new cloud paradigms, like Function-as-a-Service (QUARESMA; FIREMAN; PEREIRA, 2020). Besides understanding the impact of BTC in the context of dynamic scalability, that would allow a deeper understanding of the trade-offs, which may vary depending on the cloud provider.

Furthermore, we plan to study the impact of BTC on interactive cloud systems with many layers (i.e., more extensive fan-out). We believe that the solution is ready for that scenario, and the performance benefits are potentially much more significant.

Another critical area of research is the combination of tail-tolerant techniques. The importance of that lies in the fact that BTC focuses on background tasks, which is very important but is not the only cause of tail latency. More specifically, we plan to evaluate the impact of combining BTC with the request hedging technique.

Finally, we believe it is possible to decrease the replica unavailability by modifying interactive background tasks' to operate under the assumption that BTC is running. That performance improvement might come from simplification and optimization in implementing background tasks because the task would not need to manage the execution trigger. Furthermore, the fact that BTC does not allow request processing during BT execution enables the removal of complexity related to synchronization, i.e., write barriers in GC.

Bibliography

A, B.; M, T. Non-exponential stochastic Petri nets: an overview of methods and techniques. *COMPUTER SYSTEMS SCIENCE AND ENGINEERING*, AJMONEMARSAN M, 1987, LECTURE NOTES COMPUT, V266, P132 ALDOUS D, 1987, STOCH MODELS, V3, P467 BERTONI A, 1981, P 2 EUR WORKSH PETR BOBBIO A, 1992, J COMMUNICATIONS, V43, P27 BOBBIO A, 1993, P I INT WORKSH PERF BOBBIO A, 1994, LECTURE NOTES COMPUT, V852, P1, v. 13, n. 6, p. 339–351, 1998. ISSN 0267-6192. UT: 000078607500002.

ABRAHAM, L.; ALLEN, J.; BARYKIN, O.; BORKAR, V.; CHOPRA, B.; GEREA, C.; MERL, D.; METZLER, J.; REISS, D.; SUBRAMANIAN, S.; WIENER, J. L.; ZED, O. Scuba: Diving into data at facebook. **Proceedings of the VLDB Endowment**, VLDB Endowment, v. 6, n. 11, p. 1057–1067, ago. 2013. ISSN 2150-8097. Disponível em: <http://dx.doi.org/10.14778/2536222.2536231>.

AKDOGAN, H. **ElasticSearch Indexing**. [S.l.]: Packt Publishing, 2015. ISBN 1783987022.

Amazon Web Services. **Performance Efficiency Pillar - AWS Well-Architected Framework**. 2020. Disponível em: <https://docs.aws.amazon.com/wellarchitected/latest/performance-efficiency-pillar/welcome.html>. Acesso em: Online; Acessado: 2021-04-04.

Anderson, O.; Fortuna, E.; Ceze, L.; Eggers, S. Checked load: Architectural support for javascript type-checking on mobile processors. In: **2011 IEEE 17th International Symposium on High Performance Computer Architecture**. [S.l.: s.n.], 2011. p. 419–430.

AVIZIENIS, A.; LAPRIE, J.-C.; RANDELL, B.; LANDWEHR, C. Basic concepts and taxonomy of dependable and secure computing. **IEEE Trans. Dependable Secur. Comput.**, IEEE Computer Society Press, Washington, DC, USA, v. 1, n. 1, p. 11–33, jan. 2004. ISSN 1545-5971. Disponível em: <https://doi.org/10.1109/TDSC.2004.2>.

Azul Systems. **Java Without the Jitter: Achieving Ultra-Low Latency**. 2015. Disponível em: <https://www.azul.com/files/Java_wo_Jitter_v6.pdf>.

BALMAU, O.; DINU, F.; ZWAENEPOEL, W.; GUPTA, K.; CHANDHIRAMOORTHI, R.; DIDONA, D. SILK: Preventing latency spikes in log-structured merge key-value stores. In: **2019 USENIX Annual Technical Conference (USENIX ATC 19)**. Renton, WA: USENIX Association, 2019. p. 753–766. ISBN 978-1-939133-03-8. Disponível em: <https://www.usenix.org/conference/atc19/presentation/balmau>.

BALMAU, O.; DINU, F.; ZWAENEPOEL, W.; GUPTA, K.; CHANDHIRAMOORTHI, R.; DIDONA, D. Silk+ preventing latency spikes in log-structured merge key-value stores running heterogeneous workloads. **ACM Trans. Comput. Syst.**, Association for Computing Machinery, New York, NY, USA, v. 36, n. 4, maio 2020. ISSN 0734-2071. Disponível em: <https://doi.org/10.1145/3380905>.

BASHIR, H. M.; FAISAL, A. B.; JAMSHED, M. A.; VONDRAS, P.; IFTIKHAR, A. M.; QAZI, I. A.; DOGAR, F. R. Reducing tail latency using duplication: a multi-layered approach. In: MOHAISEN, A.; ZHANG, Z. (Ed.). **Proceedings of the 15th International Conference on Emerging Networking Experiments And Technologies, CoNEXT 2019, Orlando, FL, USA, December 09-12, 2019**. ACM, 2019. p. 246–259. Disponível em: <https://doi.org/10.1145/3359989.3365432>.

BAUER, E.; ADAMS, R. **Reliability and Availability of Cloud Computing**. 1st. ed. [S.l.]: Wiley-IEEE Press, 2012. ISBN 1118177010.

BENDECHACHE, M.; Silva, I.; Santos, G. L.; Guedes, L. A.; Svorobej, S.; Mario, M. N.; Ares, M. E.; Byrne, J.; Endo, P. T.; Lynn, T. Analysing dependability and performance of a real-world elastic search application. In: **2019 9th Latin-American Symposium on Dependable Computing (LADC)**. [S.l.: s.n.], 2019. p. 1–8.

BENDECHACHE, M.; Svorobej, S.; Endo, P. T.; Mario, M. N.; Ares, M. E.; Byrne, J.; Lynn, T. Modelling and simulation of elasticsearch using cloudsim. In: **2019 IEEE/ACM 23rd International Symposium on Distributed Simulation and Real Time Applications (DS-RT)**. [S.l.: s.n.], 2019. p. 1–8.

BENDECHACHE, M.; SVOROBEJ, S.; ENDO, P. T.; MIHAI, A.; LYNN, T. Simulating and evaluating a real-world elasticsearch system using the recap des simulator. **Future Internet**, v. 13, n. 4, 2021. ISSN 1999-5903. Disponível em: <https://www.mdpi.com/1999-5903/13/4/83>.

BIAGI, M.; CARNEVALI, L.; VICARIO, E.; PAOLIERI, M. An introduction to the oris tool. In: **Proceedings of the 11th EAI International Conference on Performance Evaluation Methodologies and Tools**. New York, NY, USA: Association for Computing Machinery, 2017. (VALUETOOLS 2017), p. 9–11. ISBN 9781450363464. Disponível em: <https://doi.org/10.1145/3150928.3158361>.

BLACKBURN, S. M.; MCKINLEY, K. S.; GARNER, R.; HOFFMANN, C.; KHAN, A. M.; BENTZUR, R.; DIWAN, A.; FEINBERG, D.; FRAMPTON, D.; GUYER, S. Z.; HIRZEL, M.; HOSKING, A.; JUMP, M.; LEE, H.; MOSS, J. E. B.; PHANSALKAR, A.; STEFANOVIK, D.; VANDRUNEN, T.; DINCKLAGE, D. von; WIEDERMANN, B. Wake up and smell the coffee: Evaluation methodology for the 21st century. **Communications of the ACM**, ACM, New York, NY, USA, v. 51, n. 8, p. 83–89, ago. 2008. ISSN 0001-0782. Disponível em: <http://doi.acm.org/10.1145/1378704.1378723>.

BRUNEO, D. A stochastic model to investigate data center performance and qos in iaas cloud computing systems. **IEEE Transactions on Parallel and Distributed Systems**, v. 25, n. 3, p. 560–569, 2014.

BURNS, B. **Designing Distributed Systems: Patterns and Paradigms for Scalable, Reliable Services**. 1st. ed. [S.l.]: O'Reilly Media, Inc., 2018. ISBN 1491983647, 9781491983645.

BUYYA, R.; BROBERG, J.; GOSCINSKI, A. M. **Cloud Computing Principles and Paradigms**. [S.l.]: Wiley Publishing, 2011. ISBN 9780470887998.

BUYYA, R.; YEO, C. S.; VENUGOPAL, S.; BROBERG, J.; BRANDIC, I. Cloud computing and emerging it platforms: Vision, hype, and reality for delivering computing as the 5th utility. **Future Generation Computer Systems**, Elsevier Science Publishers B. V., Amsterdam, The Netherlands, The Netherlands, v. 25, n. 6, p. 599–616, jun. 2009. ISSN 0167-739X. Disponível em: <http://dx.doi.org/10.1016/j.future.2008.12.001>.

CABALLERO, J.; GRIECO, G.; MARRON, M.; NAPPA, A. Undangle: Early detection of dangling pointers in use-after-free and double-free vulnerabilities. In: **Proceedings of the 2012 International Symposium on Software Testing and Analysis**. New York, NY, USA: ACM, 2012. (ISSTA 2012), p. 133–143. ISBN 978-1-4503-1454-1. Disponível em: <http://doi.acm.org/10.1145/2338965.2336769>.

CAI, B.-L.; ZHANG, R.-Q.; ZHOU, X.-B.; ZHAO, L.-P.; LI, K.-Q. Experience availability: Tail-latency oriented availability in software-defined cloud computing. **Journal of Computer Science and Technology**, Journal of Computer Science and Technology, v. 32, n. 2, p. 250, 2017. Disponível em: <http://jcst.ict.ac.cn/EN/abstract/article_2320.shtml>.

CAO, T.; BLACKBURN, S. M.; GAO, T.; MCKINLEY, K. S. The yin and yang of power and performance for asymmetric hardware and managed software. In: **Proceedings of the 39th Annual International Symposium on Computer Architecture**. USA: IEEE Computer Society, 2012. (ISCA '12), p. 225–236. ISBN 9781450316422.

CAO, T.; BLACKBURN, S. M.; GAO, T.; MCKINLEY, K. S. The yin and yang of power and performance for asymmetric hardware and managed software. **SIGARCH Comput. Archit. News**, Association for Computing Machinery, New York, NY, USA, v. 40, n. 3, p. 225–236, jun. 2012. ISSN 0163-5964. Disponível em: <https://doi.org/10.1145/2366231.2337185>.

CAO, Y.; ZHAO, L.; ZHANG, R.; YANG, Y.; ZHOU, X.; LI, K. Experience-availability analysis of online cloud services using stochastic models. In: CASETTI, C.; KUIPERS, F.; STERBENZ, J. P. G.; STILLER, B. (Ed.). **17th International IFIP TC6 Networking Conference, Networking 2018, Zurich, Switzerland, May 14-16, 2018**. IFIP, 2018. p. 478–486. Disponível em: <http://dl.ifip.org/db/conf/networking/networking2018/8A2-1570416570.pdf>.

CARVALHO, M.; BRASILEIRO, F. A user-based model of grid computing workloads. In: **2012 ACM/IEEE 13th International Conference on Grid Computing**. [S.l.: s.n.], 2012. p. 40–48. ISSN 2152-1085.

CHANG, F.; DEAN, J.; GHEMAWAT, S.; HSIEH, W. C.; WALLACH, D. A.; BURROWS, M.; CHANDRA, T.; FIKES, A.; GRUBER, R. E. Bigtable: A distributed storage system for structured data. In: **7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)**. [S.l.: s.n.], 2006. p. 205–218.

CIARDO, G.; BLAKEMORE, A.; CHIMENTO, P. F.; MUPPALA, J. K.; TRIVEDI, K. S. Automated generation and analysis of markov reward models using stochastic reward nets. In: MEYER, C. D.; PLEMMONS, R. J. (Ed.). **Linear Algebra, Markov Chains, and Queueing Models**. New York, NY: Springer New York, 1993. p. 145–191. ISBN 978-1-4613-8351-2.

Ciardo, G.; Muppala, J.; Trivedi, K. Spnp: stochastic petri net package. In: **Proceedings of the Third International Workshop on Petri Nets and Performance Models, PNPM89**. [S.l.: s.n.], 1989. p. 142–151.

CINNOBER. **Predictable Low Latency: Cinnober on GC pausefree Java applications through orchestrated memory management**. [S.l.], 2017. Online; Acessado: 2019-05-07.

CLICK, C.; TENE, G.; WOLF, M. The pauseless gc algorithm. In: **Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution Environments**. New York, NY, USA: ACM, 2005. (VEE '05), p. 46–56. ISBN 1-59593-047-7. Disponível em: <http://doi.acm.org/10.1145/1064979.1064988>.

COOPER, B. F.; SILBERSTEIN, A.; TAM, E.; RAMAKRISHNAN, R.; SEARS, R. Benchmarking cloud serving systems with ycsb. In: **Proceedings of the 1st ACM Symposium on Cloud Computing**. New York, NY, USA: ACM, 2010. (SoCC '10), p. 143–154. ISBN 978-1-4503-0036-0. Disponível em: <http://doi.acm.org/10.1145/1807128.1807152>.

CouchDB Team. **Apache CouchDB 3.1.1 Docs - Compaction**. 2020. <https://docs.couchdb.org/en/3.1.1/config/compaction.html>. Online; Accessed September 2020.

Couvillion, J. A.; Freire, R.; Johnson, R.; Obal, W. D.; Qureshi, M. A.; Rai, M.; Sanders, W. H.; Tvedt, J. E. Performability modeling with ultrasan. **IEEE Software**, v. 8, n. 5, p. 69–80, 1991.

COWAN, C.; Wagle, F.; Calton Pu; Beattie, S.; Walpole, J. Buffer overflows: attacks and defenses for the vulnerability of the decade. In: **Proceedings DARPA Information Survivability Conference and Exposition. DISCEX'00**. [S.l.: s.n.], 2000. v. 2, p. 119–129 vol.2.

DAIGNEAU, R. **Service Design Patterns: Fundamental Design Solutions for SOAP/WSDL and RESTful Web Services**. 1. ed. [S.l.]: Addison-Wesley Professional, 2011. ISBN 032154420X, 9780321544209.

DATASTAX. **Apache Cassandra 3.0 Snitches**. 2021. Https://docs.datastax.com/en/cassandra-oss/3.0/cassandra/architecture/archSnitchesAbout.html. Online; Accessed: 2021-04-03.

DEAN, J.; BARROSO, L. A. The tail at scale. **Communications of ACM**, ACM, New York, NY, USA, v. 56, n. 2, p. 74–80, fev. 2013. ISSN 0001-0782. Disponível em: <http://doi.acm.org/10.1145/2408776.2408794>.

DEGENBAEV, U.; LIPPAUTZ, M.; PAYER, H. Garbage collection as a joint venture. **Commun. ACM**, v. 62, p. 36–41, 2019. Disponível em: <https://cacm.acm.org/magazines/2019/6/236998-garbage-collection-as-a-joint-venture/pdf>.

DELIMITROU, C.; KOZYRAKIS, C. Amdahl's law for tail latency. **Commun. ACM**, Association for Computing Machinery, New York, NY, USA, v. 61, n. 8, p. 65–72, jul. 2018. ISSN 0001-0782. Disponível em: <https://doi.org/10.1145/3232559>.

DETLEFS, D.; FLOOD, C.; HELLER, S.; PRINTEZIS, T. Garbage-first garbage collection. In: **Proceedings of the 4th International Symposium on Memory Management**. New York, NY, USA: ACM, 2004. (ISMM '04), p. 37–48. ISBN 1-58113-945-4. Disponível em: <http://doi.acm.org/10.1145/1029873.1029879>.

EFRON, B. **The Jackknife, the bootstrap and other resampling plans**. Philadelphia, Pa.: Society for Industrial and applied mathematics, 1982. (Regional Conference Series in applied mathematics, 38). ISBN 898711797.

ELASTIC. **Stories from Users Like You**. 2014. <https://www.elastic.co/use-cases>. Online; Acessado: 2017-05-07.

ELASTIC. **Don't touch these settings**. 2017. <https://www.elastic.co/guide/en/elasticsearch/guide/current/_don_8217_t_touch_these_settings.html>. Online; Acessado: 2017-05-25.

ELASTIC. **Slow Log**. 2017. Https://www.elastic.co/guide/en/elasticsearch/reference/current/index-modules-slowlog.html. Online; Acessado: 2017-05-25.

ELASTIC. **Elasticsearch Reference [7.x] - Merge**. 2020. <https://www.elastic.co/guide/en/elasticsearch/reference/7.x/index-modules-merge.html>. Online; Accessed September 2020.

ENTEZARI-MALEKI, R.; TRIVEDI, K. S.; MOVAGHAR, A. Performability evaluation of grid environments using stochastic reward nets. **IEEE Trans. Dependable Sec. Comput.**, v. 12, n. 2, p. 204–216, 2015. Disponível em: <https://doi.org/10.1109/TDSC.2014.2320741>.

Envoy Project Authors. **Envoy Architecture Overview - Request Hedging**. 2021. Disponível em: <https://www.envoyproxy.io/docs/envoy/latest/intro/arch_overview/http/http_routing#arch-overview-http-routing-hedging>.

FARAWAY, J. J. Practical regression and anova using r. 2002.

FEITELSON, D. G. **Workload Modeling for Computer Systems Performance Evaluation**. Cambridge University Press, 2015. ISBN 978-1-107-07823-9. Disponível em: <http://www.cambridge.org/de/academic/subjects/computer-science/computer-hardware-architecture-and-distributed-computing/workload-modeling-computer-systems-performance-evaluation>.

FIELDING, R.; GETTYS, J.; MOGUL, J.; FRYSTYK, H.; MASINTER, L.; LEACH, P.; BERNERS-LEE, T. Rfc2616: Hypertext transfer protocol – http/1.1. RFC Editor, USA, 1999.

FIELDING, R.; RESCHKE, J. **RFC 7231 - HTTP/1.1 Semantics and Content**. 2014. <https://tools.ietf.org/html/rfc7231>. Online; Acessado: 2017-05-14.

FIREMAN, D.; LOPES, R.; BRUNET, J. A. Using load shedding to fight tail-latency on runtime-based services. In: **Brazilian Symposium on Computer Networks and Distributed Systems**. Porto Alegre, RS, Brasil: SBC, 2017. ISSN 2177-9384. Disponível em: <http://portaldeconteudo.sbc.org.br/index.php/sbrc/article/view/2677>.

FLOYD, S.; JACOBSON, V. The synchronization of periodic routing messages. **IEEE/ACM Trans. Netw.**, IEEE Press, v. 2, n. 2, p. 122–136, abr. 1994. ISSN 1063-6692. Disponível em: <https://doi.org/10.1109/90.298431>.

FOWLER, M. **Microservices**. 2011. Disponível em: <https://martinfowler.com/articles/lmax.html>. Acesso em: Online; Acessado: 2021-04-04.

GHEORGHE, R.; HINMAN, M. L.; RUSSO, R. **Elasticsearch in Action**. 1st. ed. USA: Manning Publications Co., 2015. ISBN 1617291625.

GIDRA, L.; THOMAS, G.; SOPENA, J.; SHAPIRO, M.; NGUYEN, N. NumaGiC: a Garbage Collector for Big Data on Big NUMA Machines. In: ACM SIGOPS, ACM SIGPLAN, ACM SIGARCH. **20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)**. Istanbul, Turkey: ACM, 2015. p. 661–673. Disponível em: <https://hal.archives-ouvertes.fr/hal-01178790>.

GOG, I.; GICEVA, J.; SCHWARZKOPF, M.; VASWANI, K.; VYTINIOTIS, D.; RAMALINGAM, G.; COSTA, M.; MURRAY, D. G.; HAND, S.; ISARD, M. Broom: Sweeping out garbage collection from big data systems. In: **15th Workshop on Hot Topics in Operating Systems (HotOS XV)**. Kartause Ittingen, Switzerland: USENIX Association, 2015. Disponível em: <https://www.usenix.org/conference/hotos15/workshop-program/presentation/gog>.

GORELICK, M.; OZSVALD, I. **High Performance Python: Practical Performant Programming for Humans**. O'Reilly Media, 2014. ISBN 9781449361778. Disponível em: <https://books.google.com.br/books?id=bIZaBAAAQBAJ>.

GORMLEY, C.; TONG, Z. **Elasticsearch: The Definitive Guide**. [S.l.]: O'Reilly Media, Inc., 2015.

GREGORY, K. **Managed, Unmanaged, Native: What Kind of Code Is This?** 2013. Https://www.developer.com/net/cplus/article.php/2197621/Managed-Unmanaged-Native-What-Kind-of-Code-Is-This.htm. Online; Acessado: 2019-29-01.

HAO, M.; SOUNDARARAJAN, G.; KENCHAMMANA-HOSEKOTE, D.; CHIEN, A. A.; GUNAWI, H. S. The tail at store: A revelation from millions of hours of disk and SSD deployments. In: **14th USENIX Conference on File and Storage Technologies (FAST 16)**. Santa Clara, CA: USENIX Association, 2016. p. 263–276. ISBN 978-1-931971-28-7. Disponível em: <https://www.usenix.org/conference/fast16/technical-sessions/presentation/hao>.

Performability Modelling: Techniques and Tools. In: HAVERKORT, B. R.; MARIE, R.; TRIVEDI, K. S. (Ed.). New York: John Wiley & Sons, 2001.

HEIMANN, D. I.; MITTAL, N.; TRIVEDI, K. S. Availability and reliability modeling for computer systems. In: YOVITS, M. C. (Ed.). Elsevier, 1990, (Advances in Computers, v. 31). p. 175 – 233. Disponível em: <http://www.sciencedirect.com/science/article/pii/S0065245808601540>.

HEWITT, J. **Cassandra: The Definitive Guide, 2nd Edition**. O'Reilly Media, Incorporated, 2016. ISBN 9781491933657. Disponível em: <https://books.google.com.br/books?id=yTc_nQAACAAJ>.

HUDSON, R. **Getting to Go: The Journey of Go's Garbage Collector**. 2018. <https://blog.golang.org/ismmkeynote>. Online; Acessado: 2019-05-13.

HUMBLE, C. **Twitter Shifting More Code to JVM, Citing Performance and Encapsulation As Primary Drivers**. 2011. <https://www.infoq.com/articles/twitter-java-use>. Online; Acessado: 2017-05-07.

HUNT, P.; KONAR, M.; JUNQUEIRA, F. P.; REED, B. Zookeeper: Wait-free coordination for internet-scale systems. In: **Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference**. USA: USENIX Association, 2010. (USENIXATC'10), p. 11.

JAIN, R. **The art of computer systems performance analysis - techniques for experimental design, measurement, simulation, and modeling.** [S.l.]: Wiley, 1991. I-XXVII, 1–685 p. (Wiley professional computing). ISBN 978-0-471-50336-1.

JALAPARTI, V.; BODIK, P.; KANDULA, S.; MENACHE, I.; RYBALKIN, M.; YAN, C. Speeding up distributed request-response workflows. **ACM SIGCOMM Computer Communication Review**, ACM, New York, NY, USA, v. 43, n. 4, p. 219–230, ago. 2013. ISSN 0146-4833. Disponível em: <http://doi.acm.org/10.1145/2534169.2486028>.

JAVADI, B.; KONDO, D.; VINCENT, J. M.; ANDERSON, D. P. Discovering statistical models of availability in large distributed systems: An empirical study of setihome. **IEEE Transactions on Parallel and Distributed Systems**, v. 22, n. 11, p. 1896–1903, nov. 2011. ISSN 1045-9219.

JAYASENA, S.; FERNANDO, M.; RUSIRA, T.; PERERA, C.; PHILIPS, C. Auto-tuning the java virtual machine. In: **2015 IEEE International Parallel and Distributed Processing Symposium Workshop**. [S.l.: s.n.], 2015. p. 1261–1270.

JOHNS, M. **Getting started with Hazelcast; 2nd ed.** Birmingham: Packt Publ., 2015. Disponível em: <http://cds.cern.ch/record/2050435>.

JONES, R.; HOSKING, A.; MOSS, E. **The Garbage Collection Handbook: The Art of Automatic Memory Management**. 1st. ed. [S.l.]: Chapman & Hall/CRC, 2011. ISBN 1420082795, 9781420082791.

KLEIN, M. Lyft's envoy: Experiences operating a large service mesh. In: . San Francisco, CA: USENIX Association, 2017.

KRISHNAN, S. T.; GONZALEZ, J. U. **Building Your Next Big Thing with Google Cloud Platform: A Guide for Developers and Enterprise Architects**. 1st. ed. Berkely, CA, USA: Apress, 2015. ISBN 1484210050, 9781484210055.

KURNIAWAN, D.; STUARDO, C.; SINURAT, R.; GUNAWI, H. **Notification and Prediction of Heap Management Pauses in Managed Languages for Latency Stable Systems**. [S.l.], 2020.

LAKSHMAN, A.; MALIK, P. Cassandra: A decentralized structured storage system. **SIGOPS Oper. Syst. Rev.**, Association for Computing Machinery, New York, NY, USA, v. 44, n. 2, p. 35–40, abr. 2010. ISSN 0163-5980. Disponível em: <https://doi.org/10.1145/1773912.1773922>.

LAMBERT, L. **Apache Solr Essentials**. North Charleston, SC, USA: CreateSpace Independent Publishing Platform, 2016. ISBN 1540385868.

LEE, H.; CHEN, Q.; YEOM, H. Y.; SON, Y. An efficient garbage collection in java virtual machine via swap i/o optimization. In: **Proceedings of the 35th Annual ACM Symposium on Applied Computing**. New York, NY, USA: Association for Computing Machinery, 2020. (SAC '20), p. 1238–1245. ISBN 9781450368667. Disponível em: <https://doi.org/10.1145/3341105.3373982>.

LENGAUER, P.; MöSSENBöCK, H. The taming of the shrew: Increasing performance by automatic parameter tuning for java garbage collectors. In: **Proceedings of the 5th ACM/SPEC International Conference on Performance Engineering**. New York, NY, USA: ACM, 2014. (ICPE '14), p. 111–122. ISBN 978-1-4503-2733-6. Disponível em: <http://doi.acm.org/10.1145/2568088.2568091>.

LEWIS, J.; FOWLER, M. **Microservices**. 2014. Disponível em: <https://martinfowler.com/articles/microservices.html>. Acesso em: Online; Acessado: 2019-03-25.

LI, H. **Introducing Windows Azure**. Berkely, CA, USA: Apress, 2009. ISBN 143022469X, 9781430224693.

MAAS, M.; ASANOVIć, K.; HARRIS, T.; KUBIATOWICZ, J. Taurus: A holistic language runtime system for coordinating distributed managed-language applications. **ACM SIGPLAN Notices**, ACM, New York, NY, USA, v. 51, n. 4, p. 457–471, mar. 2016. ISSN 0362-1340. Disponível em: <http://doi.acm.org/10.1145/2954679.2872386>.

MAAS, M.; ASANOVIC, K.; KUBIATOWICZ, J. Return of the runtimes: Rethinking the language runtime system for the cloud 3.0 era. In: **HotOS**. [S.l.]: ACM, 2017. p. 138–143.

MADAPPA, S.; NGUYEN, V.; MANSFIELD, S.; ENUGULA, S.; ENUGULA, A.; SIDDIQI, F. **Caching for a Global Netflix**. 2016. [Online; posted 01-March-2016]. Disponível em: <https://medium.com/netflix-techblog/caching-for-a-global-netflix-7bcc457012f1>.

MARSAN, M. A.; CONTE, G.; BALBO, G. A class of generalized stochastic petri nets for the performance evaluation of multiprocessor systems. **ACM Trans. Comput. Syst.**, Association for Computing Machinery, New York, NY, USA, v. 2, n. 2, p. 93–122, maio 1984. ISSN 0734-2071. Disponível em: <https://doi.org/10.1145/190.191>.

MASSEY, F. J. The Kolmogorov-Smirnov test for goodness of fit. **Journal of the American Statistical Association**, American Statistical Association, v. 46, n. 253, p. 68–78, 1951.

MELAMED, B.; YADIN, M. Numerical computation of sojourn-time distributions in queuing networks. **J. ACM**, Association for Computing Machinery, New York, NY, USA, v. 31, n. 4, p. 839–854, set. 1984. ISSN 0004-5411. Disponível em: <https://doi.org/10.1145/1634.322459>.

Meyer. On evaluating the performability of degradable computing systems. **IEEE Transactions on Computers**, C-29, n. 8, p. 720–731, 1980.

MEYEROVICH, L. A.; RABKIN, A. S. Empirical analysis of programming language adoption. In: **Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications**. New York, NY, USA: ACM, 2013. (OOPSLA '13), p. 1–18. ISBN 978-1-4503-2374-1. Disponível em: <http://doi.acm.org/10.1145/2509136.2509515>.

MISRA, P. A.; BORGE, M. F.; GOIRI, I. n.; LEBECK, A. R.; ZWAENEPOEL, W.; BIANCHINI, R. Managing tail latency in datacenter-scale file systems under production constraints. In: **Proceedings of the Fourteenth EuroSys Conference 2019**. New York, NY, USA: Association for Computing Machinery, 2019. (EuroSys '19). ISBN 9781450362818. Disponível em: <https://doi.org/10.1145/3302424.3303973>.

MOLLOY, M. K. Performance analysis using stochastic petri nets. **IEEE Trans. Comput.**, IEEE Computer Society, USA, v. 31, n. 9, p. 913–917, set. 1982. ISSN 0018-9340. Disponível em: <https://doi.org/10.1109/TC.1982.1676110>.

MUPPALA, J.; CIARDO, G.; TRIVEDI, K. Stochastic reward nets for reliability prediction. **Communications in Reliability, Maintainability and Serviceability**, SAE International, v. 1, n. 2, p. 9–20, jul. 1994.

MUPPALA, J.; TRIVEDI, K.; MAINKAR, V.; KULKARNI, V. Numerical computation of response time distributions using stochastic reward nets. **Annals of Operations Research**, v. 48, p. 155–184, 01 1994.

MUPPALA, J. K.; TRIVEDI, K. S.; WOOLET, S. P. Real-time systems performance in the presence of failures. **Computer**, IEEE Computer Society Press, Washington, DC, USA, v. 24, n. 5, p. 37–47, maio 1991. ISSN 0018-9162. Disponível em: <https://doi.org/10.1109/2.76285>.

NAIR, M. **How Netflix works: the (hugely simplified) complex stuff that happens every time you hit Play**. 2017. Https://is.gd/QQ5FRO. Acesso em: Online; Acessado: 2019-04-03.

NGINX. **Module ngx_http_proxy_module**. 2020. <http://nginx.org/en/docs/http/ngx_http_proxy_module.html#proxy_next_upstream>. Online; Acessado: 2020-06-14.

NGUYEN, K.; WANG, K.; BU, Y.; FANG, L.; HU, J.; XU, G. Facade: A compiler and runtime for (almost) object-bounded big data applications. **SIGPLAN Not.**, Association for Computing Machinery, New York, NY, USA, v. 50, n. 4, p. 675–690, mar. 2015. ISSN 0362-1340. Disponível em: <https://doi.org/10.1145/2775054.2694345>.

NORTON, J. An introduction to sensitivity assessment of simulation models. **Environmental Modelling & Software**, v. 69, p. 166–174, 2015. ISSN 1364-8152. Disponível em: <https://www.sciencedirect.com/science/article/pii/S1364815215001085>.

NYBLOM, J. Note on interpolated order statistics. In: **Statistics & Probability Letters**. [S.l.: s.n.], 1992. v. 14, n. 2, p. 129–131. ISSN 0167-7152 (print), 1879-2103 (electronic).

ORACLE. **JVM TI Reference**. 2007. <https://docs.oracle.com/javase/8/docs/platform/jvmti/jvmti.html>. Online; Accessed: 2017-05-07.

ORACLE. **Sun Java System Application Server Enterprise Edition 8.2 Performance Tuning Guide - Managing Memory and Garbage Collection**. 2010. <https://docs.oracle.com/cd/E19900-01/819-4742/6n6sfgmkr/index.html>. Online; Acessado: 2020-01-23.

ORACLE. **Java Platform, Standard Edition HotSpot Virtual Machine Garbage Collection Tuning Guide**. 2018. <https://docs.oracle.com/javase/10/gctuning/toc.htm>. Online; Acessado: 2019-05-13.

ORACLE. **MemoryPoolMXBean (Java SE 11 & JDK 11)**. 2018. <https://docs.oracle.com/en/java/javase/11/docs/api/java.management/java/lang/management/MemoryPoolMXBean.html>. Online; Acessado: 2019-04-12.

OUSTERHOUT, J.; AGRAWAL, P.; ERICKSON, D.; KOZYRAKIS, C.; LEVERICH, J.; MAZIèRES, D.; MITRA, S.; NARAYANAN, A.; ONGARO, D.; PARULKAR, G.; ROSENBLUM, M.; RUMBLE, S. M.; STRATMANN, E.; STUTSMAN, R. The case for ramcloud. **Commun. ACM**, Association for Computing Machinery, New York, NY, USA, v. 54, n. 7, p. 121–130, jul. 2011. ISSN 0001-0782. Disponível em: <https://doi.org/10.1145/1965724.1965751>.

PAHL, C.; JAMSHIDI, P. Microservices: A systematic mapping study. In: **Proceedings of the 6th International Conference on Cloud Computing and Services Science - Volume 1 and 2**. Portugal: SCITEPRESS - Science and Technology Publications, Lda, 2016. (CLOSER 2016), p. 137–146. ISBN 978-989-758-182-3. Disponível em: <https://doi.org/10.5220/0005785501370146>.

PARKINSON, M.; VYTINIOTIS, D.; VASWANI, K.; COSTA, M.; DELIGIANNIS, P.; MCDERMOTT, D.; BLANKSTEIN, A.; BALKIND, J. Project snowflake: Non-blocking safe manual memory management in .net. **Proc. ACM Program. Lang.**, Association for Computing Machinery, New York, NY, USA, v. 1, n. OOPSLA, out. 2017. Disponível em: <https://doi.org/10.1145/3141879>.

PETERSON, J. L. **Petri Net Theory and the Modeling of Systems**. USA: Prentice Hall PTR, 1981. ISBN 0136619835.

Portillo-Dominguez, A. O.; Wang, M.; Magoni, D.; Perry, P.; Murphy, J. Load balancing of java applications by forecasting garbage collections. In: **2014 IEEE 13th International Symposium on Parallel and Distributed Computing**. [S.l.: s.n.], 2014. p. 127–134.

Portillo-Dominguez, A. O.; Wang, M.; Murphy, J.; Magoni, D. Adaptive gc-aware load balancing strategy for high-assurance java distributed systems. In: **2015 IEEE 16th International Symposium on High Assurance Systems Engineering**. [S.l.: s.n.], 2015. p. 68–75.

QUARESMA, D.; FIREMAN, D.; PEREIRA, T. E. Controlling garbage collection and request admission to improve performance of faas applications. In: **2020 IEEE 32nd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)**. [S.l.: s.n.], 2020. p. 175–182.

REESE, W. Nginx: The high-performance web server and reverse proxy. **Linux Journal**, Belltown Media, Houston, TX, v. 2008, n. 173, set. 2008. ISSN 1075-3583. Disponível em: <http://dl.acm.org/citation.cfm?id=1412202.1412204>.

ROY, N.; GOKHALE, A.; DOWDY, L. Impediments to analytical modeling of multi-tiered web applications. In: **2010 IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems**. [S.l.: s.n.], 2010. p. 441–443.

RUMBLE, S. M.; ONGARO, D.; STUTSMAN, R.; ROSENBLUM, M.; OUSTERHOUT, J. K. It's time for low latency. In: **Proceedings of the 13th USENIX Conference on Hot Topics in Operating Systems**. USA: USENIX Association, 2011. (HotOS'13), p. 11.

RUMBLE, S. M.; ONGARO, D.; STUTSMAN, R.; ROSENBLUM, M.; OUSTERHOUT, J. K. It's time for low latency. In: **Proceedings of the 13th USENIX Conference on Hot Topics in Operating Systems**. Berkeley, CA, USA: USENIX Association, 2011. (HotOS'13), p. 11–11. Disponível em: <http://dl.acm.org/citation.cfm?id=1991596.1991611>.

SAKUMA, Y.; INOIE, A.; KAWANISHI, K.; MIYAZAWA, M. Tail asymptotics for waiting time distribution of an m/m/s queue with general impatient time. **Journal of Industrial & Management Optimization**, v. 7, n. 1547-5816_2011_3_593, p. 593, 2011. ISSN 1547-5816.

SALHI, H.; ODEH, F.; NASSER, R.; TAWEEL, A. Open source in-memory data grid systems: Benchmarking hazelcast and infinispan. In: **Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering**. New York, NY, USA: ACM, 2017. (ICPE '17), p. 163–164. ISBN 978-1-4503-4404-3. Disponível em: <http://doi.acm.org/10.1145/3030207.3053671>.

SARGENT, R. G. Verification and validation of simulation models. In: **Proceedings of the 37th Conference on Winter Simulation**. [S.l.]: Winter Simulation Conference, 2005. (WSC '05), p. 130–143. ISBN 0780395190.

SHAUGHNESSY, P. **Ruby Under a Microscope: An Illustrated Guide to Ruby Internals**. San Francisco, CA, USA: No Starch Press, 2013. ISBN 1593275277, 9781593275273.

SHVACHKO, K.; KUANG, H.; RADIA, S.; CHANSLER, R. The hadoop distributed file system. In: **Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems**

and Technologies (MSST). USA: IEEE Computer Society, 2010. (MSST '10), p. 1–10. ISBN 9781424471522. Disponível em: <https://doi.org/10.1109/MSST.2010.5496972>.

SILVA, L. G.; MARTINS, C. A. P. S.; GOES, L. F. W. Jvm configuration parameters space exploration for performance evaluation of parallel applications. **IEEE Latin America Transactions**, v. 13, n. 8, p. 2776–2784, ago. 2015. ISSN 1548-0992.

Smith, R. M.; Trivedi, K. S.; Ramesh, A. V. Performability analysis: measures, an algorithm, and a case study. **IEEE Transactions on Computers**, v. 37, n. 4, p. 406–417, 1988.

SOUZA, F. N.; ARTEIRO, R. D.; ROSA, N. S.; MACIEL, P. R. M. Using stochastic petri nets for performance modelling of application servers. In: **20th International Parallel and Distributed Processing Symposium (IPDPS 2006), Proceedings, 25-29 April 2006, Rhodes Island, Greece**. IEEE, 2006. Disponível em: <https://doi.org/10.1109/IPDPS.2006.1639650>.

SURESH, L.; CANINI, M.; SCHMID, S.; FELDMANN, A. C3: Cutting tail latency in cloud data stores via adaptive replica selection. In: **12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)**. Oakland, CA: USENIX Association, 2015. p. 513–527. ISBN 978-1-931971-218. Disponível em: <https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/suresh>.

TENNAGE, P.; Perera, S.; Jayasinghe, M.; Jayasena, S. An analysis of holistic tail latency behaviors of java microservices. In: **2019 IEEE 21st International Conference on High Performance Computing and Communications; IEEE 17th International Conference on Smart City; IEEE 5th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)**. [S.l.: s.n.], 2019. p. 697–705.

TEREI, D.; LEVY, A. A. Blade: A data center garbage collector. **CoRR**, abs/1504.02578, 2015. Disponível em: <http://arxiv.org/abs/1504.02578>.

TRIVEDI, K.; ANDRADE, E.; MACHIDA, F. Chapter 1 - combining performance and availability analysis in practice. In: HURSON, A.; SEDIGH, S. (Ed.). **Dependable and Secure Systems Engineering**. Elsevier, 2012, (Advances in Computers, v. 84). p. 1 – 38. Disponível em: <http://www.sciencedirect.com/science/article/pii/B9780123965257000010>.

TRIVEDI, K. S.; CIARDO, G.; MALHOTRA, M.; SAHNER, R. A. Dependability and performability analysis. In: DONATIELLO, L.; NELSON, R. (Ed.). **Performance Evaluation of Computer and Communication Systems**. Berlin, Heidelberg: Springer Berlin Heidelberg, 1993. p. 587–612. ISBN 978-3-540-48044-0.

TRIVEDI, K. S.; MUPPALA, J. K.; WOOLET, S. P.; HAVERKORT, B. R. Composite performance and dependability analysis. **Performance Evaluation**, v. 14, n. 3, p. 197–215, 1992. ISSN 0166-5316. Performability Modelling of Computer and Communication Systems. Disponível em: <http://www.sciencedirect.com/science/article/pii/016653169290004Z>.

TUFFIN, B.; CHOUDHARY, P. K.; HIREL, C.; TRIVEDI, K. S. Simulation versus analytic-numeric methods: Illustrative examples. In: **Proceedings of the 2nd International Conference on Performance Evaluation Methodologies and Tools**. Brussels, BEL: ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2007. (ValueTools '07). ISBN 9789639799004.

URGAONKAR, B.; Pacifici, G.; Shenoy, P.; Spreitzer, M.; TANTAWI, A. An analytical model for multi-tier internet services and its applications. **SIGMETRICS Perform. Eval. Rev.**, Association for Computing Machinery, New York, NY, USA, v. 33, n. 1, p. 291–302, jun. 2005. ISSN 0163-5999. Disponível em: <https://doi.org/10.1145/1071690.1064252>.

VAART, A. W. v. d. **Asymptotic Statistics**. [S.l.]: Cambridge University Press, 1998. (Cambridge Series in Statistical and Probabilistic Mathematics).

VERLAGUET, J.; MENGHRAJANI, A. **Hack: a new programming language for HHVM**. 2014. <https://code.facebook.com/posts/264544830379293/hack-a-new-programming-language-for-hhvm/>. Online; Acessado: 2017-05-07.

WU, Z.; YU, C.; MADHYASTHA, H. V. Costlo: Cost-effective redundancy for lower latency variance on cloud storage services. In: **12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)**. Oakland, CA: USENIX Association, 2015. p. 543–557. ISBN 978-1-931971-218. Disponível em: <https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/wu>.

XIAN, F.; SRISA-AN, W.; JIANG, H. Garbage collection: Java application servers' achilles heel. **Science of Computer Programming**, Elsevier North-Holland, Inc., Amsterdam, The Netherlands, The Netherlands, v. 70, n. 2-3, p. 89–110, fev. 2008. ISSN 0167-6423. Disponível em: <http://dx.doi.org/10.1016/j.scico.2007.07.008>.

YU, Y.; LEI, T.; ZHANG, W.; CHEN, H.; ZANG, B. Performance analysis and optimization of full garbage collection in memory-hungry environments. In: **Proceedings of the12th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments**. New York, NY, USA: ACM, 2016. (VEE '16), p. 123–130. ISBN 978-1-4503-3947-6. Disponível em: <http://doi.acm.org/10.1145/2892242.2892251>.

ZHANG, R.; YANG, Y.; ZHAO, L.; ZHOU, X.; CAI, B.; LI, K. A stochastic model for analyzing tail latency of multi-tier online cloud services. In: **9th International Symposium on Parallel Architectures, Algorithms and Programming, PAAP 2018, Taipei, Taiwan, December 26-28, 2018**. IEEE, 2018. p. 16–23. Disponível em: <https://doi.org/10.1109/PAAP.2018.00011>.

Appendix A

Factorial ANOVA for Sensitivityy Assessment

Table A.1: Results Factorial-ANOVA for the unavailability metric of the BTC simulator model's sensibility assessment.

Combination of Factors	DF	Sum Sq.	Mean Sq.	F-Value	p-value
#Replicas	4	0.10072	0.025181	11298.50	< 2e-16
RT Dur.	2	0.00670	0.003352	1503.90	< 2e-16
BT Freq.	2	0.00454	0.002269	1017.90	< 2e-16
BT Dur.	2	0.00350	0.001751	785.56	< 2e-16
#Replicas:RT Dur.	8	0.02335	0.002918	1309.52	< 2e-16
BT Freq.:#Replicas	8	0.01584	0.001980	888.44	< 2e-16
BT Freq.:BT Dur.	4	0.00054	0.000135	60.53	< 2e-16
BT Freq.:RT Dur.	4	0.00111	0.000278	124.95	< 2e-16
BT Dur.:#Replicas	8	0.01216	0.001520	682.24	< 2e-16
BT Dur.:RT Dur.	2	0.00088	0.000438	196.43	< 2e-16
BT Dur.:#Replicas:BT Freq.	16	0.00171	0.000107	47.95	< 2e-16
BT Dur.:#Replicas:RT Dur.	8	0.00282	0.000353	158.44	< 2e-16
BT Dur.:BT Freq.:RT Dur.	4	0.00015	0.000037	16.60	2.91e-13
#Replicas:BT Freq.:RT Dur.	16	0.00357	0.000223	100.23	< 2e-16
BT Dur.:BT Freq.:BT Freq.:RT Dur.	16	0.00042	0.000026	11.84	< 2e-16
Residuals	1245	0.00277	0.000002		

Table A.2: Results Factorial-ANOVA for the COUA metric of the BTC simulator model's sensibility assessment.

Combination of Factors	DF	Sum Sq.	Mean Sq.	F-Value	p-value
#Replicas	4	0.07460	0.018649	7628.446	<2e-16
RT Dur.	2	0.02219	0.011094	4537.824	<2e-16
BT Freq.	2	0.01510	0.007551	3088.827	<2e-16
BT Dur.	2	0.01107	0.005536	2264.309	<2e-16
#Replicas:RT Dur.	8	0.01765	0.002206	902.379	<2e-16
BT Freq.:#Replicas	8	0.01185	0.001482	606.123	<2e-16
BT Freq.:BT Dur.	4	0.00167	0.000417	170.555	<2e-16
BT Freq.:RT Dur.	4	0.00336	0.000840	343.634	<2e-16
BT Dur.:#Replicas	8	0.00918	0.001147	469.351	<2e-16
BT Dur.:RT Dur.	2	0.00258	0.001292	528.449	<2e-16
BT Dur.:#Replicas:BT Freq.	16	0.00132	0.000083	33.841	<2e-16
BT Dur.:#Replicas:RT Dur.	8	0.00221	0.000276	112.795	<2e-16
BT Dur.:BT Freq.:RT Dur.	4	0.00040	0.000099	40.437	<2e-16
#Replicas:BT Freq.:RT Dur.	16	0.00273	0.000170	69.710	<2e-16
BT Dur.:BT Freq.:BT Freq.:RT Dur.	16	0.00034	0.000021	8.677	<2e-16
Residuals	1245	1245	0.00304	0.000002	